A Design of Universal Intelligent Problem Solver

Thesis submitted by

Swarna Kamal Paul

Doctor of Philosophy (Engineering)

Department of Information Technology
Faculty Council of Engineering & Technology
Jadavpur University
Kolkata, India

2022

Under the Guidance of:

Dr. Parama Bhaumik

Associate Professor
Department of Information Technology
Jadavpur University
Salt Lake, Sector-3, Kolkata - 700106

List of All Publications

Conferences

1. Paul, S. K. & Bhaumik, P. (2014, September), "*A Density based clustering with Artificial Immunity inspired preprocessing*", In proceedings of 2014 International Conference on Advances in Computing, Communications and Informatics (ICACCI) (pp. 2648-2654). IEEE.

2. Paul, S. K. & Bhaumik, P. (2016, September), "*A fast universal search by equivalent program pruning*", In 2016 International Conference on Advances in Computing, Communications and Informatics (ICACCI) (pp. 454-460). IEEE.

3. Paul, S. K., Gupta, P. & Bhaumik, P. (2018, December), "*Learning to Solve Single Variable Linear Equations by Universal Search with Probabilistic Program Graphs*", In proceedings of International Conference on Innovations in Bio-Inspired Computing and Applications (pp. 310-320). Springer.

4. Paul, S. K. & Bhaumik, P. (2020, May), "*A Reinforcement Learning Agent based on Genetic Programming and Universal Search*", In proceedings of 2020 4th International Conference on Intelligent Computing and Control Systems (ICICCS) (pp. 122-128). IEEE.

5. Paul, S. K. & Bhaumik, P. (2021, December), "*Towards formalization of constructivist seed AI*". In ICCIS 2021, Springer, pp. 61-78

6. Paul, S. K. & Bhaumik, P. (Jan 2022), "*Disaster Management through Integrative AI*", In proceedings of ICDCN 2022 (pp. – 290-293). https://doi.org/10.1145/3491003.3493235

Journals

1. Paul, S. K. & Bhaumik, P. (2018), "*AIDCOR: artificial immunity inspired density based clustering with* outlier removal", International Journal of Machine Learning and Cybernetics, vol. 9, issue 2, pp. 309-334, https://doi.org/10.1007/s13042-016-0499-x. Electronic ISSN - 1868-808X, Print ISSN - 1868-8071.

2. Paul, S. K., Jana, S. & Bhaumik, P. (2020), "*A multivariate spatiotemporal model of COVID-19 epidemic using ensemble of ConvLSTM networks*", Journal of The Institution of Engineers (India): Series B, vol. 102, issue 6, pp. 1137-1142, https://doi.org/10.1007/s40031-020-00517-x. Electronic ISSN - 2250-2114, Print ISSN - 2250-2106.

3. Paul, S. K., Jana, S. & Bhaumik, P. (2021), "*On Solving Heterogeneous Tasks with Microservices*", Journal of The Institution of Engineers (India): Series B, pp. 1-9, https://doi.org/10.1007/s40031-021-00676-5. Electronic ISSN - 2250-2114, Print ISSN - 2250-2106.

4. Paul, S.K., Jana, S. & Bhaumik, P. (2021), "*Explaining Causal Influence of External Factors on Incidence Rate of Covid-19*", SN COMPUT. SCI. vol. 2, issue 6, article 465, https://doi.org/10.1007/s42979-021-00864-6. Electronic ISSN - 2661-8907.

5. Paul, S.K. & Bhaumik, P. (2021), "*Solving Partially Observable Environments with Universal Search Using Dataflow Graph-Based Programming Model*", IETE Journal of Research, https://doi.org/10.1080/03772063.2021.2004461. Print ISSN – 0377-2063, E. ISSN- 0974-780X

PROFORMA – 1

“Statement of Originality”

I, Swarna Kamal Paul, registered on April 2018, do hereby declare that this thesis entitled “A Design of Universal Intelligent Problem Solver”, contains a literature survey and original research work done by the undersigned candidate as part of Doctoral studies.

All information in this thesis has been obtained and presented in accordance with existing academic rules and ethical conduct. I declare that, as required by these rules and conduct, I have fully cited and referred all materials and results that are not original to this work.

I also declare that I have checked this thesis as per the “Policy on Anti Plagiarism, Jadavpur University, 2019”, and the level of similarity as checked by iThenticate software is ________%.

Signature of Candidate:

Date:

Certified by Supervisor:

(Signature with date, seal)

PROFORMA – 2

CERTIFICATE FROM THE SUPERVISOR

This is to certify that the thesis entitled "A Design of Universal Intelligent Problem Solver" submitted by Shri Swarna Kamal Paul (Regd. No.: 229/18/E), who got his name registered on 16/04/2018, for the award of Ph.D. (Engineering) degree of Jadavpur University, is absolutely based upon his own work under the supervision of myself and that neither this thesis nor any part of it has been submitted for either any degree/diploma or any other academic award anywhere before.

Dr. Parama Bhaumik
Associate Professor
Department of Information Technology
Jadavpur University
Salt Lake Campus, Sector-3, Kolkata-700106

To Mother Nature …

Acknowledgments

This thesis is the much-awaited outcome of my hard work and many others who have led to the successful completion of my Doctor of Philosophy in Engineering from Jadavpur University. First of all, I would like to express my honor towards Mother Nature who is the greatest teacher of all. I am thankful for her blessings and the teachings she offered. I would like to extend my sincere thanks to Dr. Parama Bhaumik, Assistant Professor, Department of Information Technology, Jadavpur University for giving me the chance to pursue my doctoral work under her guidance. I would like to thank her for her continuous support, encouragement, and for leading me to the successful completion of my thesis. I would also like to thank Professor Samiran Chattopadhyay and respected Professors of the Department of Information Technology, Jadavpur University for providing valuable guidance to pursue my work. I am thankful to my co-worker Mr. Saikat Jana, who helped me out with several important contributions. I would like to express my gratitude to Mr. Shikhar Nath Sil for providing me all the encouragement and supporting me in my tough times. Finally, my deep gratitude to my son without whom this thesis would have been incomplete. I would like to thank my family, all my friends, and staff members of Jadavpur University who have lent their helping hands in pursuing my doctoral work.

Swarna Kamal Paul

Abstract

General problem-solving in the real world requires high level of adaptability across multiple problem environments such that a problem solver should be able to solve problems in varied environments. A problem is usually solved by searching through a solution space. The search space should be of manageable size relative to the search speed and storage capacity in order to find a solution in a realistic time. However, for a majority of the real-world problems, the situation is barely the expected one. A brute force search would easily get lost in the combinatorial explosion of the search space. Thus, some sort of intelligence is required to dampen the search space and perform a focused search. Humans are good at adapting across different problem environments and consequently are effective general problem solvers. This is not quite true for artificial agents. Most of the research on artificial intelligent agents is mostly focused on domain-specific problems and achieving adaptability across environments is still a challenging task. Few pieces of research focused on designing general problem-solving agents based on strong theoretical groundings. Yet, creating a practically feasible agent which can sense and act optimally in varied environments under resource constraints of time and space is still far from trivial. Thus, we focused on designing a practically feasible general problem-solving agent. We took an integrative approach where multiple components can be integrated synergistically to build a problem solver. The solutions are represented as programs in a proposed programming model. The problem-solving agent searches through program space using generate and test approach to find solutions in varied problem environments. Solution programs can integrate multiple disparate components including sensors and actuators to interact optimally in a heterogenous problem environment. The search space is dampened using policy gradient-based incremental learning, equivalent program pruning, and genetic programming. We experimented with our designed agent in multiple problem environments and the results reinforced our claims. Comparison with the current state-of-the-art methods revealed the excellent performance of our developed agent. Last but not least we proposed a formal structure of a seed AI that has the capability to evolve into a general AI system.

Table of Contents

1 INTRODUCTION 1

1.1 MOTIVATIONS 1
1.2 GENERAL PROBLEM SOLVING 2
1.3 ARTIFICIAL GENERAL INTELLIGENCE 7
1.4 SCOPE OF THE THESIS 9
1.4.1 Design a programming model ideal for AI component integration and universal search 10
1.4.2 Design a mechanism for AI component integration 11
1.4.3 Design a problem-solving agent using universal search 12
1.4.4 Formalize seed AI 12
1.4.5 Conduct experiments with the problem-solving agent 13
1.5 ORGANIZATION OF THE THESIS 13

2 A SURVEY ON ARTIFICIAL INTELLIGENT PROBLEM SOLVING APPROACHES 15

2.1 SYMBOLIC AI 15
2.2 CONNECTIONIST AI 17
2.3 ARTIFICIAL LIFE 19
2.4 PROGRAM SEARCH BASED AI 20
2.5 INTEGRATIVE AI 22
2.6 CONCLUSION 25

3 FGPM: A PROGRAMMING MODEL FOR INTEGRATIVE AI AND UNIVERSAL SEARCH 27

3.1 EVALUATION STRATEGY OF PROGRAM GRAPHS 28
3.2 INTEGRATING MONADIC FUNCTIONS 30
3.3 AGENT ENVIRONMENT STRUCTURE 33
3.4 COMPARISON WITH CURRENT STATE OF THE ART 33
3.5 SYNTAX OF FGPM 35
3.6 DATATYPES OF FGPM 36
3.7 FUNCTIONS OF FGPM 36
3.7.1 Functions for program construction 37
3.7.2 I/O monads 40
3.7.3 First-order node functions 41
3.7.4 Higher-order node functions 50
3.8 CONCLUSION 55

4 THE INTEGRATIVE AI PLATFORM 57

4.1 AGENT-ENVIRONMENT BASED INTEGRATION PLATFORM 58
4.1.1 Middleware 60
4.1.2 Microservices 60
4.2 COMPARATIVE CASE STUDY 61
4.2.1 Learning to solve heterogeneous maze task 61
4.2.2 Speech recognition 64
4.3 SUMMARIZED COMPARISON 66
4.4 FEW CASE STUDIES ON DISASTER MANAGEMENT 67
4.4.1 Machine failure disaster prevention 68
4.4.2 Forest fire detection from satellite image 68
4.5 CONCLUSION 69

5 DESIGN OF A UNIVERSAL SOLVER 71

5.1 UNIVERSAL SEARCH 71
5.2 FGPM FOR UNIVERSAL SEARCH 73
5.2.1 Meaningful information gain in transfer learning 73
5.3 METASEARCHER 76
5.4 PROGRAM PRUNING 78
5.5 INCREMENTAL LEARNING 80
5.6 HEURISTIC PLAYOUT 83
5.7 MERGING GENETIC PROGRAMMING WITH UNIVERSAL SEARCH 83
5.7.1 Crossover and mutation 84
5.7.2 Selection 85
5.7.3 Avoiding local optima 86
5.7.4 Metasearcher with GP 86
5.8 EXPERIMENTAL RESULTS 87
5.1 CONCLUSION 93

6 APPLICATIONS OF THE UNIVERSAL SOLVER 95

6.1 LEARNING TO SOLVE SINGLE VARIABLE LINEAR EQUATIONS 95
6.1.1 Composite node functions 96
6.1.2 Generating solutions 98
6.2 NEURAL ARCHITECTURE SEARCH USING UNIVERSAL SOLVER 99
6.2.1 Search space 100
6.2.2 Search algorithm 101
6.2.3 Evaluation strategy 102
6.2.4 Metasearcher as search method for NAS 102
6.2.5 Experimental results for CIFAR-10 dataset 104
6.2.6 NAS for COVID-19 spatiotemporal forecasting model using universal solver 107
6.3 CONCLUSION 114

7 TOWARDS CONSTRUCTIVIST SEED AI 115

7.1 ABSTRACT MODELLING OF SEED AI 116
7.2 FORMALIZATION OF SEED AI 119
7.2.1 Seed AI algorithm 120
7.2.2 Constructivism 121
7.2.3 Learning 122
7.2.4 Adaptability across environments 123
7.2.5 Recursive improvement 125
7.3 SEED AI IMPLEMENTATION WITH UNIVERSAL SEARCH 126
7.3.1 Agent as an environment 128
7.3.2 Case study 129
7.4 CONCLUSION 131

8 CONCLUSION AND FUTURE WORK 132

9 REFERENCES 138

List of Figures

Figure 3.1 Illustration of a) a constant node, b) a guard node (conditional), c) an addition node, d) an apply node, e) a program graph to add 2 numbers where iW represents initial node........................28

Figure 3.2 Illustration of execution strategy of programs in FGPM 29

Figure 3.3 Illustration of an invalid FGPM program due to parallel combination of I/O monad. ...32

Figure 3.4 Illustration of a lambdagraph function in FGPM. 51

Figure 3.5 Illustration of a recurse function in FGPM 52

Figure 3.6 Illustration of a loop function in FGPM. 53

Figure 3.7 Illustration of a fmap function in FGPM...54

Figure 3.8 Illustration of aggregator function in FGPM..55

Figure 3.9 Illustration of a zip function in FGPM 55

Figure 4.1 Agent-environment model of AI integration...60

Figure 4.2 Architecture of integrative AI platform 61

Figure 4.3 The maze problem environment..62

Figure 4.4 Program graphs to solve maze problem and speech recognition problem. 65

Figure 4.5 Program graph for machine failure detection...68

Figure 4.6 Program graph for forest fire detection 69

Figure 5.1 A sample program with parallel data flow to demonstrate how storing conditional probabilities in graph helps in transfer learning. 74

Figure 5.2 A sample search graph generated by the metasearcher. 78

Figure 5.3 a) Crossover operation in a program graph. b, c) Mutation operation in program graph.. 85

Figure 5.4 A sample 20x20 maze. 87

Figure 5.5 a) Plot for comparison of runtime of metasearcher with and without equivalent program pruning. b) Plot for comparison of inverse of program probability of solutions found by metasearcher with and without incremental learning. 89

Figure 5.6 A subgraph from search graph for gridworld problem..93

Figure 6.1 Illustration of the program graph of composite function CI 97

Figure 6.2 Program graphs of composite functions.. 97

Figure 6.3 Solution program graph found by the agent for solving equation set 1 98

Figure 6.4 Solution program graph found by the agent for solving equation set 2. 98

Figure 6.5 Solution program graph found by the agent for solving equation set 3. 99

Figure 6.6 Illustration of interaction between the metasearcher and the environment for neural architecture search..103

Figure 6.7 Neural network architecture found by the metasearcher with a validation accuracy of 89.93%..105

Figure 6.8 Neural network architecture found by the metasearcher with a validation accuracy of 88.72%..106

Figure 6.9 a) Illustration of overlapping frames obtained by spatially dividing a geographical region. b) Illustration of sequence of a frame. 108

Figure 6.10 A simple RNN unfolded across multiple timesteps. 110

Figure 6.11 Plot of loss vs epoch and mean squared error (MSE) vs epoch for both training and validation data set..112

Figure 6.12 Neural architecture found by the metasearcher..113

Figure 7.1 Architecture of an embodied intelligent agent 125

Figure 7.2 Maze environment..130

Figure 7.3 Solution program graphs a) Program graph for the maze composite node. b) solution program graph found by the metasearcher for the maze problem environment……………………130

List of Tables

Table 3.1 Comparison of proposed Functional Graph Programming Model (FGPM) with Cuneiform and Apache Beam 34

Table 4.1 List of atomic microservices 62

Table 4.2 List of function nodes 63

Table 4.3 Feature Comparison 67

Table 5.1 Training sequence for the metasearcher 90

Table 5.2 Test sequence for the metasearcher. 90

Table 5.3 Comparison of US-agent with current state of art methods 91

Table 6.1 List of external functions used in the search 96

Table 6.2 Kernel Map used for Neural architecture search for CIFAR-10 dataset 103

Table 6.3 Kernel Map used for Neural architecture search for Covid-19 spatiotemporal model ... 113

Table 6.4 Model training, validation and test results 113

List of Algorithms

Algorithm 5.1 Metasearcher algorithm 77

Algorithm 5.2 Heuristic Playout algorithm...83

Algorithm 5.3 Metasearcher with GP...86

Algorithm 7.1 Seed AI 121

Algorithm 7.2 Seed AI implementation with the metasearcher 127

1 INTRODUCTION

"A problem exists when a living organism has a goal but does not know how this goal is to be reached" – Karl Duncker (1945)

1.1 MOTIVATIONS

Much of the Artificial Intelligence (AI) field today is concerned with creating algorithms or solutions to demonstrate intelligence for one or other very specialized problems like chess playing, object recognizing from image, automobile driving, etc. Whereas a general problem solver should be capable of solving problems and adapting across a wide range of environments. Engineering rule-based expert systems with a pre-loaded knowledge base quickly become practically intractable due to the sheer volume of rules and knowledge required to solve few common real-world tasks, which humans can perform with ease. Thus, it is quite evident, some form of intelligence and learning capability is required to select and generate contextual knowledge for adapting from one problem environment to another. "Artificial General Intelligence" or AGI was a term coined to represent the concepts and methods developed to deal with creation of such an intelligent general problem-solving machinery. Many authors explored this avenue and attacked the problem of general problem solving with different flavors. Tremendous scientific progress has been made and many important concepts have been developed, which allows one to create a theoretically complete artificial general intelligent agent. Yet we hardly have any practically feasible method which can seamlessly operate in a real-world across multiple problem environments. The fundamental problem in dealing with real-world environments is scarcity of resources in terms of time, space, and knowledge. Searching for solutions in an arbitrary environment from scratch almost always leads to a combinatorial explosion in one form or another. Thus, it is of utmost necessity to generate, capture and reuse knowledge as much as possible to dampen the search space during the operation life-cycle. Though this is one of the fundamental goals of artificial intelligence, yet it is far from trivial in the context of general problem-solving. Knowledge gained in one domain may not be straightforward to apply in a different domain. Also, in order to act on variety of problem environments the artificial agent

should have the capability of interacting with all the environments and generating arbitrary logical pathways. Keeping these objectives in mind, we focused on developing and improving mechanisms for efficient knowledge capture, generation, transfer, and integration across multiple environments. We utilized the power of integrative intelligence, such that reusable solutions can be easily integrated to solve problems in various environments. This would dampen the search space as searching all solutions from scratch can be avoided. However, integrative intelligence is not simple plug-n-play type solution design method. Different components need to be integrated in a synergistic way, which makes it challenging. We focused on developing methods to efficiently integrate disparate AI components to solve heterogeneous tasks and tried to design a universal intelligent problem solver which has the capability to sense, plan and act to solve problems in multiple environments. The problem solver has been designed loosely based on the steps of the general problem-solving process which remains valid both for biological and artificial agents. Lastly, we explored what is needed to build an autonomous seed AI, based on integrative intelligence, which can adapt across multiple environments without explicit manual drive.

1.2 General Problem Solving

"A problem exists when an information-processing system has a goal condition that cannot be satisfied without a search process" [1] – this is a nicely rephrased version of the original definition of a problem by Duncker [2]. The beauty of this definition lies in its broad sense as it remains valid, both for a biological organism and artificial computing machines, of course considering a biological organism as an information-processing system. This definition provides a necessary and sufficient condition for a problem to exist. The simple existence of a goal condition does not imply the existence of a problem if the solution path is known beforehand or it can be accessed without a search. The non-existence of a solution is a necessary condition for a problem to exist, where solution refers to a sequence of operations that can satisfy the goal condition [3]. The problem can be solved by searching for a solution and finding one. The same problem encountered twice will not be a problem in the 2nd encounter if the problem is solved in the 1st attempt and the solution is known during the 2nd. For example, adding two integers is a problem for a child who is still learning the addition process. A child may have multiple different sets of understanding for the

addition rules and methods, during her initial learning stage. She has to apply rules as per her different set of understanding and evaluate the result to verify if it is a correct one. That involves some level of searching. Once the rules are well understood, adding two numbers does not remain a problem as the path to obtain the solution becomes known and well-defined. Therefore, some narrow contextual problem-solving methods may need to be redefined or updated from time to time to consider them as problem-solving methods. Once a problem is solved and a solution is known it does not remain a problem. Thus, a solver designed to solve that specific problem does not remain a solver once the problem is solved.

A problem can be represented using four different components. An *initial state* – description of the state from where the problem-solving process begins; a *goal state* – state that needs to be reached in order to solve the problem; *operators* – actions that can be taken to change states; *constraint* – conditions that need to be satisfied while executing the solution path from the initial state to goal state [4]. A problem can be categorized as *well-defined,* if all the above-mentioned components are clearly specified, otherwise, it is ill-defined. Mathematical problems are mostly well-defined, whereas many of the real-world problems are ill-defined. For example, finding a solution to completely eradicate Covid-19 in a specific period of time is an ill-defined problem as an appropriate set of operators is still unknown. Full vaccination of a population may be one way but there is no guarantee that some mutated strain of the virus cannot evade the vaccine protection. An ill-defined problem may be converted to a well-defined problem by finding an appropriate set of operators, formulating exact goal conditions, defining initial conditions. In this case, problem representation itself can become another problem, and so on. A problem can also be classified as an *adversary* or *non-adversary* problem [1]. In a non-adversary problem, the solvers interact with non-responsive objects and the environment does not compete with the solver. In an adversary problem, the solver competes with an opponent in the environment who wants to defeat the purpose of the solver. A problem may be solved by finding a task or a sequence of tasks with respect to the problem-solving methods. Tasks are entities that can be executed by applying a known sequence of actions in an environment. Tasks need to be executed in an environment. A major difference between task and a problem is in a task the problem-solving methods are known whereas in a problem it is not [5]. There are multiple types of task environments categorized in the literature. Following are different types of task environments.

Fully observable vs partially observable – In a fully observable environment, the complete state information can be derived at any point in time whereas in a partially observable environment it is not.

Deterministic vs stochastic – In deterministic environment, the next state is completely determined by the current state and action whereas in a stochastic environment next state probabilistically depends on the current state and action.

Episodic vs sequential – In an episodic environment the solver's interaction with the environment are divided into episodes. In each episode, a sequence of action and observation trail is generated and in the next episode, a reset is applied in the environment.

Static vs dynamic – A static environment does not change during interaction with the solver whereas in case of a dynamic environment it is otherwise.

Discrete vs continuous – In a discrete environment, the actions and observations are limited within a discrete finite set whereas in a continuous environment it is otherwise.

Solving a problem is essentially a search process. Much of the solving process involves dealing with failed steps such that the solver gradually becomes aware of multiple routes of failure and eventually it helps to narrow down the search process. The complete steps of a problem-solving process can be enumerated as follows [6].

Detect the problem to be solved – The first step of problem-solving is to get aware that a problem exists. A problem can be assigned by an external entity in the environment or it may be realized internally. Internal realization of a problem comes from either discovering a subproblem while solving another problem or due to a drive to solve certain problems that are imprinted during the initialization of the information-processing system. For example, while solving an ill-defined problem a subproblem can be realized for discovering the appropriate set of operators, which would make the ill-defined problem a well-defined problem. In another scenario, finding ways for survival is an instinctive problem assigned to nearly all biological information-processing systems during initiation.

Formulate a concrete definition of the problem – After knowing the existence of a

problem the next step is to formulate a concrete internal representation of the problem to be solved. This involves identifying all the components of the problem. Namely, goal conditions, initial state, available operators, and constraints. Constraints can be readily provided by the environment or maybe derived through assumptions made, based on existing knowledge. In case of ill-defined problems, finding an internal representation of the problem becomes another problem that needs to be solved.

Identify a problem-solving method – Once concrete definitions of a problem are found, problems can be solved by direct state space traversal using the operators until the goal state is achieved. In this scenario, the whole problem is represented as a state-space where a path needs to be found from the initial state to the goal state by applying a sequence of operators. In another case, the problem may be divided into sub-problems and so on until fixed solutions are known for the respective sub-problems. Thereafter, individual subgoals can be achieved and aggregated to achieve the higher subgoal and so on, until the main goal is achieved. However, this method of problem-solving can also be mapped to state-space traversal, where the operators are, dividing problems into subproblems, identifying known solutions of sub-problems if there are any, and aggregating solutions. The goal state is the aggregated solution of the main problem.

Generate potential solutions – As problem solving is essentially a search operation, solutions need to be searched through generate and test method. Potential solution paths can be generated by sequencing operators. Usually, there can be an infinite number of potential solutions if no prior knowledge exists about probable solution paths. The search can easily get lost in a combinatorial explosion. However, domain knowledge, experiential knowledge, etc. plays a crucial role in dampening the search space. Learning is a mechanism to gain such knowledge while solving similar problems or contrasting problems and observing patterns across different problems. Knowledge creates initial biases during problem-solving and drives the solver towards known pathways to exploit the already learned solution. However, that might prevent the system from further learning and augment the knowledge base which eventually results in settling with suboptimal solutions in many cases. Thus, a balance between exploitation and exploration is needed for generating optimal solution pathways. Exploration is the art of controlled investigation of other solution pathways that do not

readily conform with the already learned pathways. The drive for exploration can be controlled by setting parameters during initiation of the system, experiential factors like age of the system, and internal/external problem-specific factors. For example, humans may feel bored after trying to solve a difficult problem for a while and stop exploring. In another scenario, the environment may assign a fixed time for solving a problem, and exploration gets restricted due to time constraint.

Evaluate candidate solutions and select the best solution – In order to identify the correct solution among a set of potential solutions, the solutions need to be evaluated and tested. Usually, the testing needs to be done in a simulated environment instead of an actual environment and there might be multiple criteria to evaluate the fitness of a solution. For example, fitness can be measured based on how close the solution goes toward the goal state or how much resources are consumed to implement the solution, etc. There might be certain constraints imposed by the environment or by some internal assumptions. It also needs to be checked if the solutions satisfy those constraints. In many cases, the generation of potential solutions and evaluation of solutions are interleaved and iterative. In course of carrying out these two operations, the system may learn about the problem landscape and eventually use the learnings to generate more promising potential solutions. Also, there might be multiple potential solutions generated in one pass. The system should organize the solutions in a certain order and evaluate them through a scheduling process such that promising solutions are evaluated with more priority. After evaluation, the best solution or a set of solutions is selected for implementation.

Develop a plan and implement the best solution – Once the best possible solution or set of solutions are selected, a plan needs to be made to implement them in the actual environment. Not all solutions may be executed in the actual environment due to resource constraints and other circumstances. Certain solutions may be delayed until an appropriate situation arrives. A set of solutions may be scheduled in some specific order to act on the actual environment. After implementation of a solution in the actual environment the goal may be reached and experiential learning gathers the required knowledge to improve performance in subsequent similar problems.

The steps of general problem-solving can provide high-level guidance in developing an artificial problem-solving agent. The field of Artificial General Intelligence deals with the design and development of such agents. Several works have been done and ideas have been proposed in formalizing the design of such agents. However, making it practically feasible across multiple problems is still a challenge.

1.3 Artificial General Intelligence

Biological organisms apply some sort of intelligence to handle general problem-solving. Similarly, this idea can be extrapolated for artificial computing agents. However, artificial agents imparted intelligence or trained in a narrow domain may not handle all the aspects of general problem-solving. For example, an agent carrying a trained neural network for identifying objects in an image will fail miserably if placed in a route planning problem environment. Initially, the grand vision of creating human-like intelligent digital agents was part of the scope of AI research. However, it was eventually understood the inherent difficulty in achieving human-like general intelligence across multiple domains. Thus, a typical AI research considered focusing on specific domains, and eventually, a concept of Artificial General Intelligence (AGI) or Universal Artificial Intelligence was born which clung onto the grand vision of achieving general intelligence.

An intelligent being, such as a human, has the ability to adapt in an arbitrary environment and learn from experience [7]. Machine general intelligence can be defined under similar lines. Among several definitions of "Universal Artificial Intelligence" available in the literature, we would consider the following definition to suit our goal.

Universal Artificial Intelligence measures the agent's ability to act optimally or achieve goals in a wide range of environments under constrained resources like space, time, and knowledge [8]

The above definition of AGI is most suitable in a Reinforcement Learning (RL) setting, provided the environment can generate rewards and punishments for actions taken by an agent in the environment. Acting optimally can refer to maximizing the cumulative future expected reward by the agent. Though it seems such a problem can be handled quite easily

by existing RL techniques, yet the problem is far from trivial when the environment is unknown and varied. A general intelligent agent should be able to adapt across varied environments and environments can be of different types, as mentioned in the last section. The environment may produce delayed reward, which makes it difficult for an RL agent to converge towards an optimal policy. By optimal policy, it is meant as the sequence of actions that need to be taken in an environment to achieve some goal. On top of that transfer learning from one environment to another is non-trivial. Thus, an agent trained in one environment may not be able to transfer its experience and training to a different environment which results in solving the new environment from scratch. A realistic agent should also consider resource constraints while solving a problem environment. Time, space, and knowledge may be scarce and it is expected the agent should behave optimally to the maximum extent with whatever resources are available. If the agent is unable to achieve the goal it should display graceful degradation. With all such concerns, it becomes quite an engineering task to design a general intelligent agent.

There have been several theoretical artificial general intelligent agents proposed in the literature, yet a major concern lies in practically realizing such agents mainly due to resource constraints. For example, Hutter's AIXI framework [9] is a complete theoretical definition of an AGI agent in the sense that it can optimally act in any environment given infinite computational resources. A modified framework was also proposed, named as AIXItl which runs in a bounded computation time. However, it lacks incremental learning, due to which there is no reduction in computation time by utilizing experience. Godel machine is a self-referential, self-improving optimal problem solver originally proposed by Schmidhuber [10]. A probable implementation roadmap was given using the continuous passing style (CPS) of programming and meta-circular evaluators [11]. However, proof-based systems face some challenges in POMDPs [12] and the search for proofs can also be expensive at times. Apart from these two theoretically complete architectures of AGI, there are several architectures proposed to date and multiple approaches are taken to construct an AGI architecture. The approaches are discussed in detail in Chapter 2. However, from practical feasibility perspective almost all architectures face challenges when the agents need to adapt across environments under constrained resources.

The next level challenge is to create a self-improving intelligent agent. Self-improvement by self-modification of an intelligent agent is necessary to adapt across environments and continue acting optimally. Since it is nearly impossible to create an intelligent agent with all possible rules and knowledge to act optimally in all real-world problem environments, a concept of seed AI was born. It states an agent bootstrapped with a minimal set of knowledge and an algorithm for processing the knowledge. The seed AI is expected to evolve into general AI and learn to adapt across multiple environments. Self-improvement can be recursive in the sense the seed AI program which searches for an improvement can get replaced with a better version of itself. However, such improvements are difficult to find with all possibility of getting into a combinatorial explosion. A formal structural definition of such a complete seed AI is still non-existent in literature. An abstract model of the seed AI and formalization of the seed program can guide the construction of general AI through varied approaches.

1.4 Scope of the Thesis

This thesis focuses on designing a practically feasible universal intelligent problem solver. An integrative AI approach along with program search-based methods [13] has been taken to come up with such a design. The primary focus is to address the challenges in the existing methods with respect to constrained resource usage, which seems to be one of the major roadblocks in creating an Artificial General Intelligent agent. A design of a RL agent has been proposed that can handle arbitrary general problems and the steps applied to solve a problem are loosely based on steps of general problem solving as mentioned in section 1.2. Last but not the least, a design of a seed AI has been formalized. Its properties are enumerated and an implementation roadmap has been demonstrated by implementing a prototype.

Our objectives can be summarized as follows.

To design a computable reinforcement learning agent which can solve problems in a wide range of environments provided the environment generates feedback signals in the form of rewards at minimum.

The agent should utilize limited available resources in an optimal way to solve problems

The agent should be capable to learn from experiences and exploit them wherever possible.

Derive a formal definition of seed AI and demonstrate how the proposed RL agent can be qualified as a seed AI.

While designing our agent, we focused mostly on the program search-based design, inspired by universal search [14], due to the theoretical enrichments in the field. However, several ideas of integrative AI are also used while maturing the design. Thus, our approach can be broadly classified as integrative AI. The following aspects of the problem are addressed individually while designing the agent.

1.4.1 Design a programming model ideal for AI component integration and universal search

Universal search [14] in the true sense is asymptotically optimal for all machine inversion and time-limited optimization problems. This means time and space requirement for solving these problems is a constant factor away from the current best. However, though this constant factor is independent of problem size, yet it can be immensely large and depends on the solution size. Universal search searches for a solution in a program space on some machine model. The complexity of the search process is invariant on the machine model or the language chosen. Thus, the choice of grammar for the programs only affects the search by a constant factor. However, for practical application of universal search, we need to focus on dampening the constant factor. Transfer learning is a way to dampen the combinatorial explosion in the program space which enables knowledge gained in solving one task to be transferred to another related task. Given the focus is to dampen the program search space, the choice of a programming model is of high importance. Thus, we propose a programming model, based on dataflow graphs to be used for universal search. Dataflow graphs [15] are good at implicitly capturing data dependencies and independencies among different functions in a program. The relative independencies among different functions allow the program graph to be divided into logically independent subgraphs which solve separate specific subtasks. These subgraphs can be directly reused in solving subsequent related problems, thus making the transfer learning process efficient. However, most of the existing dataflow graph-based languages [15, 16, 17] were developed either with the intention of achieving implicit parallelization of code or allowing a programmer to graphically construct

a program with ease or both. For application in universal search, we identified the necessity of several features. Like, metaprogramming – the ability to generate and modify other programs from one, dynamical program generation – generating new programs during runtime of one, reuse of arbitrary code chunk, a simple but expressive syntax, and easy reasoning of programs to handle the problem of overrepresentation [18] which occurs due to presence of semantically redundant programs in the search space. The program interpreter should have some added features like storing and managing conditional probability distribution among functions and interrupting programs based on run time. Also, the language model should be flexible enough to integrate any external components developed in another language as a module. We also plan to use the programming model for integrating disparate AI components and serve it as an integration method for an integrative AI platform. At times, such integrations might be done by a human programmer as well. Thus, high expressivity, modularity, reusability, and abstraction are of utmost necessity to reduce the engineering effort of constructing an integrative AI application. Applications developed by a human programmer through AI component integration can in turn be used by the agent for specific problem solving, which we refer to as initial bias provided by a human to the agent. In order to address all these challenges, we decided to develop a custom programming model to be used in an integrative AI platform and a universal search based problem-solving agent.

1.4.2 Design a mechanism for AI component integration

Manual integration of AI components is necessary at times to provide an initial bias to the problem-solving agent. In many scenarios, it becomes computationally expensive and unnecessary to conduct a solution search from scratch where solutions of certain sub-problems are known beforehand. There should be a scope for a human programmer to construct solutions by integrating disparate AI components with ease. This would allow humans to augment the ability of the problem-solving agent wherever possible and thereby dampen the search space. However, using AI components to solve real-world heterogeneous tasks is more than simple plug-and-play type integration and often necessitates tight coupling of components. With the current state of the art, such integration consumes large engineering effort due to problems like low expressivity, modularity, reusability, and abstraction. To alleviate these problems a platform for AI component integration is proposed. The

integration can be done using the proposed dataflow graph-based functional programming model and treating all AI components as microservices. Along with loose coupling, the integration platform also allows tight coupling of components to solve complex heterogeneous tasks without compromising modularity. A graphical method of construction of integration programs relieves a developer from learning programming syntax. The usability of the proposed method has been demonstrated by solving a heterogenous maze problem where an agent needs to solve a reinforcement learning environment, a machine vision task, and integrate them. Solving a speech recognition problem demonstrates modularity and functional abstraction of the method. Qualitative comparison with current state-of-the-art methods, based on these use cases, justifies the claims.

1.4.3 Design a problem-solving agent using universal search

Universal search allows one to create asymptotically optimal intelligent agent which can act in a wide range of computable environments. However combinatorial explosion is still lurking behind though its dependency shifted from problem size to solution size. For example, if a simple solution exists for a problem environment with a large state space size, universal search can quickly find the solution, as the time taken to find a solution is invariant with respect to the problem size. However, for many real-world problems, the solution size may not be small enough to find it in a realistic time. To combat this scenario, the proposed dataflow graph based functional programming model is used in universal search for solution program generation. We have justified the superiority of our proposed model compared to sequential token-based languages when used in universal search based intelligent agent. We have shown how applying an equivalent program pruning strategy can handle the problem of semantically redundant program generation. An incremental learning strategy based on gradient ascent is also proposed for our designed agent. Experimental results positively reinforced the theoretical justifications. We used our agent to solve some partially observable environments and compared with current state-of-the-art methods and it reveals exceptional performance of our agent.

1.4.4 Formalize seed AI

It has already been hypothesized that seed AI needs to be bootstrapped in a system that could evolve to handle multiple problem domains [19]. Such a seed AI may consist of few core

intelligent components and a seed program. However, there is no formal structural definition of seed AI in place. Thus, an abstract model of the seed AI is proposed and its generality has been proved. The formal structure of such an algorithm has also been derived. It has been discussed how and in what setting the proposed model of seed AI can achieve different properties of an intelligent agent like adaptability, constructivism, learnability, and recursive self-improvement. A prototype of seed AI has been developed using universal search to demonstrate and present guidance on the physical implementation of an agent-based system. The agent has been experimented in a heterogeneous toy problem to illustrate its usability

1.4.5 Conduct experiments with the problem-solving agent

We conducted several experiments with the problem-solving agent. Experiments with several toy problems and a comparative study with respect to the current state-of-the-art methods reveal exceptional performance of our proposed method. We also conducted experiments in two different real-world problem domains. In one case the solver needs to find solutions for simple algebraic equations and in another, it needs to find optimal neural architectures for certain computational problems. In both cases, the proposed method gave satisfactory results.

1.5 Organization of the Thesis

The thesis is organized as follows. It is recommended to follow the sequence of the chapters to maintain the continuity of the ideas presented in this thesis. However, the sequence of Chapter 6 and Chapter 7 can be interchanged if the reader finds it suitable.

Chapter 2: A survey on Artificial General Intelligence

This chapter presents a brief survey of the major works in the field of Artificial General Intelligence to date. The survey has been segmented under five different major distinct approaches that are usually taken in pursuing AGI research.

Chapter 3: FGPM: A programming model for Integrative AI and Universal Search

The proposed graph programming model is articulated in this chapter. Its syntax and semantics are presented and all primitive functions are described in detail.

Chapter 4: The Integrative AI platform

The integrative AI platform is proposed in this chapter. Two case studies implemented in this platform have been showcased and a comparative study is conducted with respect to two current state-of-the-art methods.

Chapter 5: Design of a Universal Solver

The design of a universal problem solver is proposed. The solver is built on the concepts of Universal Search. It has been experimented in few benchmark toy problems and results are compared against the current state-of-the-art methods.

Chapter 6: Applications of the Universal Solver

The universal solver is applied to two distinct problem areas, namely, finding solutions for simple algebraic equations and neural architecture search. Thereafter, results are presented.

Chapter 7: Towards Constructivist seed AI

A formal model of the seed AI is proposed and it has been demonstrated how universal solver fits into it.

Chapter 8: Conclusion and Future work

A final conclusion of the thesis and few suggested scope of future work.

2 A Survey on Artificial Intelligent Problem Solving Approaches

"The problem of moving on flat surfaces is solved quite well by wheels, but generalizing the wheel might not be the best solution to move around on general surfaces" – Ben Goertzel (March, 2008)

Biological organisms are already good at general problem solving. Humans have evolved to become the best general problem solver of all. However, mimicking natural problem-solving process and replicating it in an artificial substrate is quite a challenge. Though we have a fair understanding of the steps of general problem solving, yet we are unclear on how humans apply them across various problem domains quite efficiently. There have been several researches to create and understand general problem-solving in artificial agents. These researches mainly fall under Artificial General Intelligence, where an artificial agent needs to apply some form of intelligence to handle multiple problems across different environments/domains efficiently. The research area can be broadly classified under the following five different categories, based on their approaches [13].

2.1 Symbolic AI

Symbolic AI is an approach to AI, based on the manipulation of knowledge represented in language-like symbolic structures in which all relevant semantics is explicit in the syntax (formal structure).

A physical symbol system has been defined by Newell & Simon, 1976, p.116 [20]:

"*A physical symbol system consists of a set of entities, called symbols, which are physical patterns that can occur as components of another type of entity called an expression (or symbol structure). At any instant of time the system will contain a collection of these symbol structures. Besides these structures, the system also contains a collection of processes that operate on expressions to produce other expressions: processes of creation, modification, reproduction and destruction."*

Several AGI related projects were undertaken under Symbolic-AI paradigm. One such work was Simon and Newel's famous project called "General Problem Solver" [21]. It used heuristic search to solve problems [13]. It had a separate knowledge base in the form of rules and a solver engine which deduced a strategy for solving a problem. GPS used divide and conquer approach where it divided the original problem into subproblems and so on until a subproblem is simple enough to be solved by heuristics. GPS was able to solve some sufficiently formalized problems like the tower of Hanoi by expressing problems as well-formed formulas. But while solving any real-world problems it could get easily lost in a combinatorial explosion as there was no learning involved.

Another famous project in Symbolic-AI paradigm was the CYC project undertaken by Doug Lenat [23]. The project targeted creating AGI system by encoding all commonsense and various domain specific knowledge required for problem-solving and efficient reasoning. But the encoding effort turned out to be a huge effort to create a complete knowledge base for a full-blown AGI system that can perform efficiently in dynamic environments. However, a vast knowledge base was created in an explicit logical form and a complex inference engine was developed. Knowledge was encoded in a predicate logic syntax called as Cycl.

Allan Newell and John Laird's SOAR project targeted to build a system to grasp human-level AGI. The original theory underlying SOAR is called as problem space hypothesis [24]. The problem space hypothesis states that all goal-oriented behavior can be represented as a search through state space of the problem while attempting to achieve the goal. At each step, an operator is selected and applied, which might change the agent's internal state or modify some external environment through some actions. Currently, it seems that the SOAR project has retreated into a system for experimenting some cognitive science theories.

Pei Wang's Non-Axiomatic Reasoning System (NARS) [25] is an attempt to create uncertainty-based, symbolic reasoning system. The main objective of the system is to adapt across varied problem environments under constrained resources and knowledge. It focuses to impart three distinct features within the system. Namely, a system with finite information-processing capability, real-time handling of problems, and openness with respect to input data structure. It uses a term-oriented formal language called Narsese for knowledge

representation. It is characterized by the use of subject-predicate sentences. The truthfulness or validity of statements in its knowledge base is determined by evidential support from experience and validity of such statements, also called beliefs, gets updated from time to time. The inference engine uses a non-axiomatic logic, also known as NAL, and works on Narsese statements. It derives new pieces of knowledge from existing knowledge. It is also claimed that a certain form of 'self' can be evolved within the system, mainly through meta-level mechanisms [26].

2.2 CONNECTIONIST AI

Connectionism is an approach to achieve Artificial Intelligence by mimicking the human brain and mental activity at neuron level. An intelligent system can be built by connecting multiple relatively simpler computation units called as neurons in a certain structure. Communication among the units is achieved through signal transmission across the connections and learning is achieved by modifying the connection strengths based on experience. Stephen Grossberg [27] designed special neural network models carrying out specialized functions modeled on specific brain regions. As a learning mechanism, self-organizing parallel neural network architecture for pattern recognizer was proposed. Such pattern recognizers were used in visual perception, speech recognition, etc.

Hugo de Garis [28] introduced the idea of evolvable hardware which applies evolutionary algorithms to the generation of programmable hardware as means of building artificial brains. A FPGA based hardware known as "CAM-Brain machine" was introduced which implemented genetic algorithm to evolve cellular automata based neural network circuit. Cellular automata based neural networks are evolved in different modules which form the components of the Artificial Brain.

Due to immense progress in hardware performance and solving few engineering barriers like vanishing gradient problem, building and deploying deep neural networks to solve real-world use cases have become quite common. The power of deep neural networks has already been demonstrated in solving many domain specific tasks like machine vision, natural language understanding, etc. However, very little works has been done to create domain independent general intelligent systems using deep neural networks. Arel [29] suggested combining deep

machine learning models with reinforcement learning to create AGI architectures. Deepmind followed this concept and tried to solve the problem of AGI by combining deep learning on convolutional neural network with Q–learning, a form of model-free reinforcement learning [30]. It was tested on several video games where it received pixel data and game scores as inputs. The agent was able to surpass existing previous algorithms and achieved a performance comparable to a human game tester. Schmidhuber [31] claimed a fast deep recurrent network built for machine vision that obtained benchmark performances on several datasets, would prove an effective tool for building an AGI system. OpenAI is an organization that works on building AGI system that works for the best interest of humans, considering all safety goals. Till now they have achieved benchmark performances in few domain specific problems. For example, GPT-3 [32] is an autoregressive language model that uses deep learning to generate realistic texts that become indistinguishable from any human writing. GPT-4 [110] is a multimodal model that can accept text and image and produce text outputs. Similar to GPT3, it is a large transformer based model trained to predict next token using an excessively large corpus of data and it does not learn from experiences. Dall-e [33] is a text to image generation model that generates realistic images when given the description of the image as input. These models by themselves do not resemble AGI in any sense but they may prove useful tools in building one through integrative methods. GATO [109] is a generalist transformer model proposed by google. It has been trained on multimodal data transformed to sequence of tokens and produces sequence of tokens as output. GATO was trained and tested on multiple tasks like image captioning, game playing robotics control etc. However similar to GPT it requires excessively large corpus to train and it is also not evident how online learning will be effective in this model.

Though recent advancements in deep neural networks helped achieve benchmark results in several problems like speech recognition and machine vision and they surpassed in performance compared to almost all other previous techniques. However, though deep neural networks perform well in a specific task when designed to carry out that specific task, they are not good generalizers. Learning to solve new tasks requires immense amount of training dataset and computational power. Neural networks designed for a specific task, in general, do not perform well in different tasks. Thus, they need to be manually designed for different tasks. Recently, Kaiser et. al. [107] proposed a single neural network model which can learn

to solve multiple problems. However, it is not obvious the same model will perform well in any arbitrary problem and self modification is not applied. There are several methods developed for automatic neural architecture search [85] for different tasks, but they are also quite resource hungry. Also solving a problem may require solving several tasks and integrating the solutions in some manner. There is no straightforward automated way of integrating multiple neural networks responsible for solving individual tasks, in an arbitrary way to solve an arbitrary problem.

2.3 Artificial Life

Artificial life (ALife) studies the fundamental processes of living systems and interaction between them in artificial environments to gain a deeper understanding of complex information processing that defines such system. While synthesizing artificial life in a computing machine, it might be possible that the system may eventually evolve into a generally intelligent ALife. T.S ray was able to create a digital version of life called as Tierra [34]. Tierra used an evolutionary process and was able to create unicellular organisms. The drive for the evolutionary process was given as organisms' ability to replicate and adapt in an environment. Ray further worked on Tierra [35] to produce multicellular organisms and eventually created a distributed system that allow organisms to migrate and exploit in different environments. Few multicellular creatures were evolved in the system.

Avida [36] is a platform developed in Caltech to study and experiment with replicating and evolving digital organisms under various environments. It was created to address multiple purposes. One of which was to study evolution of digital organisms at a much faster scale which otherwise would be impossible in nature. The second important purpose was to find solutions for computational problems using digital organisms which would in fact sparked the possibility of creating a general problem solver. The Avida software is comprised of two major components. The first is an Avida core, an environment, a scheduler, and data collections components. Avida core maintains the population, replication, and evolution of digital organisms and the environment provides responses to the organisms during an interaction. The scheduler assigns computation resources to the Avida organisms. The second set of components is a collection of analysis and statistics tools. The authors proposed

several setups for experimenting with the Avida software.

ALife is an interesting approach to solve general AI problem. Though it might be possible to create a general intelligent agent which could solve real-world problems in the future, yet till now digital ALife simulators are mainly restricted in creating simple mini-organisms. These tools are good at experimenting and studying the progression of life in simple digital organisms but deploying them to adapt across real-world environments and solve problems would need further significant work.

2.4 PROGRAM SEARCH BASED AI

Program search based AGI is relatively a newer approach of designing AGI architectures. The approach was initially inspired by the seminal work of Solomonoff, Chaitin and Kolmogorov on algorithmic information theory in the 1960s, but it did not become a serious approach to practical AI until quite recently. Hutter developed a reinforcement learning agent [9] by combining Solomonoff's induction [37, 38] with sequential decision theory [39]. Hutter's AIXI framework is a complete theoretical definition of an AI agent in the sense that it can optimally act in any environment given infinite computational resources. A reinforcement learning agent with some utility function can plan its actions to maximize its future expected reward in an environment with a known probability distribution. However, problem arises when the environment distribution is unknown and this is the most common scenario for real-world problems. AIXI can optimally deal with these environments by using Solomonoff's universal prior probability as the distribution for environments. The agent's objective is to maximize some rewards across all environments. The problem of uncomputibility of the Solomonoff's prior was overcome by using a resource bounded version of it (bounded by time t and space l). The modified framework is named as AIXItl whose computation time is bounded by $t.2^l$ and this can also be immensely large at times as there is an exponential factor involved, which is proportional to solution size. It lacks incremental learning, due to which there is no reduction in computation time by utilizing experience.

Jurgen Schmidhuber made an efficient practical implementation of universal search for solving an ordered set of problems [40] known as Optimal Ordered Problem Solver (OOPS).

It employs incremental learning and tries to reuse solutions found in earlier problems to solve later problems and in the course of doing so it tries to find the most general solution solving all the ordered set of tasks. For solving the nth task, it uses half of the search time in extending and testing the previous successful programs and other half in testing fresh programs with arbitrary beginnings. In course of doing so, it tries to find out the most general program which solves all the ordered set of tasks. At the core, it uses a variant of Levin search [41]. OOPS was able to solve Towers of Hanoi with 30 disks which is unsolvable by traditional reinforcement learners. OOPS can provide possible speedup for a series of related tasks and not for a single task. It works in a resettable environment only.

Schmidhuber developed a more general theoretical framework called as Godel machine [10] which is a self-referential, self-improving optimal problem solver. The system starts with a proof searcher and an initial problem-solving code that interacts with the environment. Employing a variant of universal search, the proof searcher searches for proofs which states that a rewrite of the problem-solving code and/or the proof searcher itself is beneficial in terms of some utility function. If such proofs are found, the self-rewrite is applied. Schmidhuber points out that the Gödel machine could start out by implementing AIXI or some other program as its initial sub-program, and self-modify after it finds proof that another algorithm for its search code will be more optimal. A probable implementation roadmap was given using continuous passing style (CPS) of programming and meta-circular evaluators [11]. However, proof-based systems face some challenges in POMDPs [12] and the search for proofs can also be expensive at times.

Looks and Goertzel presented a program representation in functional style with several transformation rules to alleviate the problem of over-representation [18]. Maximizing correlation between syntactic and semantic distance among programs will help minimize chaotic execution and manage resource variance. It was claimed that the proposed program representation and transformations increase the correlation between syntactic and semantic distance.

Lukaz Kaiser [42] presented a program search based AI where the objective is to find a better algorithmic model of an arbitrary problem environment which eventually helps the agent to

take optimal actions. The agent interacts with the actual environment through sensor and actuation signals. The signals are maintained as event history and the model of the environment is searched which best explains the event history under constrained resources, namely time and space. An actor program is used to model the optimal expected behavior of the agent, which is used to calculate the response of the actor for an input event. A self-improving program search mechanism has also been described. It uses a combination of program search and proof search techniques. Thus, the problems of combinatorial explosion related to these two approaches are prevalent in this case also.

2.5 Integrative AI

Integrative AGI involves taking part or all elements from the above approaches and create a combined and synergistic system. However, an integrative approach does not refer to building each aspect of AGI as separate modules and then integrating them. The different approaches are too different from each other. Instead, a general framework of intelligent agent needs to be developed incorporating ideas from different approaches. Novamente [43] is an integrative artificial general intelligence design, which integrates aspects of many prior AI projects and paradigms, including symbolic, probabilistic, evolutionary programming, and reinforcement learning approaches. The overall architecture is unique, drawing on system-theoretic ideas regarding complex mental dynamics and associated emergent patterns. These are facilitated by a novel knowledge representation which allows diverse cognitive processes to interact effectively. Two primary cognitive algorithms are used to construct these processes: Probabilistic Term Logic (PTL), and the Bayesian Optimization Algorithm (BOA). OpenCogPrime [44] a newly designed AGI is a modification and enhancement of the Novamente Cognition Engine.

Cognitive synergy [45] theory provides a theoretical framework to achieve AGI. A collection of cognitive processes acts in a synergistic way to control a single cognitive agent. The interconnectedness among the processes aids each other in overcoming memory-type specific combinatorial explosions during diversified knowledge creation. A series of hierarchical formal models starting from a simple reinforcement learning (RL) agent in the sense proposed by Hutter [9] to a complex and more specific PrimeAGI agent, is proposed,

to achieve a step-by-step design of such a cognitive agent. The RL agent can be improved to a cognit agent by introducing a kind of introspection. Cognit agent is a collection of atomic cognits which can act on external environments as well as internal memory objects. The next phase is organizing the memory of cognit agents as labeled hypergraphs giving rise to a model of hypergraph agent. The nodes and edges of the hypergraph are called as atoms and the cognit agents are represented as either atoms or collection of atoms (subhypergraph). Thus, a diverse set of cognit agents form a connected network which will enable hierarchical and heterarchical learning. The next level of transformation is to represent hypergraph atom types with a rich language where each atom can be labeled with some programmatic operators and thus called as rich hypergraph agent. The next level in the hierarchy is Probabilistic Growth and Mining Combinations (PGMC) based agents [46]. The core idea is to have a cognitive control process for a specific cognitive process to find out ways to extend the current cognitive process transition graph based on goal-oriented pattern mining from systems history. The final level is the PrimeAGI [47, 48] cognitive architecture implemented within the software platform called as OpenCog, which works within the "PGMC driven rich hypergraph memory model" agent framework and extending it via introducing a specific set of cognitive processes. While OpenCog solves many of the problems of integrative AI yet integrating new AI capabilities needs application-programming-interface (API) development within the framework, making black-box plug-n-play type integration non-trivial. The heart of the framework relies on the Atomspace knowledge base which again involves development effort to map existing third-party knowledge graphs to Atomspace.

Honda ASIMO robot implements cognitive map architecture [49] for integrating multiple AI capabilities and uses constructionist design methodology [50]. They followed a component-based architecture where individual components interact with each other through message passing using a publish-subscribe mechanism. The components are mainly categorized based on their functional roles, namely – perception, knowledge representation, decision-making, and expression. Though the architecture solves many problems of human-computer interaction, yet it might not scale well in distributed environments like IoT or cloud-based AI integration.

Andrist et. al. [51] and Bohus et. al. [52] demonstrated a framework for development of

physically situated intelligent systems. The platform is built on .NET. It provides a programming model to integrate multiple AI components and built a situated intelligence platform that would be capable of reacting in a real-time environment. The platform provides a runtime that is capable of running programs created in the platform. The programs are internally represented as computation graphs, where a node represents components and edges represents streams of data. It also provides an ecosystem of components for multimodal audio-video sensing and processing. New custom components can also be built in the platform. Several low-level operational issues in a real-time streaming system, like synchronization across multiple channels, handling latency, etc. are handled within the runtime and abstracted away from the developer. However, integrating external components is not straightforward which needs to be mapped to native components of the framework. Since it works mainly on streams, it might not easily solve use cases that involve working on bounded datasets, like recognizing objects in a single image that can be uploaded by a user on an adhoc basis. Also, for developers, it has some learning curve involved, since integration program needs to be implemented in C#.

Thorisson proposed a theory of constructing a seed programmed intelligent system which rests on certain hypotheses of learning [53]. The theory contextualizes three high-level aspects of problem-solving, namely task-environments, cumulative learning, and seed program. The task environment represents the problem environment in which the agent needs to act upon. The agent can modify state values in a task environment through action signals and read observations through perceptions. The agent is supposed to achieve specified goals in the task environments. A seed AI is essentially a learner which is bootstrapped with few essential knowledge about the environment in which it is born. It should then be capable enough to evolve into a full-blown autonomous cumulative learner by turning gathered experience into knowledge. Knowledge is represented as causal relational models and patterns. Both are abstract and can represent knowledge at any level of detail in a hierarchical form. It can be manipulated, composed, and compared. Reasoning in all forms is used to manipulate and create knowledge at fine level of details and it is termed as micro-ampliative reasoning. Though it has been hypothesized that the proposed system would bear the characteristics of seed-programmed autonomous generality, yet a strict formalization of seed AI is not articulated.

Nivel et. al. presents an architecture blueprint for constructing a bounded seed AGI that can achieve operational autonomy in an underspecified environment [54]. They developed a prototype system AERA, which can learn complex real-world tasks. The three challenges of learning through experience, coping with resource limitations, and managing attention are addressed by learning, planning, and controlling attention mechanism, respectively. Learning and planning are not achieved by designing separate modules for the same but they are claimed to be emergent processes within the system from low-level processes. Knowledge in this system is represented by states and models. Models are executable codes that can generate new knowledge. The design of the system allows self modeling and introspection. The system was experimented in a human-to-human interaction scenario where the agent is to interact with another human effectively in real-time. The agent was able to learn proper sentence construction, gesture control, multimodal co-ordinations, and appropriate response generation. However, there is a serious open issue with this design. There is no guarantee that the system will actually pursue and achieve user-defined goals [22]. Also, it is not clear if the system would work in distributed mode, with multiple sensors and actuators not embodied in a single hardware.

Stueinbrick et. al. introduced an experience-based AI system called as EXPAI [55] which can search for safe and beneficial self-modification. Several limitations of proof-based approach of self-modification have been highlighted and an experience-based approach was chosen for self-modification. However, there are no experimental results provided to measure the performance of the proposed system.

TaskMatrix.AI [111] proposed by Microsoft is an integrative AI platform that uses a foundational model to understand the user intent and the task. Thereby, it generates solution by leveraging multiple APIs that can perform specific subtasks across different domains. TaskMatrix platform can perform both digital and physical tasks, can be augmented by adding new APIs and solutions are interpretable. It has several challenges and the foremost is building the foundational model that needs to handle multimodal tasks.

2.6 Conclusion

Multiple different approaches were taken by multiple researchers to attack the grand problem

of Artificial General Intelligence. Several important breakthroughs were made and advancements in the field seem quite promising. However, almost all of the prior-arts have some limitations in terms of the physical implementation of a real-world general problem-solving system. The majority of the problem arises due to limited resources available in real-world for solving problems. Other problems include, high engineering effort required to make the system adaptable across multiple environments, unable to work in distributed environments, etc. Thus, we focused on developing methods to overcome these challenges and tried to make an advancement in realizing a practically feasible general intelligent problem solver. We incorporated ideas from multiple prior works and tried to design an integrative Artificial Intelligent system capable of handling multiple problem domains.

3 FGPM: A PROGRAMMING MODEL FOR INTEGRATIVE AI AND UNIVERSAL SEARCH

"*The communicating of ideas marked by words is not the chief and only end of language, as is commonly supposed.*" - George Berkeley, A Treatise Concerning the Principles of Human Knowledge, (1710)

A dataflow graph is a directed acyclic graph where nodes represent some operation and directed edges represent the flow of data. The data dependency of individual operations can be nicely represented in a dataflow graph. A dataflow graph-based programming model allows a programmer to create an arbitrary computation graph in the form of a directed acyclic graph. Most of the existing dataflow graph-based languages [15, 16, 17] were developed either with the intention of achieving implicit parallelization of code or allowing a programmer to graphically construct a program with ease or both. For application in universal search, we identified the necessity of several features. Like, metaprogramming, dynamical program generation, reuse of arbitrary code chunk, a simple but expressive syntax, easy reasoning of programs to handle the problem of overrepresentation [18]. The program interpreter should have some added features like storing and managing conditional probability distribution among functions and interrupting programs based on run time. Also, the language model should be flexible enough to integrate any external components developed in another language as a module without compromising purity. Thus, we decided to develop a custom programming model specifically to be used in universal search based problem-solving agent and for integrating external AI components.

The programs in the proposed Functional Graph Programming Model (FGPM) are represented as dataflow computation graphs. A function node can have multiple input ports, each serving as a placeholder for separate input arguments. It can have only one output port that emits the computed output of the function. However multiple edges can emanate from an output port and output value is copied on each output edge. Input and output ports are type-casted. Edges can run between output port of one node and input port of another, satisfying type compatibility. A program can be composed by connecting edges between multiple nodes. The programming system is preloaded with several primitive functions for

carrying out essential operations like arithmetic/logical computations, conditional pathways, Input/Output operations, repetitions, and construction/manipulation of different datatypes/functions. An arbitrary program graph can also be stored as a function node, consequently making it reusable. Each complete executable program must start with an initial node and end with a single terminal node. The initial node is responsible for initializing the computation environment. The computation environment represents the external world and the program communicates with it through I/O nodes, named as sensors and actuators. For example, standard input/output in a file can represent an environment and the program communicates by running sensor and actuator nodes which in turn reads or writes in the file. Fig. 3.1 illustrates few of the primitive function nodes and a sample program graph. Fig. 3.1a represents a constant node that returns a constant value (denoted as a) irrespective of any input value. Fig. 3.1b represents a conditional node. If the 1st argument is True the parent

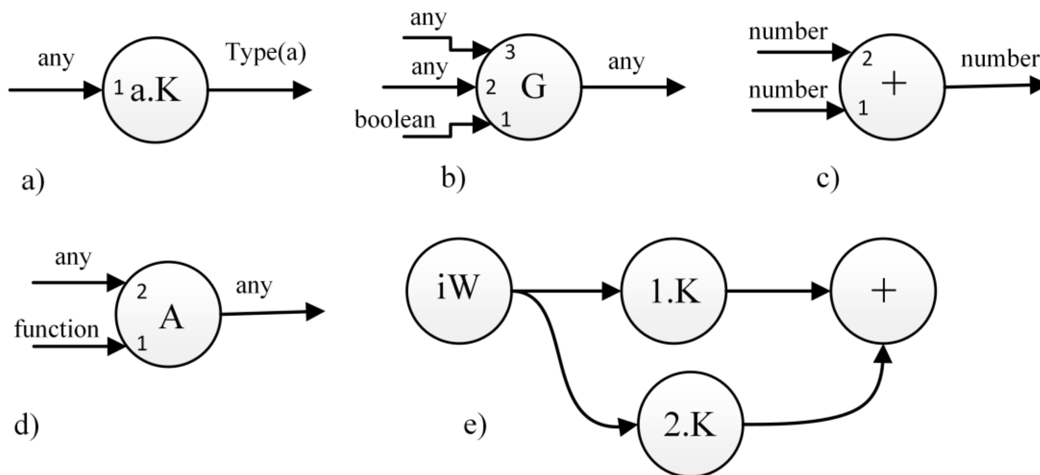


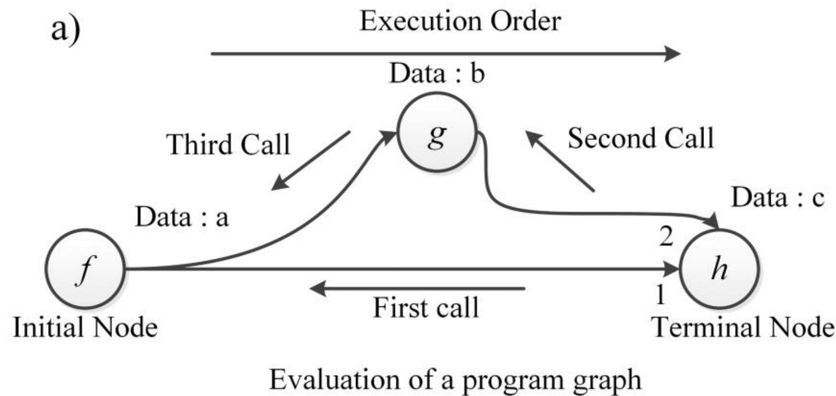
Figure 3.1 Illustration of a) a constant node, b) a guard node (conditional), c) an addition node, d) an apply node, e) a program graph to add 2 numbers where iW represents initial node

graph connected to input port# 2 is evaluated else the parent graph connected to input port# 3 is evaluated. Fig. 3.1c represents an addition node that returns the sum of two input arguments. Fig. 3.1d represents a higher-order function called as apply. It receives a program graph as a function in its first argument and applies the same to the 2nd argument. Fig. 3.1e represents a sample program graph to return a sum of two constant numbers.

3.1 EVALUATION STRATEGY OF PROGRAM GRAPHS

Every single valid executable program in FGPM should end with a single terminal node and

start with an initial node. However, a set of programs can be represented by a single graph



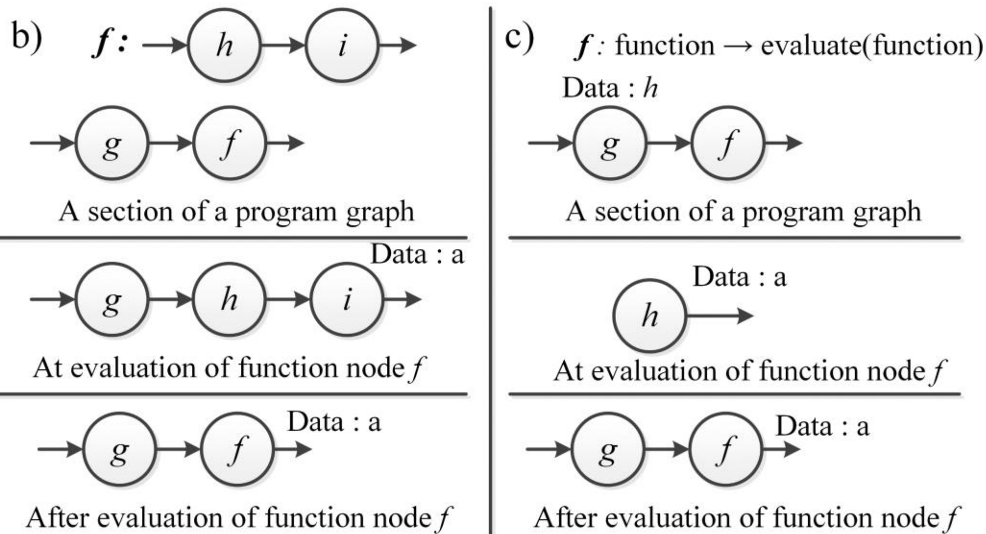


Figure 3.2 Illustration of execution strategy of programs in FGPM. a) Evaluation of a sample program graph. b) Evaluation of function node *f* where *f* is a composite function. c) Evaluation of function node *f* where *f* is a higher order function which takes another function as input and evaluates it.

with one initial node and multiple terminal nodes, where graphs between the initial node and each terminal node denote a separate program. Programs are evaluated in lazy style. The execution starts from the terminal node of a program and all the parent nodes are called recursively until the initial node is executed. Memoization is applied on each node and output value is cached during the course of the execution. The output value is copied on respective edges on each subsequent call. Fig. 3.2 demonstrates the execution techniques of FGPM programs. Fig. 3.2a illustrates execution of a sample program graph. The execution starts by calling the function *h* at the terminal node which in turn calls the function *f* first, as this is the parent node connected to input port 1 of node *h*. *f* is the initial node so it gets executed

right away and output data *a* is stored in the node. A copy of data *a* is transmitted over the outgoing edge for consumption by *h*. Next, function *g* is called by node *h* which is connected to input port 2. *g*, in turn, calls *f*. But *f* has already been executed thus it just copies the data *a* over the edge for consumption by *g*. Thereafter, g completes its execution and copies its output data *b* over the outgoing edge for consumption by *h*. Finally, *h* completes its execution and returns data 'c'. Fig. 3.2b illustrates execution of a composite function node *f*. Definition of *f* is a program subgraph constructed by composing functions *i* after *h*. At function call, *f* is copied and substituted by its definition. In the reduction phase, the program subgraph gets executed and the output data of *i* get copied as the output data of the original node *f* followed by deletion of executed program subgraph *i.h*. Fig. 3.2c illustrates execution of a higher-order function where the function takes another function as input. The definition of the higher-order function *f* states that it simply evaluates the function received as an argument. At function call, *f* receives the function *h* as data from node *g* followed by copy and substitution of itself by the function *h*. At reduction the function *h* is simply executed and output data is copied to the output port of original node *f*. Thereafter executed node *h* is deleted.

3.2 INTEGRATING MONADIC FUNCTIONS

The computation environment represents the external world with respect to the program and the program communicates with it through I/O nodes, named as *sensors* and *actuators*. For example, standard input/output in a file can represent an environment and the program communicates by running sensor and actuator nodes which in turn reads or writes in the file. Thus, in order to communicate with the external environment and to make the programs useful in real-world, I/O monads are needed. Interaction with the external environment always causes side effects. I/O monads allow these side effects without hampering the purity of the language. The output data from the I/O nodes are embellished with the environment object referred to as "*world*", to create a monadic type and finally return the monadic type. It is presumed each I/O operation modifies the environment in some way and they return the data along with the modified environment thus preventing side effect. In fact, every other function node operates on the monadic type by scooping out the data from the embellished object, operating on it, and then recreating the embellishment with the environment object.

Due to such modifications, all composition operation is carried out using Kleisli composition [56]. The Kleisli composition is defined by equation 1 and 2 which represents the bind operation of two functions and the return operation respectively. *M* represents a monadic object which in our case is the environment. The bind operator is represented by the symbol >>=.

$$\gg=: (M, A) \to \big(A \to (M, B)\big) \to (M, B) \quad (3.1)$$

The return operation at the output port of each node is modified as follows.

$$return: A \to (M, A) \quad (3.2)$$

The bind operator takes a monadic type (*M, A*) as input and a function that takes a specific data type *A* as input and returns a monadic data type (*M, B*). It extracts the datatype *A* from monadic type (*M, A*) and applies the function on datatype *A* which eventually returns a monadic type (*M, B*). In FGPM the function application operator in a node will take a monadic type from source node output port through a connected edge and apply the node function on it. The output will be returned from the target node output port. The return operation is modified as monadic return for all functions in FGPM. In FGPM we have only one monadic contextual object i.e., "*world*". Thus, we will use "*world*" in place of *M* in all future references. Only the I/O monads are allowed to modify the world object. In order to prevent ambiguity on sequencing the I/O actions on external worlds, parallel modifications of world object in a single FGPM program needs to be prevented. After all, our observable environment is strictly sequential, and different sequence of the same set of actions can modify the environment differently. Combining lazy evaluation with monadic functions is problematic, especially when parallel execution pathways are possible. It somehow always leads to some side effect. However, if we cannot prevent side effect in this situation, we will keep the side effect outside of the FGPM program. We will use memoization from design perspective of the language and this is not a flexibility that will be available to the programmer. Every function in a program by default will be memoized and there will be no way to refer to the memo table from an FGPM program. This will not prevent side effect, but it will keep the side effect out of the FGPM program. Another side effect is allowed inside the world object to prevent creation of programs that might take I/O actions in non-

deterministic order if independent paths in FGPM are executed parallelly. This constraint is applied by versioning the world object. At evaluation, each I/O monad upgrades the version of the world object which is implemented in the world object by incrementing some global (accessible by every instance of world object) sequence number by 1 and storing the same version number locally in the current instance of the world object. On evaluation, every node stores the current instance of the world object in its data section (due to memoization). While upgrading, every world object checks if the local version number (before upgrading) is same as that of the global version number. If not, it will generate an error as it signifies the world is updated parallelly elsewhere in the graph. Fig. 3.3 demonstrates an invalid FPGM program construction due to parallel combination of actuator functions. Suppose node 1 is a pure function *f* which gets a world object with version number n as input. The execution order is left to right. Suppose node 2 gets executed after receiving data from node 1. The actuator function will upgrade the world object and world version will be $n + 1$. Thereafter when node 3 tries to complete its execution after receiving data from node 1 it will match the current global world version which is $n + 1$ (as upgraded by node 2) with the local world version (n). Due to the mismatch, an error will be generated on upgrading the world object. Thus, this constraint will ensure sequential ordering of I/O monads in all valid programs.

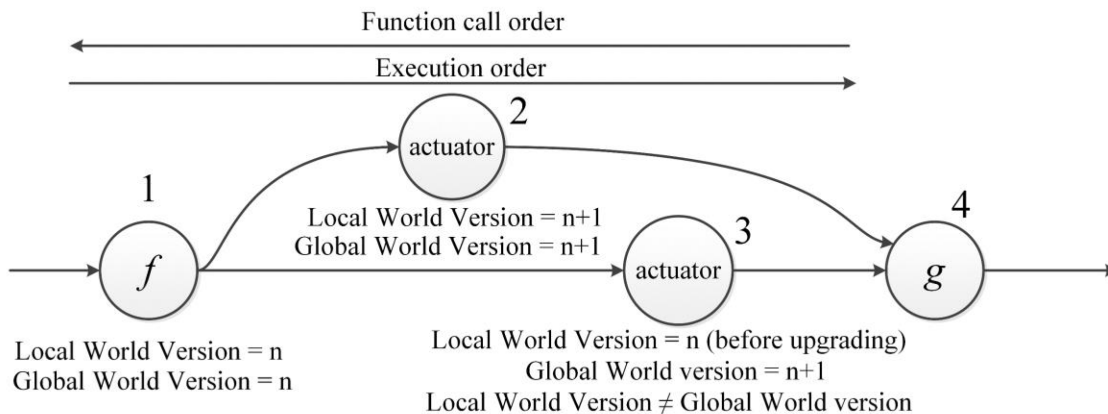


Figure 3.3 Illustration of an invalid FGPM program due to parallel combination of I/O monad.

For functions with multiple input ports in a valid program, if the world object is modified in one of the input paths, then it will cause different versions of the world object to be available at different input ports of the function. To resolve the conflict, for functions with multiple

input arguments, world object is selected from that input port only which delivers the world instance with highest version number. A special case is for the *guard* node where the world object is selected based on the Boolean value at the first input node.

Monadic function composition will be represented by the symbol $\circ$. Thus, monadic function composition of f after g will be written as $f \circ g$. From now onwards every function composition will refer to monadic function composition if not mentioned otherwise.

3.3 Agent environment structure

Every program graph in the programming model is executed in the context of an initialized environment. The program interacts with the environment through sensor and actuator nodes. In that sense, the program is treated as an agent which can take action on the environment and read perceptions from it. The environment may contain the problem context or any reusable functional components written in another language. For example, the environment may represent a physical 2D space where a bot has to maneuver itself to meet some goals. Each action signal sent through actuators can be interpreted as some movement actions in the environment. Whereas perception signals received using sensors may return current observations from the environment. The actuator signals may be simple integer values provided from the program graph. Some libraries in the environment can be used to transform it into physical actuation signals for the bot, thus allowing incorporation of external libraries into the systems without hampering the purity of the programming model (as the library always remains external with respect to the core program).

3.4 Comparison with current state of the art

Table 3.1 shows a comparison chart of the proposed programming model against two dataflow graph-based languages, namely Cuneiform [17] and Apache Beam [16]. Cuneiform is used to construct scientific workflows for large-scale data analysis and Apache beam is used to construct pipelines for streaming data processing. Each has its own set of features to serve the purpose for which they were built. However, we choose a set of comparison grounds in the context of Universal search, automatic program creation/manipulation, and applicability in solving arbitrary problem environments. On these comparison grounds, it

Table 3.1 Comparison of proposed Functional Graph Programming Model (FGPM) with Cuneiform and Apache Beam

Feature	Programming Model	Comparison
Metaprogramming - Dynamic function creation and evaluation	FGPM	Readily available using higher order primitives for program graph creation, modification and evaluation.
	Cuneiform	Not natively available.
	Beam	No core component is available to construct beam pipeline within runtime of another pipeline.
Expressiveness - Can implement any arbitrary programming logic?	FGPM	Yes, as all required primitives for arithmetic/logical operations, conditional branching, recursion, datatype creation/manipulation, and I/O operations are available.
	Cuneiform	Yes, all required primitives are available. However, I/O operations need to be done by incorporating functions written in other language.
	Beam	Using core beam transforms any arbitrary logic may not be implemented. However, it allows construction of functions in other languages which allows to inherit the expressiveness of those languages.
Program semantic reasoning	FGPM	Program reasoning can be done using simplified algebraic expressions constructed inherently for each program.
	Cuneiform	Not inherently available.
	Beam	Not inherently available.
Allows higher order function	FGPM	Yes.
	Cuneiform	Yes.
	Beam	Not using core components.
Metadata storage and manipulation on edges	FGPM	Readily available (for maintaining probability distribution).
	Cuneiform	Not readily available.
	Beam	Not readily available.
Modularity in program construction	FGPM	Program is constructed by directly adding and connecting function nodes in a graph thus making each construction step functionally modular.
	Cuneiform	Program construction is done by sequencing tokens in a lexical structure. Which reduces functional modularity as individual tokens are not functionally complete.
	Beam	Program is constructed by directly adding I/O and transformation nodes in a pipeline thus making it functionally modular.
Can build arbitrary reusable composite components?	FGPM	Yes, using a primitive called "gp" which makes any arbitrary program graph reusable by converting it into a composite node.
	Cuneiform	Yes, readily available.
	Beam	Not using core components.
Reuse of arbitrary program subpart	FGPM	Any subgraph can be functionally reused by just connecting appropriate nodes in another program graph.
	Cuneiform	Not trivial, as body of a function can be tightly coupled with the rest of the program.
	Beam	Reuse of sections of pipeline is possible.

seems both of them are not equipped with all the required features whereas the proposed

model is.

3.5 SYNTAX OF FGPM

Apart from using a graphical user interface to construct programs in FGPM, they can be constructed in textual format also. A program is initialized by invoking a command to create an empty graph object. Thereafter subsequent nodes are added to the graph object. The graph object is identified with a unique graph label which is basically a non-negative integer. The node object is created with a specific function from the list of available functions and a set of input links. A node object is identified by a unique node label. The set of input links contains node labels of parent nodes. The syntax of writing a program in FGPM is given in Backus-Naur form. The non-terminal symbols <whole numbers>, <negative integers>, <floating-point numbers>, <Boolean> and <string> are not expanded further as they hold usual meanings. The symbol <EOL> denotes the end of the line.

$< programs > ::= initGraph(< graphLabel >) < EOL > < command\ list >$

$< command\ list > ::= < null > | addNode(< graphLabel >, < Node >, <$
$inputLink > < inputLinks >)$<EOL> $< command\ list >$

$< Node > ::= Node(< nodeLabel >, < Function >)$

$< Function > ::= ' identity' | 'constant', < attr > | 'add' | 'subtract' | 'multiply'$
$| 'divide' | 'guard' | 'equal' | 'greater' | 'conjunct' | 'disjunct' | 'negate' | 'apply'$
$| 'lambdagraph' | 'recurse' | 'sensor', < type > | 'actuator', < type >$
$| 'goalchecker' | 'initWorld' | 'fmap' | 'cons' | 'nil' | 'head' | 'tail' | 'aggregator'$
$| 'zip' | 'loop' | 'join' | 'split' | 'zip' | 'worldmerge'$

$< inputLinks > ::= < null > |, < inputLink >< inputLinks >$

$< inputLink > ::= < null > | < nodeLabel >$

$< attr > ::= < number > | < string > | < Node > | < boolean >$

$< number > ::= < whole\ numbers > | < negetive\ integers > | < floating -$

$point\ numbers >$

$< graphLabel > ::=< whole\ numbers >$

$< nodeLabel > ::=< whole\ numbers >$

$< type > ::= 'number' | 'string' | 'Boolean' | 'function'$

$< null > ::=$

$< program\ graph > ::= \ getSubgraph(< graphLabel >, < nodeLabel >)$

$< evaluate\ a\ program\ graph > ::= \ evalGraph(< program\ graph >)$

3.6 DATATYPES OF FGPM

There are seven primitive types allowed in the model. Namely number, string, Boolean, list, node, graph, and null. The number type includes three subtypes namely, whole numbers, negative integers, and floating-point numbers. The string type includes all alphanumeric characters. The Boolean type includes True and False. The node and graph datatypes are special datatypes that are used to store only the nodes and program graphs in FGPM. The graph type is also synonymously called as function type as it basically represents a function in FGPM. The null type is the initial datatype. Any function input can connect with the null type. The null type contains nothing and can be compared with the empty set. We are not discussing the implementation details of these primitive datatypes as these can be implemented in any other high-level programming language which would be used to create the interpreter of FGPM. However, we will be providing the constructors for these datatypes in later sections.

3.7 FUNCTIONS OF FGPM

A function in FGPM is a reusable description of a single or related operations. There are several primitive functions available in FGPM namely first-order node functions, higher-order node functions, and I/O monads which can be grouped and composed for program construction. The semantics of the language is described in algebraic form. While writing

the semantics we will adopt the following notations. $evaluate(f)$ will represent the execution of function f, $*A$ denotes multiple optional arguments of type A. $\{A\}$ denotes single optional argument of type A, $(A, world)$ denotes type A embellished in world object, $a[world]$ denotes the world object extracted from a, $a[data]$ denotes data extracted from a, $plus(a, b)$ adds two numeric objects a and b, $minus(a, b)$ subtracts numeric object b from a, $product(a, b)$ multiplies two numeric objects a and b, $division(a, b)$ divides numeric object a by b, $equal(a, b)$ checks for equality between two objects a and b, $greater(a, b)$ checks if a is greater than b, $AND(a, b)$ checks if both the boolean objects a and b are true, $OR(a, b)$ checks if either a or b is true, $NOT(a)$ inverts the boolean object a, $read(W)$ puts a read request in the world object W and returns the input data received from the world embellished with the modified world, $write(W, a)$ puts an action request a in the world object W and returns the modified world, $check_goal(W)$ checks if the goal is reached in the world object W and returns True or False embellished in the modified world, symbols A, B and C denotes datatypes and can be any datatype unless otherwise mentioned.

3.7.1 Functions for program construction

These functions are used to construct and execute program graphs in FGPM by writing the statements in sequential textual format. These functions are not allowed to be placed within a node, and thus are not part of program graphs constructed in FGPM.

initGraph : whole number → graph. initGraph is the constructor of graph data type in FGPM. This is used to initialize a program graph. The function takes a whole number as input which is used to label the newly created program graph. The graph label is used to uniquely identify a program graph. Construction of any program or set of programs should start with initializing a graph object using the initGraph function. The semantics of the function is as follows.

$$evaluate\big(initGraph(a)\big) = \begin{cases} graph_a | a \notin A \\ undefined |\ otherwise \end{cases}, where \quad \begin{matrix} graph_a = empty\ graph\ with\ label\ a \\ A = set\ of\ all\ graph\ labels \end{matrix}$$

addNode : (whole number, node, *whole number) → graph. addNode function adds a

newly created node object in an existing program graph. It takes multiple arguments. The first argument refers to the graph label of an existing graph to which the node needs to be added. The second argument is a node object which needs to be added. Rest of the arguments are optional (the * sign in the function signature denotes optional and variable number of arguments of a specific type) and can be variable. These arguments refer to the node labels of the parent nodes of the newly added node. The input links of the newly created node will be connected with these parent nodes in the order in which they are supplied as arguments. If excess arguments are given compared to the number of input arguments required by the newly created node function, then those arguments will be ignored. All the node labels supplied in the argument should refer to only the nodes present in the graph identified by the graph label. The semantic equation is given as follows.

$$evaluate\big(addNode(a, node, *\, b)\big) =$$

$$\begin{cases} graph'_a \mid a \in G \ \wedge b \in N_a \\ undefined \mid otherwise \end{cases}, where \quad \begin{matrix} G = set\ of\ graph\ labels\ of\ all\ existing\ graphs \\ N_a = set\ of\ node\ labels\ of\ all\ nodes\ in\ graph_a \\ graph_a = graph\ identified\ by\ label\ a \\ graph'_a = modified\ graph\ after\ adding\ node\ in\ graph_a \end{matrix}$$

Node : (whole number, string, {A})→node. The Node function creates a node type that can be added to a program graph. It takes two mandatory arguments and one optional argument (the optional argument is enclosed in a second bracket in the function signature). The first argument is a whole number that denotes the node label. The second argument is the function name of the function which needs to be placed in the node. The third argument is optional and can be of any type. For constant function, the third argument is the constant value that needs to be assigned. For sensor and actuator function the third argument is the type name of the input and output type of the respective functions. For all other functions, there should be no third argument. The semantic relation is given as follows.

$$evaluate\big(Node(a, f, \{b\})\big) =$$

$$\begin{cases} node_a^f | a \notin A \wedge f = 'constant' \ \wedge b \in AnyType \\ node_a^f | a \notin A \wedge f \in \{'sensor','actuator'\} \wedge b \in T \\ node_a^f | a \notin A \wedge f \in F \wedge b \in \emptyset \\ undefined \mid otherwise \end{cases}, where \begin{matrix} node_a^f\text{=node with label a and function } f \\ T\text{=set of all available type names in string format} \\ F\text{= set of all available node function names except constant, sensor and actuator} \\ A\text{=set of all node labels} \end{matrix}$$

getSubgraph : (whole number, whole number) → graph. The function getSubgraph returns the subgraph of a graph by starting from a specific terminal node and recursively fetching all parent nodes until initial node is reached. It takes two arguments of type whole number. The first argument denotes the graph label and the second denotes the node label of a specific node in the graph. This node is taken as the terminal node based on which the subgraph is extracted. The semantic relation is given as following.

$$evaluate\big(getSubgraph(a,b)\big) = \begin{cases} graph_{ab} | a \in A \wedge b \in aB \\ undefined | \ otherwise \end{cases},$$

$$where \begin{matrix} graph_{ab}\text{=subgraph of } graph_a \text{ and terminal node label } b \\ aB\text{=set of all node labels of } graph_a \\ A\text{=set of all graph labels} \end{matrix}$$

evalGraph : graph → A. The evalGraph function is responsible for executing a terminal node in a program graph. Due to lazy evaluation scheme once a specific node is executed it calls its parents on data requirement which results in execution of its parent nodes. This continues until the initial node is reached or data is already available in a specific node (due to memoization). This function takes a graph object as argument and evaluates the terminal node of the graph to store the result and return a copy of it. If result of the execution is already available in the node, then it directly returns a copy of the result. Clearly graph object should contain only one terminal node to be executable.

$$evaluate\left(evalGraph(graph^f)\right) = evaluate(graph^f) =$$

$$\begin{cases} evaluate(f) | \ |graph_{terminalnodes}| = 1 \wedge graph_{initialnodes}^{fname} = "initWorld" \\ undefined | otherwise \end{cases},$$

$$where \begin{array}{c} graph^{f} = graph\ with\ composite\ function\ f \\ graph^{fname}_{initialnodes} = function\ name\ of\ all\ inital\ nodes\ of\ graph \\ |graph_{terminalnodes}| = number\ of\ terminal\ nodes\ in\ graph \end{array}$$

gp : graph → node. The gp function converts a subgraph to a reusable node. The node can later be reused in multiple other graphs. This is similar to the function constructor in any other language. The number of input arguments in the node is determined by number of open input nodes in the subgraph used to create the node and the order is determined by the order in which the data will be fetched from the input nodes during execution. There should always be one terminal node in the subgraph and that leads to one output port in the resultant node. The subgraph should not contain any initWorld node (stated in section 3.7.2).

$$evaluate\big(gp(graph)\big) = \begin{cases} node \mid |graph_{terminalnodes}| = 1 \wedge initWorld \notin graph \\ undefined \mid otherwise \end{cases}$$

3.7.2 I/O monads

I/O monads are functions in FGPM which are used to handle side-effects caused by interaction with the external environment, thereby preserving the purity of the language. We will use a first bracket notation to represent a monadic type. (*A, world*) represents a monadic type with data type *A* embellished with the external environment or the world object.

initWorld : null → (null, world). The initWorld (*iW*) function initializes an environment which is denoted as world object. This function takes no input argument and returns an initial world object. Every valid program graph should start with this node and there should be only one node of this type in a program. The semantic equation is given as follows.

$$evaluate(initWorld) = (null, world)$$

sensor$_A$: (world) → (A, world). The sensor$_A$ function reads some data from the external environment represented as the world object. It takes a monadic type with the world object and returns a specific type embellished with the modified world. For each specific output type, there will be a different sensor function. The sensor$_A$ function sends a read request to the world object for some data of type *A*. The returned data is embellished with the modified world object and given as output by the sensor node. A sensor node can be a child node of initWorld node or an actuator node only.

$$evaluate(sensor_A \circ f) = read(evaluate(f))$$

***actuator$_A$* : *(A, world)* → *(world)*.** The actuator$_A$ function applies some action on the world object based on the input data. The actuator$_A$ function takes a monadic type with a specific type *A* embellished with the world object. Multiple actuator functions are defined for different source datatypes. Based on the data input in the actuator function, it sends a write request to the world object to apply some action on the world object. Thereafter the modified world object is returned in the monadic type. Only a sensor or a goalchecker node can follow an actuator node.

$$evaluate(actuator_A \circ f) = write(evaluate(f))$$

***goalchecker* : *(world)* → *(boolean, world)*.** The goalchecker function checks if the goal is reached or not in the world object. The goalchecker function takes a monadic type with the world object and returns a Boolean type embellished with the modified world. The Boolean type will contain True if the goal is reached else False.

$$evaluate(Goalchecker \circ f) = check_goal(evaluate(f))$$

3.7.3 First-order node functions

These are pure functions. They might take or return any datatype available in FGPM. They are always represented by a single node in the program graph and perform atomic operations within a program execution phase. For simplicity, the type signatures of the following functions are not written in monadic form, but they can be easily extended in such form. In FGPM all functions will deal with monadic types only.

***identity* : *A* →*A*.** Identity is a polymorphic primitive function that can take any type as input and returns the same type as output. Initially, it looks pretty uninteresting function, however, in FGPM they play a crucial role in abstracting a function composition in a scenario where the inputs of multiple functions need to be joined (like recurse function). The following equivalence rule holds for this function.

$$identity \circ f = f \circ identity = f$$

***constant$_a$* : *A* → *a*.** The constant function takes any type as input but returns a constant value as output. The constant value is assigned while creating the function definition. The constant value can belong to any type. The following equivalence rule holds true.

$$evaluate(constant_a \circ f) = (a, evaluate(f)[world])$$

***add* : *number X number* → *number*.** Add is a primitive function that takes two numbers as input and produces a number type as output. It performs the operation of addition on inputs and provides the result as output. Equivalence rules are stated as below where $best_version$ operation denotes selection of the world object with the highest local version number. The following 2nd, 3rd, and 4th equations are due to the property of commutativity, property of identity and property of associativity respectively.

$$evaluate(add \circ (f \text{ X } g)) = (plus(evaluate(f)[data], evaluate(g)[data]),$$
$$best_version(evaluate(g)[world], evaluate(f)[world]))$$

$$add \circ (f \text{ X } g) = add \circ (g \text{ X } f)$$

$$add \circ (f \text{ X } constant_0) = f$$

$$add \circ (add \circ (f \text{ X } g) \text{X } h) = add \circ \left(f \text{ X } add \circ (g \text{ X } h)\right)$$

***subtract* : *number X number* → *number*.** Subtract function takes two numbers and returns a number type as output after performing the subtraction operation on two input numbers. It subtracts the second number in the product type from the first number. The following equivalence relations hold good. The following 2nd, 3rd, 4th, 5th, and 6th equations are due to the property of anticommutativity, identity, opposites, double negation, and subtraction respectively.

$$evaluate(subtract \circ (f \text{ X } g)) = (minus(evaluate(f)[data], evaluate(g)[data]),$$
$$best_version(evaluate(g)[world] , evaluate(f)[world]))$$

$$subtract \circ (f \text{ X } g) = subtract \circ (constant_0 \text{ X } subtract \circ (g \text{ X } f))$$

$$subtract \circ (f \text{ X } constant_0) = f$$

$subtract \circ (f \text{ X } f) = constant_0 \circ f$

$subtract \circ (constant_0 \text{ X } subtract \circ (constant_0 \text{ X } f)) = f$

$subtract \circ (f \text{ X } g) = \; add \circ (f \text{ X } subtract \circ (constant_0 \text{ X } g))$

multiply : number X number → number. Multiply function takes two numbers and returns a number type as output after performing the multiplication operation on two input numbers. The following equivalence relations hold. The following equations starting from 2nd are due to the property of commutativity, identity, multiplication property of zero, associativity, distributivity, distributivity, and definition of multiplication respectively.

$$evaluate(multiply \circ (f \text{ X } g)) = (product(evaluate(f)[data], evaluate(g)[data]) \\ , best_version(evaluate(g)[world], evaluate(f)[world]))$$

$multiply \circ (f \text{ X } g) = multiply \circ (g \text{ X } f)$

$multiply \circ (f \text{ X } constant_1) = f$

$multiply \circ (f \text{ X } constant_0) = constant_0 \circ f$

$multiply \circ (multiply \circ (f \text{ X } g) \text{X } h) = multiply \circ \big(f \text{ X } multiply \circ (g \text{ X } h)\big)$

$multiply \circ \big(h \text{ X } add \circ (f \text{ X } g)\big) = add \circ \big(multiply \circ (f \text{ X } h) \text{ X } multiply \circ (g \text{ X } h)\big)$

$$multiply \circ \big(h \text{ X } subtract \circ (f \text{ X } g)\big) \\ = subtract \circ \big(multiply \circ (f \text{ X } h) \text{ X } multiply \circ (g \text{ X } h)\big)$$

$multiply \circ (constant_2 \; X \; f) = add \circ (f \; X \; f)$

divide : number X number → number. The divide function takes two numbers and returns a number type as output after performing the division operation on two input numbers. It divides the second number from the first number. The following equivalence relations hold. The following equations starting from 2nd are due to property of identity, division property of zero, anticommutativity, opposites, double division, definition of division, right

distributivity, and right distributivity respectively.

$$evaluate(divide \circ (f \text{ X } g)) = (division(evaluate(f)[data], evaluate(g)[data]),$$
$$best_version(evaluate(g)[world], evaluate(f)[world]))$$

$$divide \circ (f \text{ X } constant_1) = f$$

$$divide \circ (f \text{ X } constant_0) = Undefined$$

$$divide \circ (f \text{ X } g) = divide \circ (constant_1 \text{ X } divide \circ (g \text{ X } f))$$

$$divide \circ (f \text{ X } f) = constant_1 \circ f$$

$$divide \circ (constant_1 \text{ X } divide \circ (constant_1 \text{ X } f)) = f$$

$$divide \circ (f \text{ X } g) = multiply \circ (f \text{ X } divide \circ (constant_1 \text{ X } g))$$

$$divide \circ (add \circ (f \text{ X } g) \text{ X } h) = add \circ (divide \circ (f \text{ X } h) \text{ X } divide \circ (g \text{ X } h))$$

$$divide \circ (subtract \circ (f \text{ X } g) \text{ X } h) = subtract \circ (divide \circ (f \text{ X } h) \text{ X } divide \circ (g \text{ X } h))$$

guard : Boolean X A X A → A. The guard function acts as a conditional function in FGPM. It takes three elements as input. The first element should be a Boolean and the other two can be of any type, but they should be of the same type. The output type is same as that of the second or third element of the input. Based on the Boolean value of the first element in the input it either passes the second element or the third element. If the Boolean value is True, it passes the second element else the third element. In the lazy evaluation scheme, the functions responsible for generating the second and third element of the input of the guard function is selectively called based on the Boolean value returned at the first element of the input. The following equivalence rules hold. The following 2nd equation is due to the property of fusion.

$$evaluate\big((gaurd \circ (f \text{ X } g \text{ X } h)\big) = evaluate((g + h) \circ f)$$
$$= \begin{cases} (g'[data], best_version(f'[world], g'[world]) \mid f'[data] = True \\ (h'[data], best_version(f'[world], h'[world]) \mid f'[data] = False \end{cases}$$
$$where, g' = evaluate(g), f' = evaluate(f), h' = evaluate(h)$$

$i \circ (g + h) \circ f = (i \circ g + i \circ h) \circ f$

equal : AnyType X AnyType → Boolean. As the name suggests the equal function checks equality between two inputs. It takes two elements as input and returns a Boolean type as output. The two elements can be of any type. Based on the equality test between the two input elements it returns either True or False. The following equivalence rules hold. The following equations starting from 2nd are due to property of symmetry, reflexive property, and property of cancellation.

$$evaluate((equal \circ (f \text{ X } g)) = (equal(evaluate(f)[data], evaluate(g)[data]),$$
$$best_version(evaluate(f)[world], evaluate(g)[world]))$$

$$equal \circ (f \text{ X } g) = equal \circ (g \text{ X } f)$$

$$equal \circ (f \text{ X } f) = constant_{\text{True}} \circ f$$

$$equal \circ (g \circ f \text{ X } g \circ h) = equal \circ (f \text{ X } h) \mid lambdagraph, I/O\ monads \notin g$$

greater : number X number → Boolean. The greater function takes a product type of numbers as input and returns True if the first number is greater than the second else returns False. The same function behaves like a lesser function if the arguments are reversed. The following equivalence relations hold. The following equations starting from 2nd are due to property of anticommutativity, reflexive property, and property of cancellation.

$$evaluate((greater \circ (f \text{ X } g)) = (greater(evaluate(f)[data], evaluate(g)[data]),$$
$$best_version(evaluate(f)[world], evaluate(g)[world]))$$

$$greater \circ (f \text{ X } g) = negate \circ greater \circ (g \text{ X } f)$$

$$greater \circ (f \text{ X } f) = constant_{\text{False}} \circ f$$

$$greater \circ (g \circ f \text{ X } g \circ h) = greater \circ (f \text{ X } h) \mid lambdagraph, I/O\ monads \notin g$$

conjunct : Boolean X Boolean → Boolean. The conjunct function acts like logical AND. It takes two Boolean as input and returns a Boolean type. If both the input arguments are True

then output is True else it returns False. The following equivalence rules hold. The following equations starting from 2nd are due to property of commutativity, associativity, identity, annihilation, idempotence, property of distribution of conjunct over disjunct, and property of absorption respectively.

$$evaluate((conjunct \circ (f \text{ X } g)) = (AND(evaluate(f)[data], evaluate(g)[data]),$$
$$best_version(evaluate(f)[world], evaluate(g)[world]))$$

$$conjunct \circ (f \text{ X } g) = conjunct \circ (g \text{ X } f)$$

$$conjunct \circ (f \text{ X } conjunct \circ (g \text{ X } h)) = conjunct \circ (conjunct \circ (f \text{ X } g) \text{ X } h)$$

$$conjunct \circ (f \text{ X } constant_{\text{True}}) = f$$

$$conjunct \circ (f \text{ X } constant_{\text{False}}) = constant_{\text{False}} \circ f$$

$$conjunct \circ (f \text{ X } f) = f$$

$$conjunct \circ \big(f \text{ X } disjunct \circ (g \text{ X } h)\big)$$
$$= disjunct \circ (conjunct \circ (f \text{ X } g) \text{ X } conjunct \circ (f \text{ X } h))$$

$$conjunct \circ (f \text{ X } disjunct \circ (f \text{ X } g)) = f \,|\, lambdagraph, I/O \ monads \notin g$$

disjunct : Boolean X Boolean → Boolean. The disjunct function acts like logical OR. It takes two Boolean as input and returns a Boolean type. If either of the input arguments is True, then output is True else it returns False. The following equivalence rules hold. The following equations starting from 2nd are due to property of commutativity, associativity, identity, annihilation, idempotence, property of distribution of disjunct over conjunct, and property of absorption respectively.

$$evaluate((disjunct \circ (f \text{ X } g)) = (OR(evaluate(f)[data], evaluate(g)[data]),$$
$$best_version(evaluate(f)[world], evaluate(g)[world]))$$

$$disjunct \circ (f \text{ X } g) = disjunct \circ (g \text{ X } f)$$

$$disjunct \circ \big(f \text{ X } disjunct \circ (g \text{ X } h)\big) = disjunct \circ (disjunct \circ (f \text{ X } g) \text{ X } h)$$

$$disjunct \circ (f \text{ X } constant_{\text{False}}) = f$$

$$disjunct \circ (f \text{ X } constant_{\text{True}}) = constant_{\text{True}} \circ f$$

$$disjunct \circ (f \text{ X } f) = f$$

$$disjunct \circ \big(f \text{ X } conjunct \circ (g \text{ X } h)\big)$$
$$= conjunct \circ (disjunct \circ (f \text{ X } g) \text{ X } disjunct \circ (f \text{ X } h))$$

$$disjunct \circ (f \text{ X } conjunct \circ (f \text{ X } g)) = f \,|\, lambdagraph, I/O\ monads \notin g$$

negate : Boolean →Boolean. Negate function acts like logical NOT. It takes a Boolean type and returns a Boolean type. If the input is True, then output is False else it returns True. The following equivalence rules hold. The following equations starting from 2nd are due to the property of double negation, property of complementation over conjunct, property of complementation over disjunct, DeMorgan law 1, and DeMorgan law 2 respectively.

$$evaluate(negate \circ f) = (NOT(evaluate(f)[data]), evaluate(f)[world])$$

$$negate \circ negate \circ f = f$$

$$conjunct \circ (f \text{ X } negate \circ f) = constant_{\text{False}} \circ f$$

$$disjunct \circ (f \text{ X } negate \circ f) = constant_{\text{True}} \circ f$$

$$conjunct \circ (negate \circ f \text{ X } negate \circ g) = negate \circ disjunct(f \text{ X } g)$$

$$disjunct \circ (negate \circ f \text{ X } negate \circ g) = negate \circ conjunct(f \text{ X } g)$$

nil : A →list. Nil function creates an empty list. It takes any datatype as input and returns a list datatype. The following equivalence relation holds true, where [] denotes an empty list.

$$evaluate(nil \circ f) = ([], evaluate(f)[world])$$

cons : A X list →list. The cons function appends an element of type A to a list. The element is appended at the beginning of the list. It takes two input arguments, one of any arbitrary

type A and another of list type. It returns a list type. The following equivalence relation holds true. $append(a, b)$ denotes a list append function which appends the element a in list b.

$$evaluate(cons \circ (fxg)) = (\text{ append}(evaluate(f)[data], (evaluate(g)[data]),$$
$$best_version(evaluate(f)[world], evaluate(g)[world]))$$

head : list → A. The head function pops out the first element from a list and returns the popped element. It takes a list as input argument type and returns an element of type A. The following equivalence relation holds good, where the function $pop(a)$ returns the first element from list a

$$evaluate(head \circ f) = (pop(evaluate(f)[data]), evaluate(f)[world])$$

tail : list → list. The tail function returns rest of the list except the first element. It takes a list as an input argument and returns a list. The following equivalence relation holds good where the function $tail(a)$ returns rest of the list a except the first element.

$$evaluate(tail \circ f) = (tail(evaluate(f)[data]), evaluate(f)[world])$$

$$cons \circ ((head \circ f)\ X\ (tail \circ f))\ = f$$

len : list → number. The len function returns the number of elements present in a list. It takes a list as an input argument and returns a number. The following equivalence relation holds good where the function $length(a)$ returns the length of list a.

$$evaluate(len \circ f) = (len(evaluate(f)[data]), evaluate(f)[world])$$

split : list X number → list. Split takes a list and a number as input arguments. It splits the input list at the index position specified by the input number. Finally, two lists are returned wrapped within another list. The following equivalence relation holds good where the function $split(a, b)$ returns list of split lists a in split position b.

$$evaluate(split \circ (f\ X\ g)) = (split(evaluate(f)[data], evaluate(g)[data]),$$
$$best_version(evaluate(f)[world], evaluate(g)[world]))$$

join : list X list → list. Join takes two lists as input arguments and joins both the lists. The joined list is returned as output. The following equivalence relation holds good where the function $join(a, b)$ returns a list by joining two lists a and b.

$$evaluate(join \circ (f\ X\ g)) = (\text{ join}(evaluate(f)[data], evaluate(g)[data]),$$
$$best_version(evaluate(f)[world], evaluate(g)[world])$$

$$join \circ ((head \circ split \circ (f\ X\ g))\ X\ (head \circ tail \circ split \circ (f\ X\ g)))\ = f$$

$$split \circ \Big((head \circ join \circ (f\ X\ g))\ X\ (len \circ f)\Big) = cons \circ (f\ X\ cons \circ (g\ X\ nil))$$

pop : list X number → A. Pop takes one list and one number as input arguments. It returns the element present in the input list in the index position specified by the input number. The following equivalence relation holds good where the function $pop(a, b)$ returns the bth element in list a.

$$evaluate(pop \circ (f\ X\ g)) = (\text{ pop}(evaluate(f)[data], evaluate(g)[data]),$$
$$best_version(evaluate(f)[world], evaluate(g)[world])$$

reverse : list → list. Reverse takes a list as an input argument and reverses the order of the elements within the list. The final reversed list is returned. The following equivalence relation holds good where the function $reverse(a)$ reverses the content of list a.

$$evaluate(pop \circ (f\ X\ g)) = (\text{ reverse}(evaluate(f)[data], evaluate(g)[data]),$$
$$best_version(evaluate(f)[world], evaluate(g)[world])$$

getdatatype : A → string. Getdatatype returns the datatype of the input argument in form of string. The following equivalence relation holds good where the function $getdatatype\ (a)$ returns any of the following strings based on the datatype of a. ‘number’, ’string’, ’Boolean’, ‘list’, ’node’, ’graph’, ‘null’.

$$evaluate(getdatatype\ \circ f) =$$
$$(getdatatype\ (evaluate(f)[data]), evaluate(f)[world])$$

worldmerge : A X B → A. The worldmerge function takes two arguments and returns the

first argument. This node is similar to identity except it takes two arguments. The purpose of this node is to merge parallel pathways in a graph, mainly in scenarios where the world gets modified by I/O monads in one pathway.

$$evaluate(worldmerge \circ (fxg))$$
$$= (\text{evaluate}(f)[data], best_version(evaluate(f)[world], evaluate(g)[world]))$$

3.7.4 Higher-order node functions

Higher-order functions will at least take or return a function type. Higher-order functions are represented either as a single node primitive function or as a program graph by composing other inbuilt functions (composite function). Higher-order composite functions placed in a node is copied and replaced with the respective program subgraphs during evaluation as per the evaluation strategy mentioned earlier.

apply : function X A X B → C. Apply function takes three input elements. The first element should be a function type and the next two elements can be of any type. The apply function applies the function received in its first input argument on the arguments received as second and third input elements. Clearly, the second and third elements should be type compatible with the input type of the function received as the first element. If the input function takes one argument, then the third argument is ignored, and apply returns the result of the function application on the argument received at input port 2. In case the input function takes two arguments then the function is evaluated on both of its arguments and apply returns result of the function evaluation. If the input function takes more than two arguments, then a partially evaluated function is returned by the apply function. The following equivalence relation holds good.

$$apply \circ (f \text{ X } g \text{ X } h)$$
$$= \begin{cases} f' \circ (g \text{ X } h), if\ |args(f')| >= 2 \\ f' \circ g, if\ |args(f')| = 1 \\ f', if\ |args(f')| = 0 \end{cases} \qquad \begin{matrix} where\ f' = evaluate(f) \\ |args(f')| = \ number\ of\ arguments\ of\ f' \end{matrix}$$

lambdagraph : A → function. Lambdagraph function takes an input of any type and returns a function definition. In FGPM lambdagraph function can be composed with any other

functions. It will return the function definition in the form of a graph after execution. It will return the complete subgraph ending with the parent node of the lambdagraph function. The initial node in a graph is replaced by an identity node in the returned program graph. Fig. 3.4. illustrates a lambdagraph function. The first graph is a sample program graph starting with marked initial nodes (initial nodes can be initWorld node or any other nodes marked as an initial node by a specific flag within the node) and ending with a lambdagraph function node. The second program subgraph is returned by the lambdagraph function after evaluation. The initial nodes are replaced by identity function node which is considered as the initial node of the returned program subgraph. All the child nodes of the initial identity node should have the same input type else error is raised on evaluating the lambdagraph function. Input type of the initial identity node is set as the input type of its child nodes. Lambdagraph is useful in creating arbitrary function definitions during program evaluation and using them as modules in other functions. The returned function has the same input type as that of the input type of the initial node and same output type as that of the output type of the terminal node. Consecutive composition of lambdagraph is not allowed in FGPM as they eventually do the same thing which a single lambdagraph does. The following equivalence relations hold true.

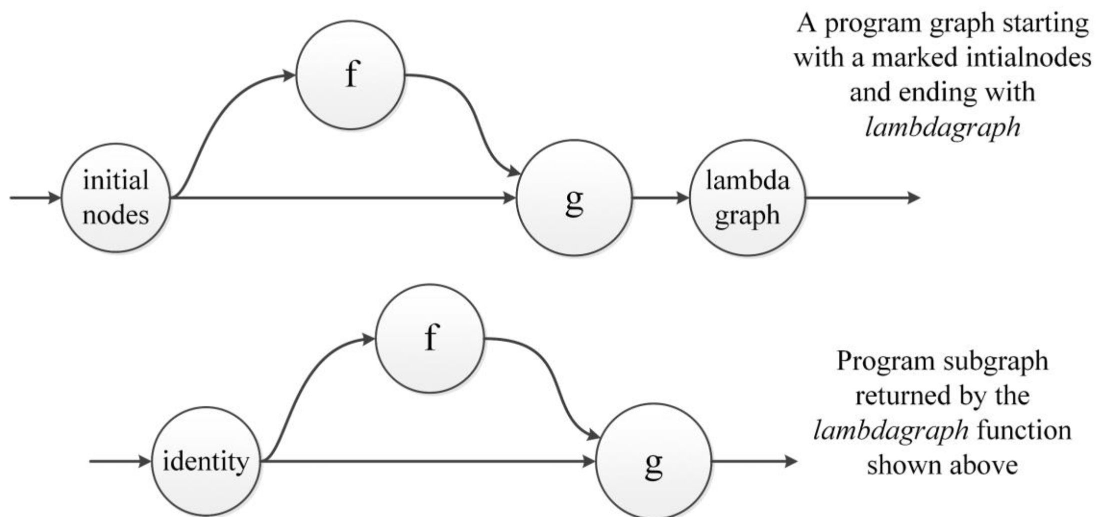


Figure 3.4 Illustration of a lambdagraph function in FGPM.

$$evaluate(lambdagraph \circ f \circ g)$$
$$= \begin{cases} (f, evaluate(g)[world]) \mid g \in \{initWorld, any\ node\ marked\ as\ initial\ node\} \\ undefined \mid otherwise \end{cases}$$

$$lambdagraph \circ lambdagraph \circ f = lambdagraph \circ f$$

recurse : function X function X A → B. The recurse function takes three inputs out of which the first two elements are function type and the third element can be of any type. The recurse function implements the looping logic using recursion. The first argument is a function that is applied to the third argument recursively until the stopping condition is met. The function for checking the stopping condition is received as the second argument. The recurse function helps generate short programs for repetitive operations. It is actually a program subgraph built with other primitive functions and itself stored as a primitive function. Fig. 3.5 illustrates the program graph of the recurse function. The sequence of input ports is in

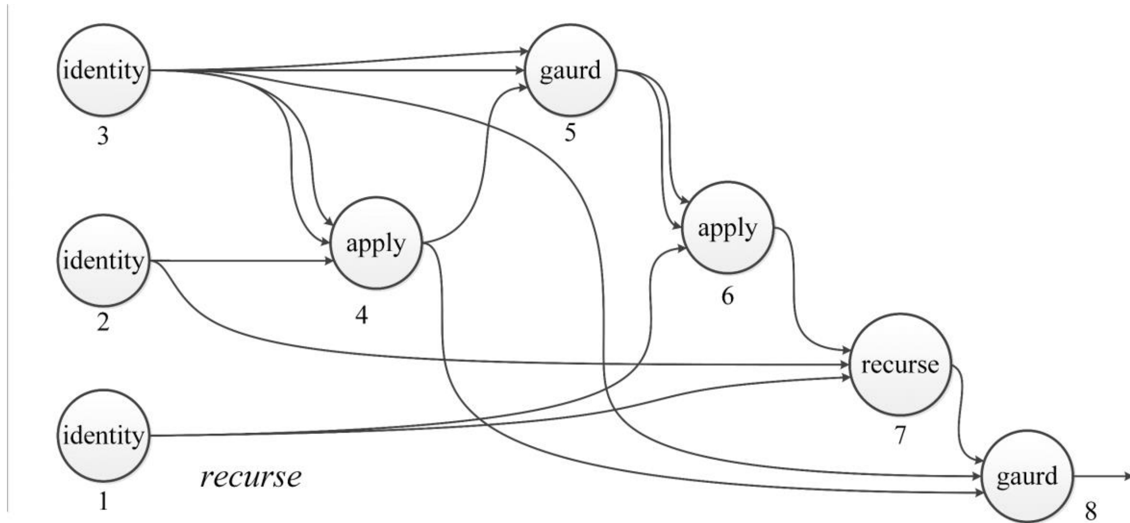


Figure 3.5 Illustration of a recurse function in FGPM

clockwise direction for all nodes. Nodes 1, 2 and 3 take the input arguments of recurse function. The function which needs to be recursed (fed to node 1) should at most take 1 argument. The same constraint applies to the function which checks the stopping condition (fed to node 2). Node 6 applies the function received at node 1 on the argument received at node 3. As the source function takes at most one argument, the third argument of node 6 is ignored. The same is true for node 4. Node 5 is used to pass the updated world object (if updated by node 4 due to some monadic function fed to node 2) to node 6. This allows to sequence monadic actions if present in functions fed at node 1 and 2. Node 7 copies and substitutes the definition of recurse function. If the stopping condition is True, then node 8 returns output from node 3 else it calls node 7. Clearly, the recursion depth will depend upon

the stopping condition. If the function for the stopping condition always returns True then the function at argument 1 will be applied on the argument available at input port 3, once. If the function for the stopping condition always returns False it will loop forever and the output will be undefined.

The following equivalence relations holds true where a program f is said to be $invalid$ if $evaluate(f) = undefined$

$$recurse \circ (f \text{ X } g \text{ X } h)$$
$$= \begin{cases} evaluate(f) \circ evaluate(h) \mid evaluate(g) = \ constant_{True} \circ i \\ invalid \mid evaluate(g) = \ constant_{False} \circ i \\ evaluate(f) \mid evaluate(f) = constant_K \bigwedge evaluate(g) \neq \ constant_{False} \circ i \\ gaurd \circ (k \text{ X } h \text{ X } recurse \circ (f \text{ X } g \text{ X } apply \circ (f \text{ X } l \text{ X } l))) \mid otherwise \end{cases}$$
$$, where\ k = apply \circ (g \text{ X } h \text{ X } h), l = gaurd(k \text{ X } h \text{ X } h)$$

loop : function X A X number → B. The loop takes 3 arguments, one of function type and another of any type, and the third one should be of number type. The input function is applied on the 2nd argument and number received as 3rd argument is decremented by 1. Thereafter the function is repeatedly applied on the output of the previous application of the function

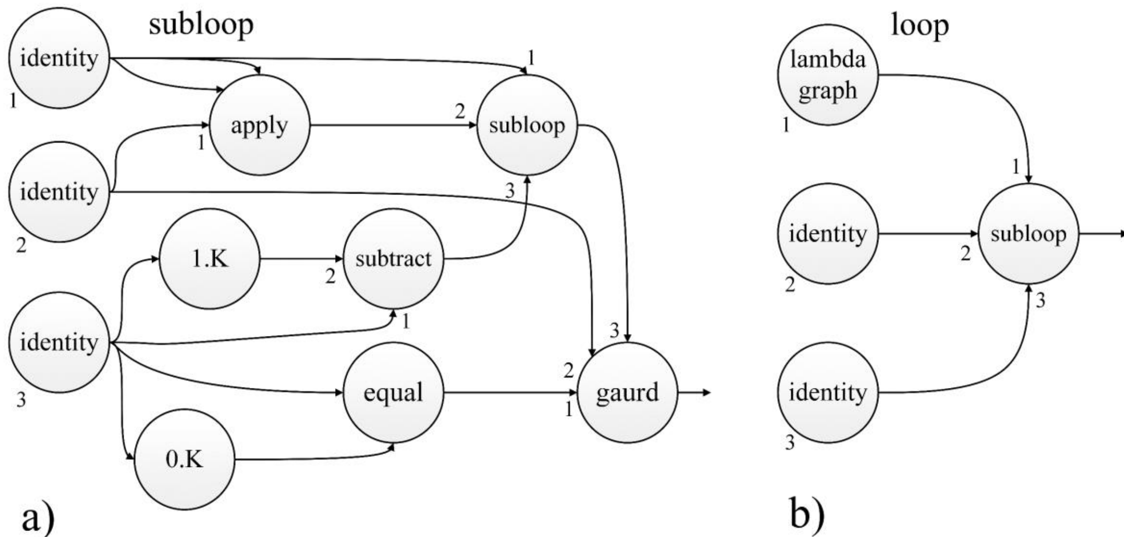


Figure 3.6 Illustration of a loop function in FGPM. a) subgraph for subloop function, b) subgraph for loop function

and the input number is decremented by 1 at each step. This continues until the value of the

input number reaches 0. The output of the final application of the input function is returned as the output of the node. Fig. 3.6 illustrates the subgraph which implements the loop node. The input port numbers are denoted by the numbers displayed beside the input ports of the nodes. It is implemented by two subgraphs. Fig 3.6a shows the subgraph for a subloop function that implements the loop logic using recursion. Fig 3.6b shows the graph for the loop node which reuses the subloop subgraph. A variant of loop node is also implemented which consists of 2 input ports. The 2nd and 3rd input ports are merged to one in this case.

fmap : function X list → list. Fmap takes a function and a list as input arguments. The function is applied on each element of the input list and output is stored in another list in the same index position. The final output list is returned. Fig 3.7 illustrates the subgraph for fmap.

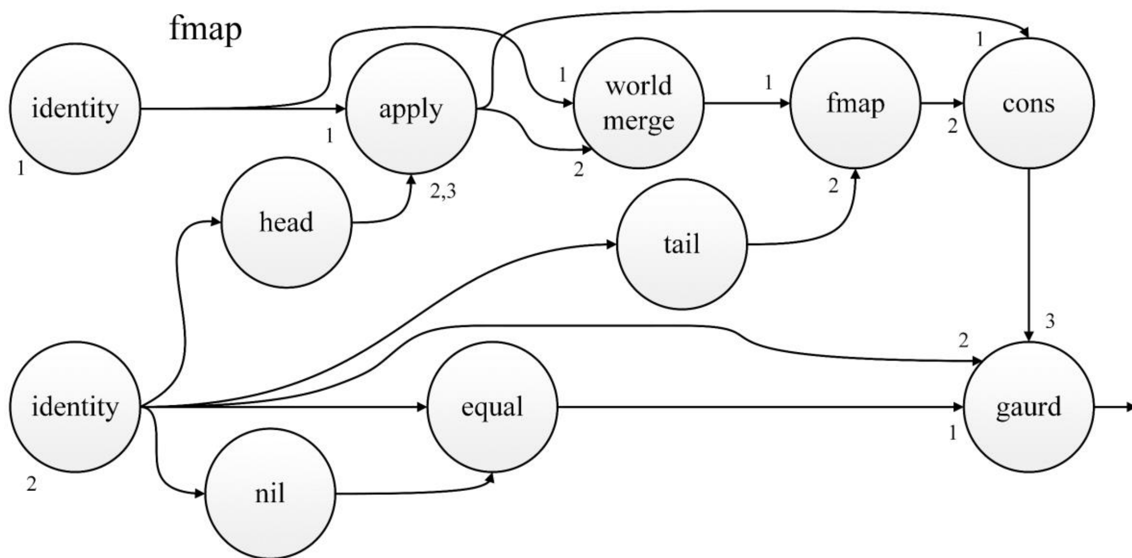


Figure 3.7 Illustration of a fmap function in FGPM

aggregator : function X list → A. Aggregator takes a function and list as input arguments. The input function is applied on each element iteratively and the output is aggregated. The final aggregated output is returned. Fig. 3.8 shows the subgraph for aggregator function.

zip : function X list X list → list. Zip takes two list type input arguments and one function type argument. It pairs each element from both the list and the input function is applied on each pair. It returns a list containing the output of the input function applied on each pair. Fig

3.9 illustrates the subgraph for zip function.

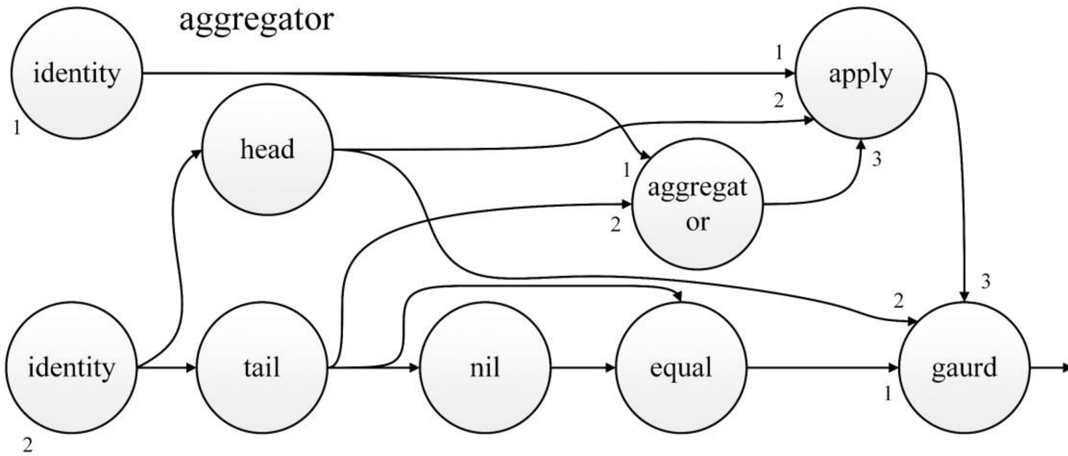


Figure 3.8 Illustration of aggregator function in FGPM

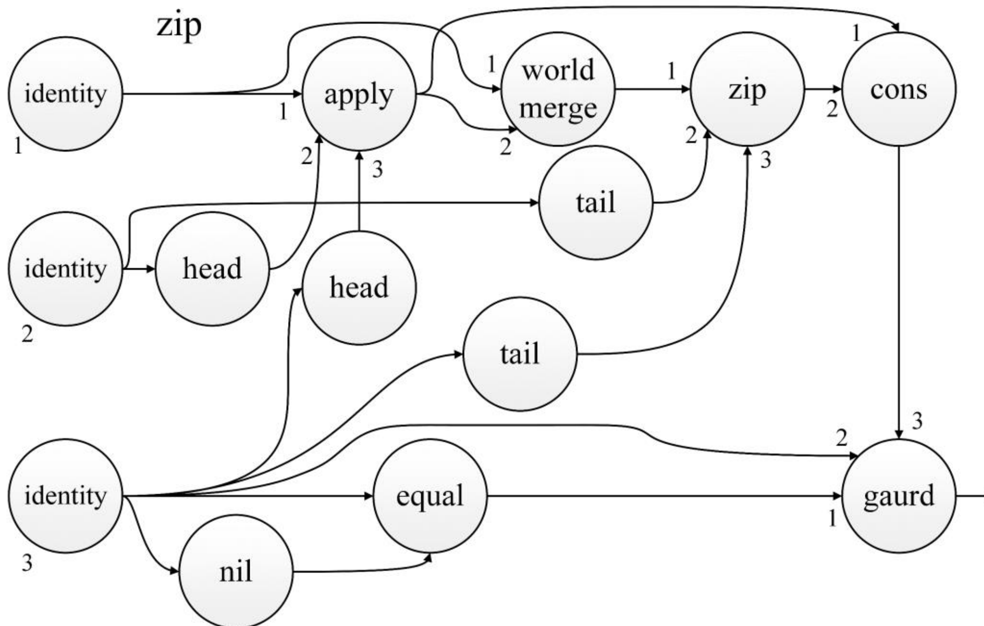


Figure 3.9 Illustration of a zip function in FGPM

3.8 CONCLUSION

In this chapter, we presented a dataflow graph based functional programming model and demonstrated its usability and applicability in the context of Universal search. Though the primary purpose of constructing this programming model is to allow universal search to

generate and test solution programs at ease, yet it can be used by human user to manually construct reusable AI applications by integrating multiple AI components. The later chapter describes how the programming model can play a vital role in the design of an integrative AI platform. This has a two-fold benefit. First, it allows easy construction of integrative AI applications through a graphical user interface. Second reusable components can be designed in the platform and eventually used in the problem-solver which would help to reduce the search space. The functional nature of the model allows every program to be written in terms of algebraic equations and consequently allows reasoning of programs.

4 THE INTEGRATIVE AI PLATFORM

"*Nature laughs at the difficulties of integration*" – Pierre-Simon Laplace, (1847)

Adopting AI to solve real-world problems requires integration of one or many domain specific AI methods due to the heterogeneous nature of the tasks. For example, a warehouse robot needs to navigate through multiple shelves in an inventory, recognize objects through vision, plan organization of objects and physically control its actuators to move objects. In some cases, it might need to understand speech commands from human. Several other examples of real-world heterogeneous problems are mentioned by Lombardi [57] which can be solved by integrative AI. Integrative AI combines disparate AI methods in a synergistic way for holistic problem-solving. Also, with the advent of Internet of Things (IoT) devices, development of distributed integrative intelligence looks very promising. Integration of IoT devices with AI methods can make Artificial Intelligence pervasive and ubiquitous.

Though integrative AI sounds very promising yet it comes with several challenges. Development of such systems are highly contextual to a problem scenario. Thus, it is mostly a manual process and quite an engineering challenge [50, 51, 58]. The developers usually work under limited resources and large-scale integration of multiple AI methodologies from scratch becomes a mammoth task [59]. The difficulty in integration mainly arises due to heterogeneity of components and methods which makes them incompatible for integration [52]. For example, a sensor camera may capture images and stream them in a certain format whereas an object recognition algorithm can process individual images in a different format. Integrating them in a live streaming situation requires some low-level development effort. Such heterogeneous integrations are usually done on adhoc basis for a problem context and reusability of solutions in a different problem context is low. Low scalability and performance brittleness are other issues that come hand in hand with such adhoc solutions. Along with these challenges, a distributed integrative intelligent system employing IoT devices comes with a new set of challenges [60]. Among these, the complexity of large-scale integration and coordination of devices with heterogeneous AI components are fundamental challenges.

In order to address the aforementioned challenges, there is a need to create a platform for integrating disparate AI methods and transducers. It should allow easy integration of disparate components using plug-n-play type integration which can largely reduce the engineering effort. Development of integrated solutions should be modular to improve reusability. Designing real-world transducer devices and different AI components as microservices [61] can overcome this problem as it provides the flexibility to treat them as black-box components. On a contrary, as argued by Thorisson [62] a real-world AI architecture may not always be divided into loosely coupled modules and they often need tight coupling. Thus, the platform should also allow enough expressivity to implement any arbitrary integration logic.

Current research efforts focus on solving some of these challenges of integrative AI. However, none addresses all of these issues. Current state-of-the-art solutions can be roughly divided into three groups. The first group focused on designing methods for building situated intelligence in a constrained environment and using native components [48, 49, 51]. Thus, ease of usage and scalability with respect to integrating external components becomes an issue. The second group focused on developing middleware for integrating IoT devices and disparate AI components [63, 64]. However, lack of expressive integration method prevents from implementing complex integration logic. The third group focused on developing integration methods based on microservices [65, 66]. Though they can provide a platform for integration of disparate components in a standardized way, yet difficulty of usage and lack of modularity in the integration method becomes a concern in this aspect. Thus, in order to address all the issues, an agent-environment based AI integration platform is proposed which considers all reusable AI components and IoT devices as microservices [67]. An intelligent system can be constructed in the platform by integrating different components using the proposed dataflow graph-based programming model as described in chapter 3. The programming model allows development of integration programs as white-box components and also provides a functional abstraction.

4.1 Agent-environment based integration platform

Thorisson suggested using small white-box components instead of black-box components

for integration [62]. However extensive use of white-box components makes an architecture messy, which might reduce modularity and consequently incremental modifications become difficult. Instead, contextual black-boxing of components can solve the above problems. Few solution modules can be atomic in all contexts. Others may be contextually atomic. Thus, the integration method should be opportunistic and support loose coupling of components wherever possible in a problem context and allow switching to white-box tight coupling as needed.

Fig. 4.1 illustrates an agent-environment model of AI integration. The environment consists of addresses and other access details of microservices that act as independent AI modules. It also includes middleware and a gateway to transform and redirect message requests or API calls to target microservices. The microservices can be hosted in any remote hosting platform or on the agent server itself. The agent serves as a host for the dataflow graph-based programming model where integration programs can be implemented. The integration programs are constructed in the proposed programming model. The functional dataflow graph-based programming model provides inherent modularity within a program. Treating each node as a complete function allows one to consider each subgraph within a program graph as a complete function. In this sense a program graph behaves as a white-box component, allowing reuse of any arbitrary subsection by just connecting edges with output ports of appropriate nodes. Programs are constructed by composing functions such that $f \circ g$ represents a program composed by connecting f after g. Every linearly composed program can be factored into any ordered combination of subset of all functions. Functions that are not composed linearly but combined parallelly can be represented by a product type. The constructed product type can be fed into multi-argument function nodes. Functions can be factored out by breaking the composition operator such that if $f' = f \circ g$, then f' can be factored into f and g. Thus, a program graph can be factored into $n + 1$ distinct functions where n is number of composition operators in the algebraic expression of the program graph. It signifies every node in a program graph represents the terminal node of a sub-function which can be reused in another program graph just by connecting the output of the corresponding node to some input port in the other program graph.

The integration program within the agent interacts with the environment through actions and

perceptions using I/O monads. Actions are API calls and perceptions read data from the environment. The environment temporarily stores the returned data by the microservices which are read by the agent through perception calls (using sensor node). An AI architecture is implemented by integrating relevant microservices with an integration program and reusable AI modules built with an integration program can be pushed into the environment as microservices. This is done by hosting the program as an API and registering it in the environment. Every integration program can be partially or completely reused in another program as a white-box function or a black-box component.

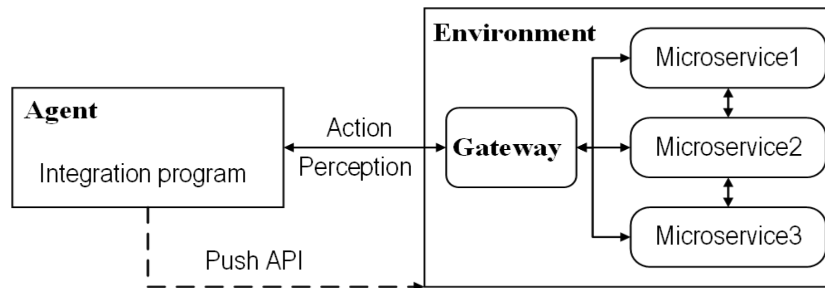


Figure 4.1 Agent-environment model of AI integration

4.1.1 Middleware

As shown in Fig 4.2 the middleware is responsible for communicating between the interpreter and the microservices. For communicating with the IoT devices, Google cloud IoT has been used. While communicating with AI services and data sources the middleware serves as an API gateway. Along with managing devices/APIs and routing data, it also applies some implicit data conversions to standardize the flow of data between the interpreter and different external components. The security layer and access control mechanisms are also implemented in the middleware. The access control mechanism ensures only authorized identities have access to respective services. The middleware is capable of both synchronous and asynchronous communications.

4.1.2 Microservices

The IoT devices, data sources, and AI services are treated as microservices within the platform. The IoT device connection is established through Google cloud IoT MQTT broker. The identity of the device is verified using an encrypted JSON web token (jwt). For each registered IoT device, a composite node is created (similar to *microcall* node as stated in

section 4.2) in the programming model which represents the API for communication with the device. The node can be used in a program graph to communicate with the device/service and implement any arbitrary programming logic. The data sources and AI services represent endpoints or APIs through which data or the AI service can be used. Similar to the IoT devices, for each data source and AI service, a composite node is created in the programming model for establishing communication with the endpoints.

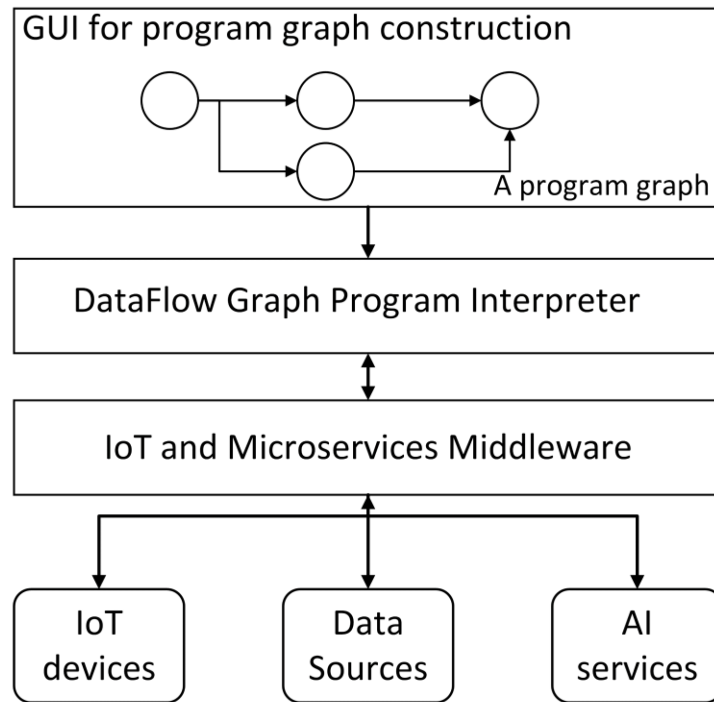


Figure 4.2 Architecture of integrative AI platform

4.2 Comparative case study

The following two case studies demonstrate the capability of designing solutions for heterogeneous problems using the agent-environment integration platform and the functional graph programming model. The features of the proposed method have been compared with Jolie [65] and Microsoft's Platform of Situated Intelligence (PSI) [51]. The demonstration and comparison have been done on the following grounds. Namely, functional abstraction, ease of integration, modularity, and expressivity.

4.2.1 Learning to solve heterogeneous maze task

A heterogeneous problem environment is constructed by combining a Reinforcement

Learning (RL) maze environment having sparse reward distribution, with a machine vision task. Fig. 4.3 illustrates the maze. A virtual agent is initially placed at coordinate 1,1 and its objective is to find a path towards a goal state marked as G, gaining maximum reward in course of its traversal. The agent can move in all four directions. Stepping into grey or white cells incurs a reward -0.5 and 0 respectively. On stepping into any of the blue cells a one-time reward 1 is generated. However, the blue cells represent the vision zone where the agent can observe the direction of the goal state from the current state in form of an image. The directions are encoded as 1 - down, 2 – right, 3 – left, 4 - up.

-0.5	-0.5	-0.5	-0.5	-0.5	-0.5	-0.5
-0.5	0	0	0	0	0	-0.5
-0.5	-0.5	0	0	-0.5	0	-0.5
-0.5	0	0	-0.5	0	0	-0.5
-0.5	0	-0.5	0	0	-0.5	-0.5
-0.5	0	0	1	1	3	-0.5
-0.5	-0.5	-0.5	2	**G**	3	-0.5
-0.5	-0.5	-0.5	-0.5	-0.5	-0.5	-0.5

Figure 4.3 The maze problem environment

Table 4.1 contains the list of microservices used as modules to solve the maze task. The value iteration module generates a policy for the maze-training module which serves as a RL training environment. The maze-training environment is same as the original maze-test environment except for the vision zone. Table 4.2 shows the list of primitives of the programming model, used to solve the maze task. The solution program has been constructed

Table 4.1 List of atomic microservices

Microservice	Code	Description	Input	Output
Value Iteration	5	Runs Value iteration and generates optimal policy	[Training-environment url, #iterations]	Policy
Digit Recognizer	6	Recognizes Arabic numerals from image	Encoded image data	Recognized digits
Maze-training	3	Maze training environment	[action, current state]	[observation, current state, reward, 0]
Maze-test	4	Maze testing environment	[action, current state]	[observation, current state, reward, goal-reached]

incrementally by divide and conquer approach. Logically separable program graphs are built as reusable functions using the *gp* node which in turn are used to construct the final program graph. gp node is similar to Lambda constructor in Lambda calculus and can convert any

arbitrary program graph to a reusable function. The reusable function created by a gp node act as a black-box component which can be directly used in another program graph.

Table 4.2 List of function nodes

Function	Code	Description	Input Type	Output Type
constant	a.K	Returns value a	any	typeof(a)
empty list	nl	Creates empty list	any	list
pop	pop	Returns nth element from the list	number X list	any
identity	id	Returns the input argument as-is	any	any
tail	tl	Returns rest of the list except 1st element	list	list
cons	cn	Appends an element in a list	any X list	list
world merge	wm	Outputs the 1st argument as is.	any	any
actuator	ac	Take action on the environment.	any	null
sensor	sn	Reads input from the environment	null	any
recurse	rc	Applies 1st function on 3rd argument until stopping condition is satisfied.	function X function X any	any
graph program	gp	Creates reusable function node with a program graph		
greater	>	Checks if 1st argument is greater than 2nd	number X number	Boolean

Fig. 4.4a represents the program graph of the function microcall, created using gp node. The function takes two input arguments. The 1st argument should specify the numeric code of the microservice (specified in Table 4.1) which needs to be called and the 2nd argument should pass the data that needs to be sent to the microservice. The sequence of actuators sends requests to the environment to call the microservice. The data returned by the microservice is temporarily stored in the environment which is eventually consumed by the sensors. The final sensor in the graph returns the data returned by the called microservice. The world merger node is placed to deterministically sequence the I/O monads [56] in a pipeline, namely sensors and actuators. Fig. 4.4b represents the program graph to determine the stopping condition of the goal search. The search in the maze is stopped when the maze test environment returns 1 in the "goal reached" flag (mentioned in Table 1). Fig. 4.4c represents the program graph for evaluating the stopping condition of executing the policy. The agent traverses through the maze following the policy until it receives a positive reward. Fig. 4.4d illustrates the program graph to execute a policy. It has one input argument to pass the policy in the form of a list. Each index of the list represents the state and the corresponding value represents the action that needs to be taken. This program graph reuses the microcall function. The microcall node returns the data returned by the maze-test microservice. The policy is appended to this list to produce the final output. Fig. 4.4e

illustrates the program graph to guide the agent within the vision zone in the maze. After each action, the maze-test environment returns an encoded image as observation. The data is sent to the digit recognizer microservice to determine the direction of the goal. The final output of the function is the data returned by the maze-test environment. Fig. 4.4f represents the integrated solver program that learns the optimal policy through value iteration in the maze-training environment and applies the policy in the maze-test environment followed by a vision task to guide the agent to reach the goal. The integration is not simple communication among loosely coupled modules rather few tight couplings are required to solve the task. Solving this environment demonstrates the ease of implementation due to low syntactic dependence of constructing integration programs and the ability to handle complex integration logic for heterogeneous tasks without compromising modularity.

Jolie [65] takes a procedural language based approach to solve the problem of development, deployment, and integration of microservices. Workflows in Jolie are used to coordinate and set dependencies among microservices. However, programming constructs to implement complex logic are not available while constructing workflows. Given the available external microservices as stated above, solving this use case is not obvious by defining a workflow, as the integration problem requires complex data transformation and coordination among the services. This can be implemented in a Jolie service using several language constructs it provides. But it follows a procedural style of text-based programming which compromises functional modularity compared to the proposed method. For example, a "loop" cannot be simply replaced with another function without modifying the rest of the code as they are semantically tightly integrated. In the proposed method any arbitrary subsection of the solution graph can be functionally reused in another program. This is usually not possible in procedural style unless every part of it is written as a reusable function, which seems like an overkill.

4.2.2 Speech recognition

This case study demonstrates the ease of integrating heterogenous components through functional abstraction. The objective is to stream audio from the microphone of an IoT device, convert it to text, and write it to a file in a file server. A similar solution [68] built on Microsoft's Platform of Situated Intelligence (PSI) has been compared.

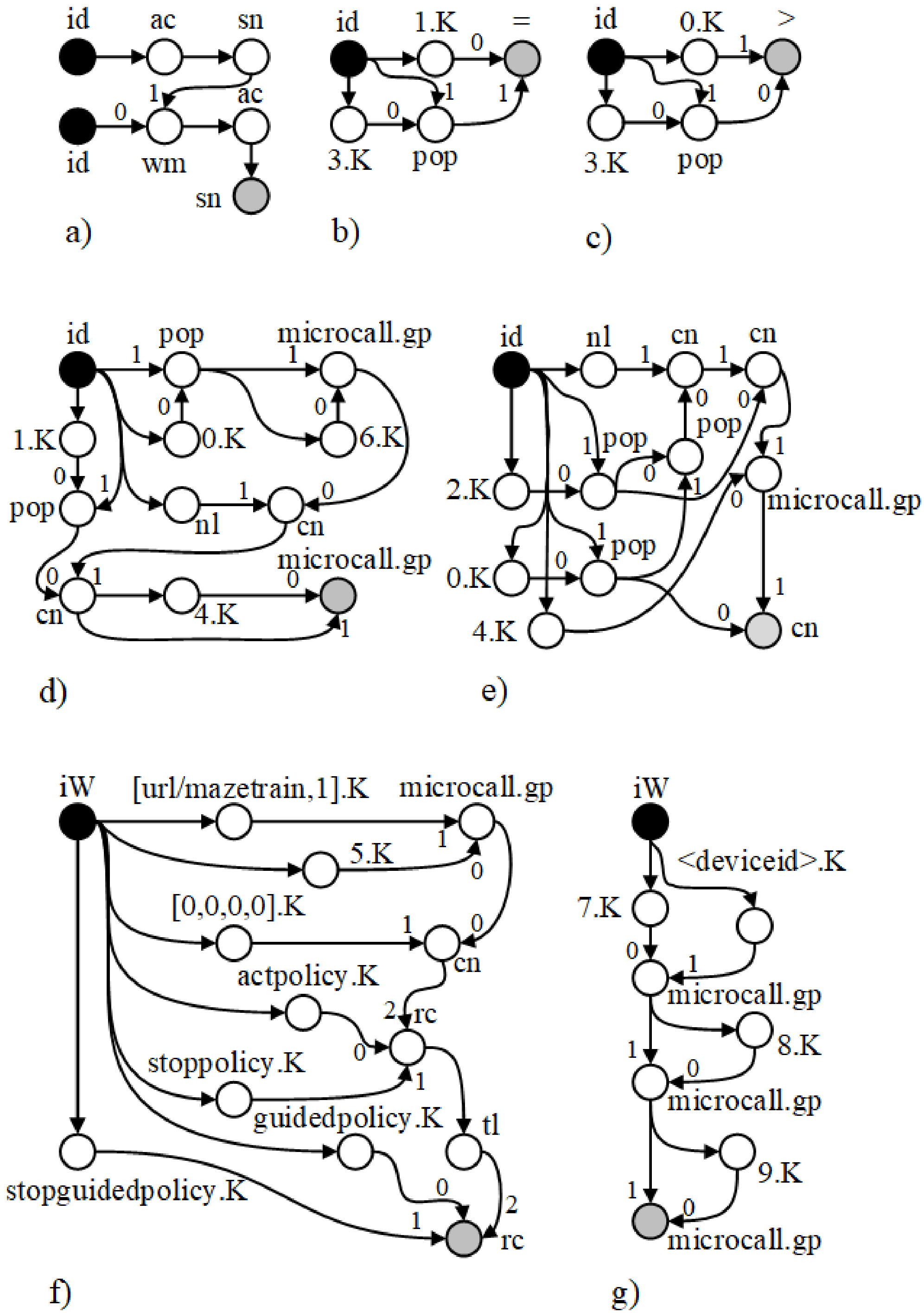


Figure 4.4 Program graphs to solve maze problem and speech recognition problem. Function nodes are represented by circles and edges by arrows. The direction of arrows represents data flow. The number on the edges represent input port# of the nodes. The black circles represent initial nodes and grey circles represent terminal nodes. a) Program graph of reusable function microcall, b) Program graph of reusable function stopguidedpolicy c) Program graph of reusable function stoppolicy, d) Program graph of reusable function actpolicy, e) Program graph of reusable function guidedpolicy, f) Program graph to solve the maze problem, g) Program graph to solve the speech recognition problem

Fig. 4.4g illustrates the program graph build for solving the speech recognition problem. It involves 3 sequential microservices calls. The first microservice (encoded as 7), connects an IoT device to capture microphone audio and return a stream. The following service (code 8) reads the stream and transcribes it, which is, in turn, returned as a stream. The final service (code 9) reads the stream data and writes it to a file server. As illustrated the same reusable function microcall is used with different parameters to read audio stream from input device, transcribe it and write it to an output device, thus providing a layer of abstraction to the application developer. From an implementation perspective, the remote streaming components are implemented using gRPC remote procedure call and IoT device communication is achieved through google cloud IoT.

In comparison to PSI the proposed platform can read and write data to remote devices, whereas from the demonstration of PSI such capability does not seem obvious. Development in PSI involves coding in C#. Syntactical and semantic knowledge of C#, as well as knowledge on the programming model of PSI, is a prerequisite for development whereas given a graphical-user-interface (GUI) the prerequisite of using the proposed platform is just semantic knowledge of the programming model. Programs are constructed graphically by creating and connecting nodes which simplifies the task. The proposed method abstracts majority of the underlying data conversion, communication, and configurations. The inherent functional modularity of the programming model allows quick reuse and modification of any arbitrary functionality of a solution by reusing or modifying the appropriate subgraph. Such functional modularization is difficult to achieve in text based procedural languages used in PSI.

4.3 SUMMARIZED COMPARISON

Solving the maze task in the proposed platform demonstrates the presence of features like modularity, expressivity, and ease of integration of heterogeneous AI components. Along with the aforementioned capabilities, the speech recognition task also demonstrated functional abstraction of components and availability of IoT devices as functions. Table 4.3 presents a comparison chart among Jolie, Microsoft PSI, and the proposed platform based on some key features. Based on all the presented features the proposed platform excels as an

integrative AI platform compared to its prior arts.

Table 4.3 Feature Comparison

Category	Feature	Jolie	Microsoft PSI	Proposed Platform
abstraction	Integration middleware available for remote AI components?	Not, readily available	Available for few built-in services mainly using azure cognitive services.	Yes, available for any Representational state transfer (REST) and gRPC based services.
abstraction	Can build arbitrary reusable composite components?	Yes, by designing Jolie services	Yes	Yes, using *gp* function
abstraction	Can connect to remote IoT devices as components?	Not readily available.	Not readily available. Components needs to be programmed	Yes, available with few configurations
ease of integration	Syntactical knowledge of a language is needed?	Yes	Yes	No, Application can be developed by graphically connecting functions
expressivity	Can implement arbitrary programming logic in integration layer?	Not in Jolie workflow used for integration.	Yes, but works for streaming data only	Yes, as all required primitives are available
modularity	Integration of arbitrary remote microservices	Yes	Not readily available. Components need to be programmed	Yes, available just by registering the endpoint
modularity	Programming style	Procedural	Procedural	Functional
modularity	can do lift and shift of arbitrary solution subpart to different problem context?	No	No	Yes, as explained in section 4 this is a consequence of using dataflow graph based functional programming

4.4 FEW CASE STUDIES ON DISASTER MANAGEMENT

Multiple use cases related to disaster prevention and response can be addressed using integrative AI. However, in this section, two use cases are discussed at a high level under disaster prevention category and prototype solutions in the integrative AI platform are presented [108]. Section 4.4.1 showcases how a disaster can be averted due to potential failure of some mission-critical machinery by detecting early signs of failure. This methodology can be categorized as machine learning based predictive maintenance [69]. Section 4.4.2 discusses how early-stage forest fires can be detected using satellite images. Due to recent advancement of deep learning techniques, detecting fire through satellite image analysis is tractable with a high degree of accuracy [70].

4.4.1 Machine failure disaster prevention

Mission-critical machines are fundamentally important to carry out a specific operation. Failure of such machines can stop those operations in an unprecedented manner which may result in a disaster including casualty. Such machines are usually equipped with several sensors to monitor the health and performance of their several subparts. The sensors can be augmented with IoT devices to transmit the captured data into the integrative AI platform. The streaming data can be pre-processed to do feature engineering after applying a time windowing strategy. The processed data can be fed into an anomaly detector API which detects anomalies on the time series data and at the same time uses the data chunk to train and update the model. Finally, if some anomaly is detected a notification is sent out to the concerned personnel for manual intervention. All these components can be made available in the platform and the application can be built by constructing a program graph. Fig. 4.5 represents a prototype program graph for machine failure detection. The graph starts with an initial node. It utilizes four microservices nodes. Namely, IoT Sensor – to capture the time series data, Data pre-processing API – for time windowing and feature engineering, Anomaly detector – for anomaly detection on real-time data, Send notification – send notifications to concerned personnel. It uses one primitive node called as conditional node (guard node). It receives a Boolean value at port number 1 and if it is true then the subgraph corresponding to port number 2 is called.

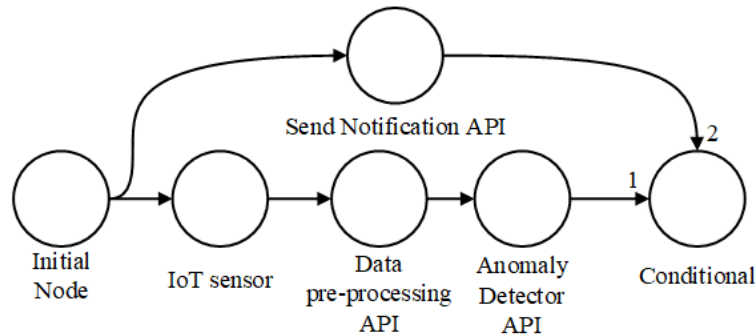


Figure 4.5 Program graph for machine failure detection

4.4.2 Forest fire detection from satellite image

Forest fires are geophysical hazards with a high damaging impact on surrounding lives. Once a forest fire spreads and gains intensity it becomes very difficult to control and, in most cases,

it subsides after taking a high toll on surrounding vegetation and animal life. Thus, it is extremely important to identify localized regions of forest fires at a very early stage in order to contain the disaster. Forest fires can be detected at an early stage from high-resolution satellite images. This method is cheap and can be applied to monitor over a large area with minimal maintenance and human effort. The idea is to train a high accuracy model to detect fires from satellite images and pinpoint the pixel region where the fire has been detected. Thereafter the pixels in the image can be mapped to latitude and longitude to identify the original geographical location. Fig. 4.6 illustrates a program graph for forest fire detection. It uses four APIs to fetch the raw image, it's metadata, identify forest fire and pinpoint geographical region of the fire.

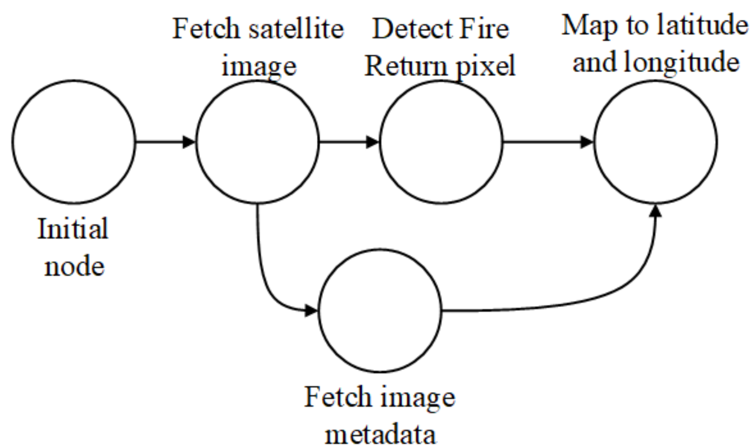


Figure 4.6 Program graph for forest fire detection

4.5 Conclusion

The proposed integration platform is demonstrated to overcome several integration challenges in solving heterogeneous AI tasks. The programming model allows developers and architects to build and implement AI architectures with low engineering efforts using black-box AI modules and IoT devices as microservices. But at the same time, it also allows construction of white-box program graphs for implementation of arbitrary integration logic. The functional abstraction of the components offloads a developer from low-level implementation details. However, a higher level of abstraction comes with some limitations. There is a limitation on the available capability of tweaking low-level implementation since most of it is abstracted using predefined methods. The next goal is to convert it into a

constructivist approach. Since the integration programs follow a functional style thus programs can be constructed by calculation and that opens the frontier of automatic programming. Machine learning models can be trained for automatic construction of integration programs to solve problem environments and that would largely reduce the manual effort of program construction. Another area of investigation would be automatic identification of reusable functions based on the usage pattern of the users to increase the level of abstraction.

5 DESIGN OF A UNIVERSAL SOLVER

"Plurality should not be posited without necessity" – Occam's Razor, (1495)

Problem-solving is essentially a search operation. Solutions need to be searched through generate and test method. Potential solution paths can be generated by sequencing multiple operators and tested in a problem environment. Usually, there can be an infinite number of potential solutions if no prior knowledge exists about probable solution paths. The search can easily get lost in a combinatorial explosion. As Universal search allows one to create an asymptotically optimal search mechanism which can handle a wide range of problems, it is a good choice for building a problem-solver. However combinatorial explosion is still lurking behind though its dependency shifted from problem size to solution size. Carefully designing the search mechanism can rapidly dampen the search space. Also, domain knowledge, experiential knowledge, etc. plays a crucial role in dampening the search space. Learning is a mechanism to gain such knowledge while solving similar problems or contrasting problems and observing patterns across different problems. Knowledge creates initial biases during problem-solving and drives the solver towards known pathways to exploit the already learned solution. Thus, we focused on designing a search mechanism inspired by Universal search and developing several strategies for dampening the search space which aids in finding solutions for many problems in a realistic time.

5.1 UNIVERSAL SEARCH

Universal search, initially proposed by Levin [41] and further improved by Hutter [71] is an asymptotically optimal method to solve machine inversion problem or time limited optimization problem. A wide variety of computational problems can be either transferred to machine inversion problem or time limited optimization problem [72]. Given a specific machine model M the machine inversion problem is to find an input string p when given as an input to M will produce the desired output string y $(M(p) = y)$. Time limited optimization problem is to find an input string p within a fixed time such that the value of p computed on M is maximum. p can be a sequence of instructions of a programming language and M can be represented as the interpreter of the programming language. The universal search follows a generate and test method to search the solution program in the program

space. It generates and tests programs in the order of Levin's complexity. The search process stops if the solution program is found for a machine inversion problem or time limit exceeds for time limited optimization problem. Levin's complexity with respect to machine model *M* is calculated by the following equation where y is the solution string which is given as output when program *p* is run on the machine model *M* and *time(p)* is the runtime of program *p*.

$$Kt_M(y) = \min\{l(p) + \log(time(p)) : l(p) = length\ of\ p\ and\ M(p) = y\} \tag{5.1}$$

If programs are represented as bit strings, then the probability that any random program *p* is a solution program can be stated as $P = 2^{-l(p)}$, which approximates the algorithmic probability [14] of the string *y*. Thus, Levin's complexity can be rewritten as

$$Kt_M(y) = min\{\log\left(\frac{time(p)}{P}\right)\} \tag{5.2}$$

The search process is carried out iteratively in phases starting from 1. Phase is incremented by 1 after every iteration. In each phase, it is expected that Levin's complexity of the solution string is equal to that phase value. Thus, while testing a program it is expected that the same program is the solution program, and time is allocated to that program according to the equation $time(p) = P\ X\ 2^{Phase}$ which is derived from equation (5.2). If the program fails to output the solution string within the allocated time, it tries the same strategy for other programs within the same phase. If no programs are left to test within a phase the phase value is incremented and the same process repeats. Clearly the search time for the solution program is bounded by $2^{Kt_M(y)} = P^{-1}time(p)$ which is proportional to the runtime of solution program p. However, there is a constant factor P^{-1} associated with it, which is independent of the problem size but is dependent on the solution size. This factor is generally huge and it needs to be dampened in order to make this search process practically feasible in real-world. Search space can be dampened by storing and updating the information gained about finding solutions in an environment through experience, in the form of conditional probability distributions or guiding probability distributions [73] among different instructions of the programming language. Successful trials will positively reinforce the probabilities of the solution programs whereas failures will negatively impact the probabilities of the concerned programs. Universal search when combined with reinforcement learning will need

probability updates based on rewards or punishments received. An efficient design of the search process and incremental learning process may dampen the constant factor enough to make it usable in practical environments.

5.2 FGPM FOR UNIVERSAL SEARCH

Universal search needs a specific programming model to generate and test solution programs. As stated in chapter 3, the proposed programming model FGPM is designed to serve this purpose. Each program in FGPM is assigned a program probability based on the conditional probabilities of the edges. A conditional probability is associated with each edge in a program graph. It signifies, given the history of graph between a node n and the initial node, what is the probability of connecting the output of node n with input port i of a new node n' such that continuity of the program from node n solves a given task. The default probability of connecting input port i of a new node n' with the output port of node n is calculated by the formula $1/\sum_{n' \in \, allnodes} \sum_{i \, \in \, input \, ports \, of \, n'} i_{n'}$, where $i_{n'} = 1$ if $i_{n'}$ is type compatible with output port of n else $i_{n'} = 0$ and $allnodes$ represent the set of all distinct function nodes to be used in the search. A distinct type of edge is identified by its source node and the target input port of a specific type of function node. The default probability is assigned to all newly created edges. The total program probability can be calculated by multiplying all the distinct edge probabilities of a program.

To implement phased search, another functionality is needed which would enable capturing the runtime of a program and abort when some time limit is exceeded. Each node gets allocated a time limit and before executing, it checks if the value of the time limit is equal to zero. If yes, then the program is aborted else the node function is executed and the value of the time limit is reduced by 1 for each atomic operation. This is applied for all nodes within a program graph including nodes within a composite higher-order function and dynamically generated nodes at runtime.

5.2.1 Meaningful information gain in transfer learning

Dataflow graphs are good at capturing independence among different functions based on data flow. This independence condition actually helps improve transfer learning. Meaningful

information can be stated as the minimum information that can be transferred from one task to another which evidently helps in reducing complexity of the later task. The information transfer or mutual information between two solution strings X and Y can be written as follows, where K denotes Kolmogorov complexity and X^* denotes the program for generating X [74].

$$I(X:Y) = K(Y) - K(Y/X^*) \tag{5.3}$$

The program probability, algorithmic probability, and Kolmogorov complexity have equal status such that $-\log(P_U(x)) \approx K(x)$ [75], where $P_U(.)$ denotes algorithmic probability and U is some Turing complete language. Thus, we can replace Kolmogorov complexity with algorithmic probability in equation 5.3,

$$I(X:Y) = \log\left(\frac{P_U(Y|X^*)}{P_U(Y)}\right) \tag{5.4}$$

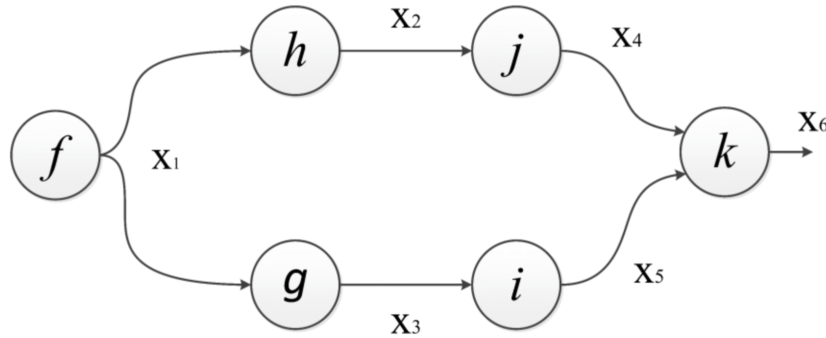


Figure 5.1 A sample program with parallel data flow to demonstrate how storing conditional probabilities in graph helps in transfer learning. f, g, h, i, j has only one input port and k have two.

Fig. 5.1 shows a sample program graph to demonstrate the effectiveness of storing conditional probability distribution of functions in dataflow graph. Let, $p_1 = j \circ h$, $p_2 = i \circ g$. Let the conditional probabilities be represented as $P(.|.)$, such that the probability of input port k of h connecting with the output port of f is given as $P(h^k|f)$. The algorithmic probability of a string is approximated as the probability of the shortest program that generates the string, so that $P_U(x_1, x_2) = P(f)P(h^1|f)$. The parallel paths in dataflow graphs are independent of each other in terms of the output data generated. The sequence x_1, x_2, x_4 and x_1, x_3, x_5 are independent of each other in the sense that they are produced by

separate functions $p_1 \circ f$ and $p_2 \circ f$ independent from each other.

Suppose the program graph G_1 in Fig. 5.1 represents a solution for a specific task $T_1$where $p_1 \circ f$ and $p_2 \circ f$ solves two subproblems s_1, s_2 independent of each other. The program G_1 produces a partially ordered set of solution string $(S, R) = (\{x_1, x_2, x_3, x_4, x_5, x_6\}, \{(x_1, x_2), (x_1, x_3), (x_2, x_4), (x_3, x_5), (x_4, x_6), (x_5, x_6)\})$. The partial order is due to independence of sub tasks s_1, s_2. Another related task can benefit from transfer learning from task T_1 if it's set of solution string contains a subset of (S, R). Let us assume another program graph G_2 solves another related task T_2 with subproblem s_2 in it. This results in inclusion of the set $(S_2, R_2) = (\{x_1, x_3, x_5\}, \{(x_1, x_3), (x_3, x_5)\})$ in its solution string, which is a subset of (S, R). Thus, the solution program will be $p_3 \circ p_2 \circ f$, where it is assumed when a function p_3 is composed with the subprogram $p_2 \circ f$ produces the solution for T_2 (composition can be done just by connecting the output of node i to input port of p_3). Let us consider X and Y be the partially ordered set of solution strings of tasks T_1 and T_2 respectively. $P(Y|X^*) = P(p_3^1|G_1) = P(p_3^1|p_2 \circ f)$ as p_3^1 is conditionally independent of any other functions in G_1. Information transfer from task $T_1$to T_2 is given as follows.

$$I_1(X{:}Y) = \log\left(\frac{P(p_3^1|p_2 \circ f)}{P(f) \text{x} P(p_2^1|f) \text{x} P(p_3^1|p_2 \circ f)}\right) = \log\left(\frac{1}{P(f) \text{x} P(p_2^1|f)}\right) \quad (5.5)$$

Now considering some type of strictly sequential programming language. The conditional probability distribution of instructions represents the probability of a symbol following a sequence of symbols and naturally, it fails to capture the independencies among different parts of a program. Writing shortest program corresponding to the program graph G_1in such a scheme will result in writing the descriptions of the functions from the set $\{f, p_1, p_2, k\}$ in some specific order without repetitions, along with some more symbols. Due to relative independence between $p_1$and p_2 the data produced by them will not affect each other. However, due to sequential ordering of symbols, a dependency between $p_1$and p_2 is captured in the conditional probability distribution. Let us assume the functions in the solution program for task T_1 is written in the following order (f, p_1, p_2, k) and ignoring other symbols. The conditional probability distribution table will contain a set of discrete probabilities $\{P(f), P(p_1|f), P(p_2|f, p_1), P(k|f, p_1, p_2)\}$ for the current program. Considering transfer learning from T_1, the solution program for task T_2 will contain the functions as per the

ordered set (f, p_1, p_2, p_4, p_3) where the functions $\{f, p_1, p_2\}$ produces the ordered set of solution string $\{x_1, x_3, x_5\}$ for s_2 along with some other strings in the set. The function p_4 extracts the string set $\{x_1, x_3, x_5\}$ from the complete ordered set of strings. Thereafter p_3 is applied on the last string in the set (x_5) to output the final solution string. Thus, in this case, information transfer is

$$I_2(X:Y) = \log\left(\frac{P(p_3|f,p_1,p_2,p_4) \text{ x } P(p_4|f,p_1,p_2)}{P(f)P(p_1|f)P(p_2|f,p_1)P(p_3|f,p_1,p_2,p_4)P(p_4|f,p_1,p_2)}\right) = \log\left(\frac{1}{P(f) \text{x } P(p_1|f) \text{ x } P(p_2|f,p_1)}\right) \quad (5.6)$$

Considering no initial bias, clearly $I_2(X:Y) > I_1(X:Y)$ which states that information transfer in the later is higher than the first. Also, the program probability of the solution program for T_2 in the sequential language should be comparatively lower than that is for the dataflow graph-based programming model (due to the presence of extra functions $p_1$and p_4). Thus, task T_2 can be solved with a program of higher probability using lower information transfer than I_2 after solving the same training task T_1. It signifies that though information transfer is higher in the sequential language yet meaningful information transfer to solve T_2 is same as that of I_1. On top of that extra computation needs to be done to extract meaningful information from the total information transfer. This process of gaining extra information and then extracting meaningful information from it increases the complexity of the solution program thus making the transfer learning process inefficient in the sequential language.

5.3 METASEARCHER

The metasearcher of the agent is responsible for implementing universal search and searches for solution in the program space for a given problem environment [76]. It starts with initializing a phase variable, a phase limit, a list of available primitive functions to be used for program generation, and a graph object called as search graph (*sg*) containing initial node. Programs are generated by adding and connecting all possible primitives with all possible combinations of successfully executed nodes present in the search graph, after satisfying type compatibility. Nodes that create syntactically redundant programs are eventually deleted. All programs are generated within the same search graph where subgraphs between each terminal node and initial node represent a unique program. The search proceeds in phases

Algorithm 5.1 Metasearcher algorithm

Procedure evaluate (sg, F_k)
  t_k = Phase * P_k / C; R_k = 0
  execute F_k for t_k time or until it halts
  set status = execution status of F_k
  if status is succeeded
    set R_k= reward gained by F_k
  if goal reached
    return F_k, R_k, status
  else:
    return null, R_k, status

Procedure extend_execute (sg, Phase, T_l, T_R)
  for all new valid programs F_k generated by adding single function node k in sg such that Phase * P_k > 1 and T_R < T_l
    L, R_k, status = evaluate(sg, F_k)
    set status in the node k in sg
    T_R= T_R + Runtime of F_k
    if $F_k \equiv F_j \mid \forall F_j \in$ sg
      Halt continuation of F_k
    if status is succeeded
      Apply incremental learning on F_k using R_k
    if status is incomplete
      delete node k
    if L != null
      return L
  return null

Procedure metasearcher (sg)
  run_Phase = 2; Phase = 2
  while run_Phase < Phase_limit
    T_l = run_Phase/C; T_R= 0
    while T_R < T_l
      L= extend_execute (sg, Phase, T_l, T_R)
      if L != null
        return L, sg
      Phase = Phase * 2
    run_Phase = run_Phase * 2
  return null, sg

and in each phase, a fraction of phase value is allotted as the runtime for a program which is proportional to its program probability. Fig. 5.2 illustrates a sample search graph generated by the metasearcher where each node represents a function in the proposed programming model. For each node in the search graph, a unique program can be extracted by recursively fetching all parent nodes starting from the given node until the initial node is reached. During execution, program graphs corresponding to each terminal node are executed.

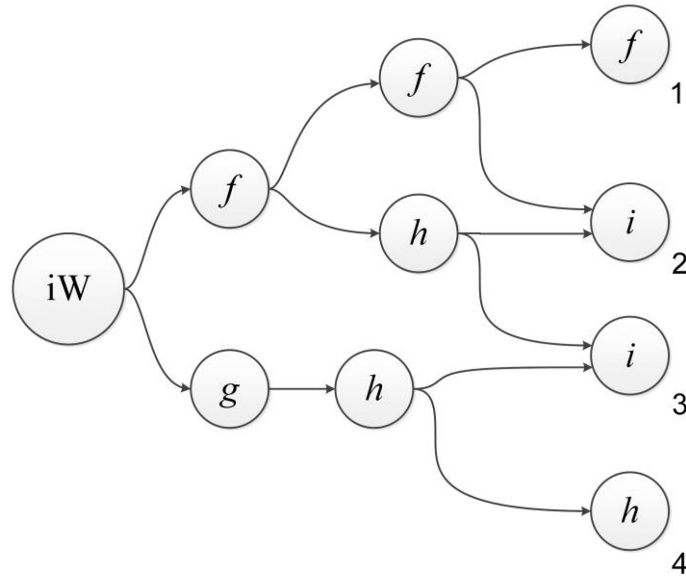


Figure 5.2 A sample search graph generated by the metasearcher. Nodes 1-4 denote terminal nodes and each terminal node denote a separate program. For each node in the search graph a unique program can be extracted by recursively fetching all parent nodes starting from a given node until the initial node

5.4 PROGRAM PRUNING

While generating all possible combinations of program graph the metasearcher creates a bunch of equivalent programs in terms of output produced for all inputs and some program produces undefined output (like non-terminating programs). As we are using functional program paradigm in FGPM so we can algebraically reason about programs for all inputs (due to the absence of side effects). Transforming programs to algebraic equations followed by simplification by applying different axioms (stated in Chapter 3 as semantic rules of different functions) helps us find equivalent programs just by comparing the algebraic expression. Programs that produce undefined output can also be recognized by simplifying the algebraic expressions. However, it is not possible to reason and find all equivalent and invalid programs due to halting problem and unknown environments. But a large number of the programs can be deterministically reasoned about, by this method. Paul & Bhaumik [77] proposed a method of pruning equivalent programs in universal search. However, their method would efficiently work in case of a single task. The strategy would not work in an agent with memory for solving incrementally a set of tasks in different but related environments. However, in this case, the semantic reasoning of functional programs

guarantees the equivalence of equivalent programs across all environments.

To implement the reasoning and pruning of programs, the nodes of FGPM are enhanced with another storage type to store the algebraic expressions of the programs which we will be calling as program expressions. For all functions, program expressions contain the results after applying evaluate function as per the semantic relations. Since all functions return a monadic type so the program expression stores a pair of data and world expression. It does not store a copy of the actual world object but a symbol for it. For monadic functions, both the data section and world section in the program expression are updated as per the semantic equation of evaluating the function by adding required symbols in the expression. For pure functions, only the expression of the data section is updated. Simplification, which essentially means reducing the number of distinct symbols in the expression in this context, is carried out on the program expression by applying various equivalence rules stated in chapter 3. Every node contains a program expression of the program of which the concerned node is the terminal node. Let us consider the following example.

A sample FGPM program can be written as $add^4 \circ (constant\ {}^2_1\ X\ identity^3) \circ sensor^1_A \circ initWorld^0$, where the superscript represents the node labels and subscript represents either a value or a type. Here type A is a numeric type. The program expression of all the nodes is written as follows, where $node^n$ represents node with label n.

$node^0 \rightarrow (world);$

$node^1 \rightarrow (read(world));$

$node^2 \rightarrow (1, read(world)[world]);$

$node^3 \rightarrow (read(world));$

$node^4 \rightarrow (plus(1, read(world)[data]), read(world)[world]);$

If two such terminal node expressions are equal then two programs are functionally equivalent. If the data section (the remaining program expression after removing the expression for world object) of a program expression is $undefined$ then such a program is

invalid.

The metasearcher, while generating new programs by adding new child nodes, checks the current program expression against the program expressions of all generated valid programs. If it finds a match, it prunes the newly generated program by marking the terminal node invalid. For example, $node^1$ and $node^3$ are functionally equivalent. Thus $node^3$ will be pruned and the above stated program will not be generated in the search graph, rather $node^4$ will directly connect with $node^1$ instead of going through $node^3$ to generate the same functionally equivalent program. It also checks for $undefined$ symbol in the current program expression. If it finds one, it marks the terminal node invalid. No child node gets connected to an invalid terminal node.

5.5 INCREMENTAL LEARNING

Incremental learning is the process of gaining mutual information by solving a related sequence of tasks which would help solve later related tasks by reducing the search space. The metasearcher operates in a reinforcement learning setting where it is assumed the environment will generate some reward or punishment after a valid program interacts with it. Incremental learning in universal search is implemented by updating the conditional probability distribution of edges in the search graph based on the rewards received. The agent will be exposed to a series of problems. Incremental learning is achieved by updating the conditional probability distribution based on gradient ascent mechanism thereby maximizing future expected rewards. We will adopt the following notations while deriving the probability update expression. Let f_i denote the ith node in the program search graph and F_i denote the program graph ending with terminal node f_i. Let f_k be the kth node of search graph which is an intermediate node in the program graph F_i. $p(f_l^j|F_k)$ denotes the probability of connecting input port j of node f_l with output port of f_k or in other words probability of composing function f_l^j after F_k which might solve the task or generate a positive reward. Thus, the total probability of a program F_i can be written as follows, where inp (f_l) denotes set of all input port numbers of f_l,

$$P(F_i) = \prod_{\forall f_l \in F_i \wedge \forall j \in \text{inp}(f_l) \wedge \forall f_k \in F_i} p\left(f_l^j \middle| F_k\right) \qquad (5.7)$$

There is always a single type of initial node (initWorld) in the search graph. Thus, the prior probability of the initial node is 1 and this term is ignored in equation 5.7. Let R_i be the reward received after executing program F_i. Given the agent has already tested a subprogram F_k of F_i the probability of receiving reward R_i in future can be given as,

$$P(R_i|F_k) = \frac{P(F_i)}{P(F_k)} \tag{5.8}$$

The total expected future reward of some horizon *i'* in the search graph can be stated as,

$$J(p) = \sum_{\forall F_k} \sum_{\forall i>k \wedge \forall i<i'} P(R_i|F_k) R_i \tag{5.9}$$

The goal is to maximize the objective function with respect to the factored conditional probability distribution (p). The optimal probability distribution can be written as,

$$p^* = \arg\max_p J(p) \tag{5.10}$$

We will be following stochastic gradient ascent method such that probability parameters are updated after each execution of a program. Thus, combining equation 5.7, 5.8, 5.9 and 5.10 the update rule for each probability parameter after execution of the program F_i can be written as,

$$p\left(f_l^j\middle|F_k\right) = p\left(f_l^j\middle|F_k\right) + \alpha p\left(f_l^j\middle|F_k\right)^{\frac{(1+sgn(-R_i))}{2}} \left(1 - p\left(f_l^j\middle|F_k\right)\right)^{\frac{(1+sgn(R_i))}{2}} \frac{\partial P(R_i|F_k)R_i}{\partial p\left(f_l^j\middle|F_k\right)} \approx$$

$$p\left(f_l^j\middle|F_k\right) + \alpha p\left(f_l^j\middle|F_k\right)^{\frac{(1+sgn(-R_i))}{2}} \left(1 - p\left(f_l^j\middle|F_k\right)\right)^{\frac{(1+sgn(R_i))}{2}} R_i \frac{P(F_i)}{P(F_k) \text{ x } p\left(f_l^j\middle|F_k\right)} \tag{5.11}$$

where α is a constant called as learning rate and the factor $p\left(f_l^j\middle|F_k\right)^{\frac{(1+sgn(-R_i))}{2}} \left(1 - p\left(f_l^j\middle|F_k\right)\right)^{\frac{(1+sgn(R_i))}{2}}$ ensures to keep the probability value within 1. To keep things simple, we approximated the probability update expression by considering the factored probability values within a program graph are independent of each other. Updating the probability values just by the specified strategy will not ensure to keep the total probability mass of the program search graph within 1. In order to keep the total probability mass consistent, the delta

increment of a probability value of an outgoing edge from node f_k needs to be balanced by reducing the same amount of probability mass from rest of the outgoing edges of node f_k. Thus, along with the above update rule the following update rule also needs to be followed where $f_{l'}^{j}$ denotes an input port j of function $f_{l'}$ compatible to be connected with the output port of f_k. Probability mass is extracted from rest of the edges in the proportion of edge probability, which means edges with highest value of conditional probability will lose the major share of the probability mass. This update rule needs to be applied for all outgoing edges of f_k such that $f_{l'}^{j} \neq f_{l}^{j}$.

$$p\left(f_{l'}^{j}\middle|F_k\right) = p\left(f_{l'}^{j}\middle|F_k\right) - \frac{\left(1 - p\left(f_{l'}^{j}\middle|F_k\right)\right)^{\frac{(1+sgn(-R_i))}{2}} p\left(f_{l'}^{j}\middle|F_k\right)^{\frac{(1+sgn(R_i))}{2}}}{\sum_{\forall f_{l'}^{j} \wedge f_{l'}^{j} \neq f_{l}^{j}} \left(1 - p\left(f_{l'}^{j}\middle|F_k\right)\right)^{\frac{(1+sgn(-R_i))}{2}} \left(1 - p\left(f_{l}^{j}\middle|F_k\right)\right)^{\frac{(1+sgn(R_i))}{2}}}$$

$$\alpha p\left(f_{l}^{j}\middle|F_k\right)^{\frac{(1+sgn(-R_i))}{2}} \left(1 - p\left(f_{l}^{j}\middle|F_k\right)\right)^{\frac{(1+sgn(R_i))}{2}} \frac{\partial P(R_i|F_k)R_i}{\partial p\left(f_{l}^{j}\middle|F_k\right)} \tag{5.12}$$

In the search graph, there can be multiple instances of node types f_l and $f_{l'}$ having input ports j and j' connected to k. Each of them might be part of separate programs. However, to maintain consistency the update rules 5.11 and 5.12 apply to all such instances simultaneously even if it is due to the execution of a single program.

As the agent uses universal search to find a solution, exploration is achieved through generating and testing multiple programs where each program is allocated a fixed time to run in each phase. Even when a solution program is found for a specific task the agent will continue to explore for other solution paths as per the universal search algorithm. Exploitation is achieved by increasing the program probability of solution programs through incremental learning, thereby allocating more time to those programs and extension of those programs for subsequent tasks.

5.6 HEURISTIC PLAYOUT

Heuristic playout is a heuristic method proposed to run the agent in a time limited optimization setting with a trained search graph. Time limited optimization in this context means gaining maximum reward while running the metasearcher for a limited amount of time. The training is carried out by running the metasearcher in a training environment and thereby applying incremental learning to update the probability distribution. The following heuristic algorithm runs the agent in a test environment with the aim to collect maximum reward in a fixed amount of time. A section of the search graph (*sg*) is selected, containing all programs (P_k) which generated positive reward (R_k). Thereafter unbounded knapsack problem is solved repeatedly to select and execute the best program. The capacity of knapsack is set as remaining time (T), cost is average runtime (t_k) of each program and value is average reward gained by each program. Following is the algorithm for test playout.

Algorithm 5.2 Heuristic Playout algorithm

```
procedure  run_knapsack (sg_sub)
   Initialize knapsack of all P_k ∈ sg_sub
      Capacity = T; value_k = average R_k; cost_k = t_k
   Plan = solve unbounded knapsack
   return Plan
procedure  playout (sg, T)
   sg_sub = subgraph of sg| R_k > 0, ∀ P_k ∈ sg_sub
   Plan = run_knapsack (sg_sub)
   R_T=0
   while T > 0
      Run best program  P_K from Plan
      R_T = R_T + R_K
      update average R_K and average t_K of P_K
      T = T - t_K
      Plan = run_knapsack (sg_sub)
      if Plan is empty
         return R_T
   return R_T
```

5.7 MERGING GENETIC PROGRAMMING WITH UNIVERSAL SEARCH

Genetic programming (GP) is a heuristic search technique which searches over program

space starting from an initial population of programs. The next generation of programs is evolved by crossover and mutation operations on the existing population followed by a selection operation based on some fitness criteria. In general, getting a good initial population of programs for a specific problem environment and finding the fitness function is quite challenging. On top of that choosing functions and terminals satisfying closure and sufficiency property [78] is another challenge. All these challenges except the need for sufficiency property, cease to exist when we combine GP with the universal search based RL agent.

Universal search starts with the initial node in a search graph and incrementally adds subsequent function nodes. It is not required to strictly initialize a population of programs. Choosing a set of functions satisfying the sufficiency property is necessary to find solutions in a problem environment. Satisfying closure property is optional, as exceptions and errors are gracefully handled in universal search. In RL agent, the utility of a program is measured directly with respect to the total reward gained. Thus, formalizing a complex fitness function can be avoided. Recursions and loops can be gracefully used as functions without stumbling due to halting problem, as Levin's search interrupts each program execution after a finite time, allocated proportional to program probability.

5.7.1 Crossover and mutation

In GP, programs are evolved by transforming them using three genetic operators, namely, mutation, crossover, and reproduction. The crossover operator arbitrarily combines sections of two parent programs and derives the child program. The mutation operator makes small arbitrary modifications in a single parent program to derive the child program. When GP is combined with universal search, crossover and mutation operators come for free. We define the crossover operation as connecting a multi-argument function node with output ports of arbitrary nodes of parent programs to create a new program. Fig. 5.3a shows a crossover operation in a program graph which involves merging of arbitrary sections of two parent programs using a multi-argument function node F. Merging of n unique non-overlapping programs with n argument function can be considered as applying the crossover operation $n-1$ times.

In this context mutation is defined as adding any single argument or multi-argument function

nodes anywhere within a single parent program, resulting in a new program. Fig. 5.3b and 5.3c shows mutation operation on a single program graph, once with a single argument function node G another with double argument function node F.

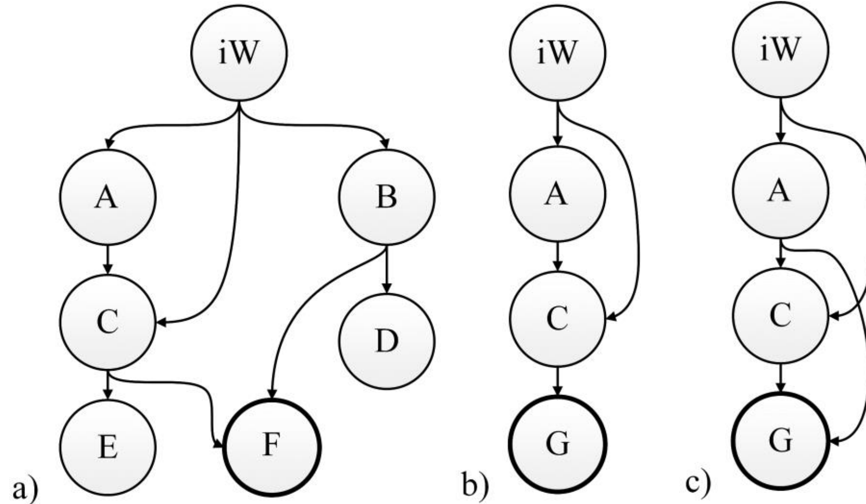


Figure 5.3 a) Crossover operation in a program graph. Nodes iW, A, C and E constitute one parent program and nodes iW, B and D constitute another. Node F is a multi-argument function node which combined parts of parent programs to generate a new program. b, c) Mutation operation in program graph. Nodes iW, A and C constitute a parent program. F is a multi-argument function node and G is a single-argument function node.

5.7.2 Selection

GP evaluates a generation of programs based on a fitness function. Fitness measures the utility of a program in the context of the problem being solved. Based on the fitness a population of programs is selected as the next generation. Others are discarded. The RL agent uses the following fitness function where f_k is the fitness of the program with terminal node k, k' denotes all parent nodes of k, γ is a decay constant such that $0 < \gamma < 1$ and R_k is the reward gained after running program with terminal node k.

$$f_k = \gamma \max_{i \in k'} f_i + R_k \tag{5.13}$$

The fitness function of a node recursively adds the max of fitness of its parents after decaying with a constant factor and in doing so younger nodes are given more importance in calculating overall fitness. The fitness function selects the historically best path within a program graph in terms of reward gained and calculates the fitness based on the rewards generated in that path.

GP is essentially a beam search through search graph. The search is focused on a section of search graph of approximately same size. During selection, the fittest programs are chosen and added to the next generation until total number of nodes added in the population exceeds a population size. Population size is another hyperparameter. Multiple programs chosen for the next generation might have common subgraphs. Nodes in common subgraphs are counted once. All chosen programs will have at least the initial node as a common node. Thus, GP is like applying selection to choose a section of search graph and continue universal search with that section by applying crossover and mutation operation using available function nodes. Gradient based learning is like fine tuning the bias in the search space whereas selection in GP is like crude adjustment of search space based on fitness.

5.7.3 Avoiding local optima

GP may suffer from premature convergence or can get stuck in local optima for a long time. The problem amplifies in a delayed reward RL environment where an agent has to take multiple different actions in order to get a reward. To alleviate this problem, a hybrid GP search is carried out in parallel with regular universal search. In each phase, the total phase time is equally divided to GP search and regular universal search in the hope that the regular universal search will pull out the agent from local peak if GP is stuck.

5.7.4 Metasearcher with GP

The following algorithm implements GP within universal search. The crossover and mutation operations are carried out by the natural program generation method of universal search. The procedure *extend_execute_gp* implements the selection operation of GP, which is based on fitness f_k and program probability P_k. Population size S_{GP} is a hyperparameter. The metasearcher has been modified to allocate half of the time to GP and half to regular universal search in each phase.

Algorithm 5.3 Metasearcher with GP

Procedure extend_execute_gp (sg, $Phase_{GP}$, T_l, T_{GP})
  Initialize empty search graph sg_{GP}
  N_{GP} = sort terminal nodes of sg by fitness f_k, P_k in descending order
  for node k in N_{GP}
    select program subgraph F_k from sg

merge F_k in sg_{GP}
if #nodes in $sg_{GP} \geq S_{GP}$
break
L= extend_execute (sg_{GP}, $Phase_{GP}$, T_l, T_{GP})
return L

Procedure metasearcher (sg)
run_Phase = 2; Phase = 2; $Phase_{GP}$=2
while run_Phase < Phase_limit
T_l = run_Phase/C; $T_R = 0$; $T_{GP} = 0$
while $T_R+T_{GP}<T_l$
L= extend_execute_gp (sg, Phase, $T_l/2$, T_{GP})
if L != null
return L, sg
$Phase_{GP} = Phase_{GP} * 2$
L= extend_execute (sg, Phase, $T_l/2$, T_R)
if L != null
return L, sg
Phase = Phase * 2
run_Phase = run_Phase * 2
return null, sg

5.8 Experimental results

The interpreter of the programming model and the metasearcher is built in python 3. We conducted two classes of experiments. One class is to prove the above theoretical claims on

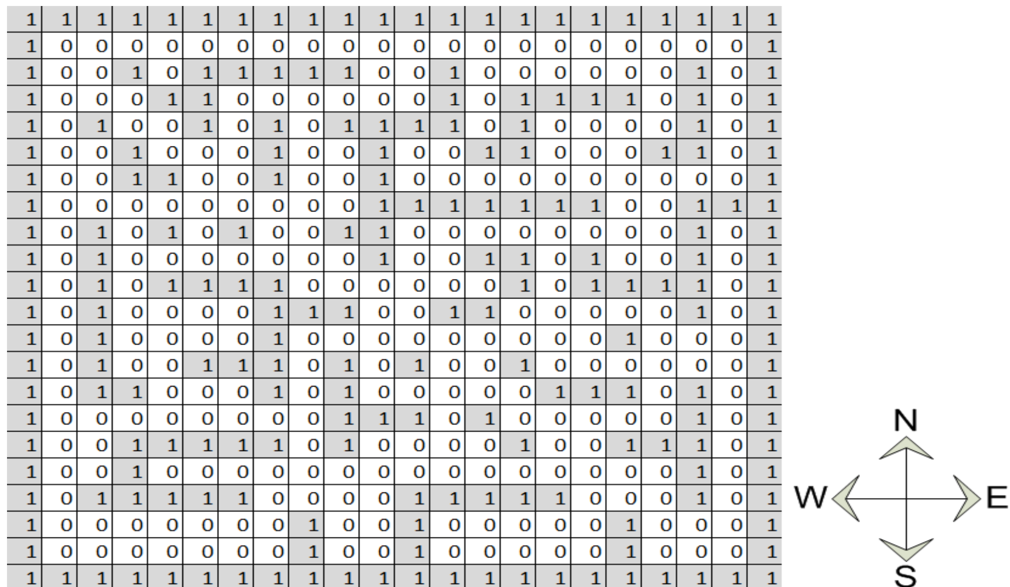


Figure 5.4 A sample 20x20 maze. A cell containing 1 denotes a blocked cell and the cells containing 0 denotes open

how the agent can handle the problem of program overrepresentation, dampen the

combinatorial explosion through effective incremental learning and find a solution in a large partially observable environment with delayed reward setting. The later set of experiments are done to compare our agent against current state-of-the-art methods on some benchmark POMDP problems. These experiments show exceptional performance statistics against current state-of-the-art methods.

For the first set of experiments, the agent is given a large partially observable maze to solve. As shown in Fig. 5.4, for experimenting with our solver agent, we used the same maze as described by Paul et. al. [77]. The maze is of dimension 20x20 with 274 open cells where 1 denotes a closed cell and 0 an open one. The objective can be stated as follows. Given an agent placed in a specific cell facing a specific direction (initial state), find a path to the goal state which is specified by the coordinates of the final cell and the final face direction. Any action in the maze can be taken through the actuator node which includes moving and changing direction. At no point, the agent can read the exact coordinates of its location in the grid. It can only check if the next cell is open or blocked and if the goal is reached or not. This creates a partially observable environment.

Within the environment, the combination of coordinates of the agent and the direction of the agent is represented as agent's state. Direction is represented by a vector of size 2. [1, 0] represents south, [0, 1] - east, [-1, 0] - north and [0, -1] - west. The counting of rows and column starts from 0. Thus, an agent state can be represented as [x, y, [a, b]], where $(0,0) \leq (x,y) \leq (21,21)$ and $a, b \in \{0,1,-1\}$.

The first experiment demonstrates the effectiveness of the equivalent program pruning on universal search without incremental learning. The agent is given an initial state [1, 1, [1, 0]] and goal state [20, 20, [1, 0]] in the environment. Metasearcher is run up to phase 65536, once with equivalent program pruning and once without it. The runtime of the metasearcher is measured as total number of atomic operations done in executing functions. Fig. 5.5a shows the plot of the runtime of metasearcher with and without program pruning and demonstrates how program pruning helps in dampening the growth rate of runtime. Constant C is chosen as 0.1.

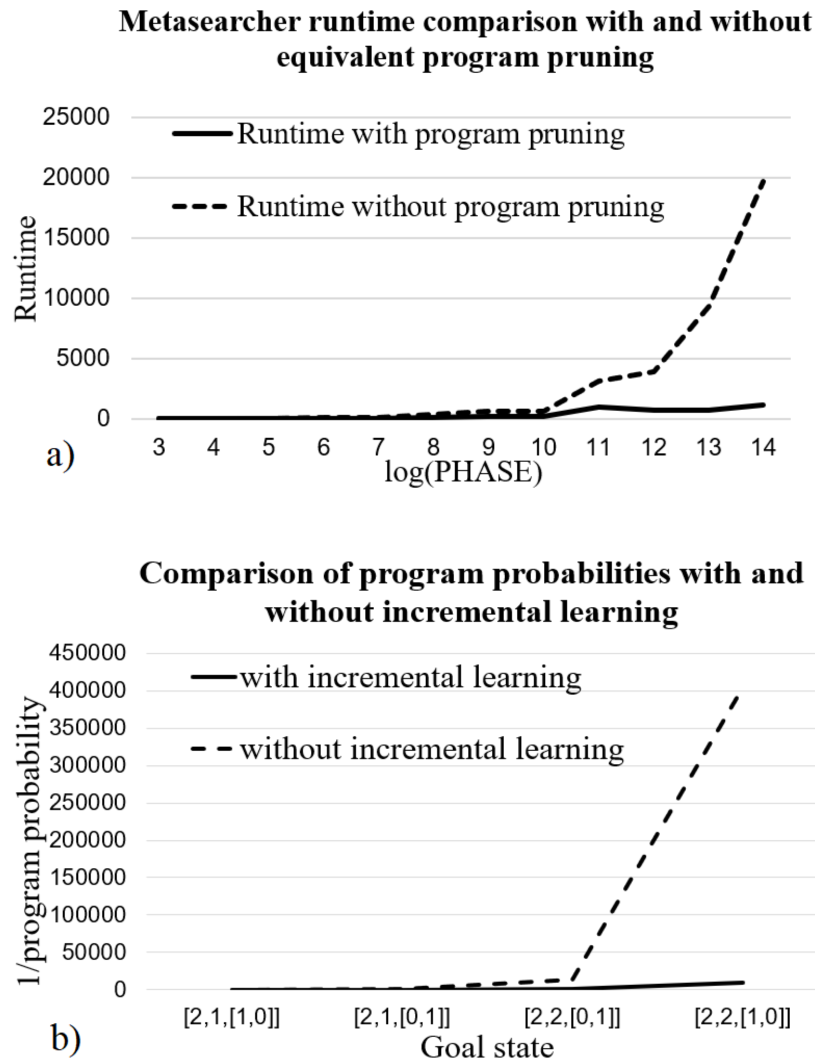


Figure 5.5 a) Plot for comparison of runtime of metasearcher with and without equivalent program pruning. b) Plot for comparison of inverse of program probability of solutions found by metasearcher with and without incremental learning.

Experiment 2 is on demonstrating the effectiveness of incremental learning in universal search. The agent is given an initial state [1, 1, [1, 0]] and 4 subsequent tasks of finding the path to goal states [2, 1, [1, 0]], [2, 1, [0, 1]], [2, 2, [0, 1]] and [2, 2, [1, 0]] respectively. On solving each task, the agent is given a reward of 1. In all other cases, it receives no reward. After execution of each program, the probability is updated according to the incremental learning strategy. Learning rate α is chosen as 0.5. Fig. 5.5b shows the plot of inverse of program probability of solutions found for different tasks.

In experiment 3 the initial state is set to [1, 1, [1, 0]]. The agent is trained multiple times with various goal states. Thereafter two related sets of tasks are given, and agent is required to find a single solution for them. Table 5.1 shows the training sequence and the program probability of the solutions found. The iteration column refers to the number of times the metasearcher was run with the same initial and goal state. The initial probability refers to program probability of the solution at the start of the first iteration in a task and the final probability refers to the program probability of the solution at the end of the final iteration in a task. Table 5.2 shows the task set for testing the agent. Two subsequent goal states are given and the agent is required to find a solution for the first goal state and then a solution for both the goal states. The unbiased program probability refers to the program probability of the solution in absence of any incremental learning. Clearly, the training sequence helped in increasing the initial and final probability of the solutions for the test tasks. The constant C is set to 0.05.

Table 5.1 Training sequence for the metasearcher

Training#	Initial state	Goal state	Initial program probability	Final program probability
1	[1,1,[1,0]]	[2,1,[1,0]]	0.0666666	0.2222222
2	[1,1,[1,0]]	[10,1,[1,0]]	0.00051440	0.0010362
3	[1,1,[1,0]]	[2,2,[0,1]]	0.00023386	0.0008527
4	[1,1,[1,0]]	[2,2,[1,0]]	2.84E-05	0.0166102

Table 5.2 Test sequence for the metasearcher.

Test#	Initial state	Goal state	Initial program probability	Final program probability	unbiased program probability
1	[1,1,[1,0]]	[10,10,[1,0]]	5.83E-05	6.13E-05	1.1429891305E-08
2	[1,1,[1,0]]	[20,20,[1,0]]	6.13E-05	6.43E-05	1.1429891305E-08

In the second set of experiments, we compared our Universal search based agent (US-agent) against Deep-Recurrent-Q-network (DRQN) [79], Monte-Carlo algorithm for online planning in large POMDPs (POMCP) [80], and Monte-Carlo AIXI (MC-AIXI) [81]. Each of the three prior arts represents three distinct approaches to attack the problem of solving POMDP environments and obtained some of the best results. Table 5.3. shows the comparison results against each method in different POMDP environments. Both the vanilla metasearcher and the GP based metasearcher [82] have been experimented and compared with the prior arts. The training and test time denotes the total number of actions taken by the agent in the environment during training and test phase respectively. Since each of the

Table 5.3 Comparison of US-agent with current state of art methods

Method	Environment	Training time	Test reward	Performance score	Hyperparameters
DRQN	Gridworld	840000	2.01	23.92857	out_size=5; time_step=8; max_steps=70; memory_size=2000; total_episodes=10000;
POMCP	Gridworld	1822496	1.299	7.127587	horizon=20; C=10; max_particles=700; reinvigorated_particles_ratio=0.05; simulation_time=5 sec
MC-AIXI	Gridworld	547700	1.04	18.98849	exploration=0.999; explore-decay=.9999; agent-horizon=8; ct-depth=96; mc-simulations=200; learning-period=5000; terminate-age=5200
Vanilla US-Agent	Gridworld	7925	4.85	6119.873	C=0.005, alpha=0.6
GP US-Agent	Gridworld	**1484**	**4.82**		C=0.005, alpha=0.6
DRQN	Tic-tac-toe	25176	44.05	17496.82	out_size=9; time_step=3; max_steps=5; memory_size=1000; total_episodes=5000;
POMCP	Tic-tac-toe	603680	43	712.2979	horizon=2; C =10; max_particles=700; reinvigorated_particles_ratio=0.05; simulation_time=0.1 sec
MC-AIXI	Tic-tac-toe	454800	-3.8	-83.5532	exploration=0.9999; explore-decay=0.999999; agent-horizon=9; ct-depth=32; mc-simulations=200; learning-period=10000; terminate-age=10200
Vanilla US-agent	Tic-tac-toe	3589	37.65	104903.8	C=0.01; alpha=0.6
GP US-Agent	Tic-tac-toe	**1726**	**42**		C=0.005, alpha=0.6
POMCP	Tiger	431292	41	950.6320	horizon=2; C =10; max_particles=700; reinvigorated_particles_ratio=0.05; simulation_time=0.1 sec
MC-AIXI	Tiger	1286200	0	0	exploration=0.99; explore-decay=.9999; agent-horizon=5; ct-depth=30; mc-simulations=200; learning-period=5000; terminate-age=5200
Vanilla US-agent	Tiger	13176	67	50850.03	C=0.01, alpha=0.6

methods majorly differ in their approach thus, to have a fair comparison the above-mentioned metric is chosen. DRQN and US-agent have distinct training phases where the agent is being trained for a specific time to generate the trained neural net and search graph respectively. Thereafter the agent is tested in the same environment for 200 actions. For US-agent, heuristic playout is being used. For POMCP, the simulation phase is considered as training, and the action taken in the original environment is considered as testing. For MC-AIXI initially, the agent is given a specific training time which includes actual action taken on the

environment and action search in the context tree. Thereafter the agent is tested in the environment for 200 actions. The following section provides a brief description of each problem environment and configurations of each method.

Gridworld: The structure of the gridworld environment is same as that of the maze environment described above (Fig. 5.4). However, the agent cannot change direction, but it can directly move in all four directions. The reward distribution is [2,1] →0.99, [2,2] → 0.99, [12,1] → 0.99, [10,10] → 0.99, [20,20] →0.99. The agent is allowed to move across all the cells but a cell containing 0 will incur a punishment of -0.01. The agent is not allowed to leave the maze. Each reward can be consumed only once during an episode (until a reset). After a reset, the agent's initial position is set to [1,1]. This reward distribution creates an environment with delayed reward. But there exists a simple optimal solution policy. The agent can only get a local observation in the grid. In our experiments, all the methods except US-agent failed to collect all the rewards in the environment during test phase. Fig. 5.6 shows section of search graph executed by heuristic playout during the test phase.

Tic-Tac-Toe: This is a conventional Tic-Tac-Toe game where a player wins if it can place same consecutive marks in a 3X3 grid. Each win generates a reward 1 and a loss incurs punishment -1. An invalid move like placing a mark on a filled cell incurs a punishment -0.05 and a no-win situation provides a reward 0.5. After generating a reward or punishment the game is reset. The agent is given an encoded observation of empty and filled cells and it cannot exactly know the position of its own marks and the opponent's marks. This creates a POMDP environment. The opponent places a mark randomly on any of the open cells.

Tiger: The player is allowed to open any one door out of 2. There is a tiger behind one door and gold behind another. A tiger behind an opened door incurs a punishment -1 and gold generates a reward 1. The agent can also choose to listen, which will allow it to know the position of the tiger with 85% accuracy. After each reward or punishment game is reset.

Table 5.3. compares different agents' performance in terms of reward gained and number of training steps, in different environments. A performance score has been derived as $\frac{\text{reward x } 10^7}{training\ steps}$. US-agents outperforms all agents in all environments based on this measure.

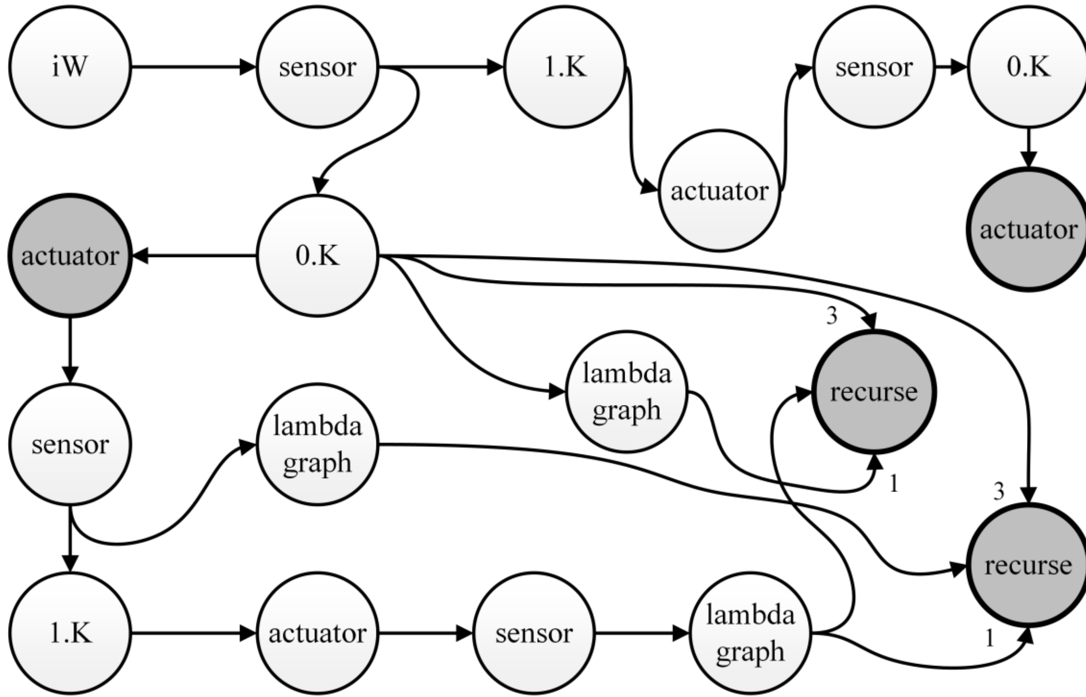


Figure 5.6 A subgraph from search graph for gridworld problem with 4 nodes (marked in gray) having total reward >0 in training phase. Out of 4 nodes, 3 are terminal nodes. Each gray node represents a separate program graph for which the respective gray node is a terminal node and iW is initial node. This search graph was executed by the heuristic playout algorithm. For this environment, allowed actuator input values are 0-3. 0 - move down, 1 - move right, 2 – move left, 3 – move up.

5.1 CONCLUSION

An agent developed based on universal search using dataflow graph based programming model with equivalent program pruning and incremental learning is able to successfully dampen the exponential factor in universal search to a large extent. We tested our agent in a large POMDP maze environment to establish the theoretical claims. The program pruning strategy improved the runtime of the search with an average improvement growth rate of around 1500 per phase. The incremental learning strategy improved the program probability of the solution program around 50 times for the final goal state in the test scenario which suggests there is a 50 times improvement in search time. The agent was also able to find an optimal solution path in the large maze even though there was delayed reward in the path. The agent is also tested against current state-of-the-art methods on some benchmark problems. Our agent outperformed all other methods in terms of total reward collected with limited training time. It outperformed the best competitor method by 250 times in gridworld,

5 times in Tic-tac-toe, and 50 times in Tiger problem, based on a performance score.

However, US-agent can be improved further. Introspection capability can be added in the programming model which would enable self-generation, self-modification, and self-evaluation of code. This would allow the metasearcher to be a part of the program corpus in the search graph, thus allowing the possibility of evolution and automatic improvement of the searcher program itself. The metasearcher program can also be added as an external function in the environment. This would allow searching through a new program space within another search and consequently allow hierarchical problem-solving. However, performance benefits need to be investigated in such scenarios.

6 APPLICATIONS OF THE UNIVERSAL SOLVER

“The key to artificial intelligence has always been the representation” – Jeff Hawkins, (2012)

The Universal solver is capable of solving wide variety of problems in RL setting. The possibility for the solver applications is numerous. It can be applied in any problem environment which provides responses in the form of rewards or punishment. It can be applied in robotics, where a bot might need to learn some sequence of optimal actions in a problem environment, like learning to walk or balancing a cart pole. It can be used to create ambient intelligent systems, where the agent needs to control its transducers optimally in a problem environment, like a smart home environment or patient care environment. As the integrative AI platform already provides an interface to integrate multiple AI components and IoT devices, deploying the solver agent in the platform would enable automatic searching of integrative AI solutions and it would be particularly useful for creating ambient intelligent systems.

In order to demonstrate its usability, we experimented in two problem domains. Namely, solving simple algebraic equations and neural architecture search. In both cases the solver was able to find satisfactory solutions in realistic time.

6.1 LEARNING TO SOLVE SINGLE VARIABLE LINEAR EQUATIONS

The problem solver was applied to solve simple algebraic linear equations [83]. The agent was given three different sets of single-variable linear equations in a sequence and it returned three different program graphs for each set, solving every equation in that set. We encoded the problem specification and the goal state in the environment. The agent can communicate with the environment with the transducer nodes. The problem environment returns the initial equation in a list, where each symbol in the equation is an element of character type in the list. For example, the list elements of the equation x + 15 = 20 will be ‘x’, ‘+’, ‘15’, ‘=’, ‘20’. The first set contain the equations x + 5 = 15, x + 80 = 100, x + 6 = 3. The agent is required to find a solution program that solves all the above equations. Solution program should return a list containing the elements in the form +x = +a or +x = -a. The second set contain the

equations x*2 + 5 = 15, x*8 + 20 = 100, x*3 + 6 = 3. The agent started with the search graph generated by the earlier search process. The second set of equations are little bit complex compared to the first, as they have a multiplicative term associated with the unknown variable. The third set contain the equations x = 5+15, x = 20 +100, x = -2 +3.

The experimental results show the efficiency of transfer learning and the agent was able to quickly find general solutions for solving simple equations. Following is the list of primitives used for the search. Namely, identity (i), constant (K), nil (n), cons (c), head (h), tail (t), reverse (r), split list (s), join list (j), fmap (F), aggregator (ag), apply (a), loop (L), goal checker (G), sensor (S). The notation for constant node is given as a.K, where a represents the constant value and a can be data as well as a function. Apart from the primitives, few custom functions are used in the form of external services. Table 6.1 lists the external services used. The external services are called using *microcall* gp node as demonstrated in Chapter 4. Separate function nodes are built for each service using the *microcall* function and constant nodes. The custom-built function nodes are directly used in the search and will be referred with the function symbol as specified in Table 6.1, in all future reference.

Table 6.1 List of external functions used in the search

Function symbol (Function Name)	input types	Output type	Operation
I (invert operator)	char	char	Takes the mathematical operators as input and inverts it. Allowed input characters are: '+','-','*','/'. '+' is inverted to '-' and '*' is inverted to '/' and vice versa
E (evaluate expression)	List(char)	List(char)	Simplifies a mathematical expression within a list. If first element of the simplified expression is not a mathematical symbol, then '+' symbol is added as the first element. Example: For Input expression x + 5 +2 it will return + x + 7
LE (Linear equation)	None	List(char)	Represents the problem environment. Returns the linear equation to be solved in form of list of characters.

6.1.1 Composite node functions

Primitive functions can be composed in arbitrary ways, satisfying type compatibility to create arbitrary composite functions. We will create such composite functions and store it as available library functions to be used by the searcher. These functions are specifically created to aid the searcher program in finding solutions for algebraic equations. This can be considered as pushing some domain knowledge into the searcher process in the sense

mentioned by Schaul & Schmidhuber [84].

Append (A). Append is a composite node function. It appends the first argument to the list object received as the second argument. If the first argument is a list type, then it joins the two lists by appending elements of the first list after the second.

Cut-Invert (CI). The function CI cuts the last two elements of a list and inverts the mathematical operator present at the second last position. Thereafter it returns the two elements in a list in the same order. It takes an input of type list and returns a list. Fig 6.1 illustrates the program graph of the CI function. In all diagrammatic representations, the input port order of a node is in clockwise direction, starting from the output port.

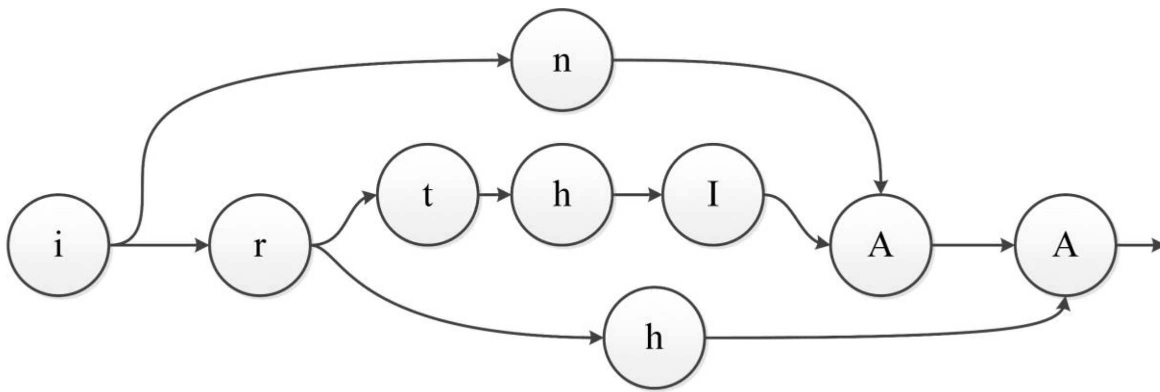


Figure 6.1 Illustration of the program graph of composite function CI

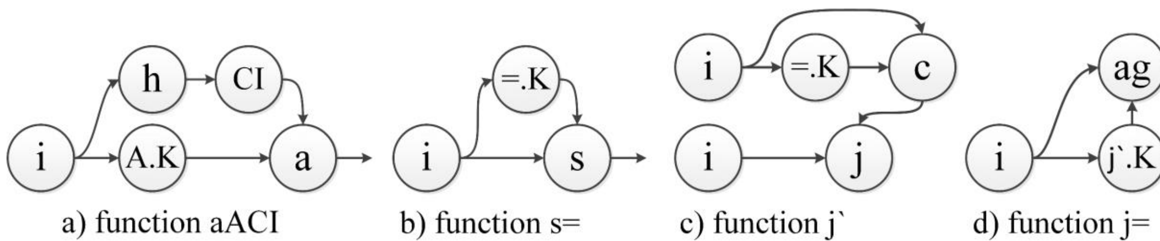


Figure 6.2 Program graphs of composite functions. a) program graph of function aACI. b) program graph of function s=. c) program graph of function j`, d) program graph of function j=.

Partial append of Cut-Invert (aACI). The function aACI takes a list of lists as input. It applies the CI function on the first element of the input list and thereafter partially applies the append function on the result. The partially evaluated function is returned as output. Fig. 6.2a illustrates the program graph of this function.

Split on equality symbol (s=). The function s= takes a list of characters as input. It applies the function s on the input list and splits the list based on "=" symbol as an element. Fig. 6.2b illustrates the program graph of this function.

Join on equality symbol (j=). The function j= takes a list of lists as input. It applies the j function on the input list and joins the lists within the list with "=" symbol as an element added in between. Fig. 6.2d illustrates the program graph of this function. It uses the helper function j` as illustrated in Fig. 6.2c.

6.1.2 Generating solutions

The vanilla metasearcher has been run to generate solutions for this problem. Following set of function nodes are used as an initial corpus to generate and search in program space. Namely, LE, E.K, s=, aACI, F, j=, L, r, 2.K. Fig. 6.3 shows the solution program graph found

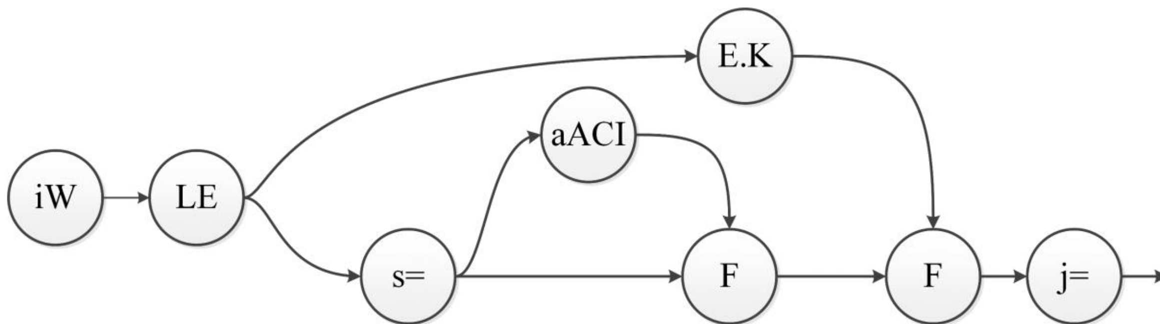


Figure 6.3 Solution program graph found by the agent for solving equation set 1

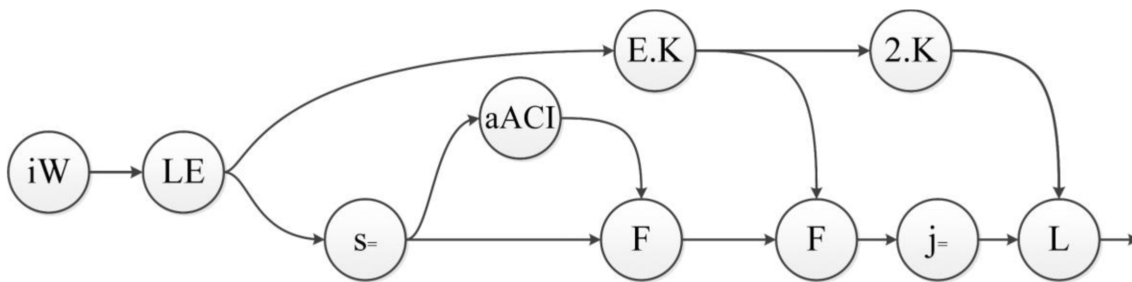


Figure 6.4 Solution program graph found by the agent for solving equation set 2.

by the agent for solving the first set of equations. The output returns the value evaluated for the variable *x* in the form of a list.

Fig. 6.4 illustrates the solution program found by the agent which solves all the equations in the second set. The second solution reuses the program graph found as the first solution. It

just executes the solution program for the 1st equation set twice, using a loop, to deduce the solution graph for the second set.

Fig. 6.5 illustrates the solution program found by the agent which solves all the equations in the third set. This program graph reuses a section of the solution program for the first set. However, it is a much simpler graph compared to the 1st program graph.

The program probability of solutions for later tasks increased due to incremental learning and thereby reduced the search time. The solutions for subsequent problems reused a section of previous solution graphs due to the presence of common subproblems between the tasks. For example, the third equation set is just a subproblem of the first and second equation sets.

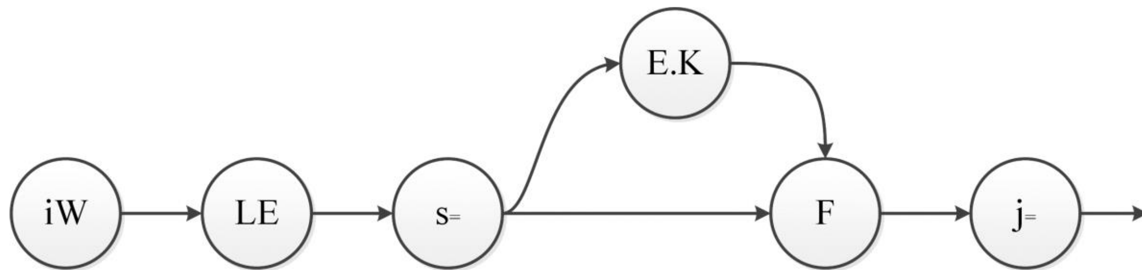


Figure 6.5 Solution program graph found by the agent for solving equation set 3.

The program graph in Fig. 6.3 is able to implicitly segregate these dependent or independent subproblems due to program representation as graph. The independencies among different functional nodes allow dividing the solution graph into several independent subgraphs, each solving a specific subtask. These subgraphs can be directly re-used for related tasks as it happened for the solution of equation set 3 in Fig. 6.5. This makes the transfer learning process efficient in program representation as data flow graphs.

6.2 NEURAL ARCHITECTURE SEARCH USING UNIVERSAL SOLVER

The performance and accuracy of a deep neural network in a problem context depend on the architecture of the network. The architecture of a deep neural network can be characterized by the number of layers in the network, type, and configurations of each layer, skip connections, etc. Designing such a network manually for a problem context is a labor-intensive and unintuitive task and there is no well-defined formula that states an ideal way

of designing a neural network for any arbitrary problem. Even with hours of effort put in by experts, in designing the network and hyper-parameter tuning – tons of potentially promising architectures remain undiscovered. Neural architecture search or NAS attempts to search for an optimal neural network architecture for a problem context, in a systematic, principled and automated way. NAS can be categorized into the following 3 dimensions [85].

Search space: Search space denotes the possible set of neural architectures that can be explored and searched through. Search space can be bounded by incorporating prior knowledge and determining a fixed set of possible configurations that would be tried while generating the architectures. This would reduce the search space and thus simplify the search. However, it would also restrict the search process from trying arbitrary novel architectures.

Search strategy: Search strategy specifies the strategy of the search process. It deals with the typical exploration-exploitation problem as the search space is usually of exponential size. It is desirable to quickly find an optimal architecture exploiting previously learned well-performing architectures. However, it is also essential to explore and try new architectures to find even better-performing architectures.

Performance estimation strategy: Performance estimation of an architecture can be done by measuring the predictive capability on unseen data. Typical measures for estimating performance can be accuracy or error on training and validation dataset.

6.2.1 Search space

The search space in NAS can be setup in multiple ways.

Sequential networks - One way of depicting/designing the search space for neural network architectures is to depict the network topologies as a list of sequential layer-wise operations [86, 87]. The serialization of network representation requires a decent amount of expert knowledge since associations between operations and layer specific parameters and constraints need to be explicitly specified. Networks are designed by chaining one layer after another. The full network can be specified by the number of layers, the type of operations in each layer, and hyperparameters associated with each layer.

Multibranch networks – Skip connections in a network allow creating complex multibranch architectures. Each layer may receive inputs from any number of arbitrary previous layers. Residual networks [88], dense networks [89], and sequential networks are all special cases of multibranch architectures. This type of efficient architecture has been generated by NAS successfully but at the cost of high computation power [90].

Cell-based architectures - These kinds of search spaces are perhaps motivated by the observation that many effective handcrafted architectures were designed with repetitions of fixed structures. Example: ResNet, Inception. The NASNet search space, as designed by Zoph et al. 2018 [90], defines the architecture of a conv net as the same cell getting repeated multiple times and each cell containing several operations as designed by the NAS algorithm. A well-designed cell module enables transferability between datasets. It is also easy to scale down or up the model size by adjusting the number of cell repeats. The NASNet search space is defined to be composed of two types of cells:

a. Normal Cell: Input and output feature maps have the same dimension.

b. Reduction Cell: Output feature maps have their width and height reduced by half. Such a strategy reduced the search space drastically as each cell contains significantly a smaller number of layers compared to the whole architecture.

6.2.2 Search algorithm

There are multiple approaches for attacking the search problem which includes Bayesian optimization, evolutionary methods, and reinforcement learning. Bergstra et. al. used Bayesian optimization to derive state of art neural architectures for vision problems [91]. Zoph et. al found competitive architectures for solving CIFAR-10 classification problem and Penn Treebanks [92]. In a reinforcement learning setting the actions are the individual network generation operators and the reward is the estimation of the performance of a generated network. For example, actions can be an operator to add a convolutional layer with certain hyperparameters or adding a skip connection. The objective is to perform a search in the action space and come up with an optimal sequence.

6.2.3 Evaluation strategy

There exist several different ways of evaluating and estimating child model performance in order to provide feedback signals to the searcher algorithm in order to optimize it. The searcher algorithm uses the evaluation scores to guide the search process and dampen the search space. This process may turn out to be prohibitively expensive and many new evaluation methods have been proposed to save time or computation. Two common methods are discussed as follows.

Training from scratch - This is the method utilized by Zoph et al. [92] wherein they trained each child model from scratch until convergence and thereafter evaluated their performance on a hold-out set. This method has the benefit of providing the most reliable performance estimates but it tends to be prohibitively slow. One complete train-converge-evaluate loop only generates a single data sample for training the RL controller.

Proxy task performance - This method involves utilizing smaller, less computationally expensive tasks as a proxy for model performance. Some widely used ones are listed as follows.

- Training on a smaller but representative dataset.

- Training for fewer number of epochs

6.2.4 Metasearcher as search method for NAS

Metasearcher can be used to find optimal neural architectures for specific problems under proper environmental settings. The search strategy in the metasearcher is based on universal search accompanied with policy gradient based incremental learning. The search space is the program space where programs are constructed using primitives of FGPM. When executed, the programs in FGPM directly translate to neural architectures. This section explains a method of constructing the problem environment which can be plugged into the metasearcher for performing NAS. However, this is just one way of representing the problem environment out of many other ways.

Fig. 6.6 illustrates an interaction between the metasearcher (agent) and the environment. The

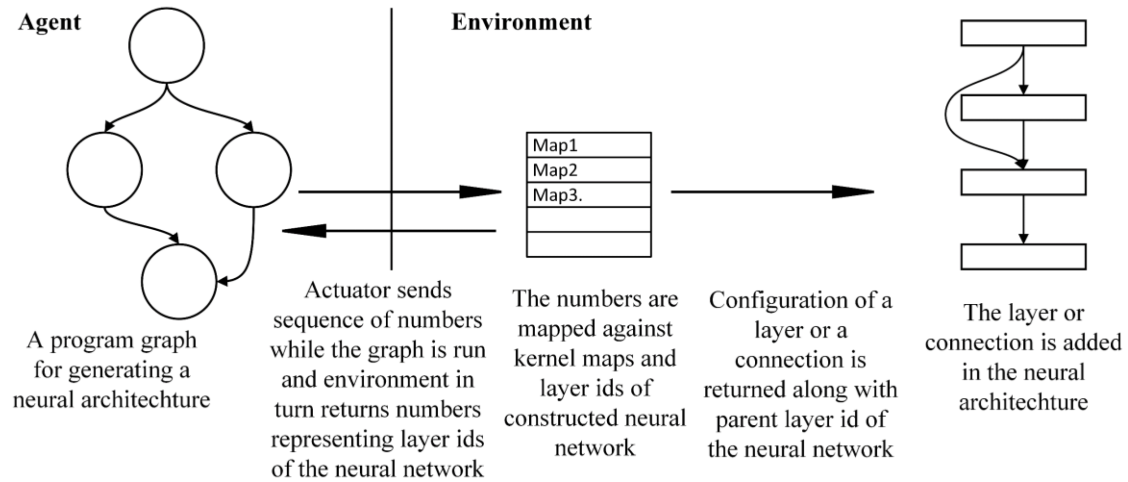


Figure 6.6 Illustration of interaction between the metasearcher and the environment for neural architecture search

metasearcher will construct arbitrary programs and test them in the context of the problem environment. An environment expects a sequence of numbers as actuation signals from the program being run. The numbers represent code for specific configurations of layers of a neural network. The environment maintains a kernel map that translates the numeric code to a specific layer configuration. Table 6.2 shows the list of kernel maps used for experimenting with the CIFAR-10 dataset. Once a specific code is received as an actuation signal, the corresponding layer is added after the previous layer. Every layer in a constructed network is identified by a unique number, called as layer id. After addition of a new layer, the

Table 6.2 Kernel Map used for Neural architecture search for CIFAR-10 dataset

code	Layer operation	Configurations
1	Convolution 2D	No. of filters = 64; kernel size = 3; activation = elu; regularizer = l2; padding = same; BatchNormalization = True
2	Maxpooling	kernel size = 3; padding = same
3	Concatenation	NA
4	Convolution 2D	No. of filters = 128; kernel size = 3; activation = elu; regularizer = l2; padding = same; BatchNormalization = True
5	Addition	NA
6	dropout	Dropout rate = 0.25
7	Convolution 2D	No. of filters = 128; kernel size = 5; activation = elu; regularizer = l2; padding = same; BatchNormalization = True
8	Convolution 2D	No. of filters = 256; kernel size = 3; activation = elu; regularizer = l2; padding = same; BatchNormalization = True
9	Average pooling	kernel size = 3; padding = same

environment returns the layer id of the added layer. Concatenation and addition are two operations that merges two different pathways in a network. Thus, for these two operations, the environment expects the layer id of one of the parent layers as the next number in the sequence. It merges that parent layer with the immediate previous layer. A minor modification was made in the metasearcher to add a special composite node (built with gp node) during the test phase of every constructed program. The node is added as a terminal node and it sends a signal to the environment to notify the end of the program. The environment in turn flattens the output of the last layer in the constructed network and adds a dense layer with 10 outputs for classifying CIFAR-10 dataset. It uses softmax activation for the dense layer. Finally, the neural network model is trained and validation accuracy is returned as a reward signal. The training was run for relatively a smaller number of epochs (10 epochs) due to high computational complexity of training a neural network. Since this is a generate and test method of searching the best architecture, an indicative relative performance measure is sufficient to get a comparative score for multiple architectures. Any invalid sequence of numbers, like numeric code not present in kernel map or invalid parent layer id, raises an error and the environment sends an error signal. After training a network and returning the reward signal, the environment is reset such that a new program can construct another neural network from scratch.

6.2.5 Experimental results for CIFAR-10 dataset

The metasearcher and the NAS problem environment were implemented in python. Keras library has been used to construct the deep neural networks in the environment. CIFAR-10 data has been obtained from [93]. The search process was run in Google Collaboratory with a fixed Phase limit and it ran for around 6 hours. The search process was running using the following set of primitives. Namely, initWorld, constant1, constant2, sensor, actuator, addition, constant3, constant4, constant5, loop. The top two models in terms of validation accuracy are selected after the completion of the search. The two models were further retrained for 500 epochs. The final accuracy of the top two models were 88.72% and 89.93% respectively. Fig 6.7 and Fig. 6.8 illustrate the top two neural network architectures found by the metasearcher. Though the accuracy of the top architecture is behind that of the current state-of-the-art method (~98%), yet we can supposedly say that the proposed method gave a

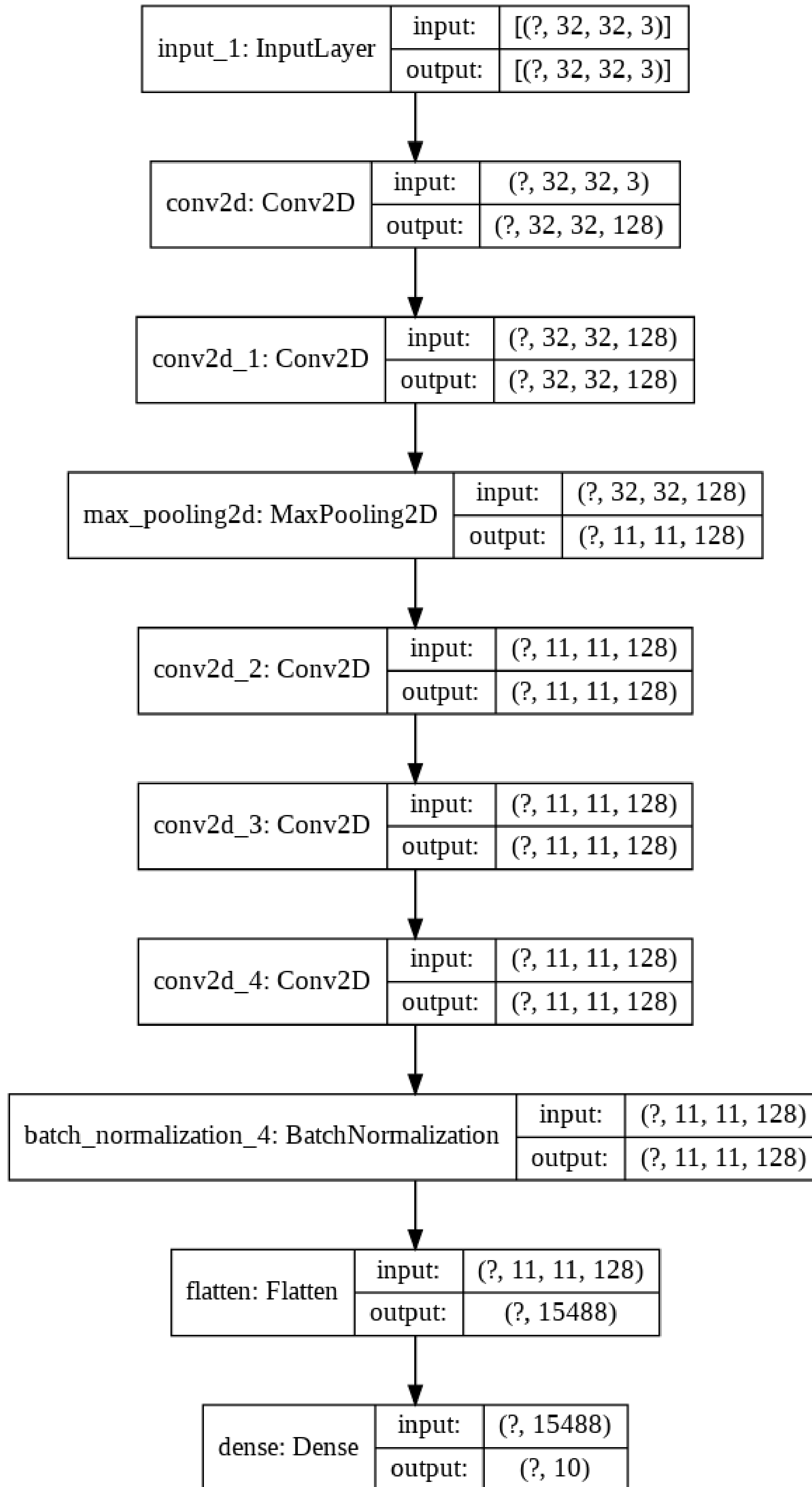


Figure 6.7 Neural network architecture found by the metasearcher with a validation accuracy of 89.93%

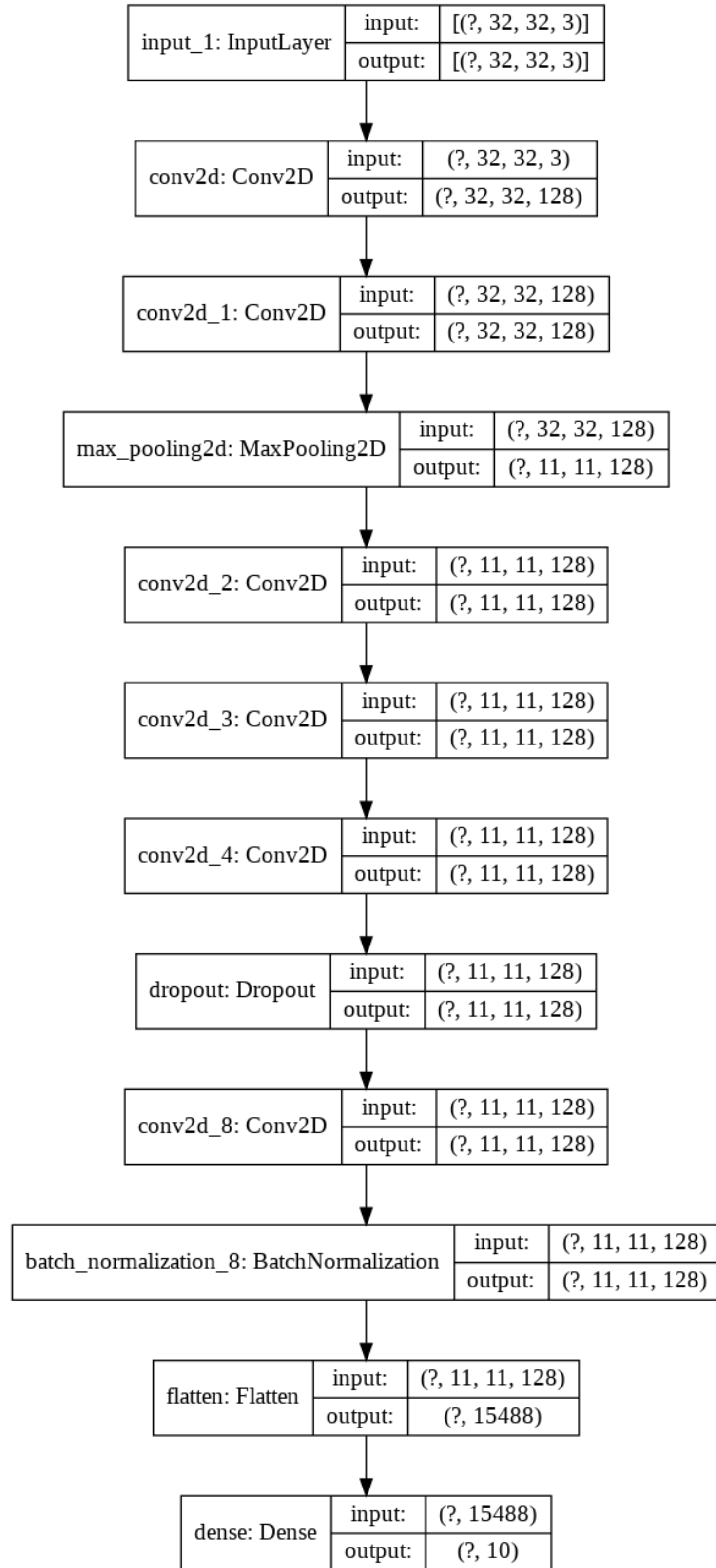


Figure 6.8 Neural network architecture found by the metasearcher with a validation accuracy of 88.72%

satisfactory solution or rather a workable solution considering the computation power it consumed. We also focused on generating only simple sequential architectures. Generating more complex architectures like adding skip connections, would take more computation power but may find better architectures in terms of accuracy.

6.2.6 NAS for COVID-19 spatiotemporal forecasting model using universal solver

Wuhan city in China initially observed an outbreak of Covid-19 disease caused by SARS-CoV-2. Eventually, it became a pandemic and more than 200 countries are fighting hard to contain the infection [94]. This has become a unique challenge for mankind as the competition has turned out to be against time due to the exponential rate of infection spread. One of the best ways to contain the infection is rapid identification of positive cases and isolation. However, due to limited resources, random and widespread testing may not be feasible in populous countries. Forecasting regional spread can help identify future hotspots and distribution of infection which would eventually help to take containment measures.

A spatiotemporal epidemic spread model can accommodate both spatial and temporal correlations in data. However, most of the models either require disease specific domain knowledge [95] or are too spatially coarse [96]. Deep learning models can learn the dynamics of epidemic spread with high spatial resolution and high degree of accuracy with minimal initial bias, due to its capability of highly nonlinear representation. Deep neural network based spatiotemporal models [97] have already been applied to predict epidemic spread. However, this model is experimented on a small localized region and the influence of external factors was ignored. Deep learning models also tend to overfit due to their high representational capability. Thus, modeling of Covid-19 spread in a wide region with high spatial and temporal resolution is challenging.

To address the problem of spatiotemporal prediction of Covid-19 spread in a large geographical region with high resolution, a Convolutional LSTM [102] based model was trained with multilayer geospatial-temporal data, transformed as a sequence of images [98, 99]. Each layer of the geospatial data corresponds to a causal factor that might influence the spread of the epidemic. The data preparation method creates geospatial images of features based on latitudes and longitudes thus avoiding the need for location specific adjacency

matrix. However, one of the major challenges is finding an optimum Conv-LSTM neural architecture for the spatio-temporal model.

6.2.6.1 Feature construction

Covid19 daily data at USA county level are filtered by a spatial region of USA as shown in Fig. 6.9c. The region is geospatially divided into M x N grids of equal sizes, bounded by calculated latitudes and longitudes. Fig. 6.9a illustrates a grid bounded by latitudes and longitudes. The dotted-line square is called as a frame. The overlapping areas in all 4 directions in a frame allows the flow of spatial influence from neighboring grids. A frame is in turn divided into L x L pixels. Each pixel represents a bounded area in the geospatial region. Each pixel contains a value mapped to a certain feature in the bounded geospatial region. Frame matrices are constructed for each feature and concatenated through a third axis called channels. For example, transmission rate (β_i) and population density are two features and they represent two separate L x L matrices in a frame concatenated across a third axis.

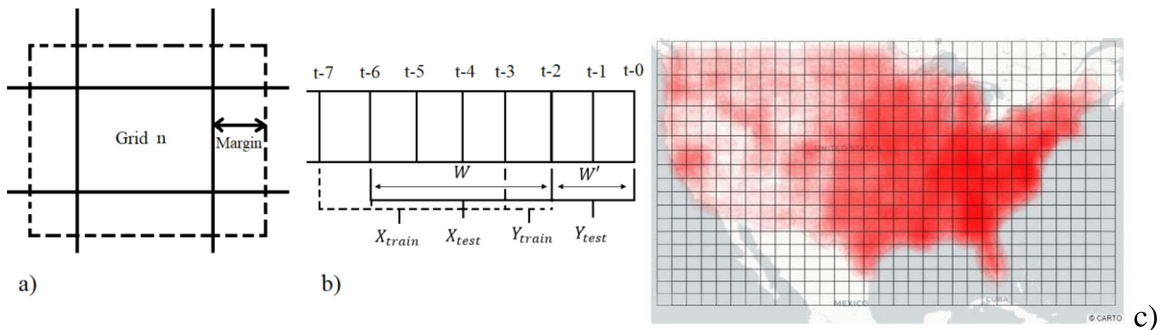


Figure 6.9 a) Illustration of overlapping frames obtained by spatially dividing a geographical region. The bold lines represent latitudes and longitudes which separate the grids. The box with dotted lines represents the frame with overlapping grids that are used for training the model. Each grid is divided LxL pixels. The margin refers to the number of pixels of overlapping regions. b) Illustration of sequence of a frame. t-0 is the most recent frame. Xtrain, Ytrain are the training samples, and Xtest, Ytest are testing samples. c) A region of USA divided into 18x30 grids. The red bubbles denote the cumulative number of Covid-19 cases.

Some features like transmission rate, active infection fraction, weather etc. are distributed spatio-temporally. Whereas other features like population density, female fraction, median age are assumed time-invariant and have no temporal component. Thus, they are only distributed spatially and copied along the temporal axis. Population density has been log transformed to reduce skewness and normalized. Other features are only normalized on 0-1 scale. Daily transmission rate and removal rates at pixel level have been calculated as follows, where $i \in \{1..n\}$ denotes each pixel, $\Delta I_i^+(t)$ and $\Delta R_i(t)$ are fraction of new cases

in infected class and new individuals in removed class respectively, at time t in pixel i. N_i represents population in pixel i. $S_i(t)$ and $I_i(t)$ are fraction of susceptible and infected respectively, at time t in pixel i. $\delta_i(t)$ is the removal rate.

$$\beta_i(t) = \Delta I_i^+(t) I_i(t) / [S_i(t-1) I_i(t-1) + N_i^{-2}]$$

$$\delta_i(t) = \Delta R_i(t) / [I_i(t-1) + N_i^{-1}] \tag{6.1}$$

Each training sample of a frame is represented by a tensor of dimension $T\, x\, L\, x\, L\, x\, C$, where T is the total time span and C is number of channels or features. As shown in Fig. 6.9b each training sample is generated by sliding a time window of size $W+1$, by 1, leaving behind a test case sample of time window size W' in the most recent period. The number of training samples for a frame can be calculated as $T - W' - W - 1$. Thus, the total number of training samples S_{train} for all frames can be calculated as $S_{train} = (T - W' - W - 1) * M * N$.

The forecasting problem is framed as a supervised learning problem. Given a sequence of observed multichannel frames of spatial data as matrices $X_1, X_2 \ldots X_t$, the objective of the model is to predict the next single channel frame Y_{t+1}. The training samples are divided into input sequences of length W and output frames. The model forecasts the transmission rate in each pixel in a frame for each timestep. Thus, the output frame consists of only 1 channel. The input training dataset (X_{train}) can be represented as a tensor of size $S_{train}\, x\, W\, x\, L\, x\, L\, x\, C$ and the output dataset (Y_{train}) as $S_{train}\, x\, W\, x\, L\, x\, L\, x\, 1$. For training, the input sequences are selected from all frames having a non-zero total infection count. Fig. 6.9b illustrates the sequence of a frame. The frames *t-7* to *t-3* represents an input training sequence (X_{train}) of length W. The output frame (Y_{train}) for this training sample is *t-2*. Other training samples are generated by sliding the window $W+1$ backward in time by 1. The most recent images *t-0* and *t-1* represent the test output images (Y_{test}) and immediate sequence of images *t-6* to *t-2* is the test input sample (X_{test}). The test set X_{test} is represented by a tensor of size $(M * N)\, x\, W\, x\, L\, x\, L\, x\, C$ and Y_{test} by $(M * N)\, x\, W'\, x\, L\, x\, L\, x\, 1$.

6.2.6.2 Conv-LSTM model of transmission rate

Recurrent neural networks (RNN) are a class of artificial neural networks with nodes having feedback connections, thereby allowing them to learn patterns in variable length temporal

sequences. A simple RNN has a feedback loop that is associated with hidden state weights. Fig. 6.10 illustrates an RNN unfolded in time. It has hidden states *h(t)* which change with time. The inputs and outputs are represented by *x(t)* and *y(t)* respectively. The dependency on historical values of the sequence is captured by the relationship between the hidden states. *Wh, Wx, Wy* are the weights that are learned through backpropagation during training. However, it becomes difficult to learn long-term dependencies for traditional RNN due to the vanishing gradient problem [100]. LSTMs [101] solve the problem of learning long-term dependencies by introducing a specialized memory cell as a recurrent unit. The cell can selectively remember and forget long-term information in its cell state through some control gates. We presume there might be long-term complex dependencies of several factors on the transmission rate. Thus, LSTM seems to be a preferred choice for time series modeling. In convolutional LSTM [102] a convolution operator is added in state-to-state transition and input to state transition. All inputs, outputs, and hidden states are represented by 3D tensors having 2 spatial dimensions and 1 temporal dimension. This allows the model to capture

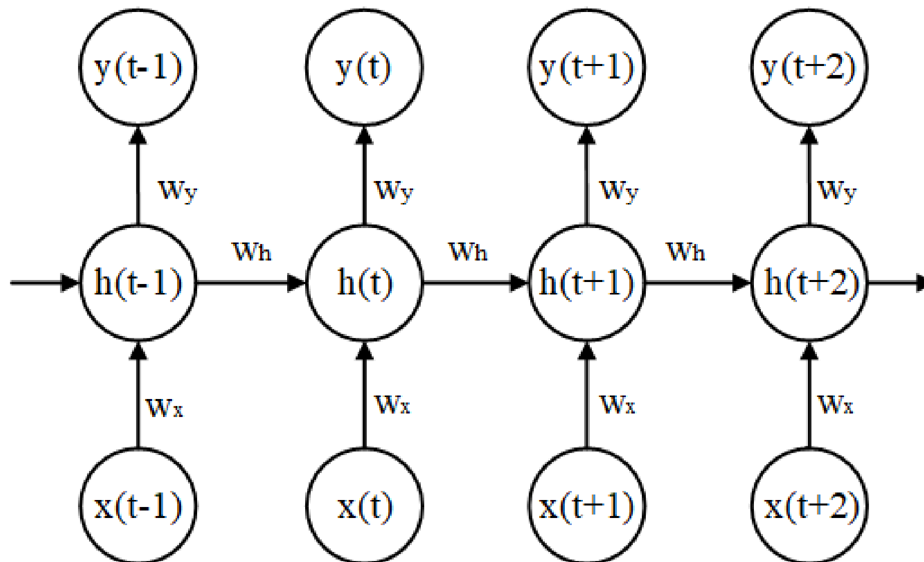


Figure 6.10 A simple RNN unfolded across multiple timesteps.

spatial correlation along with the temporal one. In our model, we configured multichannel input, such that distinct features can be passed through different channels. Multiple convolutional LSTM layers are stacked sequentially to form a network with high nonlinear representation. The final layer is a 2D convolutional layer having one filter which constructs a single channel output image as the next predicted frame.

The model is tested by feeding in an input sequence of frames and the next frame is predicted, which in turn is combined with other features along the channel and appended with the input sequence. The new input sequence is fed to the model again to get the next predicted frame. This continues until forecasting completes for a desired time period. "Mean absolute percent error" (MAPE) and Kullback-Liebler (KL) divergence are used to measure the accuracy of the model. The model predicts the transmission rate for a future time period for each pixel, which in turn is used to calculate daily new infection cases $\Delta I_i^+(t)$. The removal rate is estimated as the running average of previous 3-days and daily removed cases are calculated. The active infection cases ($I_i(t)$) and susceptibles ($S_i(t)$) are also calculated. Cumulative infection cases ($\sum \Delta I_i^+(t)$) are calculated by summing up all new infection cases up to a certain day. MAPE of transmission rate is calculated at pixel level for the prediction period and averaged. The pixels with 0 susceptible population count are filtered out while calculating MAPE and KL divergence. Pixel MAPE is calculated as per equation 6.2, where G is set of all grids and G' is set of all pixels such that the frame for each corresponding grid have non zero cumulative infection count, W' is prediction time period, $W'' = T - W'$ is total time period in training set, $\hat{\beta}_i'(t)$ and $\beta_i'(t)$ are predicted and actual scaled transmission rate for ith pixel at time t respectively.

$$MAPE_{pixel} = \frac{1}{W'|G'|} \sum_{W'} \sum_{\forall i} \frac{|\hat{\beta}_i'(t) - \beta_i'(t)|}{\beta_i'(t)} \mid i \in G' \tag{6.2}$$

KL divergence at pixel level is calculated for modified transmission rate in the prediction period to measure the dissimilarity of distribution of predicted transmission rate with respect to actual. σ is softmax function applied after scaling a series in 0 to 1 scale and $P(X)$ is probability distribution of X. Softmax is applied to convert transmission rate as probability distribution across pixels. Since KL divergence measures the dissimilarity between two distributions thus a lower value of it indicates better performance of the model.

$$D_{KL}^{pixel} = \sum_{\forall i} P\left(\sigma(\hat{\beta}_i'(t))\right) \log\left(\frac{P\left(\sigma(\hat{\beta}_i'(t))\right)}{P\left(\sigma(\beta_i'(t))\right)}\right) \tag{6.3}$$

6.2.6.3 Experimental results

Initially, a spatiotemporal forecasting model has been constructed manually by stacking

multiple layers of ConvLSTM. An optimal architecture has been derived manually after several hit and trials. Thereafter neural architecture search has been run using metasearcher to find an architecture and both the architectures are compared against each other. The manually constructed architecture consists of four ConvLSTM layers stacked sequentially and terminating with a Convolutional 2D layer. The final layer is followed by an exponential linear unit as activation. The input and other hidden Convolutional LSTM layers are followed by sigmoid activation. Each Convolutional LSTM layer has 32 filters and kernel size 3x3. The input layer is configured to take tensors of size 16x16x8. Eight input features are

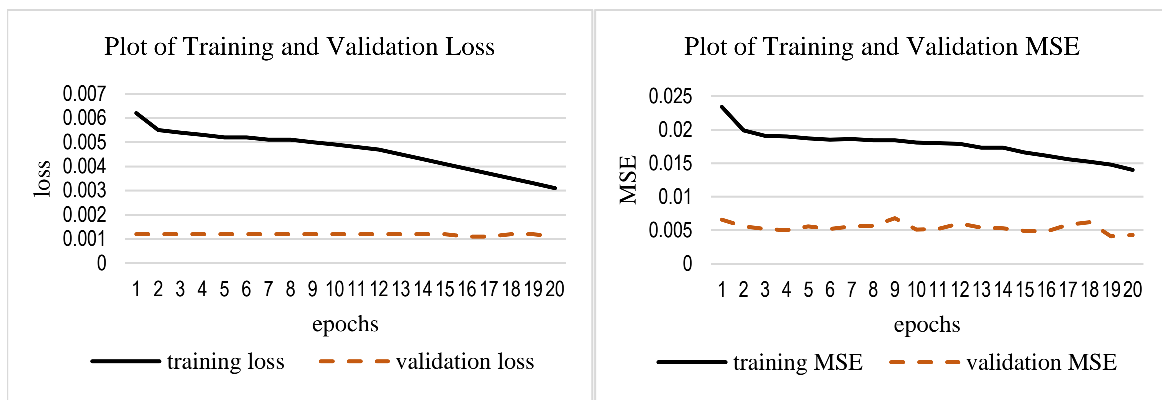


Figure 6.11 Plot of loss vs epoch and mean squared error (MSE) vs epoch for both training and validation data set

constructed and fed into the model as separate channels. Namely transmission rate, population density, female fraction, median age, active infection fraction, average temperature, temperature standard deviation, and average relative humidity. The model is trained for 20 epochs with a batch size of 50 and mean squared error as loss function. Out of 11378 samples, 10809 are used for training the model and 569 are for validation.

The dataset has a time span of 51 days starting from 2020-03-21, out of which data from 42nd to 51st day is used for testing the model and the rest for training and validation. Fig. 6.11 illustrates the plot of training/validation loss and training/validation mean squared error (MSE). Both the loss and MSE consistently decreased for training and validation as epochs increased.

The NAS setup for metasearcher is similar to the previous use case (CIFAR-10). However, for this use case, the kernel map is modified as specified in Table 6.3. Only sequential neural

architectures are constructed and tested. The search process was run in Google Collaboratory with a fixed Phase limit and it ran for around 2 hours. The search process was run using the

Table 6.3 Kernel Map used for Neural architecture search for Covid-19 spatiotemporal model

code	Layer operation	Configurations
0	ConvLSTM	No. of filters = 32; kernel size = 3; activation = sigmoid; padding = same
1	ConvLSTM	No. of filters = 64; kernel size = 3; activation = sigmoid; padding = same
2	ConvLSTM	No. of filters = 32; kernel size = 5; activation = sigmoid; padding = same
3	ConvLSTM	No. of filters = 32; kernel size = 3; activation = elu; padding = same
4	ConvLSTM	No. of filters = 64; kernel size = 3; activation = elu; padding = same
5	ConvLSTM	No. of filters = 32; kernel size = 5; activation = elu; padding = same

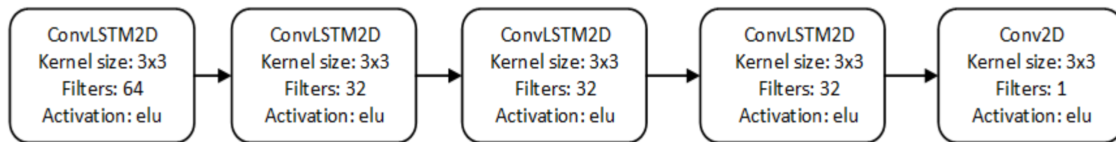


Figure 6.12 Neural architecture found by the metasearcher

following set of primitives. Namely, initWorld, constant1, constant2, constant3, constant4, constant5, constant0, sensor, actuator, loop. The reward signal for each tested architecture is computed as $(1 - \min(\text{mean absolute training error}))/1.5$.

The top model in terms of reward gained is selected after completion of the search. The top model is further retrained for 20 epochs. Fig. 6.12 illustrates the top neural architecture found by the metasearcher. Table 6.4 shows the test results for the manually constructed architecture and the neural architecture found by the metasearcher. Based on the different performance metrics it can be deduced that metasearcher was able to find a near-optimal architecture whose performance is nearly the same as that of the carefully handcrafted optimal architecture.

Table 6.4 Model training, validation and test results

Metric	Manually constructed architecture	Best architecture found through NAS
Training mean absolute error	0.0140	0.0132
Validation mean absolute error	0.0043	0.0044
Pixel KL divergence	8.306x10^-9	8.338x10^-9
Pixel MAPE	7.95%	10.2%
Grid MAPE	8.39%	10.2%
Country MAPE	0.19%	0.6 %
Predicted total cases (1330525)	1331175	1306171

6.3 CONCLUSION

This chapter presents two different problem environments and showcases how the universal solver handles them. The solver was able to generate satisfactory solutions in both the problem environments. Due to immense use of deep neural networks in current times, the problem of Neural Architecture Search is of prime importance. NAS is usually prohibitively complex in terms of the computation power it uses. Most of the current solutions are based on deep neural networks operating in a RL setting. We have showcased how a program search based approach could also generate usable architectures with low computation effort. The solver can be experimented in multiple other configurations of the problem environment for NAS. It can be used to find other types of neural architectures also, like RNN, generative networks, etc.

Though the solver is capable of handling multiple problem environments, yet the search time and space can be quite large for certain problems. Specially, problems which don't have relatively shorter solutions or rather there are little observable patterns in the solutions. However, in such scenarios accurate and frequent reward signals can help to focus the search and reduce the search time. But in a delayed reward environment it becomes quite difficult to converge the search in such cases. The learning mechanism needs to be improved to better exploit the past experiences in the form of historical search graphs when immediate rewards signals are not available.

.

7 TOWARDS CONSTRUCTIVIST SEED AI

"*No matter how tall or wide a tree is, it started off a seed*." - Matshona Dhliwayo

Though there have been several major advancements in the last few decades in domain specific artificial intelligence, yet portability of an intelligent system from one problem environment to another is inefficient and requires manual reconstruction. Real-world problems are barely restricted to a small domain. For example, driving a car requires complex interconnections among visual processing, auditory processing, and motor skills. These complex systems are mostly hand-engineered by following a constructionist approach [50]. The constructionist method of designing an AI has its own limitations like restricted learnability and portability which calls for a new design approach, known as constructivist approach [62]. Constructivist systems are empowered with the automatic construction of solutions in a given arbitrary problem environment with limited resources and little external bias. In order to optimize usage of available resources the system is expected to self-modify and improve its learnability and portability. To create such an artificial intelligent agent, it needs to be bootstrapped with a seed program [19] and intelligent core, which would eventually improve itself and adapt to a variety of environments. Even for the constructivist approach a seed AI is necessary to let the agent know at least the method of construction. We discuss a principled approach of creating a seed program and provided a formal grounding [112]. Questions like, what is the minimum structure required to construct such a seed AI and how it achieves multiple properties of an intelligent system are answered. There are multiple architectures present in the literature [54, 9, 10, 26, 45, 53] to achieve general intelligence, however, any of them barely discusses the abstract minimal structural requirement to achieve different properties of general intelligence namely, constructivism, learnability, recursive self-improvement and adaptability across environments.

A practical implementation of seed AI based on Universal search is also demonstrated. Universal search is a good candidate to design a constructivist framework due to its generate and test approach in program space. It has been shown how the metasearcher structurally resembles the seed AI. The implemented system is demonstrated to solve a heterogenous toy problem environment.

7.1 ABSTRACT MODELLING OF SEED AI

Category theoretical approach has been used for abstract modeling of seed AI. Milewski [103] provides an excellent introduction to category theory. A useful seed AI is one that enables an agent to act optimally in a problem environment and learns to adapt across varied problem environments. It is assumed that all the problem environments in which an agent interacts in its lifetime would be computable. Thus, every problem environment can be represented as strings over some set of characters. Deductively, considering the agent as a part of the environment is also a computable one and all of its parts can be represented as strings. The environment can be considered as a single object monoidal category over a set of primitives in a Turing complete language, and a composition operator. The objects in the environment are programs. Programs can be created by composing primitives and other programs. Thus, the category is equipped with a bi-functor $\sigma_e: E \otimes E \to E$, an identity or empty string I_e, where E represents environment object. In a general sense, there can be infinitely many possible problems in any arbitrary environment, represented by different unique programs e or elements of object E. However, only those problems are practically solvable by an agent which allows the agent to interact with it and halts on every input. On interaction with the agent, a problem e can produce some output such that $e(x) \to y$. The output y can be interpreted by the agent to identify reward signals r. Considering the objective of the agent is to act optimally, so as to maximize future expected total reward, let us assume a set of programs p exists in another Turing complete language that can find optimal solutions for all such problems e.

At least one optimal policy exists for all solvable problem environments

Proof: Let us consider a cybernetic agent which runs a program p in a Turing complete language, identified as policy for problem e. The agent is coupled with the environment in a sense mentioned in [9]. The problem environment's output is consumed by the agent and vice-versa in an iterative manner. The environment's output y is interpreted as a product of observation and reward received, $o \times r$. Considering the agent's goal is to maximize future expected reward on some arbitrary horizon m, an optimal policy is defined as,

$$p^* = \arg\max_p V_{1m}^{pe}, where\ V_{km}^{pe} = \sum_{e'} \mu(e') \sum_{i=k}^{m} r_i^{pe'} \tag{7.1}$$

V_{km}^{pe} is the utility of the policy p on problem environment e in cycles k to m. e is a probabilistic mixture of problem environments e' with probability distribution $\mu(e')$. If e is deterministic then $e' = e$ and $\mu(e') = 1$. As per initial assumption, both e and p are programs constructed in two Turing complete languages U_e and U_p, respectively. As they are Turing complete so they are Turing equivalent. Thus, any program e have a semantically equivalent program p. Thus, every solvable problem environment e have an equivalent program p_e in U_p. V_{km}^{pe} can be calculated for any p and finite m if e is known, by running e in coupled fashion with p for m steps. Considering in each step p is allowed to write a fixed length (l) string, a finite set of p can be tested generating all possible combination of strings of a finite length ml and p^* can be selected generating maximum V_{1m}^{pe}. From the above argument, it is clear that once the problem environment e is known there is a fixed algorithmic method to derive the optimal policy. Thus, every p^* is a fixed functional extension of p_e. For every e in U_e, there is a semantically equivalent p_e in U_p and consequently a p^*, although there might be many unsolvable e and p_e for which p^* does not make any sense.

Solution of any arbitrary solvable problem environment can be computed using an intermediate Turing complete language, a set of two lax monoidal functors and a monoidal product operator.

Proof: Let P denote a category of all such p_e which semantically maps with every e such that p^* can be constructed by functionally extending p_e. For a single e there can be multiple p_e which are semantically equivalent. To make a surjective functional map from E to P the p_e having minimum length is chosen for each e. All p can be represented by strings constructed by a finite set of alphabets A_p for U_p. Similar to E, P is also a monoidal category equipped with a bi-functor $\sigma_p: P \otimes P \to P$ and an identity object or empty string I_p.

Let us define a lax monoidal functor between categories E and P that signifies the map of every problem from E to P. A lax monoidal functor comes with a functor that maps objects between categories, a morphism to map identity, and a natural transformation to map

functorial product which satisfies the usual associativity and unitality conditions.

$$F : E \to P \tag{7.2}$$

$$\in : I_P \to F(I_E) \tag{7.3}$$

$$\eta_F : F(e_1) \otimes_P F(e_2) \to F(e_1 \otimes_E e_2), \forall e_1, e_2 \in E \tag{7.4}$$

Functor F maps semantically equivalent programs from E to P. One way of doing it is taking any program in U_e and convert it into U_p by adding a required interpreter of U_e written in U_p. However monoidal product σ_p keeps only the shortest among semantically equivalent programs. Thus F is a set of translation functions that translates any program in E to shortest equivalent program in P and preserves compositionality. As stated in equation 7.4 the usual commutative relation holds for F and monoidal products of E and P. Functor G derives the program corresponding to the optimal policy for problem environment E using the simulated image P and maps that to semantically equivalent program in E. This can be done by adding a specific code piece in P to derive the optimal policy using the model of the environment that can be interpreted by U_e. The code piece added by G should also contain logic to preserve product operation σ_e in E.

$$\begin{array}{ccccc} e_1 \otimes e_2 & \xrightarrow{F \otimes F} & p_1 \otimes p_2 & \xrightarrow{G \otimes G} & e_4 \otimes e_5 \\ \downarrow \sigma_e & \Downarrow \eta_F & \downarrow \sigma_p & \Downarrow \eta_G & \downarrow \sigma_e \\ e_3 & \xrightarrow{F} & p_3 & \xrightarrow{G} & e_6 \end{array} \tag{7.5}$$

Equation 7.5 states that for any arbitrary solvable problem in E there exist an optimal policy in E itself which can be derived by applying functor $G \circ F$. The policy can also be constructed from existing sub-policies (e_4, e_5) which may be full or partial, using the same monoidal product operator σ_e in E. If a problem environment e_1 transforms into a more complex environment e_3 the commutative diagram states that solution policy e_6 can be derived by multiple pathways. One by applying $G \circ F$, which signifies deriving the solution from

scratch. The other two solution pathways are $G \circ \sigma_p \circ F \otimes F$ and $\sigma_e \circ G \otimes G \circ F \otimes F$. Both pathways reuse part of solutions found while solving subproblem e_1. In many cases solving e_1 and reusing solutions for e_1 to solve e_3 might be less costly in terms of resources like time and space. Thus, any solution can be derived by using functor F, functor G, monoidal product σ_p and intermediate Turing complete language U_p. Hence proved.

7.2 FORMALIZATION OF SEED AI

We define seed AI as the minimum algorithm which is capable of evolving and finding optimal solutions for a wide range of problem environments under resource constraints like time, space and prior knowledge. Following the abstract model, any arbitrary problem environment can be represented in a Turing complete language U_e. The solutions can be symbolically represented in another Turing complete language U_p which can be simulated in U_e. The seed AI can be represented as just another program in U_e. We choose the pathway $G \circ \sigma_p \circ F \otimes F$ in our design of seed AI to map solutions to environment. Consequently, individual morphisms need to be implemented in U_e to realize a concrete implementation of seed AI.

The functor F can be understood as an estimator of any problem environment e in U_e, represented as program p in U_p. Ideally p should be exactly semantically equivalent with respect to e. But in real-world scenario, exact description of e may not be known and e can be estimated only by interacting with it. The quality of estimation depends on other environmental factors like resource constraints. Among multiple descriptions of the environment, the selection needs to be done based on some score that measures fitness or utility of the solution found using the description of the environment p in the context of a specific problem environment e. The functor uses a helper function ν in U_e to measure the utility in a specific context. Implementation of F in U_e is defined as follows,

$$f : e \times \nu \rightarrow p \tag{7.6}$$

σ_p denotes a function constructor using composition operator, whose domain consists of all semantically unique and shortest programs in U_p. Considering U_p a functional programming

language with a set of primitive functions or a generator set ω, making P a free monoid, all program equivalence can be calculated by algebraically simplifying program expressions due to absence of side effects. Implementation of σ_p in U_e is defined as,

$$\sigma_p : \begin{cases} p_1 \times p_2 \rightarrow p_2 \circ p_1, \ if \ U_p(p_2 \circ p_1) \not\equiv U_p(p_1) \wedge U_p(p_2 \circ p_1) \not\equiv U_p(p_2) \\ p_1 \times p_2 \rightarrow p_2, \ if \ U_p(p_2) \equiv U_p(p_2 \circ p_1) \\ p_1 \times p_2 \rightarrow p_1, \ if \ U_p(p_1) \equiv U_p(p_2 \circ p_1) \end{cases} \tag{7.7}$$

The natural transformation η_F transforms the estimator function f of a solved problem to find a new estimator for a new problem environment by reusing the estimator of the solved problem. The natural transformation in U_e is derived using the estimator function f and the function constructor σ_p. Implementation of η_F in U_e is defined as,

$$\eta_f \equiv \sigma_p \circ (f \times f) \tag{7.8}$$

The functor G computes the optimal solution program p_{sol} or the policy, using the environment description p and transforms it to an executable e_{sol} in U_e. This can be done by adding the interpreter of U_p written in U_e so that p_{sol} can be interpreted in U_e. G uses the helper function ν to find p_{sol} and consequently e_{sol}. Implementation of G in U_e is given as follows.

$$g : p \times \nu \rightarrow e_{sol} \tag{7.9}$$

7.2.1 Seed AI algorithm

For any solvable problem environment, the seed AI searches for a solution represented as a program in a Turing complete language. For any given problem environment e, the seed AI generates multiple programs as estimated representation of the problem environment and allocates execution time proportional to the fitness of solution programs e_{sol}. This continues until a maximum age is reached. The fitness of a solution program is calculated by a helper function ν, which might use the interaction history ($trace(e_{sol})$) of the program with the problem environment to calculate the same. The fitness of the solution program also determines the fitness of the estimated environments (p_1, p_2). f selects two of the best estimations based on fitness and σ_p combines them to produce a new estimation. In order to

bootstrap, f is supplied with a generator set which represents the primitives of U_p. g evaluates the new estimation by deriving the solution program and running it for a fraction of the total allocated runtime (T_R) in a phase. The runtime allocation is based on expected fitness of the solution. If the program does not halt within the allocated runtime it is interrupted. Similar to the Levin's search [41] the dynamic runtime allocation to individual programs alleviates the halting problem and makes the algorithm computable. T_p is the total runtime of the program e_{sol} and $\mathbb{E}$ represents the evaluator corresponding to U_e. After completion of each phase T the total runtime allocation is doubled and the same process repeats.

Algorithm 7.1 Seed AI

Function seedAI($e, \nu, f, \sigma_p, g,$ maxage)
While T < maxage
$T_R = T$
While $T_R \leq 0$
$f: e \times \nu \times T_R \rightarrow p_1$
$f: e \times \nu \times T_R \rightarrow p_2$
$\sigma_p: p_1 \times p_2 \rightarrow p_3$
$g: p_3 \times \nu \times T_R \rightarrow \mathbb{E}(e_{sol}, T_R) \times trace(e_{sol}) \times T_p$
$T_R = T_R - T_p$
$T = T * 2$

7.2.2 Constructivism

Constructivism is at the heart of the seed AI. Thorisson [62] already justified the need for constructivism in building artificial general intelligent systems. The seed AI achieves autonomy and adaptability by constructing estimations of problem environments, based on recorded action-perception history. One usual problem of the constructivist approach is facing exponential search space in order to find the right architecture or solution for a problem environment. The seed AI may also face this problem at its birth with minimal experiential knowledge as it needs to construct all possible programs and test them in some order. In course of doing so, gaining experiential knowledge will help dampen the search space and with the proper choice of ν the agent will gradually converge towards an optimal or near-optimal solution. The capability of constructing solutions for any solvable problem environment imparts a good generalization property to the seed AI. Yet, practically usable solutions may not be constructed from scratch in resource constrained environments. For

example, a specific problem environment may demand recognizing objects from captured images and picking up the objects which are recognized as apples using a robotic arm. Solving such an environment requires solving the subproblems of object recognition and robotic arm control. Creating solutions for these subproblems from scratch, in time constrained environment may not be practically feasible. Deep neural networks are already proved excellent in vision processing problems. Pretrained deep neural nets acting as object recognizers can be placed in the environment as a module. A robotic arm controller can also be placed as a library in the environment. The seed AI then constructs a program to integrate and control these modules in order to achieve the final goal. The addition of modules can be anytime without breaking the operation state of the seed AI, thus allowing integration of human intelligence with artificial intelligence. Such a level of abstraction in solution construction can also be achieved automatically by the seed AI. A seed AI may run another seed AI program which may construct a solution for a subproblem and place it in the environment as a module. Other constructed solution programs may reuse that module and achieve a level of abstraction.

7.2.3 Learning

Learning is the process of gaining experience with constrained resources based on interaction history with an environment so that the environment can be modified in the desired way as much as possible. Learning is analogous to the prediction problem with constrained resources. It can be defined as finding the best possible machine model M for an environment with limited interaction history of action/perception, time, and space. The learned machine model M can be used to predict an optimal sequence of actions for desired perceptions. Given a set of interaction histories an agent can find the best possible estimation of an environment using brute force. But it might use unlimited amount of time to find the model. Similarly, an expert system may tag outputs against inputs as rules, but it might need immensely large set of interaction history for any arbitrary environment to realistically model it, let alone assuming there is no effect of noise in measuring perceptions. In both cases it can be comfortably said that there is least amount of learning involved. Learning is essentially an optimization problem of finding patterns in interaction history with the environment, using limited resources and thereby modeling it to get maximum desired perceptions. For example,

while driving a car through a highway it is necessary to recognize and differentiate an animal lying on a road or a specific sign is written in the road. However, to avoid a collision it is irrelevant to identify if the animal is a wild animal or a pet of one of the car owner's neighbors. While training the object recognizer to get that information may be useful but it will obviously need more training data and/or time and will not contribute to improving the desired results with respect to the car driving problem.

The seed AI has inherent learning capability as it tries to find an estimate of the problem environment (p_3) so as to find out the program (e_{sol}) that generates desired set of actions. The usage of limited time to learn the model is already embedded in the logic as time allocated to evaluate a model of environment is directly proportional to the expected fitness of the solution. The choice of ν is important to drive the learning process. Proper choice of ν would allow progressive learning such that it balances exploration and exploitation and the learned model improves with time. ν can also control the usage of limited space in the learning process. For example, ν considers models with smaller size more fit comparatively. The design of seed AI inherently adds metalearning capability. For example, suppose a problem environment requires finding neural architectures which provide high accuracy with low training effort and use minimal training examples. Then in such scenario ν helps the seed AI to learn to find out neural architectures with improved learning capabilities. Also, suppose the objective of a problem environment is to find out seed AI with better learning capabilities, it may generate a ν which may improve the child seed AI when evaluated by the parent one. This adds the capability of metalearning of metalearning and so on.

7.2.4 Adaptability across environments

Let us assume the seed AI acts optimally in problem environment e_1 which transitions to environment e_3 by combining with e_2 in certain way. By equation 7.5 there is p_1 corresponding to e_1 in the representative language U_p. If seed AI already learned p_1 it needs to search for p_2 corresponding to e_2 which can be combined to form p_3 and eventually e_{sol3} which represents solution of e_3. Thus, the seed AI effectively reuses solutions of solved problem environments to adapt to new problem environments if the search process is less costly for a given ν. Otherwise, it may directly search p_3 without reusing p_1. In either case,

the seed AI is capable of transitioning to the new solution program when the problem environment changes, based on utility derived from action-perception trace. Making an instance of seed AI callable from programs constructed by the parent seed AI makes the search hierarchical. A problem may be automatically subdivided into subproblems and solved by child seed AIs and reused in the solution found by the parent seed AI.

A practical embodied intelligent agent is born in a single continuous environment. The embodiment itself serves as the internal environment for the seed AI of which seed AI is a part. It might include several sensors, actuators, knowledge bases, other soft skills, etc. Sensors, actuators may be physical or virtual but they are meant to communicate with the environment outside embodiment. However, there should be a problem in its internal environment that should give the agent a drive for its intelligent behavior. It is solely an internal problem within the embodiment. The root level seed AI communicates with this problem environment and tries to search for the optimal solution. Now the question is what should be this problem? This problem can be anything for any arbitrary embodied agent depending on the objective the agent is created to serve. For a general intelligent agent, the problem can simply be "act optimally using all functional components present in the internal environment" or "strive to maintain a homeostatic condition of the internal environment". This gives an internal drive for the agent. The internal environment can be influenced and changed by external perception signals which indirectly affect the drive of the seed AI. But the seed AI never communicates with the external environment which makes it impossible to control directly by an external being.

Though creating an internal problem environment for a general intelligent agent may sound simple yet it is an engineering challenge. The root problem may consist of multiple problems and components. Actions and perceptions need to be properly routed between subproblems and seed AI. Some perceptions may get priority over others or multiple perceptions may get merged and transformed. For example, a physical damage in any part of the agent may generate a strong punishment signal irrespective of the problem in which the seed AI is currently engaged. Fig 7.1. Illustrates an embodied agent constructed using seed AI. The root problem represents the primary problem in the internal environment of the agent which may consist of multiple subproblems. Some subproblems may in turn communicate with external

environment. The seed AI communicates with the root problem and searches for solution.

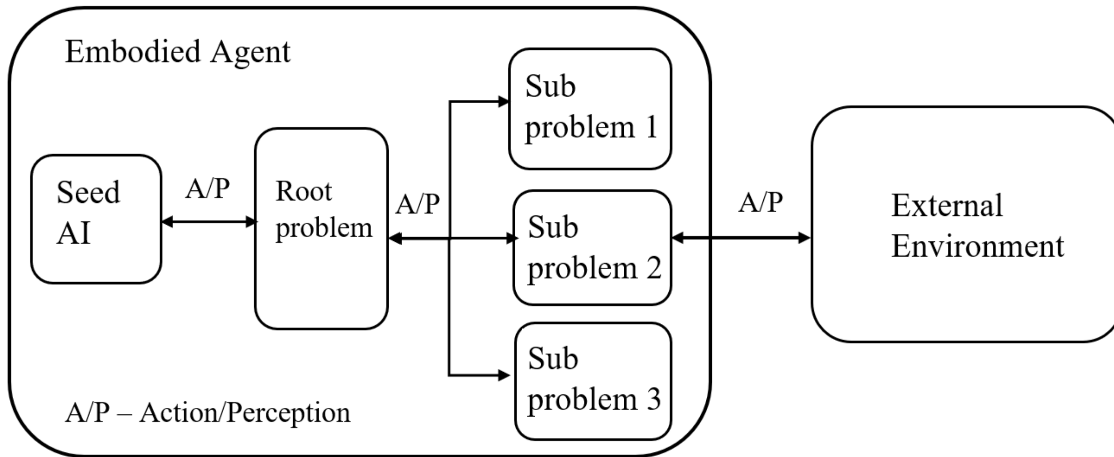


Figure 7.1 Architecture of an embodied intelligent agent

7.2.5 Recursive improvement

Self-improvement by self-modification of an intelligent agent is necessary to adapt across environments and continue acting optimally [104]. Arguments provided by Hall establishes the usefulness of self-improvement in achieving universal intelligence and render the bootstrap fallacy as generalization of experience with brains and systems below the level of universality [105]. The seed AI is just a program synthesis and scheduling algorithm which makes it a synthesizer, selector, and evaluator of specific programs. Due to this property self-modification is inherent to seed AI. If a specific program solves a specific problem it is going to get maximum time allocation. On changing the problem environment, the seed AI will eventually shift time allocation to some other program that serves as the solution for the new problem.

Recursive self-improvement requires improving the self-improvement algorithm itself aka the seed AI. Recursive self-improvements can be found by creating a problem environment using the seed AI itself. Let us consider a problem environment e_ξ, where ξ denotes an instance of seed AI. e_ξ can run another instance of seed AI ξ' with any ν, f, g and σ_p when interacting with ξ, considering ξ can generate/modify arbitrary ν, f, g and σ_p in U_e which serves as parameters for ξ'. e_ξ generates a positive reward if the fitness or utility of instance ξ' exceeds all previous fitness when measured with ν with which ξ is running. Considering ξ is also running within e_ξ, e_ξ generates reward if $\nu(\xi') > \nu(\xi)$. $\nu(\xi)$ measures the utility

of ξ with respect to the root problem. ξ' can be represented by a program generated by ξ while interacting with subproblem e_ξ. Increment of the utility of ξ in general indicates improvement of ξ itself. ν can be a function whose one component can represent future expected reward which makes one of the goals of ξ is to maximize total future expected reward with respect to root problem. If e_ξ is a part of root problem then finding self-improvement of itself is a part of the goal of ξ. In such scenario, if ξ' performs better than ξ with respect to root problem in general then $\nu(\xi') > \nu(\xi)$. Getting rewards for ξ' from e_ξ positively reinforces the value of $\nu(\xi')$ and its fitness gradually increases. As fitness of ξ' increases, ξ allocates more time to ξ' and it may eventually converge to a limit. Thus, it essentially means ξ' evolved as the improved version of ξ and occupies a major share of the execution time. This can happen recursively until no further improvements are found with respect to the root problem. Making the agent a part of the environment itself imparts the ability to self-modify [106]. Thus, the trick of achieving recursive improvement is making the solution part of the problem environment such that improvement of the solution can be automatically found by solving a problem environment which essentially asks for improvement of the prior solution.

7.3 Seed AI implementation with universal search

The metasearcher is a physical implementation of seed AI using universal search. The metasearcher implements the synthesizer, the fitness function, and the evaluator. The evaluator serves as the interpreter of the programming model. The following functions demonstrate the implementation of each of the sub-methods of the seed AI. The metasearcher generates a search graph to solve a problem environment. The search graph consists of a cluster of generated programs in the programming model. Individual programs are executed and perceptions from the environment are recorded and reward signals are extracted for each program. The sequence of rewards constitutes the output trace of each program which in turn is used to update the fitness. The search graph loosely resembles a model of the problem environment which can be used to estimate expected reward on executing a sequence of actions as defined by a specific program graph.

The fitness of a program is calculated by the program probability which in turn indicates the

fitness of a search graph. The maximum fitness among all programs in a search graph denotes the fitness of the search graph itself. The program probabilities are assigned and updated in the same strategy as mentioned in Chapter 5. Program probability distribution is updated incrementally based on the sequence of rewards received by individual programs. The probability distribution is updated using gradient ascent so as to maximize the total future expected reward, as defined by the objective function $J(p)$. x_k is a program in search graph with index k, i' represents the horizon and $p(r_i|x_k)$ represents conditional probability for gaining reward r_i when program x_k is extended to x_i.

The synthesizer selects program graphs (x) from search graph based on fitness and extends them by adding another selected function node (y). The $\sigma(.,.)$ is the composition operator which composes two program graphs and produces a third program graph. If the new synthesized program is not semantically redundant, it is added to the search graph and resultantly modifies it. Semantically redundant programs with lower fitness are not added to the search graph to avoid the problem of over-representation in the sense mentioned by Looks and Goertzel [18]. The synthesizer implements the natural transformation η_F in the seed AI. The evaluator selects the optimal program graph from the search graph, based on fitness and evaluates it using the interpreter of the programming model. The evaluator is allocated a time proportional to the fitness of the program graph where c is a predefined constant value and $0 < c \leq 1$. If the runtime of the program exceeds the allocated time it is interrupted else the program is marked as executed and runtime (T_r) is recorded.

The metasearcher implements the main body of the seed AI using the helper functions, namely, fitness, synthesizer, and evaluator. The metasearcher is executed in phases and in each phase, a specific total runtime is allocated (T_R). Once the total allocated time is consumed it is doubled in each subsequent phase and this continues until the maximum age (T_{limit}) is reached.

Algorithm 7.2 Seed AI implementation with the metasearcher

Function fitness(X, x)
  If $x \notin X$
    Assign default probability to x and add in p_X^*
  $J(p) = \sum_{\forall x_k \in X} \sum_{\forall i > k \wedge \forall i < i'} p(r_i|x_k) r_i$

$p_X^* \leftarrow \arg\max_p J(p)$
$p_x^* \leftarrow$ probability of x from p_X^*
Return p_x^*

Function synthesizer(X,T, y)
$x = \arg\max_x fitness(\text{X}, \sigma(x,y)) | \forall_{x \in X \text{ and } x \text{ is executed}} fitness(\text{X}, \sigma(x,y)) * T > 1 \wedge \sigma(x,y) \notin X$
If x exists
Add $\sigma(x,y)$ in X
Return

Function evaluator(X, T)
$x = \arg\max_x fitness(\text{X}, \text{x}) | \forall_{x \in X \text{ and } x \text{ is not executed}} fitness(\text{X}, \text{x}) * T > 1$
$T_r \leftarrow eval(x, fitness(\text{X}, \text{x}) * T/c)$
If x halts in $fitness(\text{X}, \text{x}) * T/c$
Mark x as executed
Return T_r

Function metasearcher(T_{limit}, Y)
$T = 2$
While $T < T_{limit}$
$T_R = T$
While $T_R > 0$
For each y in Y
synthesizer(X,T, y)
$T_r =$ evaluator(X, T)
$T_R = T_R - T_r$
$T = T * 2$

7.3.1 Agent as an environment

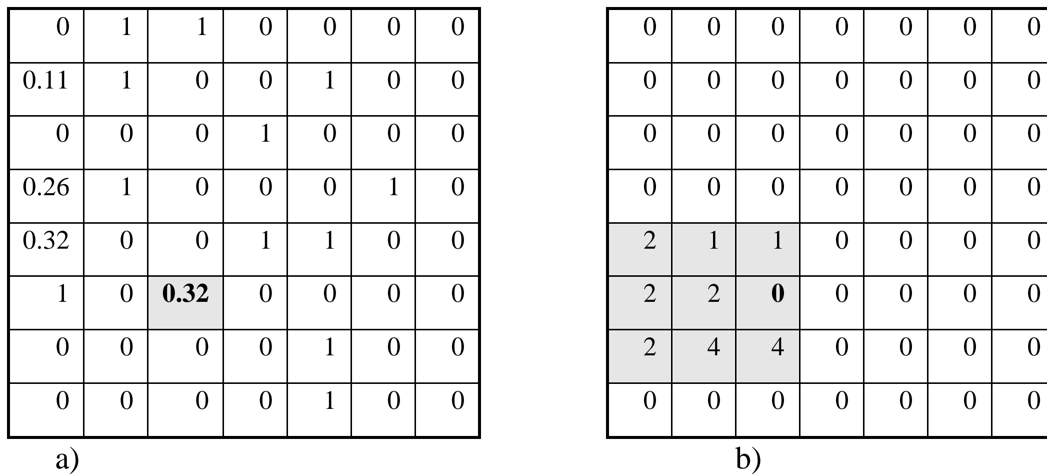
After introducing an implementation of seed AI naturally, a question arises – What constitutes an intelligent agent? Is it the seed AI? No! The seed AI is a part of the environment or rather the internal environment of an embodied agent. The seed AI acts as a controller or integrator of various components of its internal environment including a copy of itself. The seed AI may generate a better version of itself and assign maximum resources to it. An agent can be constructed by building and connecting multiple components which constitutes its internal environment. An embodied agent may contain several sensors, actuators, internal knowledge base, reasoning capability, etc. All have some functional interfaces along with some subproblems to train the agent to use these capabilities. The agent has a root problem that gives the internal drive to do everything that an agent does. While acting optimally against the root problem it may reuse solution programs for individual

subproblems and integrate them with other programs. Interaction with the external environment is optional, yet to make an agent useful in real-world, it is necessary. Communication with the external environment is carried out with functionalities available in the internal environment of the agent. The embodied agent may be supplied with various inbuilt capabilities to act optimally against the external environment and the list may be augmented from time to time. The seed AI achieves autonomy by learning to integrate and control the functional capabilities available within the embodied agent so as to make it useful in an external environment.

7.3.2 Case study

Experiments are conducted with the prototype in a heterogenous maze environment. Fig. 7.2 illustrates the maze environment. The prototype acts as an agent in the maze environment whose objective is to traverse through the cells in the maze and reach the goal state. The rewards and images of direction in the vision zone help guide the search. This is a resettable environment and the agent is initially placed in [0,0] cell. Five actions are allowed in the environment, each shifting the agent in a specific direction. Namely 1 – move front, 2 – move left, 3 – move right, 4 – move back, 0 – return current cells observation. The vision zone returns base64 encoded image of the direction of the goal state. The reward distribution and the vision zone together make it a heterogeneous problem environment. In order to reach the goal state, the agent has to initially follow the reward distribution and in later stage, it has to perform optical character recognition to decide the direction of movement. The agent searches through the program space and finds a program, executing which would generate a sequence of actions to lead the agent to the goal state. The problem environment and components external to the programming model are implemented as microservices. All the microservices are configured to take an action request as input and returns observation, reward pair as output. The agent interacts with these microservices through an interface that serves as the root problem environment for the agent. The following functions are used as generator set for the metasearcher. iW, 1.K, 2.K, 3.K, 4.K, maze, ocreco, lp. The maze and ocreco are composite node functions. Each contains a similar program graph as illustrated in Fig. 7.3a. For ocreco the node maze.K is replaced with ocreco.K. The composite nodes act as functions to interact with the maze environment and OCR component. The maze node

sends the input argument as an action to the maze environment and returns the observation. The cumulative reward obtained by running a program is used by the fitness function to update the probability distribution. The ocreco node takes an image encoded in base64 and returns the identified character in it. For any other input, it returns 0. Fig 7.3b. illustrates the program graph found by the metasearcher. There is a pattern in the solution which has been identified in the solution. The movement pattern is repeated 4 times in a loop to reach the goal state. The reward distribution is used to update the probability distribution of the programs and expedite the search process. The solution program took direct movement actions in the maze environment as well as used the OCR component wherever necessary to solve this heterogenous problem environment optimally.



a)

0	1	1	0	0	0	0
0.11	1	0	0	1	0	0
0	0	0	1	0	0	0
0.26	1	0	0	0	1	0
0.32	0	0	1	1	0	0
1	0	**0.32**	0	0	0	0
0	0	0	0	1	0	0
0	0	0	0	1	0	0

b)

0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0
2	1	1	0	0	0	0
2	2	**0**	0	0	0	0
2	4	4	0	0	0	0
0	0	0	0	0	0	0

Figure 7.2 Maze environment a) reward distribution. The grey cell is the goal state. Cells containing 1 generate -0.5 punishment. All other cells generate rewards as specified in the illustration. b) The grey region denotes the vision zone. Each cell in the grey region returns a base64 encoded image of the number specified as observation. All other cells return 0 as observation

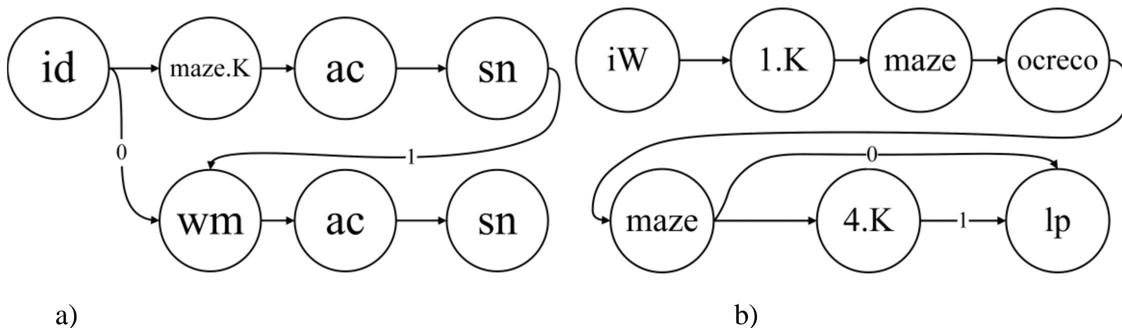


Figure 7.3 Solution program graphs a) Program graph for the maze composite node. b) solution program graph found by the metasearcher for the maze problem environment.

7.4 Conclusion

The presented formal architecture of seed AI provides guidance to implement any general intelligent system. It has been proved that the presented model of the seed AI is theoretically capable of finding a solution for any solvable problem. The seed AI algorithm states that a properly designed synthesizer of arbitrary computation logic and a scheduler is sufficient to achieve general AI. This alleviates the problem of finding an AGI system design from scratch through different approaches, as any system could evolve into AGI system if they are structurally similar to the seed AI. The seed AI guarantees to achieve generality by adapting across different solvable problem environments. It has also been discussed how the seed AI could achieve different properties of general intelligent systems, like learning, self-improvement, constructivism, etc. An implementation roadmap of the seed AI is also demonstrated and the developed prototype is experimented in a toy problem. However physical implementation of a full-fledged general intelligent system is still an engineering challenge, especially designing an appropriate fitness function, the core internal components, the subproblem environments, and the root problem. Multiple subproblems need to be designed based on the situatedness of the system which would allow communicating with the external environment. On principle, the seed AI should be able to generate subproblems. But its engineering setting, feasibility, and applicability need to be investigated. The limitations of the seed AI in an embodied agent also needs to be investigated.

8 CONCLUSION AND FUTURE WORK

Artificial Intelligence has revolutionized various fields and made a significant impact on the way we solve problems. General Problem Solving (GPS) is an important area of research in AI as it seeks to develop intelligent systems that can solve a wide range of problems, like how humans do. GPS is not just about finding a solution to a specific problem, but it is about developing intelligent systems that can apply knowledge from one problem domain to another, generalize knowledge to new situations, and adapt to new environments.

The importance of GPS lies in its potential to automate complex tasks that are currently done by humans. This could lead to a significant reduction in time and cost, increased efficiency, and the ability to solve previously unsolvable problems. For example, GPS could help in medical diagnosis, robotics operations, and scientific research, where the amount of data to be processed is enormous and requires extensive human effort.

However, achieving GPS is a challenging task. There are many reasons for this difficulty. One major challenge is the inherent complexity of many problems that require a deep understanding of the domain knowledge. This domain knowledge is often not explicitly stated and requires sophisticated methods for discovering, modeling, and integrating it into the problem-solving process. Additionally, many problems require reasoning with uncertainty, which adds another layer of complexity.

Another challenge is the need for a flexible and adaptable system that can learn and reason across multiple domains. This requires not only the ability to learn from data but also to apply that learning to new situations, generalize knowledge to new domains, and integrate different types of knowledge.

Moreover, building a GPS system requires the integration of different AI techniques such as knowledge representation, reasoning, planning, learning, and perception. Combining these techniques into a cohesive and efficient system is a significant challenge.

In the first part of this thesis, we reviewed the current state of the art in general problem solving, using AI. We discussed the limitations of traditional AI approaches and the need for

more advanced techniques to solve complex problems. We also presented an overview of different approaches to problem solving, including classical AI, connectionist AI, program search based AI and integrative AI. The current AI systems lack a robust and flexible cognitive architecture, which is necessary to support GPS. The cognitive architecture must be able to integrate knowledge and reasoning from multiple sources, learn from experience, and adapt to new situations and act optimally under constrained resources like space, time and knowledge. Developing such an architecture requires a deep understanding of cognitive processes, which is still an active area of research.

The survey chapter provides a comprehensive review of the existing AI prior arts and their limitations in tackling the general problem-solving task. From the analysis of the various techniques and approaches employed in AI, it is evident that none of the existing methods are completely practically feasible to handle wide range of general problem-solving tasks, specially under constrained resources. While they have been successful in solving specific problems, they lack the flexibility and adaptability to solve new problems under constrained resources, that they have not been explicitly designed for.

The review of the existing AI methods reveals that most of them rely on a combination of techniques such as search algorithms, rule-based systems, and machine learning models. These methods are tailored towards solving specific problems and are designed to optimize for specific objectives, such as accuracy or speed. They lack the ability to generalize and adapt to new problem domains under constrained resources, which limits their usefulness in real-world applications.

We proposed several methods and techniques to handle the task of general problem solving in a resource constrained environment, making it practically feasible. Though our work may not be considered as the final solution for general-problem solving, yet our contributions lie in providing few novel methods and frameworks that can be used in further development and enhancement of general-problem solving agents.

The functional dataflow graph programming model proposed in this thesis has proven to be an effective tool in building an integrative AI platform that addresses the challenges of solving heterogeneous AI tasks. This platform allows the use of black-box AI modules and IoT devices as microservices, as well as the construction of white-box program graphs for

implementing arbitrary integration logic. The platform's functional abstraction of components offers a high level of abstraction, allowing developers and architects to build and implement AI architectures with low engineering effort. However, this higher level of abstraction comes with some limitations. The available capability of tweaking low-level implementation is limited, since most of it is abstracted using predefined methods. Nonetheless, the platform opens the possibility of automatic programming, where machine learning models can be trained for automatic construction of integration programs to solve problems. Due to functional nature of the programming model, it is easier to construct programs mathematically. Large language models can be trained to generate more accurate programs in this programming model based on natural language inputs.

The integration platform's future scope is promising, especially in the area of General Problem Solving (GPS) systems. The platform's ability to handle heterogeneous tasks through an integrative approach can be used to construct GPS systems. Due to the graphical nature of the programming model, an user interface constructed on top of this platform would provide a low-code platform to integrate and construct AI solutions graphically. Treating AI modules and IoT devices as microservices helps building truly distributed AI architectures within the platform.

The development of a solution searcher for general problems is a significant advancement in the field of AI for general problem solving. The use of a functional dataflow graph programming model, metasearcher, incremental learning, and genetic programming helped in creating an efficient and effective solution searcher that can tackle the problem of combinatorial explosion quite well during solution search.

The functional dataflow graph programming model provides a flexible and scalable approach for building AI architectures. This allowed the development of a modular and reusable architecture that could be easily integrated with other AI modules. The use of a metasearcher helped in enhancing the search capabilities of the system by combining multiple search techniques.

Incremental learning was used to improve the performance of the system by continuously updating the knowledge base. This allowed the system to adapt and learn from new data and experiences. The use of genetic programming helped in evolving solutions by creating new

combinations of existing solutions. This approach helped in tackling the problem of combinatorial explosion by reducing the search space.

The solution searcher developed using these techniques has shown promising results in solving complex problems. However, there is still much work to be done to improve the performance of the system. The system could be further optimized for parallel processing to improve the speed of the search.

The proposed theoretical structure of seed AI provides a promising path for the construction of general problem solving systems. The seed AI model presents a formal architecture that guides the implementation of a general intelligent system that is theoretically capable of solving any solvable problem. The model suggests that a properly designed synthesizer of arbitrary computation logic and a scheduler is sufficient to achieve general AI, thereby alleviating the problem of finding an AGI system design from scratch through different approaches.

Furthermore, the seed AI algorithm guarantees the achievement of generality by adapting across different solvable problem environments, as well as achieving different properties of general intelligent systems such as learning, self-improvement, and constructivism. An implementation roadmap of the seed AI has been demonstrated, and a prototype based on the metasearcher has been developed and experimented within a toy problem.

However, it is important to note that the physical implementation of a full-fledged general intelligent system is still an engineering challenge, and designing an appropriate fitness function, the core internal components, the subproblem environments, and the root problem, present significant challenges. Additionally, multiple subproblems need to be designed based on the situatedness of the system, which would allow communication with the external environment. While the seed AI should be able to generate subproblems, its engineering setting, feasibility, and applicability require further investigation.

Nonetheless, the proposed theoretical structure of seed AI provides a solid foundation for the construction of general problem solving systems, and its principles can guide future research towards achieving the goal of building truly general AI systems capable of solving complex and diverse problems. By using the seed AI model as a guide, researchers can explore new

avenues of research to overcome the limitations of existing AI prior arts in tackling general problem solving.

The research presented in this thesis opens up many avenues for future work. Here are some potential areas of investigation:

1) **Real-world application of the integrative AI platform:** The proposed integrative AI platform has shown great potential in solving heterogeneous AI tasks. However, further research is needed to test its practical application in real-world settings. An user interface can also be built to make it a low code platform for constructing AI architectures. It would be interesting to investigate how the platform can be used in complex problem environments, such as healthcare, finance, and transportation.
2) **Improvement of the proposed functional dataflow graph programming model:** The functional dataflow graph programming model proposed in this thesis has proved to be effective representational language in building the integrative AI platform. Parallelizing execution flows can significantly speed up evaluation of programs. It may be explored how program executions can parallelized. Functions for program construction and manipulation needs to be part of the programming model, so that programs can generate and manipulate other programs. It can be explored how to make them as node functions without hampering the purity of the language.
3) **Extension of the proposed solution searcher for general problems:** The proposed solution searcher for general problems using metasearcher, incremental learning, and genetic programming has shown promising results in addressing the problem of combinatorial explosion. However, there is room for further research on how to improve the efficiency and effectiveness of the algorithm, as well as its applicability to different types of problem environments. Introspection capability can be added in the programming model which would enable self-generation, self-modification, and self-evaluation of code. This would allow the metasearcher to be a part of the program corpus in the search graph, thus allowing the possibility of evolution and automatic improvement of the searcher program itself. The metasearcher program can also be added as an external function in the environment. This would allow searching through a new program space within another search and consequently allow hierarchical

problem solving. Another challenge with the metasearcher is, it is still slow for many real-world problems. However, the searcher can be parallelized and that would significantly improve its runtime.

4) **Further investigation on seed AI:** The theoretical structure of seed AI proposed in this thesis provides a promising direction for constructing general intelligent systems. One of the major parts of the seed AI is the fitness function. It may be explored if a general fitness function can be formalized that would on principle provide the drive for a general problem solver. Agents can be built and tested using other methods, like neural networks, following the structure of the seed AI.
5) **Investigating the use of the proposed metasearcher for other types of problems:** The proposed metasearcher has shown promising results in finding solutions for different problem instances. Future research can investigate the applicability of the proposed approach to other types of problems, such as finding integrative AI architectures for situated intelligence. It can be tested in problems where the problem space changes over time, such as in robotics or adversary planning problems.

The future research directions outlined above are just a few of the many avenues that can be explored to further advance the state-of-the-art in this field.

9 REFERENCES

1 Gilhooly, K. J., (Ed.). (2012), *Human and machine problem solving*, Springer Science & Business Media.

2 Duncker, K., & Lees, L. S., (1945), *On problem-solving*, Psychological monographs, 58(5), i.

3 Holyoak, K. J., (1990), *Problem solving*, Thinking: An invitation to cognitive science, 3, pp. 117-146.

4 Newell, A., & Simon, H. A., (1972), *Human problem solving,* (Vol. 104, No. 9). Englewood Cliffs, NJ: Prentice-hall.

5 Breuker, J., (1994, September), *Components of problem solving and types of problems*, In International Conference on Knowledge Engineering and Knowledge Management, Springer, Berlin, Heidelberg, pp. 118-136.

6 **https://web.njit.edu/~lipuma/GPS.htm**

7 U. Neisser, G. Boodoo, T. J. Bouchard, Jr., A. W. Boykin, N. Brody, S. J. Ceci, D. F. Halpern, J. C. Loehlin, R. Perloff, R. J. Sternberg, and S. Urbina, (1996), *Intelligence: Knowns and unknowns*, American Psychologist, 51(2), pp. 77–101.

8 Legg, Shane & Hutter, Marcus, (2007), *Universal intelligence: A definition of machine intelligence*, Minds and Machines 17.4, pp. 391-444.

9 Hutter, Marcus, (2003), *A Gentle Introduction to The Universal Algorithmic Agent {AIXI}*, Technical report, Manno-Lugano, Switzerland: IDSIA, https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.11.8797&rep=rep1&type= pdf.

10 Schmidhuber, Jürgen, (2007), *Gödel machines: Fully self-referential optimal universal self-improvers*, Artificial general intelligence. Springer Berlin Heidelberg, pp. 199-226.

11 Steunebrink, B. R., & Schmidhuber, J., (2011, August), *A family of Gödel machine implementations*, In International Conference on Artificial General Intelligence, Springer, Berlin, Heidelberg, pp. 275-280.

12 Steunebrink, B. R., Thórisson, K. R., & Schmidhuber, J., (2016, July), *Growing recursive self-improver*, In International Conference on Artificial General Intelligence, Springer, Cham, pp. 129-139.

13 Goertzel, Ben., Ed. Cassio Pennachin, (2007), *Artificial general intelligence,* Vol. 2. New York: Springer.

14 Li, M., & Vitányi, P., (2013), *An introduction to Kolmogorov complexity and its applications*, Springer Science & Business Media.

15 Johnston, W. M., Hanna, J. R., & Millar, R. J., (2004), *Advances in dataflow programming languages*, ACM computing surveys (CSUR), 36(1), pp. 1-34.

16 Akidau, T., Bradshaw, R., Chambers, C., Chernyak, S., Fernández-Moctezuma, R. J., Lax, R., ... & Whittle, S., (2015), *The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing*, Proceedings of the VLDB Endowment, Vol. 8, No. 12, pp. 1792-1803.

17 Brandt, J., Reisig, W., & Leser, U., (2017), *Computation semantics of the functional scientific workflow language Cuneiform*, Journal of Functional Programming, 27, E22

18 Looks, M., & Goertzel, B., (May 2009), *Program representation for general intelligence*, Proc. of AGI (Vol. 9).

19 Yudkowsky, E., (2007), *Levels of organization in general intelligence. In Artificial general intelligence*, Springer, Berlin, Heidelberg, pp. 389-501.

20 Newel, A., & Simon, H. A. (1976), *Computer science as empirical inquiry: Symbols and search*, Communications of the ACM, *19*(3), pp. 113-126.

21 Newell, Allen, John C. Shaw, and Herbert A. Simon, (1959), *Report on a general problem solving program*, IFIP Congress. Vol. 256.

22 Nivel, E., Thórisson, K. R., Dindo, H., Pezzulo, G., & Rodriguez, M., (2013), *Autocatalytic endogenous reflective architecture*, http://alumni.media.mit.edu/~kris/ftp/AERA-RUTR-SCS13002.pdf

23 Guha, R. V., Lenat., DB, (1990), *Cyc: A midterm report,* AI Magazine 11.3 (6), pp. 33-59.

24 Newell, Allen, (1994), *Unified theories of cognition*, Harvard University Press.

25 Wang, Pei, (1995), *Non-Axiomatic Reasoning System| Exploring the Essence of Intelligence*, Diss. Indiana University.

26 Wang, P., Li, X., & Hammer, P. (2018), *Self in NARS, an AGI System*, Frontiers in Robotics and AI, 5, 20.

27 Stephen Grossberg, (1992), *Neural Networks and Natural Intelligence*, MIT Press.

28 de Garis H, Korkin M, (2002), *The CAM-Brain Machine (CBM): An FPGA Based Hardware Tool which Evolves a 1000 Neuron Net Circuit Module in Seconds and Updates a 75 Million Neuron Artificial Brain for Real Time Robot Control,* Neurocomputing, 42(1-4), pp. 35–68

29 Arel, I., (2012), *Deep reinforcement learning as foundation for artificial general intelligence*, In Theoretical Foundations of Artificial General Intelligence (pp. 89-102). Atlantis Press, Paris.

30 Mnih, Volodymyr; Kavukcuoglu, Koray; Silver, David, (26 February 2015), *Human-level control through deep reinforcement learning*, Nature. 518 (7540), pp. 529–33. Bibcode:2015 Natur.518..529M. doi:10.1038/nature14236. PMID 25719670.

31 Schmidhuber, J., Cireşan, D., Meier, U., Masci, J., & Graves, A., (2011, August), *On fast deep nets for AGI vision*, In International Conference on Artificial General Intelligence, Springer, Berlin, Heidelberg, pp. 243-246

32 Floridi, L., & Chiriatti, M., (2020), *GPT-3: Its nature, scope, limits, and consequences*, Minds and Machines, *30*(4), pp. 681-694.

33 Ramesh, A., Pavlov, M., Goh, G., Gray, S., Voss, C., Radford, A., ... & Sutskever, I., (2021), *Zero-shot text-to-image generation*, arXiv preprint arXiv:2102.12092.

34 Ray, T. S., (1994), *An evolutionary approach to synthetic biology: Zen and the art of creating life*, Artificial Life 1(1/2): pp. 195-226.

35 Thomas S. Ray., (1995), *A Proposal to Create a Network-Wide Biodiversity Reserve for Digital Organisms*, Technical Report ATR Technical Report TR-H-133, ATR,

36 Ofria, C., & Wilke, C. O., (2004), *Avida: A software platform for research in computational evolutionary biology,* Artificial life, *10*(2), pp. 191-229.

37 Solomonoff, Ray J., (1964), *A formal theory of inductive inference. Part I,* Information and control 7.1, pp. 1-22

38 Solomonoff, Ray J., (1964), *A formal theory of inductive inference. Part II,* Information and control 7.2, pp. 224-254

39 Schmitz, NORBERT J., (1991), *Sequential decision theory*, Handbook of sequential analysis. New York: Marcel Dekker, pp. 407-28.

40 Schmidhuber, Jürgen, (2004), *Optimal ordered problem solver,* Machine Learning 54.3, pp. 211-254.

41 Levin, L.A., (1973), *Universal sequential Search Problems*, Problemy Peredaci Informacii 9, pp. 115–116. Translated in Problems of Information Transmission 9, pp. 265–266

42 Kaiser, L., (2007), *Program search as a path to artificial general intelligence*, In Artificial general intelligence, Springer, Berlin, Heidelberg, pp. 291-326.

43 Looks, Moshe, Ben Goertzel, and Cassio Pennachin, (2004), *Novamente: An integrative architecture for general intelligence,* AAAI fall symposium, achieving human-level intelligence.

44 Goertzel, Ben, (2008), *OpenCog Prime: Design for a Thinking Machine*, Online wikibook at http://opencog. org.

45 Goertzel, B., (2017, August), *A formal model of cognitive synergy*, In International Conference on Artificial General Intelligence. Springer, Cham, pp. 13-22.

46 Goertzel, B., (2016), *Probabilistic growth and mining of combinations: A unifying meta-algorithm for practical general intelligence*, In International Conference on Artificial General Intelligence, Springer, pp. 344–353.

47 Ben Goertzel, Cassio Pennachin, and Nil Geisweiller, (2013), *Engineering General Intelligence, Part 1: A Path to Advanced AGI via Embodied Learning and Cognitive Synergy*, Springer: Atlantis Thinking Machines.

48 Ben Goertzel, Cassio Pennachin, and Nil Geisweiller, (2013), *Engineering General Intelligence, Part 2: The CogPrime Architecture for Integrative, Embodied AGI*, Springer: Atlantis Thinking Machines.

49 Ng-Thow-Hing, V., Thórisson, K. R., Sarvadevabhatla, R. K., Wormer, J. and List, T., (2009), *Cognitive map architecture: Faciliation of human-robot interaction in humanoid robots*, IEEE Robotics & Automation 16(1), pp. 55–667.

50 Thórisson, K. R., Benko, H., Arnold, A., Abramov, D., Maskey, S. and Vaseekaran, A., (2004), *Constructionist design methodology for interactive intelligences*, A.I. Magazine 25(4), pp. 77– 90.

51 Andrist, S., Bohus, D., & Feniello, A., (2019, March), *Demonstrating a framework for rapid development of physically situated interactive systems*, In 2019 14th ACM/IEEE International Conference on Human-Robot Interaction (HRI), IEEE, pp. 668-668.

52 Bohus, D., Andrist, S., Feniello, A., Saw, N., Jalobeanu, M., Sweeney, P., ... & Horvitz, E., (2021), *Platform for situated intelligence*, arXiv preprint arXiv:2103.15975.

53 Thórisson, K. R., (2020, September), *Seed-Programmed Autonomous General Learning*, In International Workshop on Self-Supervised Learning, PMLR, pp. 32-61.

54 Nivel, E., Thórisson, K. R., Steunebrink, B. R., Dindo, H., Pezzulo, G., Rodríguez, M., ... & Chella, A., (2014, August), *Bounded seed-AGI,* In International Conference on Artificial General Intelligence, Springer, Cham, pp. 85-96.

55 Steunebrink, B. R., Thórisson, K. R., & Schmidhuber, J. (2016, July), *Growing recursive self-improvers,* In International Conference on Artificial General Intelligence (pp. 129-139). Springer, Cham.

56 Hughes, J., (2000), *Generalising monads to arrows*, Science of computer programming, 37(1-3), pp. 67-111.

57 Michele Lombardi, (June 22, 2020), *A simple guide to Integrative AI*, Published on the AI4EU platform: http://ai4eu.eu.

58 Kirchner, F., (2020), *AI-perspectives: the Turing option*, AI Perspectives, 2(1), pp. 1-12.

59 Hart, D., & Goertzel, B., (2008, March), *Opencog: A software framework for integrative artificial general intelligence*, In AGI, pp. 468-472.

60 Atlam, H. F., Walters, R. J., & Wills, G. B., (2018, July), *Intelligence of things: opportunities & challenges*, In 2018 3rd Cloudification of the Internet of Things (CIoT), IEEE, pp. 1-6.

61 Thönes, J., (2015), *Microservices*, IEEE software, 32(1), pp. 116-116.

62 Thórisson, K. R., (2012), *A new constructivist AI: from manual methods to self-constructive systems*, In Theoretical Foundations of Artificial General Intelligence. Atlantis Press, Paris, pp. 145-171.

63 Dipsis, N., & Stathis, K., (2019), *A RESTful middleware for AI controlled sensors, actuators and smart devices*, Journal of Ambient Intelligence and Humanized Computing, pp. 1-24.

64 Lin, Y. W., Lin, Y. B., Liu, C. Y., Lin, J. Y., & Shih, Y. L., (2019), *Implementing AI as cyber IoT devices: The house valuation example*, IEEE Transactions on Industrial Informatics, 16(4), pp. 2612-2620.

65 Guidi, C., Lanese, I., Mazzara, M., & Montesi, F., (2017), *Microservices: a language-based approach*, In Present and Ulterior Software Engineering. Springer, Cham, pp. 217-225.

66 Rademacher, F., Sorgalla, J., Wizenty, P., Sachweh, S., & Zündorf, A., (2020), *Graphical and textual model-driven microservice development*, In Microservices. Springer, Cham, pp. 147-179.

67 Paul, S. K., Jana, S., & Bhaumik, P., (2021), *On Solving Heterogeneous Tasks with Microservices*, Journal of The Institution of Engineers (India): Series B, pp. 1-9.

68 https://github.com/Microsoft/psi-samples/tree/main/Samples/SpeechSample

69 Kamat, P., & Sugandhi, R., (2020), *Anomaly detection for predictive maintenance in industry 4.0-A survey*, In E3S Web of Conferences (Vol. 170, p. 02007). EDP Sciences.

70 Vani, K., (2019, December), *Deep learning based forest fire classification and detection in satellite images*, In ICoAC, IEEE, pp. 61-65.

71 Hutter, M., (2002), *The fastest and shortest algorithm for all well-defined problems*, International Journal of Foundations of Computer Science, 13(03), pp. 431-443.

72 Solomonoff, R. J., (1989, December), *A system for incremental learning based on algorithmic probability,* In Proceedings of the Sixth Israeli Conference on Artificial Intelligence, Computer Vision and Pattern Recognition, pp. 515-527.

73 Solomonoff, R. J., (2010), *Algorithmic probability, heuristic programming and AGI,* Advances in Intelligent Systems Research, 10, pp. 151-157.

74 Grünwald, P. D., & Vitányi, P. M., (2008), *Algorithmic information theory*, Handbook of the Philosophy of Information, pp. 281-320.

75 Cover, T. M., & Thomas, J. A., (2012), *Elements of information theory*, John Wiley & Sons.

76 Swarna Kamal Paul & Parama Bhaumik, (2021), *Solving Partially Observable Environments with Universal Search Using Dataflow Graph-Based Programming Model*, IETE Journal of Research, DOI: 10.1080/03772063.2021.2004461

77 Paul, S.K., & Bhaumik, P., (2016), *A fast universal search by equivalent program pruning*, In ICACCI, pp. 454-460.

78 Nordin, P., Keller, R. E., & Francone, F. D., (1998), *Genetic programming,* W. Banzhaf (Ed.). Springer.

79 Hausknecht, M., & Stone, P., (September 2015), *Deep recurrent q-learning for partially observable mdps*, In *2015* AAAI Fall Symposium Series.

80 Silver, D., & Veness, J., (2010), *Monte-Carlo planning in large POMDPs*, In Advances in neural information processing systems, pp. 2164-2172.

81 Veness, J., Ng, K. S., Hutter, M., Uther, W., & Silver, D., (2011), *A monte-carlo aixi approximation*, Journal of Artificial Intelligence Research, 40, pp. 95-142.

82 S. K. Paul and P. Bhaumik, (2020), *A Reinforcement Learning Agent based on Genetic Programming and Universal Search*, In ICICCS, pp. 122-128.

83 Paul, S. K., Gupta, P., & Bhaumik, P., (2018, December), *Learning to Solve Single Variable Linear Equations by Universal Search with Probabilistic Program Graphs*, In ICIBCA, Springer, Cham, pp. 310-320

84 Schaul, T., & Schmidhuber, J., (2010, June), *Towards practical universal search,* In Proceedings of the Third Conference on Artificial General Intelligence, Lugano.

85 Elsken, T., Metzen, J. H., & Hutter, F., (2019), *Neural architecture search: A survey*, The Journal of Machine Learning Research, 20(1), pp. 1997-2017.

86 Bowen Baker, Otkrist Gupta, Nikhil Naik, and Ramesh Raskar, (2017a), *Designing neural network architectures using reinforcement learning*, In International Conference on Learning Representations.

87 Masanori Suganuma, Shinichi Shirakawa, and Tomoharu Nagao, (2017), *A genetic programming approach to designing convolutional neural network architectures*, In Genetic and Evolutionary Computation Conference.

88 He, K., Zhang, X., Ren, S., & Sun, J., (2016), *Deep residual learning for image recognition*, In Proceedings of the IEEE conference on computer vision and pattern recognition, pp. 770-778.

89 Huang, G., Liu, Z., Van Der Maaten, L., & Weinberger, K. Q., (2017), *Densely connected convolutional networks*, In Proceedings of the IEEE conference on computer vision and pattern recognition, pp. 4700-4708.

90 Zoph, B., Vasudevan, V., Shlens, J., & Le, Q. V., (2018), *Learning transferable architectures for scalable image recognition*, In Proceedings of the IEEE conference on computer vision and pattern recognition, pp. 8697-8710.

91 James Bergstra, Dan Yamins, and David D. Cox., (2013), *Making a science of model search: Hyperparameter optimization in hundreds of dimensions for vision architectures*, In ICML.

92 Barret Zoph and Quoc V. Le., (2017), *Neural architecture search with reinforcement learning*, In International Conference on Learning Representations.

93 Alex Krizhevsky, (2009), *Learning Multiple Layers of Features from Tiny Images*, http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.222.9220&rep=rep1&type=pdf

94 https://www.who.int/docs/default-source/coronaviruse/situation-reports/20200401-sitrep-72-covid-19.pdf?sfvrsn=3dd8971b_2).

95 Neilan, R. M., & Lenhart, S., (2011), *Optimal vaccine distribution in a spatiotemporal epidemic model with an application to rabies and raccoons*, Journal of mathematical analysis and applications, 378(2), pp. 603-619.

96 Pitzer, V. E., Viboud, C., Simonsen, L., Steiner, C., Panozzo, C. A., Alonso, W. J., ... & Grenfell, B. T., (2009), *Demographic variability, vaccination, and the spatiotemporal dynamics of rotavirus epidemics*, Science, 325(5938), pp. 290-294.

97 Xi, G., Yin, L., Li, Y., & Mei, S., (2018, November), *A deep residual network integrating spatial-temporal properties to predict influenza trends at an intra-urban scale*, In Proceedings of the 2nd ACM SIGSPATIAL International Workshop on AI for Geographic Knowledge Discovery, pp. 19-28.

98 Paul, S.K., Jana, S. & Bhaumik, P., (2020), *A Multivariate Spatiotemporal Model of COVID-19 Epidemic Using Ensemble of ConvLSTM Networks*, J. Inst. Eng. India Ser. B., 102(6), pp. 1137-1142

99 Paul, S.K., Jana, S. & Bhaumik, P., (2021), *Explaining Causal Influence of External Factors on Incidence Rate of Covid-19*, SN COMPUT. SCI. 2**,** 465.

100 Hochreiter, S., (1998), *The vanishing gradient problem during learning recurrent neural nets and problem solutions*, International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems, 6(02), pp. 107-116.

101 Hochreiter, S., & Schmidhuber, J., (1997), *LSTM can solve hard long time lag problems*, In Advances in neural information processing systems, pp. 473-479.

102 Xingjian, S. H. I., Chen, Z., Wang, H., Yeung, D. Y., Wong, W. K., & Woo, W. C., (2015), *Convolutional LSTM network: A machine learning approach for precipitation nowcasting,* In Advances in neural information processing systems, pp. 802-810.

103 Milewski, B., (2018), *Category theory for programmers*, https://cdn.jsdelivr.net/gh/it-ebooks-0/it-ebooks-2018-11to12/Category%20Theory%20for%20Programmers.pdf

104 Nivel, E., Thórisson, K. R., Steunebrink, B. R., Dindo, H., Pezzulo, G., Rodriguez, M., ... & Jonsson, G. K., (2013), *Bounded recursive self-improvement*, arXiv preprint arXiv:1312.6764

105 Hall, J. S., (2007), *Self-improving AI: An analysis*, Minds and Machines, 17(3), pp. 249-259.

106 Everitt, T., Filan, D., Daswani, M., & Hutter, M., (2016, July), *Self-modification of policy and utility function in rational agents*, In International Conference on Artificial General Intelligence, Springer, Cham, pp. 1-11.

107 Kaiser, L., Gomez, A. N., Shazeer, N., Vaswani, A., Parmar, N., Jones, L., & Uszkoreit, J., (2017), *One model to learn them all,* arXiv preprint arXiv:1706.05137.

108 Paul, S. K. & Bhaumik, P. (Jan 2022), "*Disaster Management through Integrative AI*", In proceedings of ICDCN 2022 (pp. – 290-293). https://doi.org/10.1145/3491003.3493235

109 Reed, Scott et. al., (Nov 2022), *"A Generalist Agent"*, TMLR, https://openreview.net/forum?id=1ikK0kHjvj

110 OpenAI, (Mar 2023), "*GPT-4 Technical Report*", https://arxiv.org/abs/2303.08774

111 Liang, Yaobo et. al., (Mar 2023), "*TaskMatrix.AI: Completing Tasks by Connecting Foundation Models with Millions of APIs*", https://arxiv.org/pdf/2303.16434.pdf

112 Paul, S. K. & Bhaumik, P. (2021, December), "*Towards formalization of constructivist seed AI*". In ICCIS 2021, Springer, pp. 61-78