Design and Development of Universal Search based Problem Solver using Functional Graph Programming Language

A Thesis Submitted in Partial Fulfilment of the requirements for the

Degree of Master of Engineering in Software Engineering

by

PALLABI CHATTERJEE

Class Roll Number: 001711002016

Exam Roll Number: M4SWE19002

Registration Number: 140972 of 2017-2018

under the supervision of

DR. PARAMA BHAUMIK

Associate Professor

Department of Information Technology
Jadavpur University

DEPARTMENT OF INFORMATION TECHNOLOGY

FACULTY OF ENGINEERING AND TECHNOLOGY

JADAVPUR UNIVERSITY
Salt Lake Campus, Kolkata - 7000098

MAY 2019

DEPARTMENT OF INFORMATION TECHNOLOGY

FACULTY OF ENGINEERING AND TECHNOLOGY

JADAVPUR UNIVERSITY

<u>CERTIFICATE OF SUBMISSION</u>

I hereby recommend that the thesis, entitled **"Design and Development of Universal Search based Problem Solver using Functional Graph Programming Language"**, prepared by Pallabi Chatterjee (Registration Number: 140972 of 2017-2018) under my supervision, be accepted in partial fulfilment of the requirements for the degree of Master of Engineering in Software Engineering from the Department of Information Technology under Jadavpur University.

Date:

Dr. Parama Bhaumik
Associate Professor
Department of Information Technology
Jadavpur University

Countersigned by:

Head of the Department
Information Technology
Jadavpur University

Dean
Faculty of Engineering and Technology
Jadavpur University

DEPARTMENT OF INFORMATION TECHNOLOGY

FACULTY OF ENGINEERING AND TECHNOLOGY

JADAVPUR UNIVERSITY

<u>CERTIFICATE OF APPROVAL</u>

(Only in case the Thesis is approved)

The thesis at instance is hereby approved as a creditable study of an Engineering subject carried out and presented in a manner satisfactory to warrant its acceptance as a prerequisite to the degree for which it has been submitted. It is understood that by this approval the undersigned does not necessarily endorse or approve any statement made, opinion expressed or conclusion drawn therein, but approve this thesis for the purpose for which it is submitted.

Signature of the Examiner

Signature of the Superviser
Dr. Parama Bhaumik
Associate Professor
Department of Information Technology
Jadavpur University

Date:

DECLARATION OF ORIGINALITY AND COMPLIANCE OF ACADEMIC ETHICS

I hereby declare that this thesis contains literature survey and original research work done by me, as a part of my Master of Engineering in Software Engineering studies.

All information in this document have been obtained and presented in accordance with academic rules and ethical conduct.

I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name: PALLABI CHATTERJEE
Class Roll No.: 001711002016
Examination Roll No.: M4SWE19002
Registration No.: 140972 of 2017-2018
Thesis Title: *Design and Development of Universal Search based Problem Solver using Functional Graph Programming Language*

Signature with date

ACKNOWLEDGEMENTS

I would like to express my heartiest gratitude to my supervisor and guide **Dr. Parama Bhaumik** (Associate Professor, Department of Information Technology, Jadvpur University) for her guidance and support during the entire course of my post graduate studies and ME thesis work. Her encouraging words and positive reinforcements helped me to remain confident in myself and motivated me to keep working towards my thesis when I felt lost. The constructive and insightful discussions that I have had with her over the past one year, she inspired me to complete this work successfully. I have always been benefited by her constant enthusiasm and zeal for research. It is certain the project would not have achieved the success it did without her help and support. It has been a privilege to have her as my mentor.

I would also like to thank **Swarna Kamal Paul** (PhD Scholar, Department of Information Technology,Jadvpur University), who was involved deeply with the validation of the project and developed the idea of it.

My sincere obligation goes to **Dr. Bhaskar Sardar**, Head of the Department, Department of Information Technology, Jadavpur University for his support. My whole hearted thanks to the staffs of the Department of Information Technology, Jadavpur University, and my friends for their help.

I would also like to thanks Jadavpur University, for giving me this opportunity to pursue this ME program including this research. I also would like to thank all of my friends for offering words of encouragement and support in times when I most needed it.

Finally, heartfelt thanks to my Parents and the Almighty for showing me the right direction out of the blue, to help me stay calm in the oddest of the times and keep moving even at times when there was no hope.

Regards,

Pallabi Chatterjee

ABSTRACT

KEYWORDS: Universal Search; Reinforcement Learning; Positive and Negative Reward; Graph Programming Language; Metasearcher Algorithm; Probability; Message Passing Interface (MPI); Cart-pole Problem solver.

We have designed a universal search-based problem solver using Graph Programming for problems where for a given problem statement in the problem space, we can find out the best probable and optimized method to reach the destination or goal state. For this work, we have used levin's search approach. Functional Graph Programming Language is designed previously with some basic features. Here, in this work, we have introduced some new functional inputs to implement some balancing problems (i.e. cartpole problem in our case). To train the model we have used Reinforcement learning to help the agent learn by trial and error method awarding positive reward for correct evaluation and negative reward for wrong finding. The incremental learning concept is introduced to update the program probability of each subgraph to utilize the reward concept. We have also worked on stand alone machine as well as in distributed environment environment to check it's effectiveness in master slave architecture.We have used MPI (Message Passing Interface) to approach the distributed environment so the performance could get enhanced and achieved better time bound results in distributed methodology.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

Introduction

Universal or Levin Search is an algorithm for solving inversion problems in asymptotically optimal way. Universal Search is the asymptotically fastest way of finding a program that calculates a solution to a given problem, provided nothing is known about the problem except that there is a fast way of verifying solutions (Schaul and Schmidhuber, 2010). The algorithm has the property that the total time taken to find a solution is $O(t^*)$, where t^* is the time used by fastest program p^* to compute the solution. The search time of the whole process is at most a constant factor larger than t^*; typically this depends on the encoding length of p^*. The algorithm itself is very simple: It consists in running all possible programs in parallel, such that the fraction of time allocated to program p is $2^{-l(p)}$, where $l(p)$ is the size of the program.

AI problem can be entitled to a resource bounded prediction problem. In order to maximize future expected rewards, the agent has to predict optimal actions in a given environment (Paul and Bhaumik, communicated), (Legg and Hutter, 2007). Universal search if combined with reinforcement learning can create an effective learning mechanism for an agent. It guarantees asymptotic optimality in finding models for generating optimal sequence of actions proportionate to rewards gained in a certain environment. Reinforcement learning is about taking suitable action to maximize reward in a particular situation. It is employed by various software and machines to find the best possible path it should take in a specific situation. Incremental learning is a machine learning paradigm where the learning process takes place whenever new example(s) or new attribute(s) merge or must be deleted from dataset and the solutions already obtained are only modified.

In case of universal search, to diminish the exponential factor the guiding probability distribution in the sense mentioned by (Solomonoff, 2010) needs to be stored and updated properly. Keeping this objective in mind a Functional Graph Programming Language (FGPL) is designed, based on dataflow graph to be used by universal search for solution program generation. Generated source code is represented as directed acyclic computational graph. Representing program as a dataflow graph helps devising better transfer learning process for transferring more meaningful information from one task to another thereby reducing the complexity of subsequent related tasks.

There have been several dataflow graph based languages already developed for practical purposes. One can follow the survey article by Johnston et. al. (Johnston et. al. 2004) for a comparative study on many such languages. However most of these languages were developed either with the intention of achieving implicit parallelization of code or allowing programmer to graphically construct a program with ease or both. For application in universal search we identified the necessity of following features in a language. A language where programs can generate another program dynamically which can be reused multiple times and called recursively. A language where programs can be auto generated and stored by another application with ease and it is not expected that a human coder will frequently write a program in this language. A simple but powerful syntax to construct programs such that constructing every syntactically correct program should be straightforward and generation of syntactically incorrect programs can be easily avoided. But at the same time the language should be expressive enough to act on wide variety of environments. A simple but powerful syntax to construct programs such that constructing every syntactically correct program should be straightforward and generation of syntactically incorrect programs can be easily avoided. But at the same time the language should be expressive enough to act on wide variety of environments.

Reasoning about the programs can be done without much complications mainly to handle the problem of over representation in the sense mentioned by Looks and (Looks and Goertzel, 2009). The program constructs itself should have some added features like storing and managing conditional probability distribution, interrupting programs based on run time etc. To handle all these scenarios, a dataflow Functional Graph Programming Language(FGPL) has been developed specifically to be used in universal search based general problem solving agent. Hence featuring this FGPL to further levels and to test invert pendulum problems such as cartpole problem by universal search based AI agent or solver.

CHAPTER 2

Literature Survey

Hutter's *HSearch algorithm* combines Universal Search in program space with simultaneous search for proofs about time bounds on their runtime (Hutter, 2007). The algorithm is asymptotically optimal, but replaces the multiplicative slowdown by an additive one. It may be significantly faster than Universal Search for problems where the time taken to verify solutions is nontrivial. The additive constant depends on the problem class, however, and may still be huge. A way to dramatically reduce such constants in some cases is a universal problem solver called the Godel Machine.

Functional programming paradigm eliminates usage of variables and mutable data, thus preventing side effects. Bird and Wadler (Bird *et al.*, 1998) provided a good introduction to functional programming. In this paradigm every program is represented as a composition of functions. Functions without side effect are much easier to compose, evaluate and reason about. Due to explicit composition instead of implicit communication among functions every description of communication among functions is a part of the program created which helps is easier composition of semantically correct programs. In generate and test method where same functions might need to be reused in multiple programs or multiple times in a same program, evaluation becomes easier due to referential transparency where the function can be directly replaced by its output value for same inputs. Implicit parallelism can be achieved among independent functions. Unlike imperative programming, functional programming does not deal with state transitions (Wadler, 1995). It is more like a declarative style of programming and works by evaluating functions on arguments which often produces shorter codes compared to imperative counterpart. This is clearly beneficial in universal search where problem complexity is considered as proportional to solution size.

2.1 Related Concept

2.1.1 Universal Search

Universal search is devised by Leonid A. Levin (1973,1984)(Levin, 1973) related to Levin Complexity, a computable, time bounded version of algorithmic complexity.

If there exists a program p, of length $l(p)$, that can solve the problem in $time(p)$, Universal Search will solve the problem in a time $2^{l(p)+1} + time(p)$ at most. This exponential growth of computational cost in the algorithmic complexity $l(p)$ of the fastest solver makes practical applications of Universal Search problematic.

2.1.2 Algorithmic information theory (AIT)

It is the information theory of individual objects, using computer science, and concerns itself with the relationship between computation, information, and randomness. The information content or complexity of an object can be measured by the length of its shortest description (Hutter, 2007).

"010101010101010101010101010101" is a string which has a short description of "15 repetitions of 01" but "1100100001100001110111101110" has no such short description.

2.1.3 Algorithmic Kolmogorov Complexity (AC)

The Algorithmic "Kolmogorov" Complexity (AC) of a string is defined as the length of the shortest program that computes or outputs where the program is run on some fixed reference universal computer.

Kolmogorov defined the complexity of a string x as the length of its shortest description p on a universal Turing machine U

$$K(x) = \min\{l(p) : U(p) = x\} \tag{2.1}$$

A string is simple if it can be described by a short program, like "the string of one million ones", and is complex if there is no such short description, like for a random string whose shortest description is specifying it bit-by-bit.

Kolmogorov complexity is a key concept in (algorithmic) information theory (Li and Vitányi, 2013). An important property of K is that it is nearly independent of the choice of U.

2.1.4 Algorithmic Solomonoff Probability (AP)

Solomonoff (1964) considered the probability that a universal computer outputs some string when fed with a program chosen at random. This Algorithmic "Solomonoff" Probability (AP) is key in addressing the old philosophical problem of induction in a formal way. It is based on

- Occam's razor (choose the simplest model consistent with the data),
- Epicurus' principle of multiple explanations (keep all explanations consistent with the data),
- Bayes's Rule (transform the a priori distribution to a posterior distribution according to the evidence, experimentally obtained data),
- Universal Turing machines (to compute, quantify and assign codes to all quantities of interest), and
- Algorithmic complexity (to define what simplicity / complexity means).

Solomonoff defined the closely related universal a priori probability $M(x)$ as the probability that the output of a universal Turing machine U starts with x when provided with fair coin flips on the input tape (Solomonoff, 1989). M can be used as a universal sequence predictor that outperforms (in a certain sense) all other predictors.

2.1.5 Levin Complexity

Levin complexity of a string s can be defined as the minimum value of the sum of the length of the program p that computes s and the logarithm of it's runtime. We can test it using Turing machine U. Levin complexity is useful for some programs which might not halt and process might never get completed.

$$Kt_U(s) = \min\{|p| + \log t(p,s) : f_{U(p)=s}\} \tag{2.2}$$

We can start testing all programs starting from lowest Levin complexity. This means testing all programs with decreasing program probability. To have a time bound solution even if some programs don't halt they can be pre-empted when their run time exceeds their time bound complexity measure. Total search time for a solution program p is bounded by $2^{Kt(s)} = 2^{|p|}t(p,s)$.

2.1.6 Universal prefix Turing machine

A prefix Turing machine is one unidirectional input tape (read only), one unidirectional output tape (write only), and some bidirectional work tapes (initially filled with zeros).

Turing machine T halts on input p with output x, and write $T(p) = x$ if p is to the left of the input head and x is to the left of the output head after T halts. The set of p on which T halts forms a prefix code. Such codes p are called self-delimiting programs. There exists a universal prefix Turing machine U which simulates prefix Turing machine T_i with input $y'q$ if fed with input $y'i'q$, i.e.

$$U(y'i'q) = T_i(y'q) \quad \forall i, q \tag{2.3}$$

where, $x' = \langle x \rangle$ is a prefix code of x with $l(x') \leq l(x) + 2\log l(x) + O(1)$,
U is the reference universal Turing machine,
prefix Kolmogorov complexity is defined as the shortest program p, for which the universal prefix Turing machine U outputs x (given y)

$$K(x) := \min\{l(p) : U(p) = x\} \tag{2.4}$$

and

$$K(x|y) := \min\{l(p) : U(y'p) = x\} \tag{2.5}$$

2.1.7 Reinforcement Learning

Reinforcement learning is based on how to map situations to actions to maximize a numerical reward signal. The learner is not told which actions to take, but instead must discover which actions will gather them most rewards. The important part is actions may affect not only the immediate reward but also the next situation and all subsequent rewards.

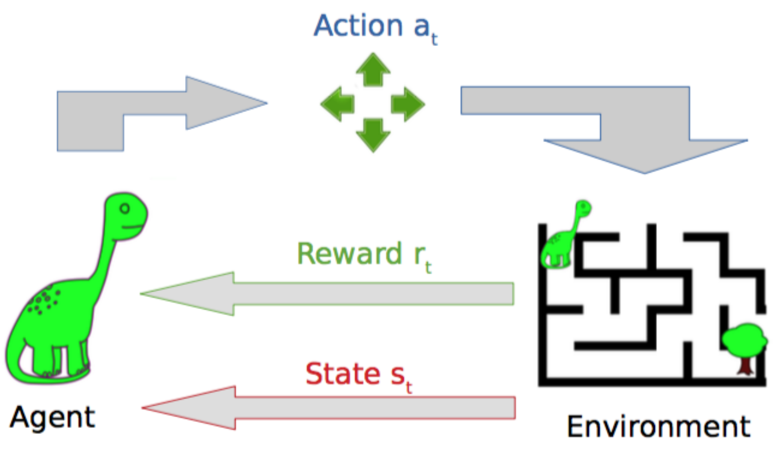


Figure 2.1: Interaction between agent and environment in Reinforcement Learning.

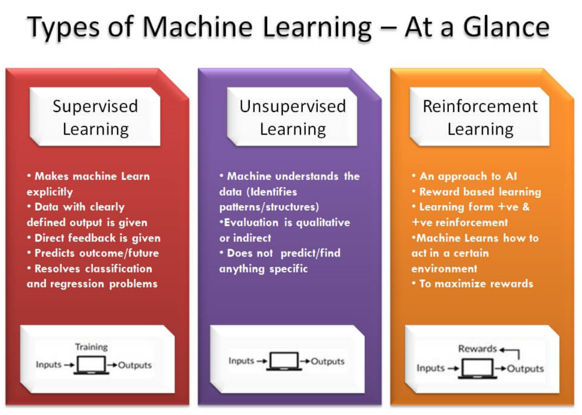
- A learning agent must be able to sense the state of its environment to some extent and must be able to take actions that affect the state. The agent also must have a goal or goals relating to the state of the environment. Markov decision processes are intended to include just the three aspects *sensation, action,* and *goal* in their simplest possible forms without trivializing any of them. Any method that is well suited to solving such problems we consider to be a reinforcement learning method.

- **Trial-and-error search** and **delayed reward** are the two most important distinguishing features of reinforcement learning.

- Reinforcement learning takes **the opposite tack, starting with a complete, interactive, goal-seeking agent**. All reinforcement learning agents have explicit goals, can sense aspects of their environments, and can choose actions to influence their environments. Moreover, it is usually assumed from the beginning that the agent has to operate despite significant uncertainty about the environment it faces.

- Reinforcement learning is part of a decades long trend within artificial intelligence and machine learning **toward greater integration with statistics, optimization, and other mathematical subjects**. For example, the ability of some reinforcement learning methods to learn with parameterized approximators addresses the classical "curse of dimensionality" in operations research and control theory.

- Formalizing the problem of reinforcement learning using ideas from dynamic decision systems theory known as Markov decision process (MDP). The environment is typically formulated as a Markov decision process (MDP). The main difference between the classical dynamic programming methods and reinforcement learning algorithms is that the latter do not assume knowledge of an exact mathematical model of the MDP and they target large MDPs where exact methods become infeasible.


Types of Machine Learning – At a Glance

Supervised Learning

• Makes machine Learn explicitly
• Data with clearly defined output is given
• Direct feedback is given
• Predicts outcome/future
• Resolves classification and regression problems

Unsupervised Learning

• Machine understands the data (Identifies patterns/structures)
•Evaluation is qualitative or indirect
• Does not predict/find anything specific

Reinforcement Learning

• An approach to AI
• Reward based learning
• Learning form +ve & +ve reinforcement
•Machine Learns how to act in a certain environment
• To maximize rewards

Figure 2.2: Supervised, Unsupervised and Reinforcement Learning.

- Reinforcement learning is considered as one of three machine learning paradigms, alongside supervised and unsupervised learning (refer fig. 2.2).

Examples of Reinforcement Learning

A good way to understand reinforcement learning is to consider some of the examples and possible applications that have guided its development.

- A master chess player makes a move. The choice is informed both by planning, anticipating possible replies and counter replies and by immediate, intuitive judgements of the desirability particular positions and moves (Sutton *et al.*, 1998).
- An adaptive controller adjusts parameters of a petroleum refinery's operation in real time. The controller optimizes the yield/cost/quality trade-off on the basis of specified marginal costs without sticking strictly to the set points originally suggested by engineers.
- A gazelle calf struggles to its feet minutes after being born. Half an hour later it is running at 20 miles per hour.
- A mobile robot decides whether it should enter a new room in search of more trash to collect or start trying to find its way back to its battery recharging station. It makes its decision based on the current charge level of its battery and how quickly and easily it has been able to find the recharger in the past.

Types of Reinforcement: Positive and Negative

Positive Reinforcement is defined as when an event, occurs due to a particular behaviour, increases the strength and the frequency of the behaviour or it has a positive effect on the behaviour.

Negative Reinforcement is defined as strengthening of a behaviour because a negative condition is stopped or avoided.

	Positive Reinforcement Learning	Negative Reinforcement Learning
Advantages	• Maximizes Performance • Sustain Change for a long period of time	• Increases behaviour • Provide defiance to minimum standard of performance
Disadvantages	Too much Reinforcement can lead to overload of states which can diminish the results.	It only provides enough to meet up the minimum behaviour.

Monte Carlo vs TD Learning methods

We have two ways of learning:

Monte Carlo Learning: When the episode ends (the agent reaches a “terminal state”), the agent looks at the total cumulative reward to see how well it did. Rewards are only received at the end of the game. Then, we start a new game with the added knowledge. The agent makes better decisions with each iteration.

Temporal Difference (TD) Learning : Learning at each time step. TD Learning, on the other hand, will not wait until the end of the episode to update the maximum expected future reward estimation: it will update its value estimation V for the non-terminal states S_t occurring at that experience. This method is called TD(0) or one step TD (update the value function after any individual step). TD methods only wait until the next time step to update the value estimates. At time $t + 1$ they immediately form a TD target using the observed reward $R_t + 1$ and the current estimate $V(S_t + 1)$.

Deep Reinforcement Learning

Deep Reinforcement Learning introduces deep neural networks to solve Reinforcement Learning problems, hence the name “deep”.

Various Practical applications of Reinforcement Learning

1. RL can be used in robotics for industrial automation.
2. RL can be used in machine learning and data processing.
3. RL can be used to create training systems that provide custom instruction and materials according to the requirement of students.

2.1.8 Cartpole Problem

Cartpole, known also as an Inverted Pendulum, is a pendulum with a center of gravity above its pivot point. It's unstable, but can be controlled by moving the pivot point under the center of mass. The goal is to keep the cartpole balanced by applying appropriate forces to a pivot point. A pole is attached by an un-actuated joint to a cart, which moves

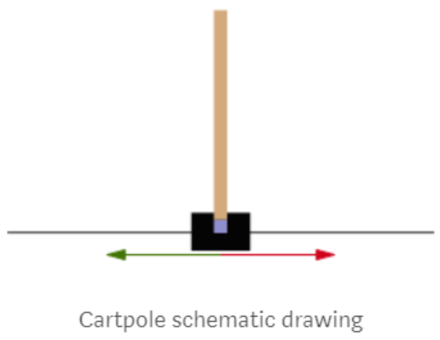


Figure 2.3: Cartpole schematic drawing. Violet square indicates a pivot point. Red and green arrows show possible horizontal forces that can be applied to a pivot point.

along a frictionless track. The system is controlled by applying a force of $+1$ or -1 to the cart. The pendulum starts upright, and the goal is to prevent it from falling over. A reward of $+1$ is provided for every time-step that the pole remains upright. The episode ends when the pole is more than 15 degrees from vertical, or the cart moves more than 2.4 units from the center (Chan, 2016).

- The Cart-Pole world consists of a cart that moves along the horizontal axis and a pole that is anchored on the cart. At every time step, you can observe **its position** (x)**, velocity** (x_{dot})**, angle** (θ)**, and angular velocity** (θ_{dot}). These are the observable states of this world. At any state, the cart only has two possible actions: move to the left or move to the right.

- In other words, the state-space of the Cart-Pole has four dimensions of continuous values and the action-space has one dimension of two discrete values.

- In this environment, a reward is given as long as the pole is still somewhat upright and the cart is still within the bound. An episode is over as soon as the pole falls beyond a certain angle or the cart strays too far off to the left or right. The problem is considered "solved" when it stays upright for over 195 time steps, 100 times consecutively.

Finding the optimal policy

A policy is simply the action that can be taken at a given state. Here, to find the policy that can maximize reward or wanting to find the optimal policy for each possible state. **Q-Learning** is a method of finding these optimal policies. Essentially, through trials-and-errors, we can find a Q-value for each state-action pair. This Q-value represents the desirability of an action given the current state. Over time, if the world is static, the Q-values would converge and the optimal policy of a given state would be the action with the largest Q-value.

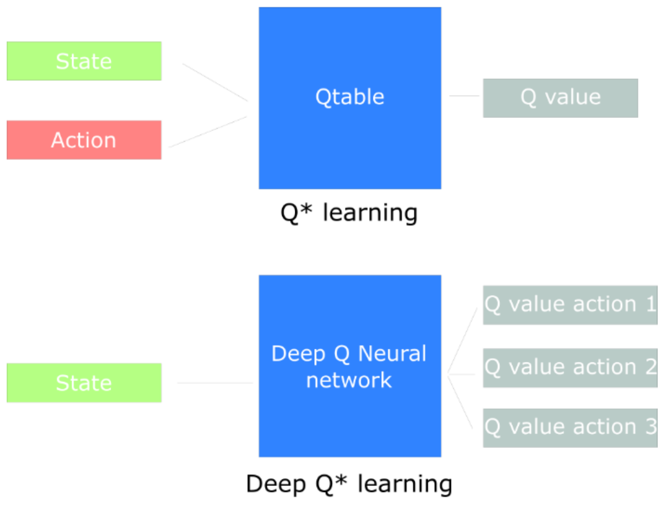


Figure 2.4: Block diagram of Q^* learning and Deep Q^* learning.

2.1.9 Functional Graph Programming Language (FGPL)

One aspect of intelligence is, how well an agent can dampen the exponential factor in the course of time by achieving experience on solving set of related problems in a problem class (Erwig, 2001). In case of universal search, to dampen the exponential factor the guiding probability distribution in the sense mentioned by R.J Solomonoff (2010) needs to be stored and update properly. Effective reuse of previous successful solutions as a whole or by part and pruning functionally equivalent programs can also help in dampening the search space and thus the search time. Keeping this objective in mind a FGPL is designed previously based on dataflow graph to be used by universal search for solution program generation. We are working on it's framework to add some more features as per need to solve various problem statements in the problem domain effectively.

Why FGPL?

There have been several dataflow graph based languages already developed for practical purposes. There is the survey article by Johnston et. al. (Johnston et. al. 2004) for a comparative study on many such languages. However most of these languages were developed either with the intention of achieving implicit parallelization of code or allowing programmer to graphically construct a program with ease or both. For application in universal search we identified the necessity of following features in a language.

- A language where programs can generate another program dynamically which can be reused multiple times and called recursively.
- A language where programs can be autogenerated and stored by another application and it is not expected that a human coder will frequently write a program in this language.
- A simple but powerful syntax to construct programs such that constructing every syntactically correct program should be straightforward and generation of syntactically incorrect programs can be easily avoided.
- The language should be expressive enough to act on wide variety of environments. Reasoning about the programs can be done without much complications mainly to handle the problem of over representation.
- The program constructs itself should have some added features like storing and managing conditional probability distribution, interrupting programs based on run time etc.

Functional programming paradigm **eliminates usage of variables and mutable data, prevents side effects**. Bird and Wadler (Bird *et al.*, 1998) provided a good introduction to functional programming. In this paradigm every program is represented as a composition of functions. Functions without side effect are much easier to compose, evaluate and reason about.

Due to explicit composition instead of implicit communication among functions every description of communication among functions is a part of the program created which helps **is easier composition of semantically correct programs**.

In generate and test method where same functions might need to be reused in multiple programs or multiple times in a same program, evaluation becomes easier due to **referential transparency** where the function can be directly replaced by its output value for same inputs.

Implicit parallelism can be achieved among independent functions. Unlike imperative programming, functional programming does not deal with state transitions. It is more like a declarative style of programming and works by evaluating functions on arguments which often produces shorter codes compared to imperative counterpart (Hu et. al. 2015). This is clearly beneficial in universal search where problem complexity is considered as proportional to solution size.

To implement universal search a **metasearcher** can be created which generates and tests program in the newly proposed language FGPL. It **searches for a solution program for a given problem environment in the program space**. To dampen the growth rate of search space, type compatibility checking is done before extension of a program and program pruning is applied based on functional equivalence of programs.

2.1.10 Message Passing Interface (MPI) for Distributed environment

Message passing is a programming paradigm used widely on parallel computer architectures and networks of workstations. One of the attractions of the paradigm is that it will not be made obsolete by architectures combining the shared- and distributed-memory views, or by increases in network speeds.

Why MPI?

We have trained a model providing some small goals to check their time taken to reach by trial and error method in an individual machine. Now for huge dataset or problem statement we need to try distributed environment engaging master slave architecture to reduce time and improve performance.

MPI Basics

Although MPI is a complex and multifaceted system, we can solve a wide range of problems using just six of its functions, which initiate and terminate a computation, identify processes, and send and receive messages:

`MPI_INIT()` : Initiate an MPI computation.

`MPI_FINALIZE()` : Terminate and shut down a computation.

`MPI_COMM_SIZE(comm, size)` : Determine number of processes in a computation.

`MPI_COMM_RANK(comm, pid)`: Determine the process identifier of the current process.
`MPI_SEND(buf, count, datatype, dest, tag, comm)`: Send a message.
`MPI_RECV(buf, count, datatype, source, tag, comm, status)`: Receive a message.

MPI Python implementations include: pyMPI, mpi4py, pypar, MYMPI and the MPI submodule in ScientificPython. pyMPI is notable because it is a variant python interpreter, while pypar, MYMPI, and ScientificPython's module are import modules. They make it the coder's job to decide where the call to `MPI_INIT` belongs.

2.2 Related Work

The Optimal Ordered Problem Solver (OOPS) incrementally searches a space of programs that may reuse programs solving previously encountered problems. OOPS was able to learn universal solvers for the Tower of Hanoi puzzle in a relatively short time, a problem that the other learning algorithms have repeatedly failed to solve. In a probabilistic variant of Universal Search called Probabilistic Search uses a language with a small but general instruction set to generate neural networks with exceptional generalization properties.

Levin search has optimal order of complexity (total search time equals to the time to execute and verify the solution multiplied by a constant factor) yet the constant factor associated with it is exponential in nature with respect to solution size. Thus Levin search is practically effective only if there is a shorter implementation of the solution is available (Paul and Bhaumik, 2016). Hutter search (Hutter, 2002) combines Universal search in program space (solution implementation) with simultaneous search for proofs and optimally distributes resources between execution of provably correct programs and enumeration of all proofs. It has been able to reduce the constant multiplicative factor drastically but introduced an additive constant which still may be a large one.

Schmidhuber made a practical implementation of a probabilistic version of Levin search for the first time. The probabilistic search algorithm randomly generates programs written in a general assembler-like programming language based on sequences of integers (Schmidhuber, 2004). Each program computes a solution candidate which is tested on the training data. While generating and testing programs, candidates with low Levin complexity are preferred over high Levin complexity. An incremental version of the Levin search called as Adaptive Levin search uses experiences collected from solutions of previous problems to adaptively modify Levin search by modifying underlying probability distributions of solution programs. Whenever Levin search finds a solution program for a given problem the probabilities of its corresponding instructions are increased proportionately. Thereafter for subsequent related problems the Levin search runs with the modified probability distribution. This is called as bias shifts or changes in learner's policy.

Making Universal Search Practical

The more domain knowledge we have, the more we can shape or restrict the space of programs we need to search. Here we make Universal Search practically useful by devising a domain-specific language that encodes plausible (according to prior knowledge) programs by relatively few bits, thus reducing the slowdown factor to an acceptable size.

Dropping assumptions

Universal Search makes a number of assumptions about the language L. We will keep the assumption that L is a prefix-free binary code, and drop the following ones:

- L is Turing-complete.
- Every encoding corresponds to a valid program.
- L is infinite.

This does not mean that the opposites of those assumptions are true, only that they are not necessarily true (L is still allowed to be infinite or Turing-complete). Thus, for Practical Universal Search (PUnS), L can encode an arbitrary set of programs, all of which can be domain-specific.

Marcus Hutter made a serious attempt in solving the problem of AGI by combining Solomonoffs theory of universal induction and decision theory (Hutter M. 2003). 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 known probability distribution. AIXI can optimally deal with these environments by using Solomonoff's universal prior probability as the distribution for 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 $t \cdot 2^l$.

Schmidhuber made an efficient practical implementation of universal search for solving an ordered set of problems (Schmidhuber 2004) 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. 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 resettable environment only.

Godel machine (Schmidhuber 2003, 2007) is a self-referential, self-improving optimal problem solver originally proposed by Jurgen Schmidhuber. The system starts with an initial problem-solving code which interacts with the environment and a proof searcher. 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. A probable implementation roadmap was given using continuous passing style (CPS) of programming and meta-circular evaluators (Thórisson *et al.*, 2014).

To alleviate the problem of combinatorial explosion in program space for universal search a conceptual program representation scheme was proposed by Moshe Looks and Ben Goertzel (Looks and Goertzel, 2009). A normal form program representation was presented with several transformation rules to alleviate the problem of over representation. Maximizing correlation between syntactic and sematic distance among programs will help minimizing chaotic execution and managing resource variance. It is claimed that the proposed program representation and transformations increases the correlation between syntactic and semantic distance.

Some of the difficulties in the design of graph algorithms can be overcome by using a functional language. Burton and Yang (Burton and Yang, 1990) using a lazy functional language represent their graphs by heaps. The heaps are implemented with balanced binary trees. The heaps are also used for holding visited markings on vertices, which leads to having logarithmic time graph traversal. One drawback of this is that each function must take a heap and return an updated heap. Kashiwagi and Wise (Kashiwagi and Wise, 1991)'express their graph algorithms in Haskell. To overcome the problem of requiring side effects they present graph algorithms as the fixed point of a set of recursive equations. The recursive equations are derived directly from the formal specification of the problem.

Dataflow graph based programming has been a major research topic around 70s. A dataflow application is then a composition of processing blocks, with one or more initial source blocks and one or more ending blocks, linked by a directed edge. **SISAL** (Burns and Gaudiot, 2002), (Feo *et al.*, 1990)) is a functional programming language with Pascal like syntax and based on dataflow graph. Source codes are translated to executable dataflow graphs with the intent of achieving implicit parallelism over multiple parallel paths across the generated program graph. **GPL** (Graphical Programming Language) (Davis et. al. 1981) was one of the earliest practical visual dataflow graph based programming language where program can be designed directly as a graph. Each node in the graph can be an atomic node or can be a subgraph. **Prograph** (Cox et. al. 1989) is another graphical programming language based on dataflow graph and object-oriented programming. The methods of objects are defined as dataflow diagrams (Johnston et. al. 2004). **Tensorflow** (Abadi et. al. 2016) is a dataflow graph based computation model which follows an imperative style of programming. It is being used in machine learning research with a focus on training on deep neural networks. Its power lies in parallelizing computation over multicore CPUs and general-purpose GPUs.

CHAPTER 3

Background

3.1 Functional Graph Programming Language (FGPL)

Graphs are a fundamental data structure in computer science because a lot of problems can be modelled with them. Graphs are used to model the relationships among a collection of objects that provide a direct, intuitive and mathematically precise way of describing complex structures. FGPL can be described as an accumulation of operators in terms of nodes and edges to perform operations provided by the environment. The nodes are primitive functions which needs to be connected through edges (another primitive function) to employ the functionality of a DAG(directed acyclic graph).

3.1.1 Nodes in FGPL

The nodes are capable of storing composite functions(both conceptual data and computational information). Nodes can have fixed number of input and output ports. Function of input data decides the order of input port. Each port consists of one connection only. Multiple connection in a single port can't be considered. Now these nodes create an hierarchy of parent-child relationship by establishing proper connection.

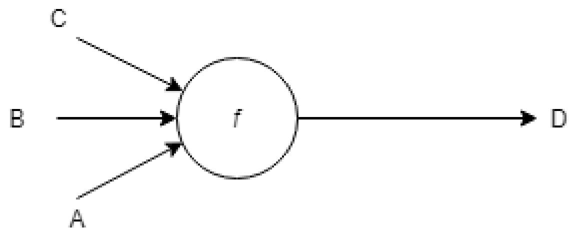


Figure 3.1: Graphical representation of function f with 2 input arguments of type A and B. Output type is C.

3.1.2 Edges in FGPL

An edge can be parameter causing connection between nodes in FGPL. In case of directed graph, an edge transmits the data from source node to the destination node. There is a condition for edge formation i.e. no self loop should get formed. The port type signature is updated after execution of the node based on the type of the data available at the port. Edges should be one directional and can be more than one between two nodes.

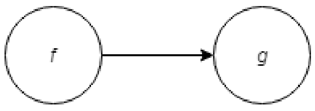


Figure 3.2: Representation of function composition of two functions in FGPL.

3.1.3 Integrating Monadic functions in FGPL

Interaction with the external environment always causes side effects. I/O monads allow these side effects without any hamper. In universal search all generated programs will start with an initial node which returns a “world” object and other I/O monads must to interact with this external environment. Thus, every program eventually becomes a monadic function. Every function in a program by default will be memorized and there will be no necessity of referring to the memo table from an FGPL program. This will not prevent side effect, but it will keep the side effect out of the FGPL program.

3.1.4 Data types of FGPL

There are 6 primitive types allowed in the system. Namely number, string, boolean, node, graph and null. There are 3 compound types available in FGPL. Namely product, sum, and function type.

Product Type

Product type is a created by ordered combination of other primitive or compound types. Product type can be created by parallel combination of functions. In FGPL, Product type can be described as the dangling output edges having same or different functions with same or different output types.

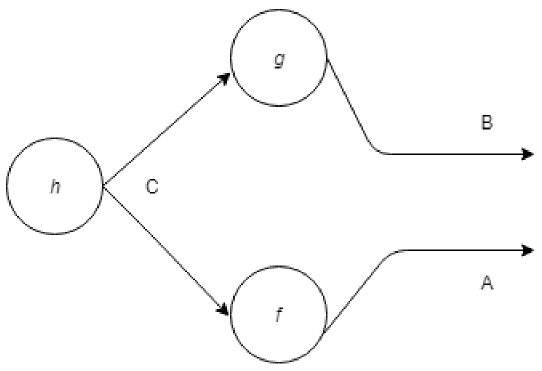


Figure 3.3: Construction of a product data type in FGPL.

Function Type

Function is a method which maps elements of a given type A to elements of type B where A represents the domain and B represents the range type of the function. A function of type $f : A \rightarrow B$ can be written as B^A which denotes there can be B^A different maps from A to B. If input type of function f is $A \times B$ and output type is C. Function type will be represented by the following notation. $f : A \times B \rightarrow C$ or $c^{A \times B}$.

Sum Type

Sum type is represented by combining functions parallelly whose output type is same but input type can be different. They can be joined together using a conditional function, namely Gaurd. Figure 3.5 represents combination of A and B sum type.

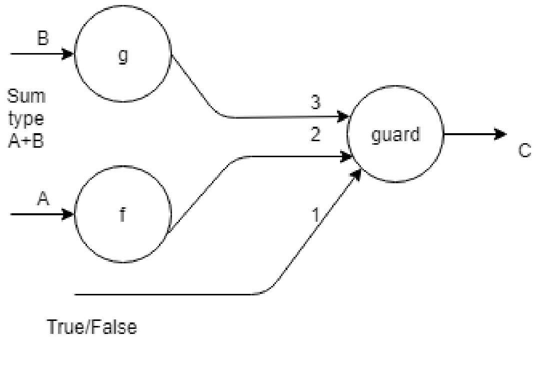


Figure 3.4: Representation of a sum data type in FGPL.

3.1.5 Functions of FGPL

1. $initGraph$: whole number $\rightarrow$ graph. $initGraph$ is the constructor of graph data type in FGPL. This can initialize a program graph. InitGraph function is used to label the newly created program graph.

$$evaluate(initGraph(a)) = \{graph_a | a \notin A,\ undefined\ otherwise\} \tag{3.1}$$

where, $graph_a$ = empty graph with label a and A = set of all graphs.

2. $addNode$: (whole number, node, *whole number) $\rightarrow$ graph. This function adds a newly created node object in an existing program graph. It takes multiple arguments. The first argument refers to an existing program 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 and can be variable.

$$\begin{aligned} evaluate(addNode(a, node, *b)) = \{&graph'_a | a \in G \wedge b \in N_a, \\ &undefined\ otherwise\} \end{aligned} \tag{3.2}$$

where, 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 and $graph'_a$ = modified graph after adding node in $graph_a$.

3. $Node$: (whole number, string, {A})$\rightarrow$node. The Node function creates a node type which can be added into a program graph. It takes two mandatory arguments and one optional argument.

$$\begin{aligned} &evaluate(Node(a, f, \{b\})) \\ &\quad = \{node^f_a | a \notin A \wedge f = \text{`}constant' \wedge b \in AnyType, \\ &\quad\quad node^f_a | a \notin A \wedge f \in \{\text{`}Sensor', \text{`}Actuator'\}, \\ &\quad\quad node^f_a | a \notin A \wedge f \in F \wedge b \in \phi, \\ &\quad\quad undefined\ otherwise\} \end{aligned} \tag{3.3}$$

4. $getSubgraph$: (whole number, whole number)$\rightarrow$ graph. The function returns the subgraph of a graph by starting from a node and recursively finds all parent nodes until initial node is reached. It takes 2 arguments. The first argument denotes the graph label and the second denotes the node label of a specific node in the graph.

$$\begin{aligned} evaluate(getSubgraph(a, b)) = \{&graph_{ab} | a \in a \wedge b \in aB, \\ &undefined\ otherwise\} \end{aligned} \tag{3.4}$$

where, $graph_{a,b}$ =subgraph of $graph_a$ and terminal node label b, aB =set of all node labels of $graph_a$ and A =set of all graph labels.

5. $evalGraph$: graph$\rightarrow$ A. The evalGraph function executes 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.

$$evaluate\Big(evalGraph(graph^f)\Big) = evaluate(graph^f) \tag{3.5}$$

3.1.6 I/O monads

- $\underline{initWorld\text{: null}\rightarrow\text{(world)}}$. The initWorld function initializes an environment. This function takes no input argument and returns an initial world object. Every valid program graph should start with this node and this is used only once in a program.

$$evaluate(initWorld) = (world) \tag{3.6}$$

- $\underline{Sensor_A \text{ : (world)}\rightarrow\text{(A,world)}}$. The $Sensor_A$ function reads some data from the environment as world object. It takes world object and returns a specific type modified world. The $Sensor_A$ function sends read request to world object for some data of type A. A Sensor node can be a child node of initWorld node or an Actuator node only.

$$evaluate(Sensor_A \circ f) = read(evaluate(f)) \tag{3.7}$$

- $\underline{Actuator_A \text{ : (A,world)}\rightarrow\text{(world)}}$. The $Actuator_A$ function takes a monadic type with a specific type A with world object. Actuator sends a write request to the world object to apply some action on the world object. Only a Sensor or a Goalchecker node can be child node of an Actuator node.

$$evaluate(Actuator_A \circ f) = write(evaluate(f)) \tag{3.8}$$

- $\underline{Goalchecker \text{ : (world)}\rightarrow\text{(boolean,world)}}$. This function checks if the goal is reached or not in the world object. The function takes world object and returns a boolean type which will contain True if goal is reached else False.

$$evaluate(Goalchecker \circ f) = check_goal(evaluate(f)) \tag{3.9}$$

3.1.7 First order node functions

- $\underline{Identity \text{ : A}\rightarrow\text{A}}$. Identity is a primitive function which takes any type as input and returns the same type as output. In FGPL, this functions is used where the inputs of multiple function need to be joined.

$$Identity \circ f = f \circ Identity = f \tag{3.10}$$

- $\underline{Constant_a \text{ : A}\rightarrow\text{a}}$. Constant function takes any type as input but returns a constant value as output.

$$evaluate(Constant_a \circ f) = (a, evaluate(f)[world]) \tag{3.11}$$

- $\underline{Add : \text{number}\times\text{number}\rightarrow\text{number}}$. *Add* is a primitive function which takes a product type of two numbers as input and produces a number type as output. It performs addition on inputs and gives the result as output. Equivalence rules are stated as below where *best_version* denotes the world object with highest local version number.

$$\begin{aligned} &evaluate(Add \circ (f \times g)) \\ &\qquad = (plus(evaluate(f)[data], evaluate(g)[data]), \\ &\qquad\quad best\_version(evaluate(g)[world], evaluate(f)[world])) \end{aligned} \tag{3.12}$$

- $\underline{Subtract : \text{number}\times\text{number}\rightarrow\text{number}}$. *Subtract* function takes product type of two numbers and returns a number type as output. It subtracts the second number in the product type from the first number.

$$\begin{aligned} &evaluate(Subtract \circ (f \times g)) \\ &\qquad = (minus(evaluate(f)[data], evaluate(g)[data]), \\ &\qquad\quad best\_version(evaluate(g)[world], evaluate(f)[world])) \end{aligned} \tag{3.13}$$

- $\underline{Multiply : \text{number}\times\text{number}\rightarrow\text{number}}$. *Multiply* function takes product type of two numbers and returns a number type as output. This operate as multiplication operation on two input numbers.

$$\begin{aligned} &evaluate(Multiply \circ (f \times g)) \\ &\qquad = (multiply(evaluate(f)[data], evaluate(g)[data]), \\ &\qquad\quad best\_version(evaluate(g)[world], evaluate(f)[world])) \end{aligned} \tag{3.14}$$

- $\underline{Divide : \text{number}\times\text{number}\rightarrow\text{number}}$. *Divide* function takes product type of two numbers and returns a number type as output. It divides the second number in the product type from the first number.

$$\begin{aligned} &evaluate(Divide \circ (f \times g)) \\ &\qquad = (divide(evaluate(f)[data], evaluate(g)[data]), \\ &\qquad\quad best\_version(evaluate(g)[world], evaluate(f)[world])) \end{aligned} \tag{3.15}$$

- $\underline{Gaurd : \text{boolean}\times\text{A}\times\text{A}\rightarrow\text{A}}$. The *Gaurd* function, a conditional function in FGPL, takes a polymorphic product type of three elements as input. The first element must be a boolean and other two can be of any type, but they should be of same type so as the output type. If the boolean value is True it passes the second element else passes the third element.

$$evaluate\Big(gaurd \circ (f \times g \times h)\Big) = evaluate\Big((g + h) \circ f\Big)$$
$$= \begin{cases} \Big(g'[data], best\_version(f'[world], g'[world] | f'[data])\Big) = True \\ \Big(h'[data], best\_version(f'[world], h'[world] | f'[data])\Big) = False \end{cases} \tag{3.16}$$

where, $g' = evaluate(g)$, $h' = evaluate(h)$ and $f' = evaluate(f)$.

- $\underline{Equal : \text{AnyType}\times\text{AnyType}\rightarrow\text{boolean}}$. *Equal* takes a polymorphic product type of two elements as input(any type) and returns a boolean type as output. It returns either True or False.

$$\begin{aligned} &evaluate(equal \circ (f \times g)) \\ &\qquad = (equal(evaluate(f)[data], evaluate(g)[data]), \\ &\qquad\quad best\_version(evaluate(g)[world], evaluate(f)[world])) \end{aligned} \tag{3.17}$$

- $\underline{Greater : \text{number}\times\text{number}\rightarrow\text{boolean}}$. *Greater* function takes a product type of numbers as input and returns True if the first number is greater than second else returns False.

$$\begin{aligned} &evaluate(Greater \circ (f \times g)) \\ &\qquad = (greater(evaluate(f)[data], evaluate(g)[data]), \\ &\qquad\quad best\_version(evaluate(g)[world], evaluate(f)[world])) \end{aligned} \tag{3.18}$$

- $\underline{Conjunct : \text{boolean}\times\text{boolean}\rightarrow\text{boolean}}$. *Conjunct* function acts like logical AND. It takes a product type of two boolean. It returns true if both the input arguments are True else it returns False.

$$\begin{aligned} &evaluate(Conjunct \circ (f \times g)) \\ &\qquad = (and(evaluate(f)[data], evaluate(g)[data]), \\ &\qquad\quad best\_version(evaluate(g)[world], evaluate(f)[world])) \end{aligned} \tag{3.19}$$

- $\underline{Disjunct : \text{boolean}\times\text{boolean}\rightarrow\text{boolean}}$. *Disjunct* function acts like logical OR. It takes a product type of two boolean and returns True if either of the input arguments are True, else it returns False.

$$\begin{aligned} &evaluate(Disjunct \circ (f \times g)) \\ &\qquad = (or(evaluate(f)[data], evaluate(g)[data]), \\ &\qquad\quad best\_version(evaluate(g)[world], evaluate(f)[world])) \end{aligned} \tag{3.20}$$

- $\underline{Negate : \text{boolean}\rightarrow\text{boolean}}$. This function acts like logical *NOT*. It takes a Boolean type and returns a boolean type.

$$evaluate(Negate \circ f) = (not(evaluate(f)[data]), evaluate(f)[world]) \tag{3.21}$$

3.1.8 Higher order node functions

- $\underline{\textit{Apply}: \text{function}\times\text{A}\times\text{B}\rightarrow\text{C}}$. *Apply* function takes a product type of three elements. The first element is a function type and the next two elements is of any type but type compatible with the input type of the function received as the first element. If the input function takes two arguments then the function is evaluated on both of its arguments and *Apply* returns result of function evaluation. If the input function takes more than two arguments, then a partially evaluated function is returned by the *Apply* function.

$$Apply \circ (f \times g \times h) = \begin{cases} f' \circ (g \times h), & if\ |args(f')| \geq 2 \\ f' \circ g, & if\ |args(f')| = 1 \\ f', & if\ |args(f')| = 0 \end{cases} \tag{3.22}$$

  where, $f' = evaluate(f)$ and $|args(f')|$ =number of arguments of f'.

- $\underline{\textit{Lambdagraph}: \text{A}\rightarrow\text{function}}$. *Lambdagraph* function takes input of any type. In FGPL it will return the complete subgraph ending with the parent node of the *Lambdagraph* function. 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 same input type. Input type of the initial *Identity* node is set as the input type of its child nodes. Consecutive composition of *Lambdagraph* is not allowed in FGPL.

$$\begin{aligned} &evaluate(Lambdagraph \circ f \circ g) \\ &= \begin{cases} (f, world0) | g \in \{Sensor_A \circ initWorld, goalchecker \circ initWorld\} \\ undefined |\ otherwise \end{cases} \end{aligned} \tag{3.23}$$

- $\underline{\textit{Recurse}: \text{function}\times\text{function}\times\text{A}\rightarrow\text{B}}$. The *Recurse* function takes a product type as input 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 which is applied on the third argument recursively until the stopping condition is met. The function for checking the stopping condition is received as second argument. The *Recurse* function helps generating short programs for repetitive operations.

$$\begin{aligned} &Recurse \circ (f \times g \times h) \\ &= \begin{cases} evaluate(f) \circ evaluate(h) |\ evaluate(g) = Constant_{True} \circ i \\ invalid |\ evaluate(g) = Constant_{False} \circ i \\ evaluate(f) |\ evaluate(f) = Constant_K \wedge evaluate(g) \neq Constant_{False} \circ i \\ Gaurd \circ (k \times h \times \times Recurse \circ (f \times g \times Apply \circ (f \times l \times l))) | otherwise \end{cases} \end{aligned} \tag{3.24}$$

  where, $k = Apply \circ (g \times h \times h)$ and $l = Gaurd(k \times h \times h)$.

3.2 FGPL for Universal Search

With the help of all these functionality, FGPL can be made effective for universal search platforms. The main difficulty with FGPL is that the subgraphs created in each step by depth first search might be reachable via different edges. Whereas the restriction here is to visit a node only once. To make the job easier there comes the concept of assigning program probability for each program. Now a particular phase time is allocated for each program. Within this time bound FGPL should find it's goal based on this program probability. So program probability(conditional probability) helps to decide a solution path will exists or will be aborted. Default probability is assigned to all newly created edges initially. A node can store two set of probability values. One set contains the probabilities of all possible distinct type of edges and the other set contains the distinct factored probabilities (set of distinct objects) of the program between the given node and the initial node.

Advantage of using dataflow graph as language model for universal search Data flow graphs are good at capturing independence among different functions based on data flow. This independence condition actually helps improving transfer learning which evidently helps in reducing complexity of the later task between the solution strings.

3.3 Incremental Learning

Incremental learning is the process of gaining mutual information by solving related sequence of tasks which would help solving later related tasks by reducing the search space. Incremental learning in universal search is implemented by updating conditional probability distribution of function nodes based on some rewards received. It has been designed problem-solving agent in a reinforcement learning setting. The agent will be exposed to a series of problems with increasing difficulty. Solving a task will generate a positive reward. The objective is to maximize the total future expected reward. Incremental learning is achieved by updating the conditional probability distribution based on gradient ascent mechanism thereby maximizing future expected rewards.

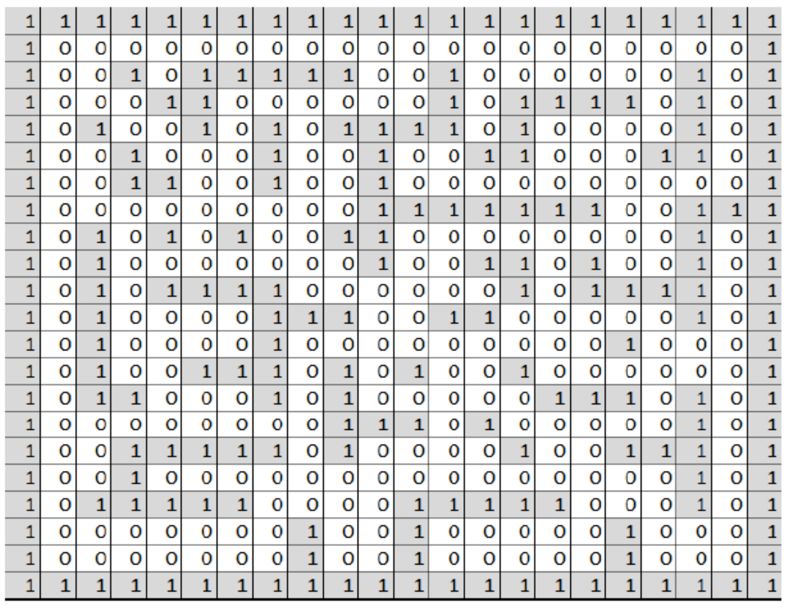
3.4 The Metasearcher

The metasearcher of the agent is responsible for implementing universal search and searches for solution in the FGPL program space for a given problem environment. *The metasearcher is outside of FGPL and it starts with initializing a phase variable, a phase limit, a list of available FGPL functions to be used for program generation and an initial FGPL graph object if it is not supplied as input.* In absence of incremental learning a new graph object is created with the node initWorld added as initial node. Thereafter the search proceeds in phases by generating and executing unique programs in FGPL. *Programs are generated by adding and connecting all possible nodes with all possible combinations of successfully executed nodes present in the graph object after satisfying type compatibility.* Redundant programs are eventually deleted. All the programs generated gets connected to the same search graph. In each phase, a fraction of phase value is allotted as runtime to a program, proportional to its program probability. *Terminal nodes of programs are marked as failed which raises an error during execution.* Once the goal is reached the search process stops and the terminal node corresponding to the solution program along with the complete search graph is returned.

Results

The agent is experimented in a maze problem.

1. The *Sensor monad* sends a read request to the maze environment. If the next cell along the current direction of the agent is blocked, then the environment returns True else False.
2. The *Actuator* monad sends an action request to the environment with a parameter. If the parameter is 1 the agent moves one step forward along its current direction, If 2 then it turns left, if 3 it turns right.
3. The *Goalchecker* sends read request to the environment to check if the goal is reached or not and the environment responds in boolean. If the current state of the agent matches with the goal state given by the environment, it returns True else returns False.
4. Here, [1,0] represents south, [0, 1] represents east, [-1, 0] represents north, [0, -1] represents west.



1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
1	0	0	1	0	1	1	1	1	1	0	0	1	0	0	0	0	0	0	1	0	1
1	0	0	0	1	1	0	0	0	0	0	0	1	0	1	1	1	1	0	1	0	1
1	0	1	0	0	1	0	1	0	1	1	1	1	0	1	0	0	0	0	1	0	1
1	0	0	1	0	0	0	1	0	0	1	0	0	1	1	0	0	0	1	1	0	1
1	0	0	1	1	0	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	1
1	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	0	0	1	1	1
1	0	1	0	1	0	1	0	0	1	1	0	0	0	0	0	0	0	0	1	0	1
1	0	1	0	0	0	0	0	0	0	1	0	0	1	1	0	1	0	0	1	0	1
1	0	1	0	1	1	1	1	0	0	0	0	0	0	1	0	1	1	1	1	0	1
1	0	1	0	0	0	0	1	1	1	0	0	1	1	0	0	0	0	0	1	0	1
1	0	1	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1	0	0	0	1
1	0	1	0	0	1	1	1	0	1	0	1	0	0	1	0	0	0	0	0	0	1
1	0	1	1	0	0	0	1	0	1	0	0	0	0	0	1	1	1	0	1	0	1
1	0	0	0	0	0	0	0	0	1	1	1	0	1	0	0	0	0	0	1	0	1
1	0	0	1	1	1	1	1	0	1	0	0	0	0	1	0	0	1	1	1	0	1
1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1
1	0	1	1	1	1	1	0	0	0	0	1	1	1	1	1	0	0	0	1	0	1
1	0	0	0	0	0	0	0	1	0	0	1	0	0	0	0	0	1	0	0	0	1
1	0	0	0	0	0	0	0	1	0	0	1	0	0	0	0	0	1	0	0	0	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

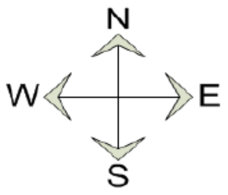


Figure 3.5: A 20X20 maze. 1 is blocked and 0 is opened cell.

These configurations can be considered as the initial bias given to the agent based on the domain knowledge. 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. Learning rate is chosen as 0.5.

Training	initial state	goal state	initial program probability	final program probability	iteration
Training 1	[1,1,[1,0]]	[2,1,[1,0]]	0.0666666	0.2222222	1
Training 2	[1,1,[1,0]]	[10,1,[1,0]]	0.000514403	0.001036283	9
Training 3	[1,1,[1,0]]	[2,2,[0,1]]	0.000233863	0.000852707	1
Training 4	[1,1,[1,0]]	[2,2,[1,0]]	2.84E-05	0.016610265	7

Table 3.1: Training sequence for the metasearcher.

Task	initial state	goal state	initial program probability	final program probability	unbiased program probability
Test 1	[1,1,[1,0]]	[10,10,[1,0]]	5.83E-05	6.13E-05	1.1429891305E-08
Test 2	[1,1,[1,0]]	[20,20,[1,0]]	6.13E-05	6.43E-05	1.1429891305E-08

Table 3.2: Test sequence for the metasearcher

Fig. 3.9 shows the solution programs drawn from the training sequence and the test sequence. The complexity of each program graph increases as per shown below. Reaching goal or destination for each task helped increasing the probability of each edges in the solution. This helped in increasing the probability of the solution as a whole for the test tasks and eventually reduce the search space and time.

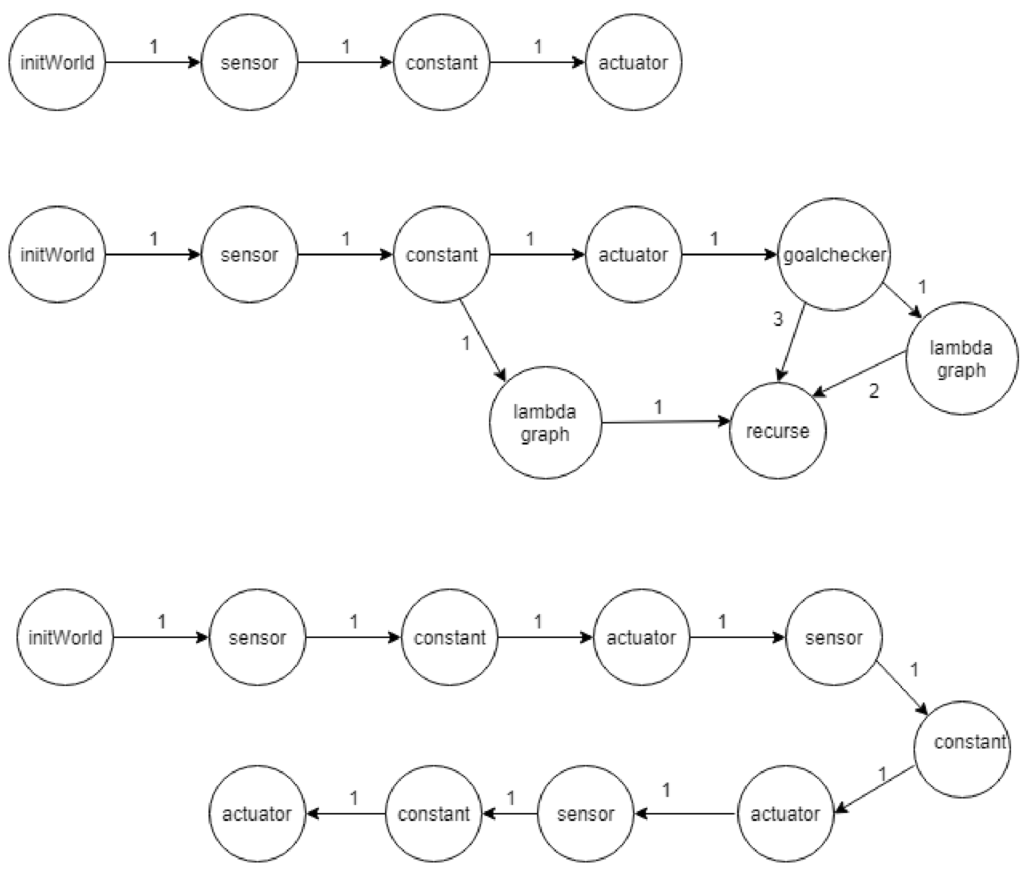


Figure 3.6: Solution program for training using FGPL

The node names in Fig. 3.9 carries meaning explained above. The first graph is simplest among the three. It can be described for maze environment that the world object is found by initWorld node and it can be followed by sensor which takes the data as well as the object from environment through it's parent node. The sensor is connected with the constant node. The one directional edges confirm the connection. The value of constant node is inherited to the actuator node which is the goal state for the program graph.

In 2nd case the value in actuator is checked by the goalchecker to send the third input of recurse node as a True/False condition. And both the lambdagraph acts in port no. 1 and 2 to send the operational value as input. When the boolean satisfies then it reaches the goal.

For the 3rd graph value of constant node is generating at different level. All the values are evaluated by sensor and streamed by actuator. Hence the child node waits for the parent node to generate and the parent node connects with all of it's children node. No duplicate data is entertained in FGPL. Program paths must be unique in the program space. Redundancy can hamper the performance of the agent.

So it can be seen that solution path does not depend on the problem size (maze dimensions). Due to usage of recursion without any hardcoded stopping condition in the solution program the same program can be used as solution for other maze problems if same pattern exists in those solutions irrespective of the dimension of the maze.

Thus, for a 1000 X 1000 maze with exactly similar pattern and similar extended problem (initial state = [1,1,[1,0]], goal state = [1000,1000,[1,0]]) as the current one, the solution program will also be same as the current one (though the runtime of the solution program will be greater than the current one due to longer solution path). This shows the capability of the FGPL language in creating small solutions for problems with excessively large state space.

CHAPTER 4

Problem Statement

The problem statement of our work aims to design a problem solver using above mentioned FGPL and data flow diagram to solve universal search problems in the problem space. We need to solve such problems in optimized way by training the model we designed using reinforcement learning, so that the model can remember the paths that it experienced previously as well as explore the paths which are yet to be experienced by trial and error process. While testing, the goal or destination must be reached through best probable path. We have taken cartpole problem or inverted pendulum problem as sample program to solve in an individual machine as well as in distributed framework to verify effectiveness of FGPL framework in different environment, to establish a master-slave architecture to divide the heavy computational workload among all the processors used and to improve the time difference and performance graph.

CHAPTER 5

Proposed Work and Solution Statement

5.1 Proposed Framework

1. We have implemented the whole work in *python3* using *openai Gym* library. *Gym* library is a collection of test problems – environments – where we have used reinforcement learning or trial and error methodology. *Gym* makes no assumption of the intelligent agent's structure.
   19

   Installation of *Gym* has been done using **pip install gym** command.

2.
   ```
   import gym
   env = gym.make('CartPole-v0')
   env.reset()
   for _ in range(1000):
     env.render()
     env.step(env.action_space.sample()) # take random action
   env.close()
   ```

   Here $env.step()$ function returns 4 values, which are: **observation, reward, done, info**.

3. We have implemented our FGPL in this gym environment to train the model which can interact with environment just by applying **positive reward on correct evaluation** and punishing negative reward for wrong evaluation.

4. For testing FGPL in this framework we have executed small test programs to get the output and draw the graph in **R studio for visualization** purpose.

5. We have designed the cartpole problem using FGPL property and have trained our model to return the correct graph while testing.

6. We have used *metasearcher* algorithm which starts with initializing a phase variable, a phase limit, a list of available FGPL functions to be used for program generation and an initial FGPL graph object if it is not supplied as input.

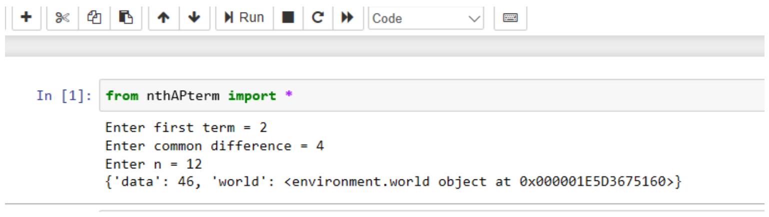
7. Programs are generated by adding and connecting all possible nodes with all possible combinations of successfully executed nodes present in the graph object after satisfying type compatibility.

8. We have also worked on distributed approach to increase the performance. To communicate with multiple server we have used MPI technique.

9. Here in master slave architecture of MPI small subgraphs will be generated in each slave machine after reaching their respective goals.

10. All the small subgraphs will be gathered back to the master computer to make a complete graph in FGPL. So the communication establishment is the key to such architecture.

We have first tested for nth AP term program using FGPL in python3 and visualized the graph in R studio. This is a small program where 1st term and common difference is given to find out nth AP term. We have also implemented the recursive version of this using recurse function.



```
In [1]: from nthAPterm import *

        Enter first term = 2
        Enter common difference = 4
        Enter n = 12
        {'data': 46, 'world': <environment.world object at 0x000001E5D3675160>}
```

Figure 5.1: Output of the nth AP term is 46.

And here is the graph flow of the above problem using Rstudio.

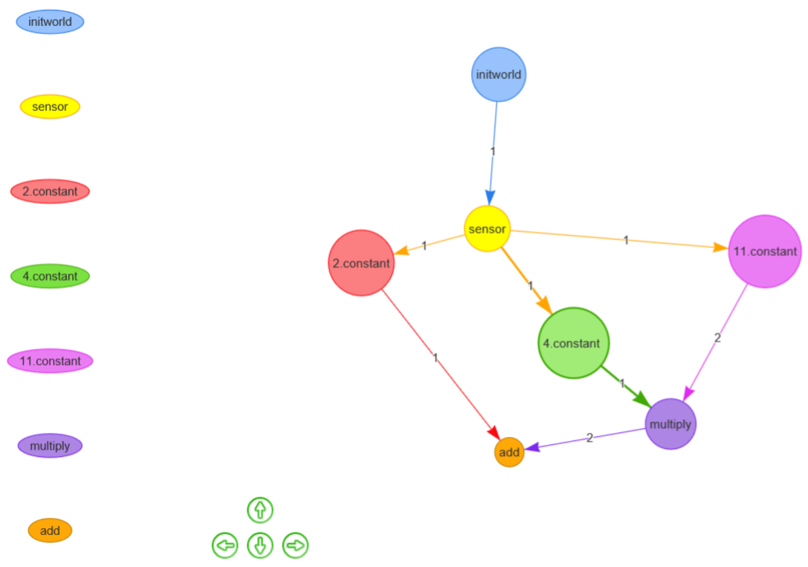


Figure 5.2: Nth term of AP series in graphical representation using FGPL.

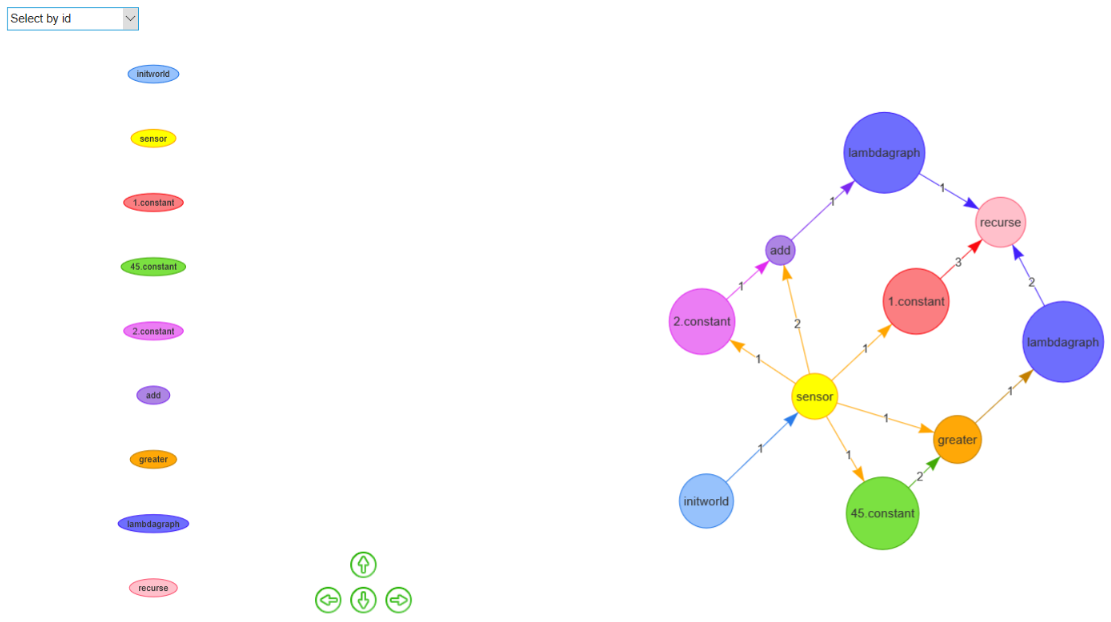


Figure 5.3: Recursive function Representation for finding Nth AP term In FGPL.

We have implemented few new monadic functions apart from the existing functions as an extension of our work. We have implemented List operation and related functions like Head, Tail, Nil, Fmap, Zipping, and Aggregate and Power.

- Head: list -> list. Head function takes a list as input and returns the first element or head element as output in form of list.

$$evaluate(head([a_0, a_1, a_2, a_3])) = (evaluate[a_0][data]), \\ version(evaluate[a_0][world]). \tag{5.1}$$

- Tail: list -> list. Tail function takes a list as input and returns the rest of the element except head as output in form of list.

$$evaluate(tail([a_0, a_1, a_2, a_3])) \\ = (evaluate[a_1, a_2, a_3][data]), best\_version \\ (evaluate[a_1, a_2, a_3][world]). \tag{5.2}$$

- A -> list. Nil takes any data type as input and returns empty list as output.

$$evaluate(nil(A)) = (evaluate[][data]), best_version(evaluate[][world]) \tag{5.3}$$

- Fmap: function X list -> list. Fmap takes polymorphic product type of 2 elements as input. The first element of the product type should be function type and the next element needs to be a list type. This function implements the function operation given in input port 1, to each of the elements of input list and perform individual operation. So, the output of fmap function will be always a list.

$$\begin{aligned}&evaluate(fmap(f \circ [a_0, a_1, a_2, a_3]) = \\ &\qquad (evaluate(f \circ [a_0])[data], evaluate(f \circ [a_1])[data], \\ &\qquad evaluate(f \circ [a_2])[data], evaluate(f \circ [a_3])[data]), \\ &\qquad best\_version(evaluate(f \circ [a_0]))[world], evaluate(f \circ [a_1])[world], \\ &\qquad evaluate(f \circ [a_2])[world], evaluate(f \circ [a_3])[world]).\end{aligned} \tag{5.4}$$

- Zipping: function X list X list -> list. Zipping takes polymorphic product type of 3 elements as input. The first element of the product type should be function type and the next two input elements should be of list type. This function implements the function operation given in i/p port 1, to both the lists in sequence. So, it returns a list as output.

$$\begin{aligned}&evaluate(zipping(f \circ ([a_0, a_1, a_2, a_3] \circ [b_0, b_1, b_2, b_3])) = \\ &\qquad (evaluate(f \circ [a_0, b_0])[data], evaluate(f \circ [a_1, b_1])[data], \\ &\qquad evaluate(f \circ [a_2, b_2])[data], evaluate(f \circ [a_3, b_3])[data]), \\ &\qquad best\_version(evaluate(f \circ [a_0, b_0]))[world], evaluate(f \circ [a_1, b_1])[world], \\ &\qquad evaluate(f \circ [a_2, b_2])[world], evaluate(f \circ [a_3, b_3])[world]).\end{aligned} \tag{5.5}$$

- Aggregate: function X list -> any. Aggregate takes polymorphic product type of 2 elements as input. The first element of the product type should be function type and the next input needs to be of list type. And the output can be of any type.

$$\begin{aligned}&evaluate(aggregate(f \circ [a_0, a_1, a_2, a_3]) = \\ &\qquad (evaluate(f \circ [a_0, a_1, a_2, a_3])[data], \\ &\qquad best\_version(evaluate(f \circ [a_0, a_1, a_2, a_3]))[world].\end{aligned} \tag{5.6}$$

- Cons: any X list -> any. Cons takes polymorphic product type of 2 elements as input. The first element of the product type should be any type and the next input would be of list type. In this function appending of the list is done at 0th position.

$$\begin{aligned}&evaluate(cons(x \circ [a_0, a_1, a_2, a_3]) = \\ &\qquad (evaluate([a_0, a_1, a_2, a_3])[data].append(x)[data], \\ &\qquad best\_version(evaluate([a_0, a_1, a_2, a_3])[world].\end{aligned} \tag{5.7}$$

Here we have tried showing the internal representation of the nodes (graphical representation) we have added i.e fmap, zipping and aggregate for cartpole problem we want to solve.

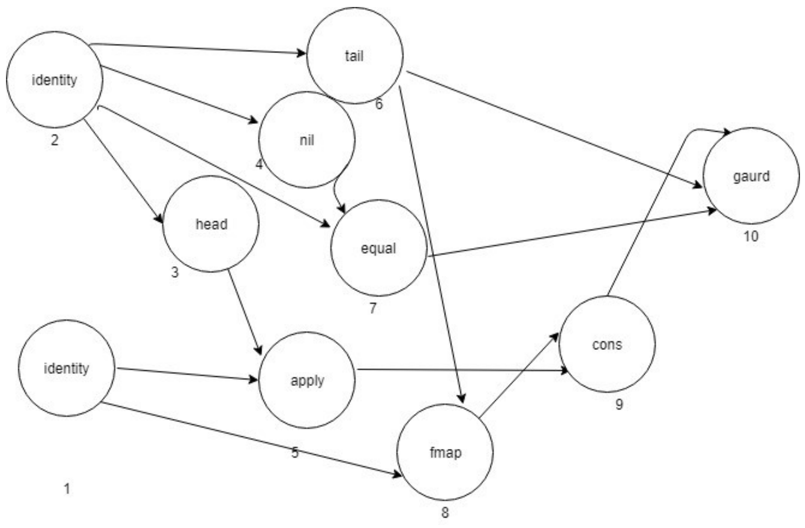


Figure 5.4: Representation of fmap function and it's graphical connectivity.

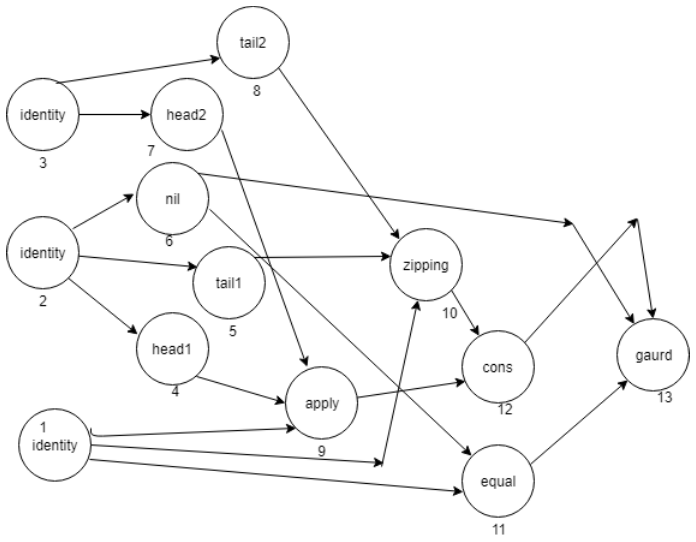


Figure 5.5: Representation of zipping function and it's graphical connectivity.

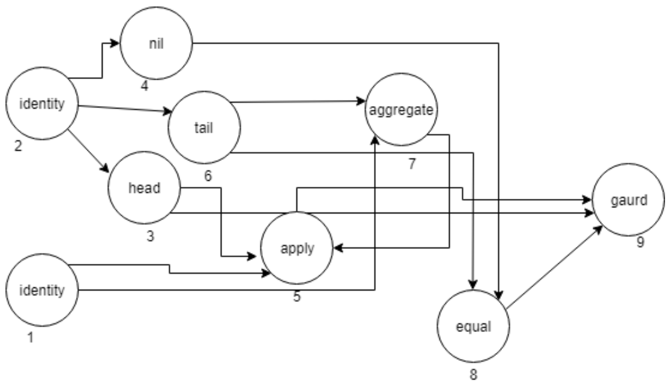
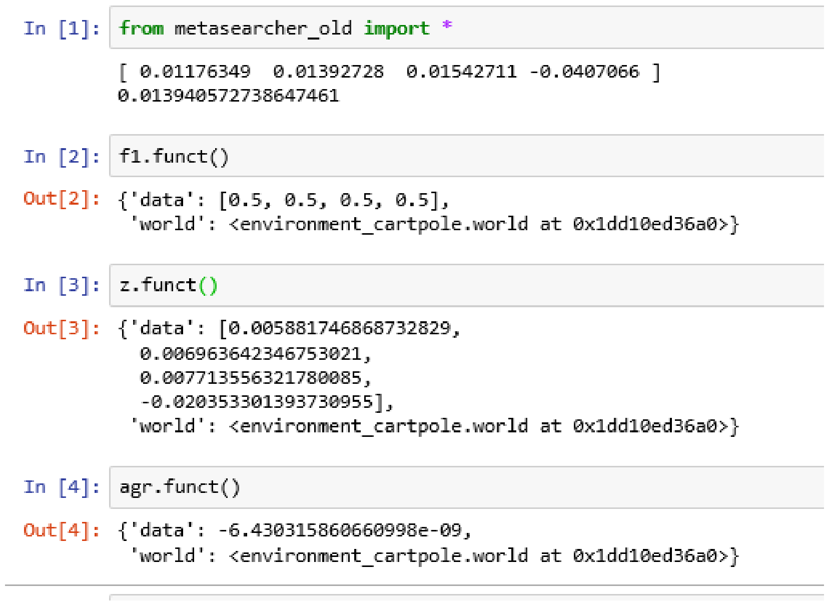


Figure 5.6: Representation of aggregate function and it's graphical connectivity.

Here a short test program output for checking the o/p and graph by fmap, zipping and aggregate is attached.



```
In [1]:  from metasearcher_old import *
         [ 0.01176349  0.01392728  0.01542711 -0.0407066 ]
         0.013940572738647461

In [2]:  f1.funct()

Out[2]:  {'data': [0.5, 0.5, 0.5, 0.5],
          'world': <environment_cartpole.world at 0x1dd10ed36a0>}

In [3]:  z.funct()

Out[3]:  {'data': [0.005881746868732829,
           0.006963642346753021,
           0.007713556321780085,
           -0.020353301393730955],
          'world': <environment_cartpole.world at 0x1dd10ed36a0>}

In [4]:  agr.funct()

Out[4]:  {'data': -6.430315860660998e-09,
          'world': <environment_cartpole.world at 0x1dd10ed36a0>}
```

Figure 5.7: Output of a test program with fmap, zipping and aggregate function implemented.

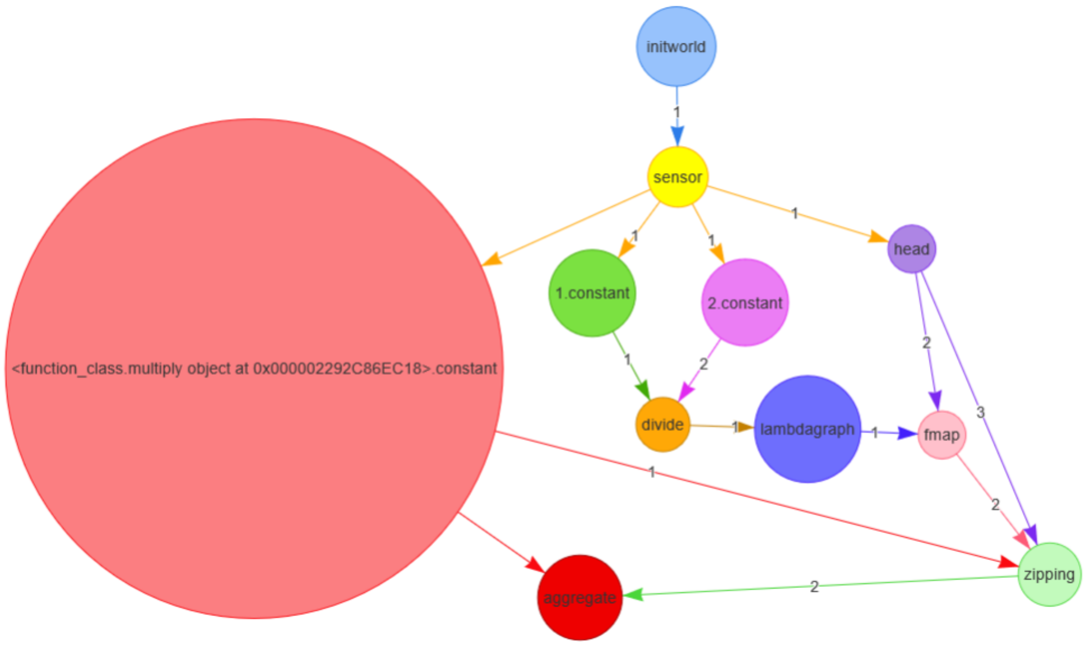


Figure 5.8: Graphical representation of a test program with fmap, zipping and aggregate node implemented combinedly.

5.2 Proposed Solution

Now we have all the functionality available to draw the graph of cartpole problem in FGPL. With the help of the I/O monads, First order node functions and higher order bode functions, as discussed previously, the data types and newly implemented functions we are going to propose a solution graph for Cartpole problem. Cartpole problem solution graph that we have designed in FGPL and after all the inclusion of functions the correct graph could be drawn in Rstudio. The graph is attached below.

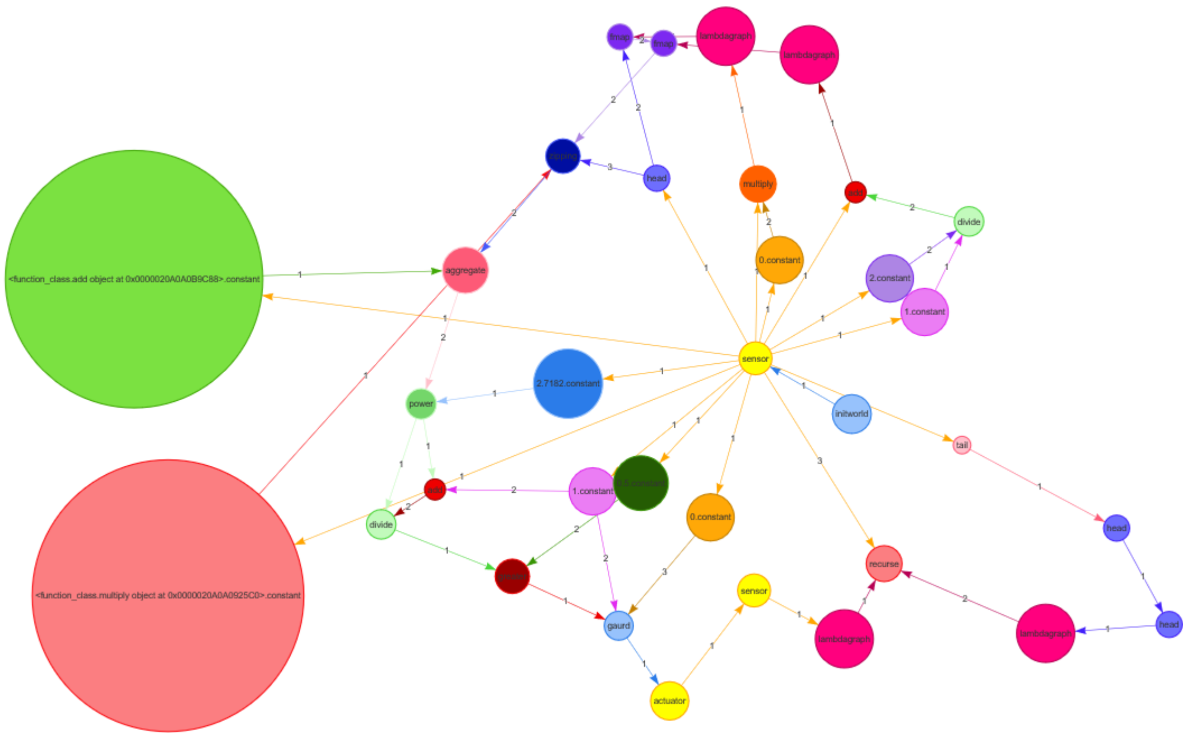


Figure 5.9: Cartpole problem in graphical representation using FGPL.

<u>Use of MPI</u>: MPI (Message Passing Interface)is needed for connecting more than 1 computers simultaneously to establish master slave architecture. The basic need is to divide the workload in all the processors where the sender or master computer will send some information to other slave or receiver computers and will get all the responses back to the master accordingly. Here we have first established the MPI connection in python in 6 machines (24 processors) and tested if they are working successfully or not.

CHAPTER 6

Results and Discussion

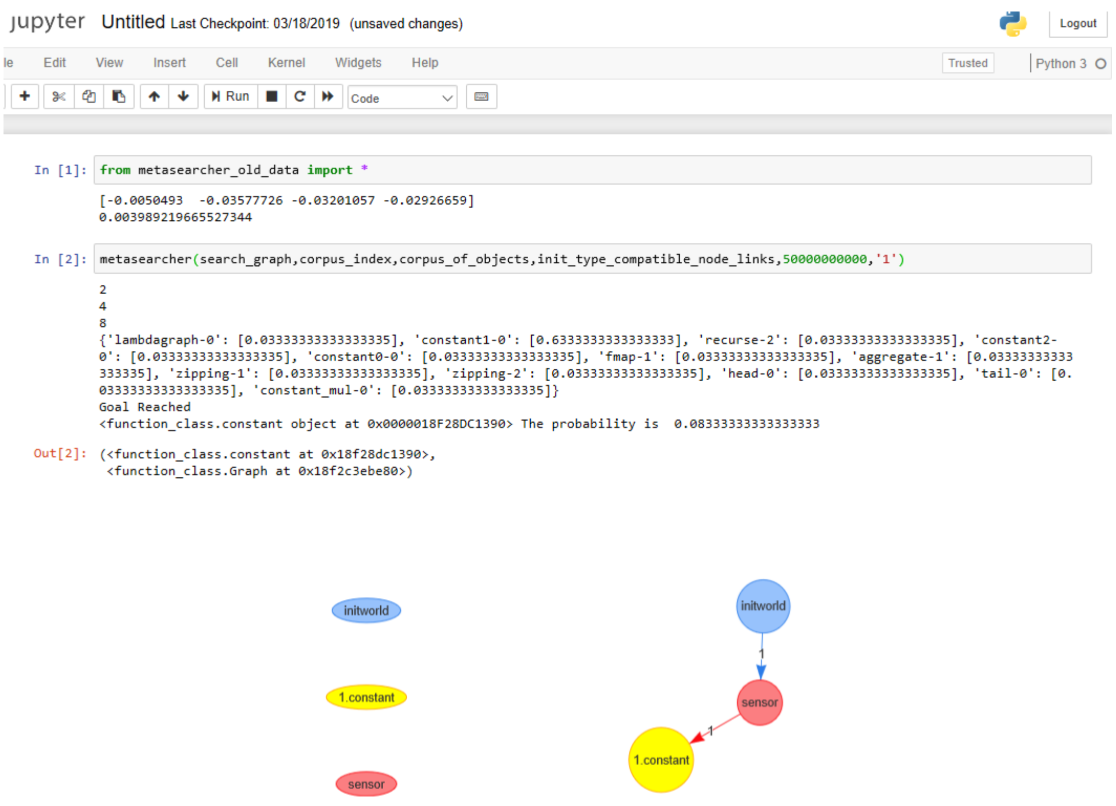
Here we have attempted Universal search problems to be solved using reinforcement learning by training an agent or model with some particular labelled data and making it intelligent enough to interact with the environment. The problem statement is tested for standalone machine as well as distributed environment. The performance of distributed system is clearly better as the computational load is scattered into all the processors of all the machines we have engaged. One master computer is sending the job request to each and every slave machines. They will receive request and proceed to find the solution graph and will finish the alloted task it is given to do. After completion each slave machine send their solution subgraphs back to the master. In this case three points are important.

- If a parent node has more than one child node and all the child nodes are generated in different subgraphs then there must be only once the parent node can be generated. There must no be duplicate nodes for each subgraph. One node shouldn't be replicated multiple times to increase space and time complexity.
- If a child node has more than one parent node then both the parent node should get generated before the child node gets executed. All the predecessor nodes must be present to create the next successor node.
- All the possible combinations should be present to create all the possible subgraphs for generating the optimal probable solution. So the least probable path will also be there in the problem space but to avoid computational deficiency we must reduce that edge's probability. So no active pruning is considered.

Here are few data, screenshot of output and constructed view of the graphs are attached as a part of result data of the experiment.

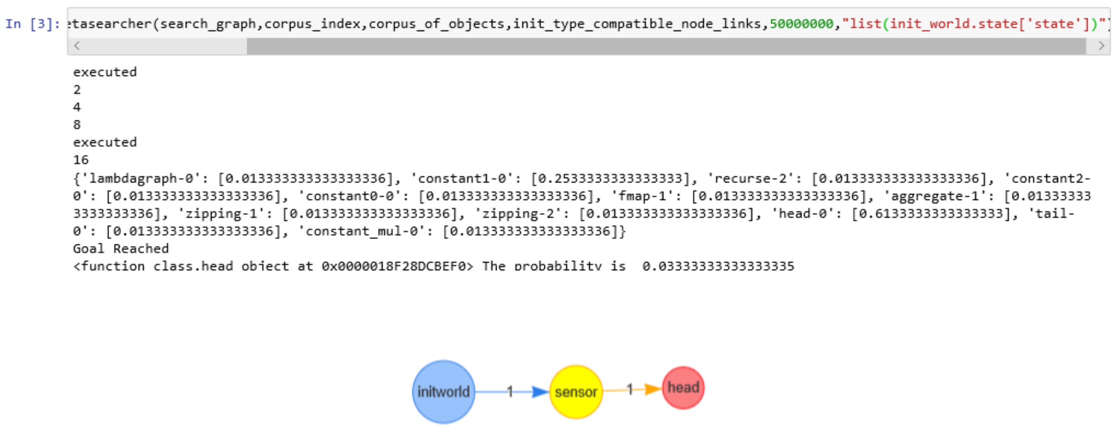
For the cartpole problem we run the metasearcher algorithm, it starts with initializing a phase variable, a phase limit, a list of available FGPL functions to be used for program generation and an initial FGPL graph object if it is not supplied as input. The search proceeds in phases by generating and executing unique programs in FGPL. *Programs are generated by adding and connecting all possible nodes with all possible combinations of successfully executed nodes present in the graph object after satisfying type compatibility.*

Nodes which creates syntactically redundant programs are eventually deleted. Here I have shown images for constant, head node, fmap node, divide node, and their respective output while running the metasearcher algorithm.


jupyter Untitled Last Checkpoint: 03/18/2019 (unsaved changes)
Logout
le Edit View Insert Cell Kernel Widgets Help
Trusted
Python 3
Run
Code
initworld
1.constant
sensor
initworld
1
sensor
1
1.constant

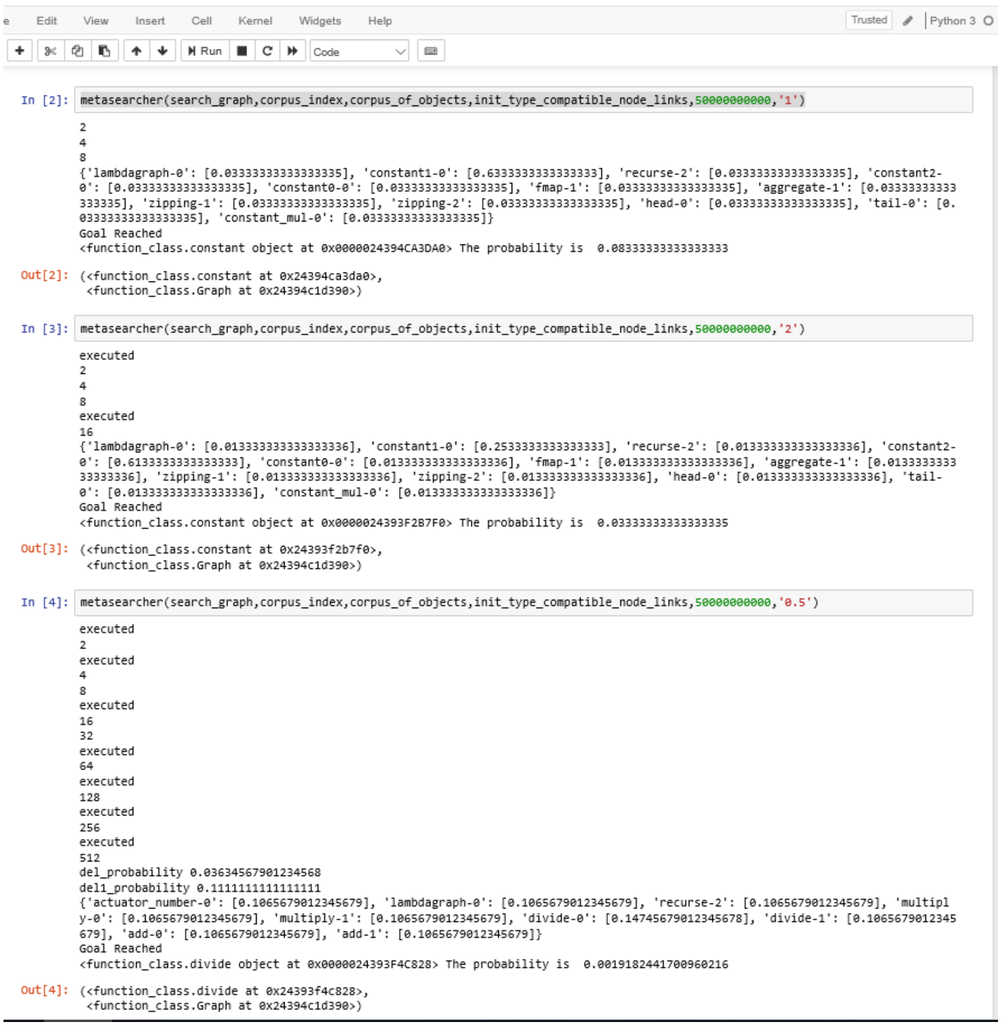
```
In [1]: from metasearcher_old_data import *

        [-0.0050493  -0.03577726 -0.03201057 -0.02926659]
        0.003989219665527344

In [2]: metasearcher(search_graph,corpus_index,corpus_of_objects,init_type_compatible_node_links,50000000000,'1')

        2
        4
        8
        {'lambdagraph-0': [0.03333333333333335], 'constant1-0': [0.6333333333333333], 'recurse-2': [0.03333333333333335], 'constant2-
        0': [0.03333333333333335], 'constant0-0': [0.03333333333333335], 'fmap-1': [0.03333333333333335], 'aggregate-1': [0.03333333333
        333335], 'zipping-1': [0.03333333333333335], 'zipping-2': [0.03333333333333335], 'head-0': [0.03333333333333335], 'tail-0': [0.
        03333333333333335], 'constant_mul-0': [0.03333333333333335]}
        Goal Reached
        <function_class.constant object at 0x0000018F28DC1390> The probability is  0.08333333333333333

Out[2]: (<function_class.constant at 0x18f28dc1390>,
         <function_class.Graph at 0x18f2c3ebe80>)
```

Figure 6.1: Goal reached for constant value given as 1 and the subgraph created attached.



```
In [3]: :tasearcher(search_graph,corpus_index,corpus_of_objects,init_type_compatible_node_links,50000000,"list(init_world.state['state'])"

        executed
        2
        4
        8
        executed
        16
        {'lambdagraph-0': [0.013333333333333336], 'constant1-0': [0.2533333333333333], 'recurse-2': [0.013333333333333336], 'constant2-
        0': [0.013333333333333336], 'constant0-0': [0.013333333333333336], 'fmap-1': [0.013333333333333336], 'aggregate-1': [0.01333333
        3333333336], 'zipping-1': [0.013333333333333336], 'zipping-2': [0.013333333333333336], 'head-0': [0.6133333333333333], 'tail-
        0': [0.013333333333333336], 'constant_mul-0': [0.013333333333333336]}
        Goal Reached
        <function class.head object at 0x0000018F28DCBEF0> The probability is  0.03333333333333335
```

Figure 6.2: Goal reached for head function and the subgraph created attached.



```
In [2]: metasearcher(search_graph,corpus_index,corpus_of_objects,init_type_compatible_node_links,50000000000,'1')

        2
        4
        8
        {'lambdagraph-0': [0.03333333333333335], 'constant1-0': [0.6333333333333333], 'recurse-2': [0.03333333333333335], 'constant2-
        0': [0.03333333333333335], 'constant0-0': [0.03333333333333335], 'fmap-1': [0.03333333333333335], 'aggregate-1': [0.03333333333
        333335], 'zipping-1': [0.03333333333333335], 'zipping-2': [0.03333333333333335], 'head-0': [0.03333333333333335], 'tail-0': [0.
        03333333333333335], 'constant_mul-0': [0.03333333333333335]}
        Goal Reached
        <function_class.constant object at 0x0000024394CA3DA0> The probability is  0.08333333333333333

Out[2]: (<function_class.constant at 0x24394ca3da0>,
         <function_class.Graph at 0x24394c1d390>)

In [3]: metasearcher(search_graph,corpus_index,corpus_of_objects,init_type_compatible_node_links,50000000000,'2')

        executed
        2
        4
        8
        executed
        16
        {'lambdagraph-0': [0.013333333333333336], 'constant1-0': [0.2533333333333333], 'recurse-2': [0.013333333333333336], 'constant2-
        0': [0.6133333333333333], 'constant0-0': [0.013333333333333336], 'fmap-1': [0.013333333333333336], 'aggregate-1': [0.0133333333
        33333336], 'zipping-1': [0.013333333333333336], 'zipping-2': [0.013333333333333336], 'head-0': [0.013333333333333336], 'tail-
        0': [0.013333333333333336], 'constant_mul-0': [0.013333333333333336]}
        Goal Reached
        <function_class.constant object at 0x0000024393F2B7F0> The probability is  0.03333333333333335

Out[3]: (<function_class.constant at 0x24393f2b7f0>,
         <function_class.Graph at 0x24394c1d390>)

In [4]: metasearcher(search_graph,corpus_index,corpus_of_objects,init_type_compatible_node_links,50000000000,'0.5')

        executed
        2
        executed
        4
        8
        executed
        16
        32
        executed
        64
        executed
        128
        executed
        256
        executed
        512
        del_probability 0.03634567901234568
        del1_probability 0.1111111111111111
        {'actuator_number-0': [0.1065679012345679], 'lambdagraph-0': [0.1065679012345679], 'recurse-2': [0.1065679012345679], 'multipl
        y-0': [0.1065679012345679], 'multiply-1': [0.1065679012345679], 'divide-0': [0.14745679012345678], 'divide-1': [0.1065679012345
        679], 'add-0': [0.1065679012345679], 'add-1': [0.1065679012345679]}
        Goal Reached
        <function_class.divide object at 0x0000024393F4C828> The probability is  0.0019182441700960216

Out[4]: (<function_class.divide at 0x24393f4c828>,
         <function_class.Graph at 0x24394c1d390>)
```

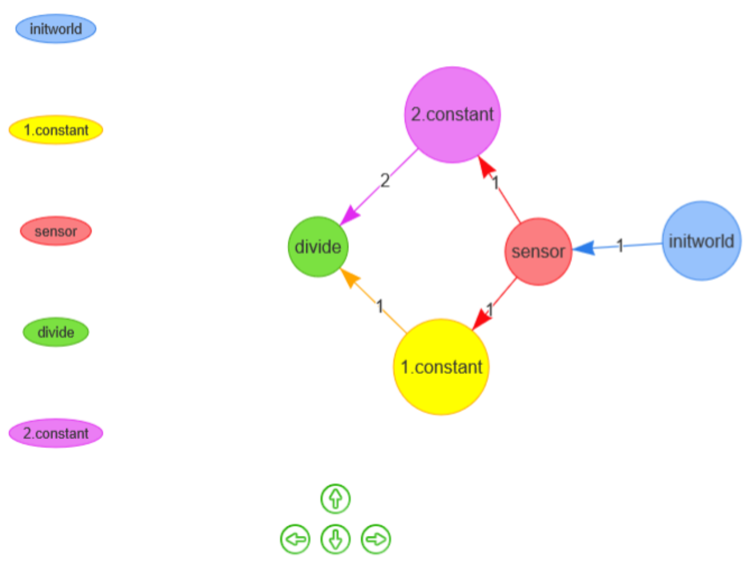
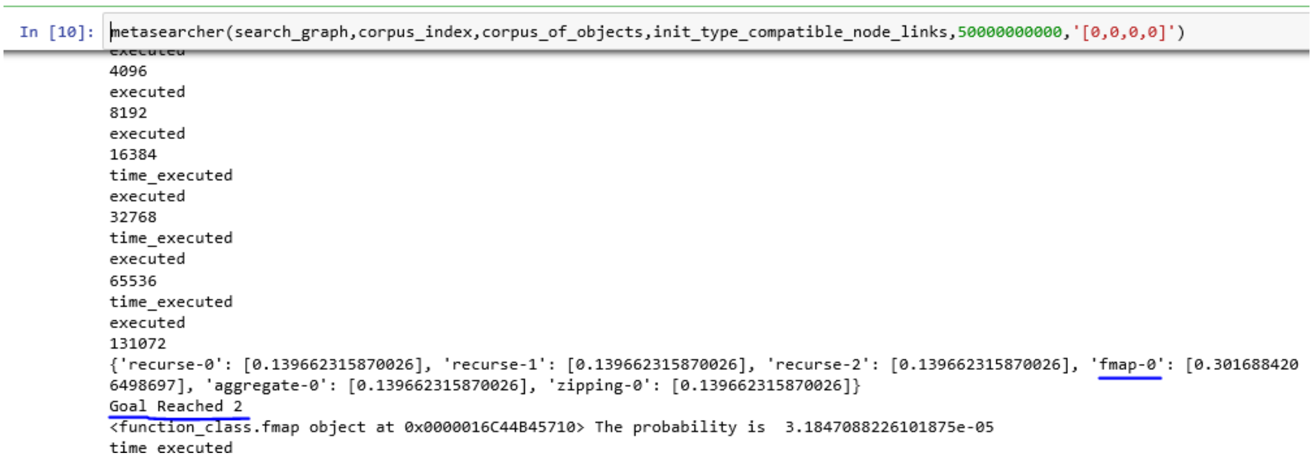


Figure 6.3: Goal reached for divide function given data as 0.5 and the subgraph created attached.



```
In [10]: metasearcher(search_graph,corpus_index,corpus_of_objects,init_type_compatible_node_links,50000000000,'[0,0,0,0]')
         executed
         4096
         executed
         8192
         executed
         16384
         time_executed
         executed
         32768
         time_executed
         executed
         65536
         time_executed
         executed
         131072
         {'recurse-0': [0.139662315870026], 'recurse-1': [0.139662315870026], 'recurse-2': [0.139662315870026], 'fmap-0': [0.301688420
         6498697], 'aggregate-0': [0.139662315870026], 'zipping-0': [0.139662315870026]}
         Goal Reached 2
         <function_class.fmap object at 0x0000016C44B45710> The probability is  3.1847088226101875e-05
         time executed
```

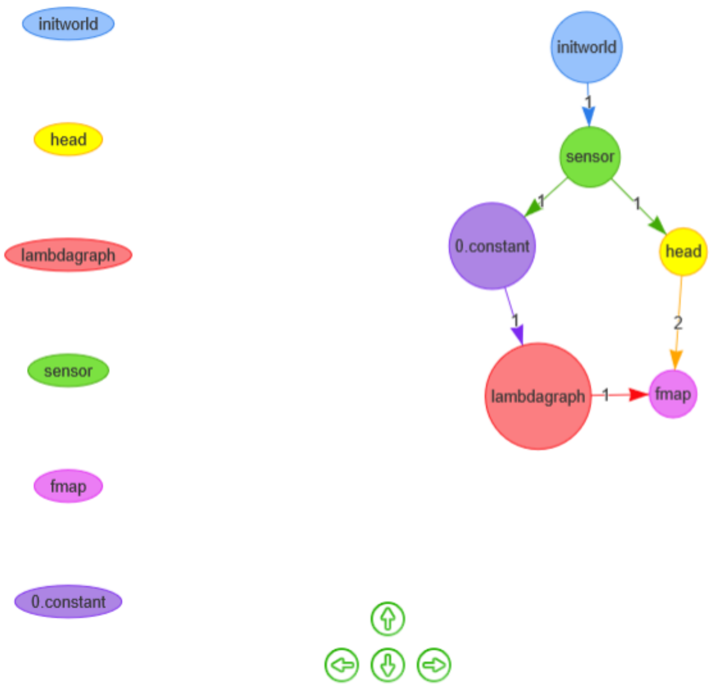
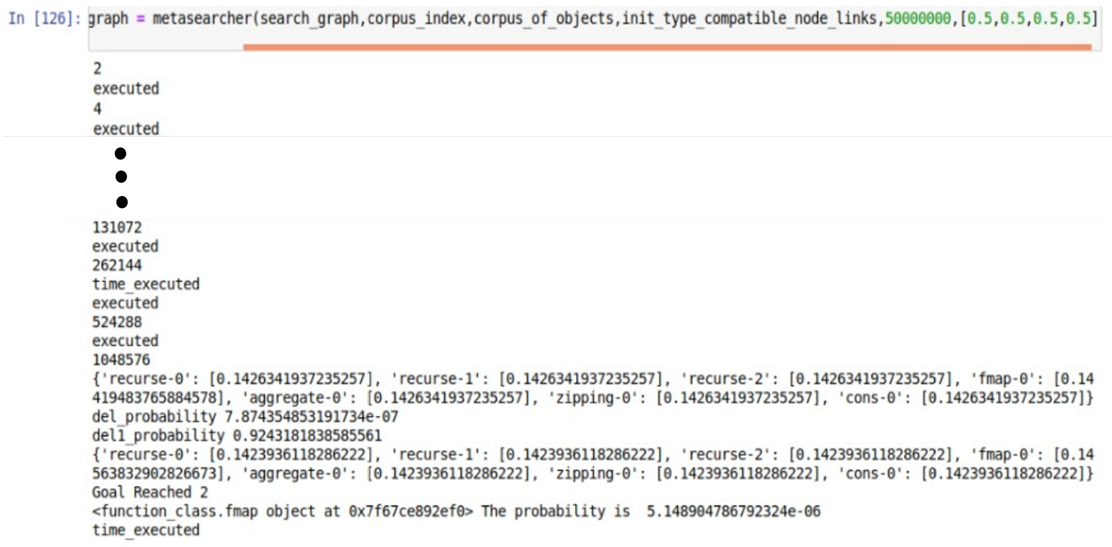


Figure 6.4: Goal reached for fmap function given input list as [0,0,0,0] and the subgraph created attached.



```
In [126]: graph = metasearcher(search_graph,corpus_index,corpus_of_objects,init_type_compatible_node_links,50000000,[0.5,0.5,0.5,0.5]

2
executed
4
executed
●
●
●
131072
executed
262144
time_executed
executed
524288
executed
1048576
{'recurse-0': [0.1426341937235257], 'recurse-1': [0.1426341937235257], 'recurse-2': [0.1426341937235257], 'fmap-0': [0.14
419483765884578], 'aggregate-0': [0.1426341937235257], 'zipping-0': [0.1426341937235257], 'cons-0': [0.1426341937235257]}
del_probability 7.874354853191734e-07
del1_probability 0.9243181838585561
{'recurse-0': [0.1423936118286222], 'recurse-1': [0.1423936118286222], 'recurse-2': [0.1423936118286222], 'fmap-0': [0.14
563832902826673], 'aggregate-0': [0.1423936118286222], 'zipping-0': [0.1423936118286222], 'cons-0': [0.1423936118286222]}
Goal Reached 2
<function_class.fmap object at 0x7f67ce892ef0> The probability is  5.148904786792324e-06
time_executed
```

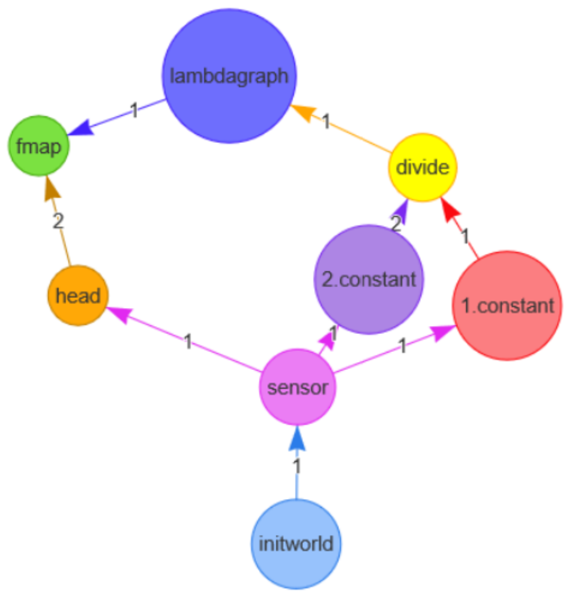


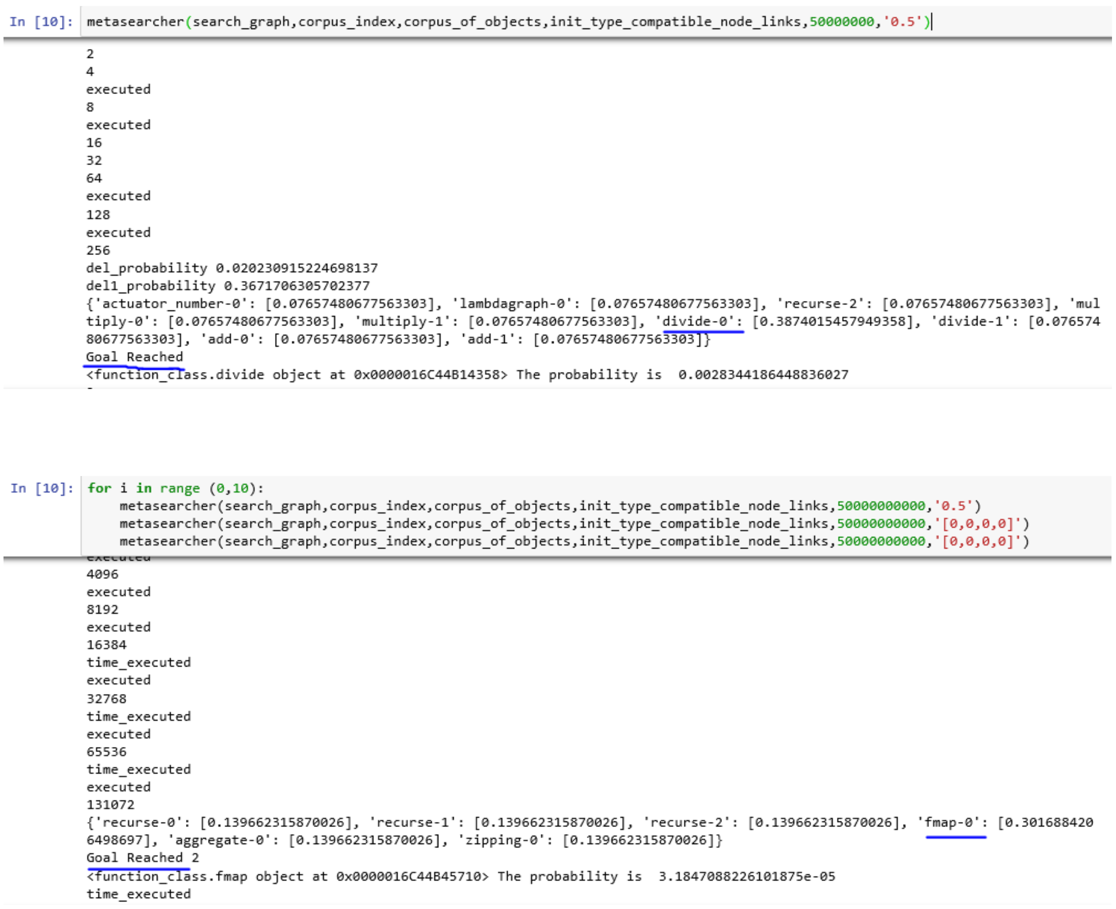
Figure 6.5: Goal reached for fmap function given input list as [0.5,0.5,0.5,0.5] and the subgraph created attached.

Explanation:

- 1st node of connection between environment and agent is ‘initworld’. Then comes ‘sensor’. Now from the first line of execution by importing metasearcher code to run in jupyter notebook we can see the observation state values such as [-0.0050493, -0.03577726, -0.03201057, -0.2926659] which is the head value of the data in the sensor.
- Now we have added ‘constant’ 1 and 2 in the subgraph.
- Next we have passed data 0.5 in the metasearcher’s argument that can perform the ‘divide’ property between 1 and 2 and creates total 5 nodes, initWorld, sensor, constant1, constant2, divide.
- Next comes fmap1 which is basically the head data each of which is multiplied by 0 and returns [0, 0, 0, 0]. So we have given this data in metasearcher to satisfy ‘fmap1’.
- For fmap2, as per cartpole fig 27, 0.5 needs to be added to [0, 0, 0, 0] so it will return [0.5, 0.5, 0.5, 0.5]. So we have given this data in metasearcher to satisfy ‘fmap2’.

Now the next step that is ‘zipping’ node could not be evaluated in this way. The main reason behind this can be **computational complexity**. Now upto fmap2 we have done some **forced training** to reach the goal within some limited phase value. If we don’t do any biased training and as mentioned in the explanation above, we try to train the model then it takes much more time due to computational overhead. And we are unable to reach the goals as smoothly as expected.

Here is some data to attach as a proof of forced learning. Because if the probability of a particular node is increased then the next successor node will be reached early. That is why we needed to increase the probability.

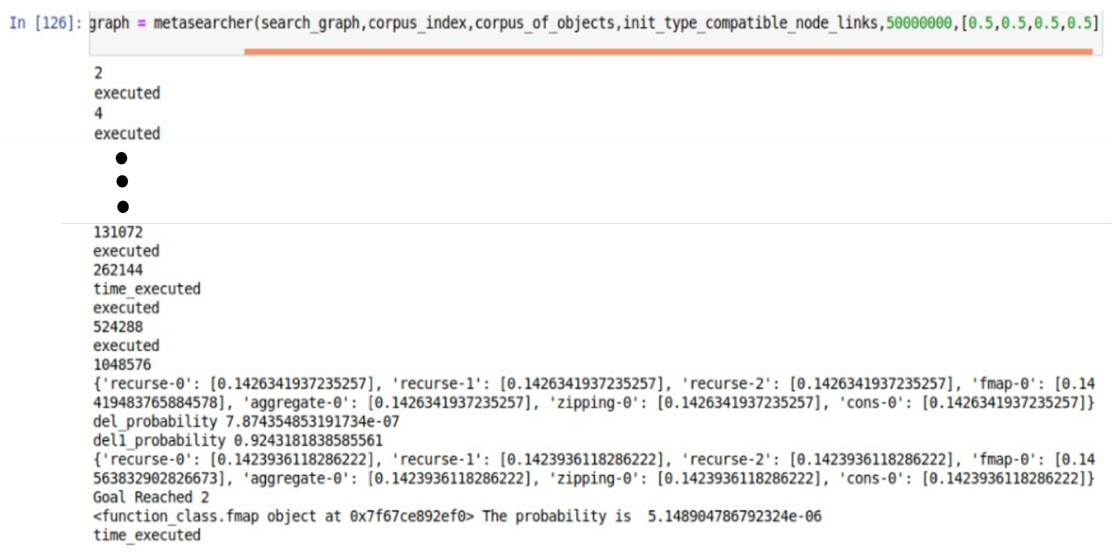


```
In [10]: metasearcher(search_graph,corpus_index,corpus_of_objects,init_type_compatible_node_links,50000000,'0.5')

2
4
executed
8
executed
16
32
64
executed
128
executed
256
del_probability 0.020230915224698137
del1_probability 0.3671706305702377
{'actuator_number-0': [0.07657480677563303], 'lambdagraph-0': [0.07657480677563303], 'recurse-2': [0.07657480677563303], 'mul
tiply-0': [0.07657480677563303], 'multiply-1': [0.07657480677563303], 'divide-0': [0.3874015457949358], 'divide-1': [0.076574
80677563303], 'add-0': [0.07657480677563303], 'add-1': [0.07657480677563303]}
Goal Reached
<function_class.divide object at 0x0000016C44B14358> The probability is  0.0028344186448836027
```

```
In [10]: for i in range (0,10):
             metasearcher(search_graph,corpus_index,corpus_of_objects,init_type_compatible_node_links,50000000000,'0.5')
             metasearcher(search_graph,corpus_index,corpus_of_objects,init_type_compatible_node_links,50000000000,'[0,0,0,0]')
             metasearcher(search_graph,corpus_index,corpus_of_objects,init_type_compatible_node_links,50000000000,'[0,0,0,0]')
executed
4096
executed
8192
executed
16384
time_executed
executed
32768
time_executed
executed
65536
time_executed
executed
131072
{'recurse-0': [0.139662315870026], 'recurse-1': [0.139662315870026], 'recurse-2': [0.139662315870026], 'fmap-0': [0.301688420
6498697], 'aggregate-0': [0.139662315870026], 'zipping-0': [0.139662315870026]}
Goal Reached 2
<function_class.fmap object at 0x0000016C44B45710> The probability is  3.1847088226101875e-05
time_executed
```

Figure 6.6: increased probability of divide and fmap by forced training.

Now as 'fmap2' is depended on both 'divide' and 'fmap1' so the increment of probabilities helped accelerating to reach 'fmap2'.

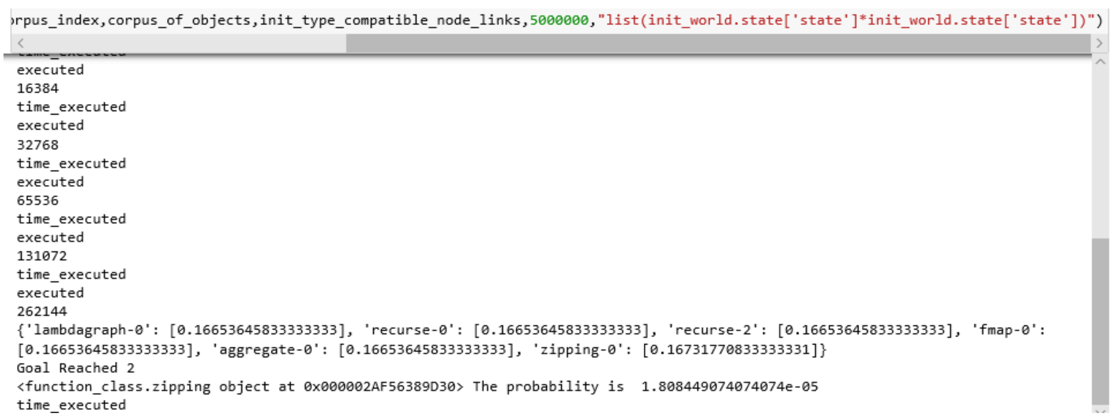


```
In [126]: graph = metasearcher(search_graph,corpus_index,corpus_of_objects,init_type_compatible_node_links,50000000,[0.5,0.5,0.5,0.5]

2
executed
4
executed
●
●
●
131072
executed
262144
time_executed
executed
524288
executed
1048576
{'recurse-0': [0.1426341937235257], 'recurse-1': [0.1426341937235257], 'recurse-2': [0.1426341937235257], 'fmap-0': [0.14
419483765884578], 'aggregate-0': [0.1426341937235257], 'zipping-0': [0.1426341937235257], 'cons-0': [0.1426341937235257]}
del_probability 7.874354853191734e-07
del1_probability 0.9243181838585561
{'recurse-0': [0.1423936118286222], 'recurse-1': [0.1423936118286222], 'recurse-2': [0.1423936118286222], 'fmap-0': [0.14
563832902826673], 'aggregate-0': [0.1423936118286222], 'zipping-0': [0.1423936118286222], 'cons-0': [0.1423936118286222]}
Goal Reached 2
<function_class.fmap object at 0x7f67ce892ef0> The probability is  5.148904786792324e-06
time_executed
```

Figure 6.7: Fmap2 function generation after forced training

But after reaching fmap2 we realized the issue of over-fitting while working on zipping. We have incremented the probability of the particular path so high that the model is unable to evaluate other possible paths. In this case the the constant -multiplication node could never be generated so we could never reach to zipping node.

For this issue to overcome we started training the model in a parallel way so that all the paths in a particular level should get trained simultaneously so that possibility of any path should not get decreased so much that the goal can't be generated. In this method we have again trained the model and got such data.



```
›rpus_index,corpus_of_objects,init_type_compatible_node_links,5000000,"list(init_world.state['state']*init_world.state['state'])")
```

```
executed
16384
time_executed
executed
32768
time_executed
executed
65536
time_executed
executed
131072
time_executed
executed
262144
{'lambdagraph-0': [0.16653645833333333], 'recurse-0': [0.16653645833333333], 'recurse-2': [0.16653645833333333], 'fmap-0':
[0.16653645833333333], 'aggregate-0': [0.16653645833333333], 'zipping-0': [0.16731770833333331]}
Goal Reached 2
<function_class.zipping object at 0x000002AF56389D30> The probability is  1.808449074074074e-05
time_executed
```

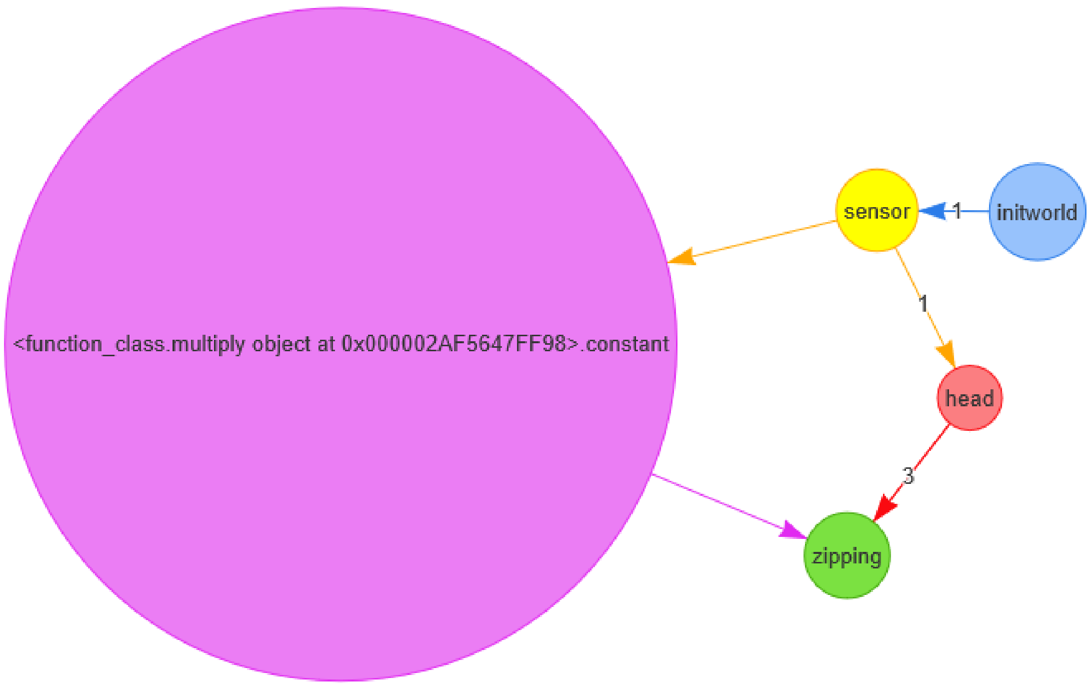
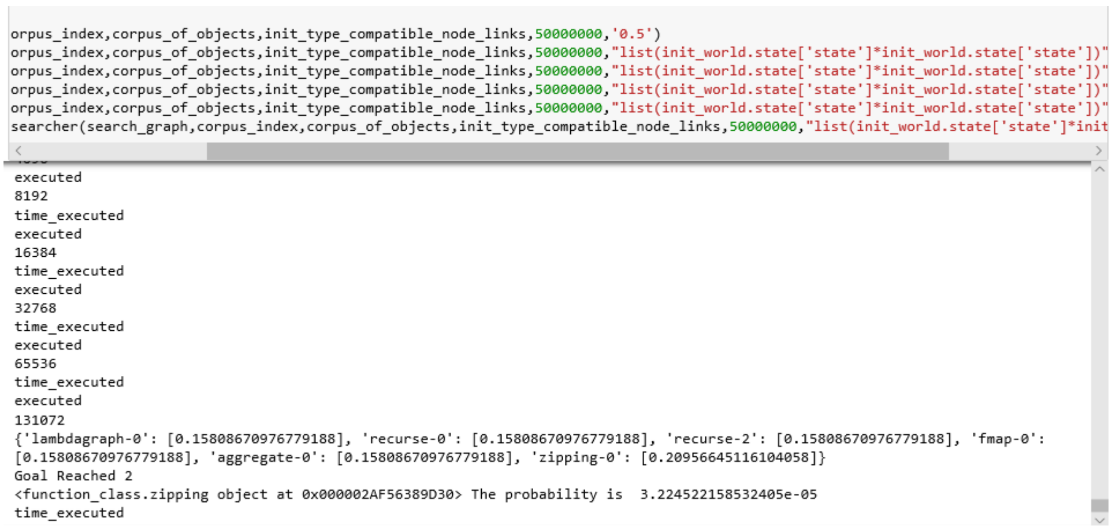
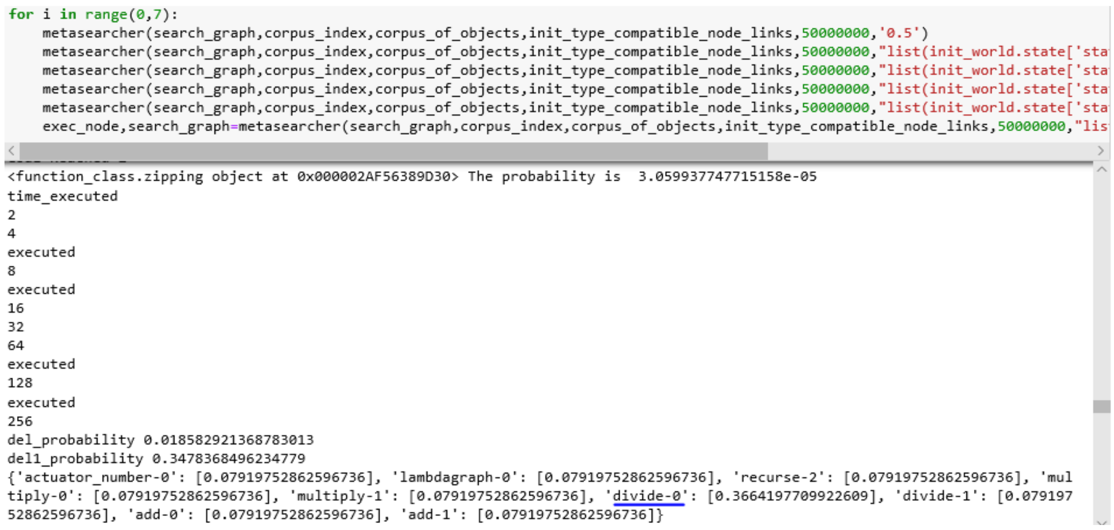

sensor
1
initworld
1
head
3
zipping
<function_class.multiply object at 0x000002AF5647FF98>.constant

Figure 6.8: Subgraph created by parallel training



```
orpus_index,corpus_of_objects,init_type_compatible_node_links,50000000,'0.5')
orpus_index,corpus_of_objects,init_type_compatible_node_links,50000000,"list(init_world.state['state']*init_world.state['state'])"
orpus_index,corpus_of_objects,init_type_compatible_node_links,50000000,"list(init_world.state['state']*init_world.state['state'])"
orpus_index,corpus_of_objects,init_type_compatible_node_links,50000000,"list(init_world.state['state']*init_world.state['state'])"
orpus_index,corpus_of_objects,init_type_compatible_node_links,50000000,"list(init_world.state['state']*init_world.state['state'])"
searcher(search_graph,corpus_index,corpus_of_objects,init_type_compatible_node_links,50000000,"list(init_world.state['state']*init
```

```
executed
8192
time_executed
executed
16384
time_executed
executed
32768
time_executed
executed
65536
time_executed
executed
131072
{'lambdagraph-0': [0.15808670976779188], 'recurse-0': [0.15808670976779188], 'recurse-2': [0.15808670976779188], 'fmap-0':
[0.15808670976779188], 'aggregate-0': [0.15808670976779188], 'zipping-0': [0.20956645116104058]}
Goal Reached 2
<function_class.zipping object at 0x000002AF56389D30> The probability is  3.224522158532405e-05
time_executed
```



```
for i in range(0,7):
    metasearcher(search_graph,corpus_index,corpus_of_objects,init_type_compatible_node_links,50000000,'0.5')
    metasearcher(search_graph,corpus_index,corpus_of_objects,init_type_compatible_node_links,50000000,"list(init_world.state['sta
    metasearcher(search_graph,corpus_index,corpus_of_objects,init_type_compatible_node_links,50000000,"list(init_world.state['sta
    metasearcher(search_graph,corpus_index,corpus_of_objects,init_type_compatible_node_links,50000000,"list(init_world.state['sta
    metasearcher(search_graph,corpus_index,corpus_of_objects,init_type_compatible_node_links,50000000,"list(init_world.state['sta
    exec_node,search_graph=metasearcher(search_graph,corpus_index,corpus_of_objects,init_type_compatible_node_links,50000000,"lis
```

```
<function_class.zipping object at 0x000002AF56389D30> The probability is  3.059937747715158e-05
time_executed
2
4
executed
8
executed
16
32
64
executed
128
executed
256
del_probability 0.018582921368783013
del1_probability 0.3478368496234779
{'actuator_number-0': [0.07919752862596736], 'lambdagraph-0': [0.07919752862596736], 'recurse-2': [0.07919752862596736], 'mul
tiply-0': [0.07919752862596736], 'multiply-1': [0.07919752862596736], 'divide-0': [0.3664197709922609], 'divide-1': [0.079197
52862596736], 'add-0': [0.07919752862596736], 'add-1': [0.07919752862596736]}
```

Figure 6.9: Parallel training to generate nodes.

Still the issue persisted as by parallel force training too the phase value is too high $[10^{-}9]$ to meet the goal for zipping. So due to computational restrictions we were not able to generate the whole program graph but methodically we were correct to solve it.

Now to check whether we can work it on distributed environment or not we took the help of MPI (Message Passing Interface)

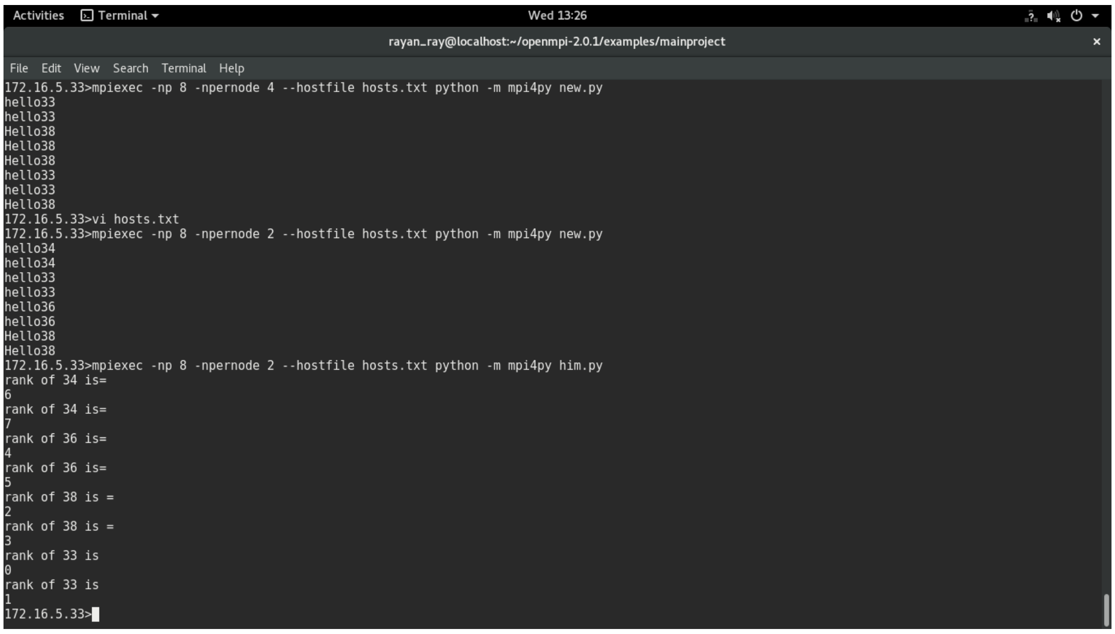
Here in our experiment 172.16.5.33 is the master computer and 34, 35, 36, 37, 38 are the other slave computers. There is a hostfile created in master computer naming hosts.txt and all the IP addresses of other machines are written in the file. Here is the command for mpi communication within the processors in standalone machine

Mpiexec -n 4 python -m mpi4py file.py

And to communicate with multiple computer and multiple processors, the below mentioned command should be executed in master computer. (i.e 172.16.5.33 in our case)

Mpiexec -n 8 -npernode 2 –hostfile hosts.txt python -m mpi4py file.py

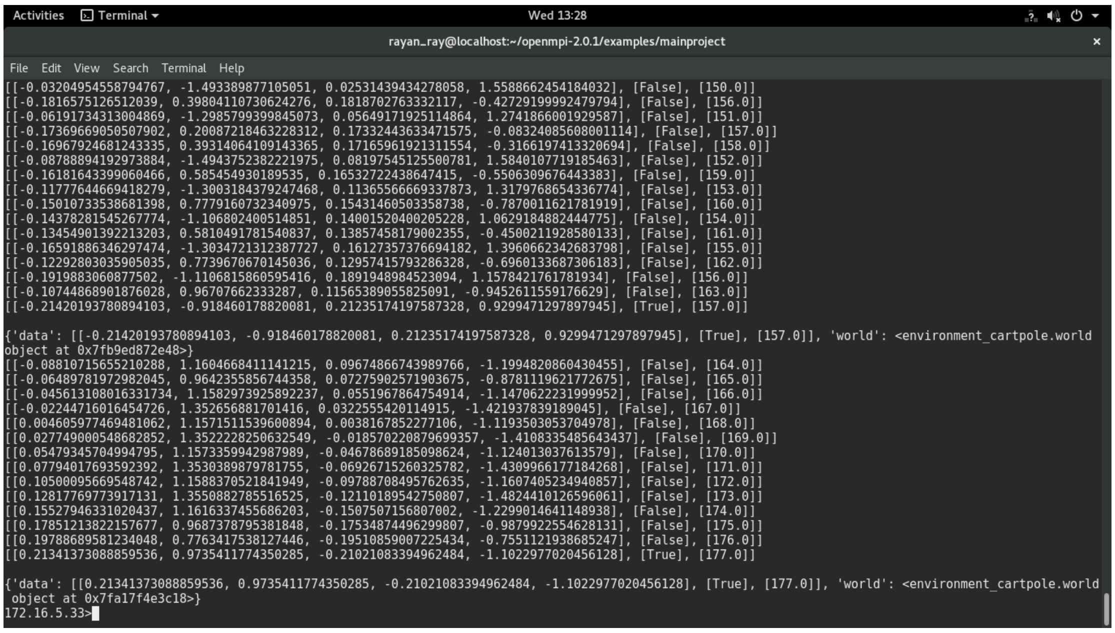
Here the connection between master and slave computers are established. And the screenshots are attached.


Activities
Terminal
Wed 13:26
rayan_ray@localhost:~/openmpi-2.0.1/examples/mainproject
File Edit View Search Terminal Help

```
172.16.5.33>mpiexec -np 8 -npernode 4 --hostfile hosts.txt python -m mpi4py new.py
hello33
hello33
Hello38
Hello38
Hello38
hello33
hello33
Hello38
172.16.5.33>vi hosts.txt
172.16.5.33>mpiexec -np 8 -npernode 2 --hostfile hosts.txt python -m mpi4py new.py
hello34
hello34
hello33
hello33
hello36
hello36
Hello38
Hello38
172.16.5.33>mpiexec -np 8 -npernode 2 --hostfile hosts.txt python -m mpi4py him.py
rank of 34 is=
6
rank of 34 is=
7
rank of 36 is=
4
rank of 36 is=
5
rank of 38 is =
2
rank of 38 is =
3
rank of 33 is
0
rank of 33 is
1
172.16.5.33>
```

Figure 6.10: connection between master and slave computers are established

Here is the cartpole problem where all the processors of total 6 machines are used and code is executed until the goal is reached and it returns true. Pole will be dislodged and episode will end.

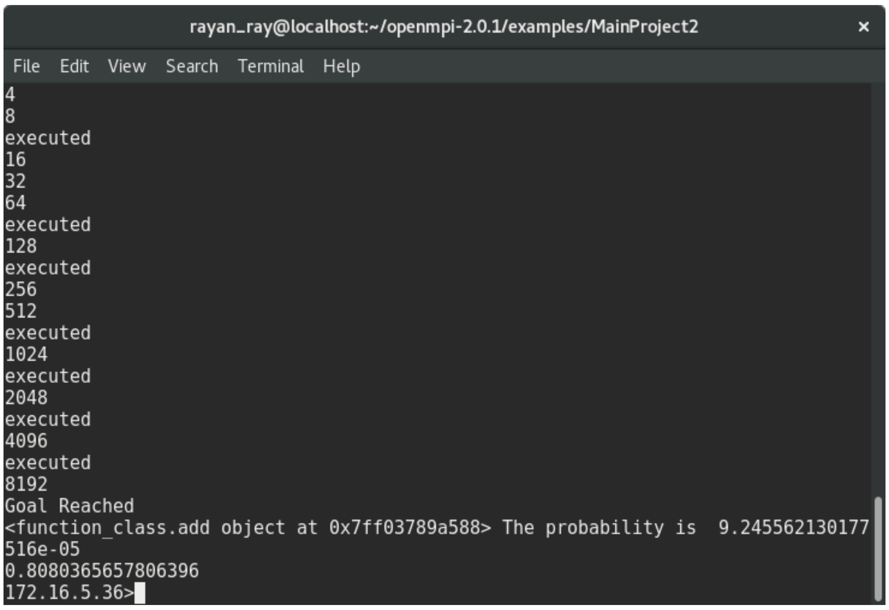


```
[[-0.03204954558794767, -1.493389877105051, 0.02531439434278058, 1.5588662454184032], [False], [150.0]]
[[-0.1816575126512039, 0.39804110730624276, 0.1818702763332117, -0.42729199992479794], [False], [156.0]]
[[-0.06191734313004869, -1.2985799399845073, 0.05649171925114864, 1.2741866001929587], [False], [151.0]]
[[-0.17369669050507902, 0.20087218463228312, 0.17332443633471575, -0.08324085608001114], [False], [157.0]]
[[-0.16967924681243335, 0.39314064109143365, 0.17165961921311554, -0.3166197413320694], [False], [158.0]]
[[-0.08788894192973884, -1.4943752382221975, 0.08197545125500781, 1.5840107719185463], [False], [152.0]]
[[-0.16181643399060466, 0.585454930189535, 0.16532722438647415, -0.5506309676443383], [False], [159.0]]
[[-0.11777644669418279, -1.3003184379247468, 0.11365566669337873, 1.3179768654336774], [False], [153.0]]
[[-0.15010733538681398, 0.7779160732340975, 0.15431460503358738, -0.7870011621781919], [False], [160.0]]
[[-0.14378281545267774, -1.106802400514851, 0.14001520400205228, 1.0629184882444775], [False], [154.0]]
[[-0.13454901392213203, 0.5810491781540837, 0.13857458179002355, -0.4500211928580133], [False], [161.0]]
[[-0.16591886346297474, -1.3034721312387727, 0.16127357376694182, 1.3960662342683798], [False], [155.0]]
[[-0.12292803035905035, 0.7739670670145036, 0.12957415793286328, -0.6960133687306183], [False], [162.0]]
[[-0.1919883060877502, -1.1106815860595416, 0.1891948984523094, 1.1578421761781934], [False], [156.0]]
[[-0.10744868901876028, 0.96707662333287, 0.11565389055825091, -0.9452611559176629], [False], [163.0]]
[[-0.21420193780894103, -0.918460178820081, 0.21235174197587328, 0.9299471297897945], [True], [157.0]]

{'data': [[-0.21420193780894103, -0.918460178820081, 0.21235174197587328, 0.9299471297897945], [True], [157.0]], 'world': <environment_cartpole.world
object at 0x7fb9ed872e48>}
[[-0.08810715655210288, 1.1604668411141215, 0.09674866743989766, -1.1994820860430455], [False], [164.0]]
[[-0.06489781972982045, 0.9642355856744358, 0.07275902571903675, -0.8781119621772675], [False], [165.0]]
[[-0.045613108016331734, 1.1582973925892237, 0.0551967864754914, -1.1470622231999952], [False], [166.0]]
[[-0.02244716016454726, 1.352656881701416, 0.0322555420114915, -1.421937839189045], [False], [167.0]]
[[0.004605977469481062, 1.1571511539600894, 0.0038167852277106, -1.1193503053704978], [False], [168.0]]
[[0.027749000548682852, 1.3522228250632549, -0.018570220879699357, -1.4108335485643437], [False], [169.0]]
[[0.05479345704994795, 1.1573359942987989, -0.04678689185098624, -1.124013037613579], [False], [170.0]]
[[0.07794017693592392, 1.3530389879781755, -0.06926715260325782, -1.4309966177184268], [False], [171.0]]
[[0.10500095669548742, 1.1588370521841949, -0.09788708495762635, -1.1607405234940857], [False], [172.0]]
[[0.12817769773917131, 1.3550882785516525, -0.12110189542750807, -1.4824410126596061], [False], [173.0]]
[[0.15527946331020437, 1.1616337455686203, -0.1507507156807002, -1.2299014641148938], [False], [174.0]]
[[0.17851213822157677, 0.9687378795381848, -0.17534874496299807, -0.9879922554628131], [False], [175.0]]
[[0.19788689581234048, 0.7763417538127446, -0.19510859007225434, -0.7551121938685247], [False], [176.0]]
[[0.21341373088859536, 0.9735411774350285, -0.21021083394962484, -1.1022977020456128], [True], [177.0]]

{'data': [[0.21341373088859536, 0.9735411774350285, -0.21021083394962484, -1.1022977020456128], [True], [177.0]], 'world': <environment_cartpole.world
 object at 0x7fa17f4e3c18>}
172.16.5.33>
```

Activities
Terminal
Wed 13:28
rayan_ray@localhost:~/openmpi-2.0.1/examples/mainproject
File Edit View Search Terminal Help

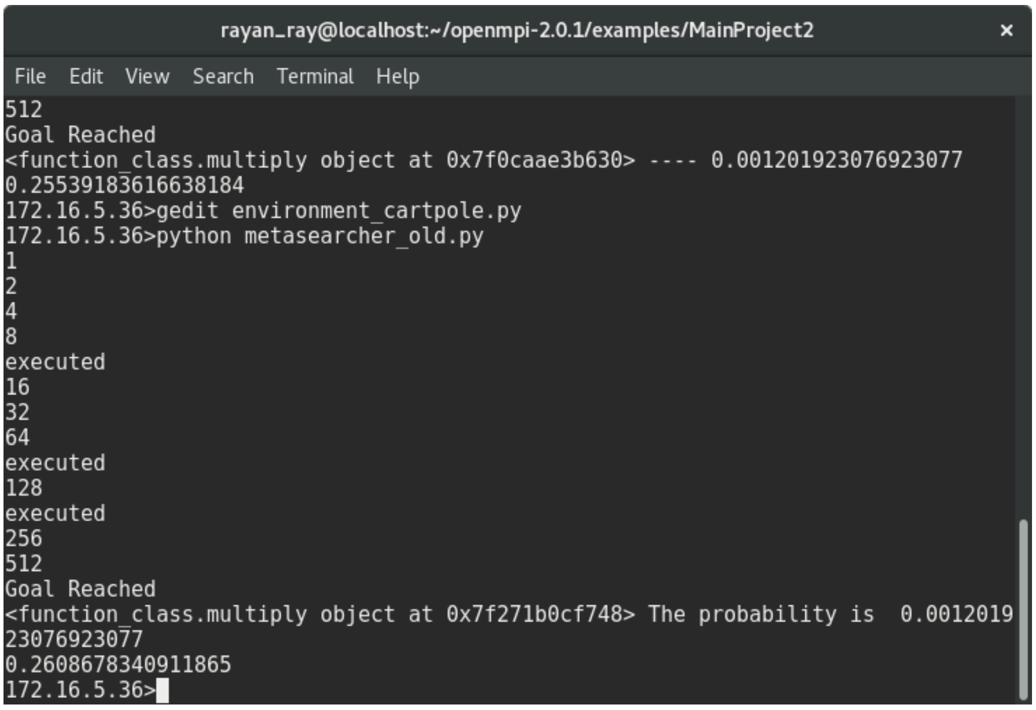
Figure 6.11: Cartpole problem code is run in MPI to reach the goal

Now the metasearcher algorithm is updated to run different test problems where reward is given if correct subgraph is made in the slave computers and gathering all the subgraphs back in master computer by MPIcomm.Send() and MPIcomm.receive() messages to reach the destinations. The screenshots of all the output given for input data and the graphs created after reaching the goals are attached here.

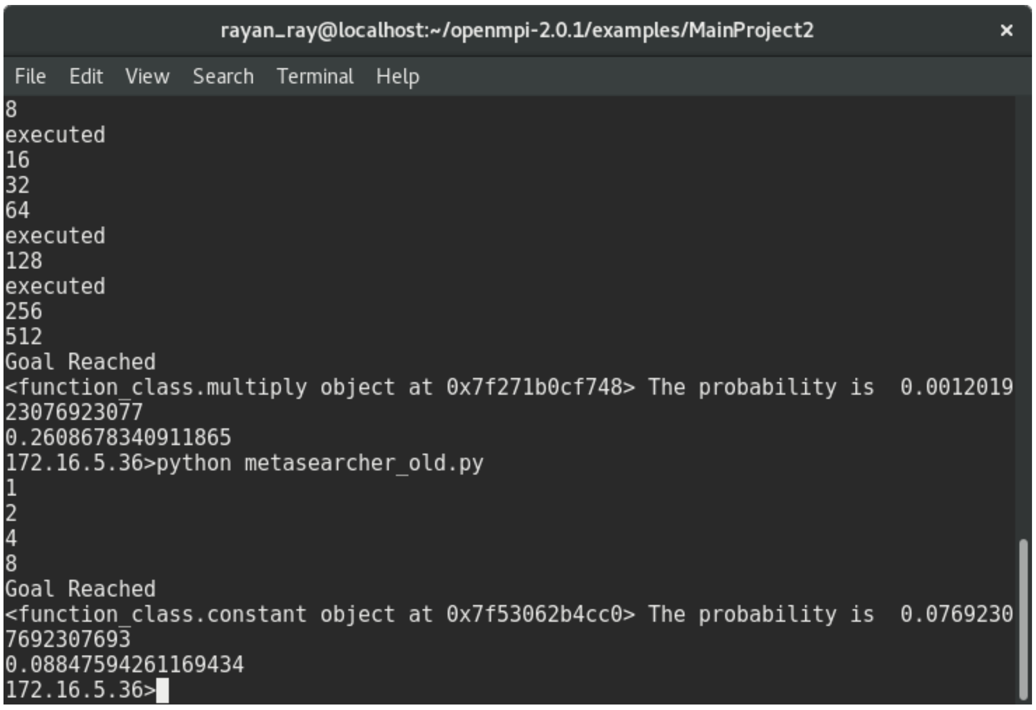


```
4
8
executed
16
32
64
executed
128
executed
256
512
executed
1024
executed
2048
executed
4096
executed
8192
Goal Reached
<function_class.add object at 0x7ff03789a588> The probability is  9.245562130177
516e-05
0.8080365657806396
172.16.5.36>
```

rayan_ray@localhost:~/openmpi-2.0.1/examples/MainProject2
File Edit View Search Terminal Help


rayan_ray@localhost:~/openmpi-2.0.1/examples/MainProject2
File Edit View Search Terminal Help

```
512
Goal Reached
<function_class.multiply object at 0x7f0caae3b630> ---- 0.001201923076923077
0.25539183616638184
172.16.5.36>gedit environment_cartpole.py
172.16.5.36>python metasearcher_old.py
1
2
4
8
executed
16
32
64
executed
128
executed
256
512
Goal Reached
<function_class.multiply object at 0x7f271b0cf748> The probability is  0.0012019
23076923077
0.2608678340911865
172.16.5.36>
```


rayan_ray@localhost:~/openmpi-2.0.1/examples/MainProject2
File Edit View Search Terminal Help

```
8
executed
16
32
64
executed
128
executed
256
512
Goal Reached
<function_class.multiply object at 0x7f271b0cf748> The probability is  0.0012019
23076923077
0.2608678340911865
172.16.5.36>python metasearcher_old.py
1
2
4
8
Goal Reached
<function_class.constant object at 0x7f53062b4cc0> The probability is  0.0769230
7692307693
0.08847594261169434
172.16.5.36>
```

Figure 6.12: Goal reached for add, multiply, constant function using metasearcher algorithm using MPI environment and subgraph are created.

So in this experiment we have done

- Implemented list operator so that we can take the observation state of cartpole as a list and the environment can be connected through sensor to have this data to reach goal.
- We have added fmap, zipping, aggregate, head, tail etc all the relevant functionality.
- We have designed the model of cartpole problem in FGPL using graph flow paradigm.
- We have implemented small arithmetic operations changing the reward values in standalone as well as distributed environment to compare the data when the goal is reached.
- We have tried learning the cartpole problem to a model so that when we provide the test data, it can accordingly draw the whole graph that we have designed previously.
- We worked on comparing the time and performance enhancement of the cartpole problem using normal python code in single machine running the metasearcher algorithm as well as in distributed system in MPI communication system.
- Through the $initWorld()$ we have taken $[[x_1, x_2, x_3, x_4], Boolean, Reward]$ initial state value of cartpole from the environment. Where $x_1$is position of the cart, $x_2$is the velocity of the cart, x_3 angle of the pole, x_4 velocity of the pole. Whenever for a particular state value and reward the pole is not dislodged then it will return false and as soon as it is disbalanced, it will return boolean true and the episode will end and goal will be reached.
- Thus we have worked on developing the whole graph of cartpole problem.

Here are 2 tables of time taken in distributed system(WITH MPI) and stanalone system(WITHOUT MPI) to reach the goal for some small arithmatic operations(eg: addition, multiply, divide) using metasearcher algorithm for distributed environment and metasearcher algorithm for non-distributed environment respectively. The table is adjusted in increasing order of reward.

Reward	Time taken without MPI (second)	Maximum Phase value
2	0.9301	50000000
3	69.17	50000000
4	376.82	50000000
5	369.77	50000000

Table 6.1: WITHOUT MPI

Reward	Time taken with MPI (second)	Maximum Phase value
2	0.594	50000000
3	45.8280	50000000
4	156.24	50000000
5	156.80	50000000

Table 6.2: WITH MPI

CHAPTER 7

Conclusion and Future Work

We have done our experiment in python to test the effectiveness of FGPL in this cartpole problem. And we have been successfully implemented incremental learning concept using program probability of the subgraphs created in each level. We have also successfully established distributed environment and are able to communicate through all the machines using proper master slave architecture. We have obvious enhanced performance results for some short arithmetical problems. But as cartpole problem is being too computationally heavy in stand alone machines that to reach the particular phase value for meeting the goal we had to forcefully increment program probabilities of possible program path. This tends to over-fitting issue and we could not view the complete graph at a stretch without biased training. This was the limitation of our work. We have thus come up with a solution of parallel training for the model such that the probability of any particular problem path should neither get too much decreased , nor too much increased to avoid over-fitting. However we need to improve FGPL framework further. FGPL should be capable of self- generation, self-modification or self-evaluating codes. We need to test the model for different problem and challenges so that we can make the language more flexible and more effective.

APPENDIX A

Code snippets

Finding Nth Term Of An AP Series

```python
import cProfile
import sys
sys.path.append("./FGPL/UIPS-master/UIPS-master")
from metasearcher_new import *
from graph_draw import *
a=int(input("Enter first term = "))
d=int(input("Enter common difference = "))
n=int(input("Enter n = "))
k=initGraph(1)
i0=Node(1,'initWorld')
s1=Node(2,'sensor','number')
c1=Node(5,'constant',a)
c2=Node(7,'constant',d)
c3=Node(9,'constant',n-1)
#c3=Node(11,'constant',4)
mul=Node(12,'multiply')
addition=Node(13,'add')
addNode(1,s1,1)
addNode(1,c1,2)
addNode(1,c2,2)
addNode(1,c3,2)
addNode(1,mul,7,9)
addNode(1,addition,5,12)
print(evalGraph(k))
draw_graph(k)
```

Recursive approach to find Nth Term Of An AP Series

```
import cProfile
import sys
sys.path.append("./FGPL/UIPS-master/UIPS-master")
from metasearcher_new import *
from graph_draw import *

k=initGraph(1)
i=Node(1,'initWorld')
s1=Node(2,'sensor','number')
#s1.funct()
#s1.data = 0
c1=Node(3,'constant',1)
c2=Node(4,'constant',45)
c3=Node(5,'constant',2)
less=Node(7,'greater')
addition=Node(8,'add')
l1=Node(9,'lambdagraph')
l2=Node(10,'lambdagraph')
r=Node(11,'recurse')

addNode(1,s1,1)
addNode(1,c1,2)
addNode(1,c2,2)
addNode(1,c3,2)
addNode(1,addition,5,2)
addNode(1,less,2,4)
addNode(1,l1,8)
addNode(1,l2,7)
addNode(1,r,9,10,3)
print(evalGraph(k))
draw_graph(k)
```

Implementation of head operator in function class

```
class head(node):

# Node class for head function
  def __init__(self,label, *links):
    node.__init__(self,label,'head',1,{'function':{'input':
       ['list'],'output':['any']}},links)
    #super().update_program_expression('head',None)

  def funct(self):
    super().funct()
    temp_list = self.links[0].funct()
    if self.data == None:
      link1_in = self.links[0].funct()
      if not isinstance(link1_in['data'],list):
        raise Exception ('Invalid Input Type for head')
      try:
        self.data = link1_in['data'][0]
      except IndexError:
        self.data = None
      self.world = link1_in['world']
      self.world_version = self.links[0].world_version
      update_node_type(self)
      super().update_program_expression('head',self.data)
    return {'data':pickle.loads(pickle.dumps(self.data,-1)),
       'world':self.world}
```

Implementation of tail operator in function class

```
class tail(node):

# Node class for tail function
  def __init__(self,label, *links):
    node.__init__(self,label,'tail',1,
    {'function':{'input':['list'],'output':['list']}},links)
    #super().update_program_expression('tail',None)

  def funct(self):
    super().funct()
    list_var = self.links[0].funct()
    if self.data == None:
      link1_in = self.links[0].funct()
      temp_list = link1_in['data']
      if not isinstance(temp_list,list):
        raise Exception ('Invalid Input Type tail')
      self.data = temp_list[1:len(temp_list)]
      self.world = link1_in['world']
      self.world_version = self.links[0].world_version
      #update_node_type(self)
      super().update_program_expression('tail',self.data)
    return {'data':pickle.loads(pickle.dumps(self.data,-1)),
    'world':self.world}
```

Implementation of nil operator in function class

```
class nil(node):
# Node class for nil function
  def __init__(self,label,*links):
    node.__init__(self,label,'nil',1,
    {'function':{'input':['any'],'output':['list']}},links)
    #super().update_program_expression('nil',None)

  def funct(self):
    super().funct()
    if self.data == None:
      link1_in = self.links[0].funct()
      self.data = []
      self.world = link1_in['world']
      #self.world_version = self.world.version
      self.world_version = self.links[0].world_version
      #update_node_type(self)
      super().update_program_expression('nil',self.data)
    return {'data':pickle.loads(pickle.dumps(self.data,-1)),
    'world':self.world}
```

Implementation of cons operator in function class

```
class cons(node):
# Node class for cons function
  def __init__(self,label, *links):
    node.__init__(self,label,'cons',2,
    {'function':{'input':['any','list'],
    'output':['list']}},links)
    #super().update_program_expression('cons',None)

  def funct(self):
    super().funct()
    if self.data == None:
      link1_in = self.links[1].funct()
      #if not isinstance(link1_in['data'],list):
        #  raise Exception ('Invalid Input Type for cons')
        link2_in = self.links[0].funct()
        link1_in['data'].insert(0,link2_in['data'])
        self.data = link1_in['data']
        self.world = link1_in['world']
        self.world_version =
           max(self.links[0].world_version,self.links[1].world_version)
        #update_node_type(self)
        super().update_program_expression('cons',self.data)

    return {'data':pickle.loads(pickle.dumps(self.data,-1)),
    'world':self.world}
```

Implementation of fmap operator in function class

```
class fmap(node):

  def __init__(self,label, *links):
    node.__init__(self,label,'fmap',2,
    {'function':{'input':['function','list'],
    'output':['list']}},links)
    #super().update_program_expression('fmap',None)

  def funct(self):
    global globalvars
    #print(str(self.links[1].program_expression['data']))
    super().funct()
    node_list_dict ={}
    if self.data == None:
      self.g = Graph(globalvars.graph_label)
      node_list_dict[1] = identity(globalvars.node_label)
      node_list_dict[2] = identity(globalvars.node_label)
      node_list_dict[3] = head(globalvars.node_label)
      node_list_dict[4] = nil(globalvars.node_label)
      node_list_dict[5] = apply(globalvars.node_label)
      node_list_dict[6] = tail(globalvars.node_label)
      node_list_dict[7] = equal(globalvars.node_label)
      node_list_dict[8] = fmap(globalvars.node_label)
      node_list_dict[9] = cons(globalvars.node_label)
      node_list_dict[10] = gaurd(globalvars.node_label)

      for node_i in node_list_dict.values():
          node_i.update_exp = 0

      self.g.add_node(node_list_dict[1])
      self.g.add_node(node_list_dict[2])
      self.g.add_node(node_list_dict[3],node_list_dict[2])
      self.g.add_node(node_list_dict[4],node_list_dict[2])
      self.g.add_node(node_list_dict[5],node_list_dict[1],
      node_list_dict[3])
      self.g.add_node(node_list_dict[6],node_list_dict[2])
      self.g.add_node(node_list_dict[7],node_list_dict[2],
      node_list_dict[4])
      self.g.add_node(node_list_dict[8],node_list_dict[1],
      node_list_dict[6])
      self.g.add_node(node_list_dict[9],node_list_dict[5],
      node_list_dict[8])
      self.g.add_node(node_list_dict[10],node_list_dict[7],
      node_list_dict[6],node_list_dict[9])
      self.g.glinks = self.links

      for i in range(len(self.links)):
          self.g.initialnodes[i].links = (self.links[i],)

      output = self.g.terminalnodes[0].funct()
      self.data = output['data']
      self.world = output['world']
      super().update_program_expression('fmap',self.data)
      del(node_list_dict)
      del(output)
      self.world_version =
         self.g.terminalnodes[0].world_version
         #self.world.version
    return {'data':pickle.loads(pickle.dumps(self.data,-1)),
    'world':self.world}
```

Implementation of zipping operator in function class

```
class zipping(node):

  def __init__(self,label, *links):

    node.__init__(self,label,'zipping',3,
    {'function':{'input':['function','list','list'],
    'output':['list']}},links)

  def funct(self):
    global globalvars
    super().funct()
    node_list_dict ={}
    if self.data == None:
      self.g = Graph(globalvars.graph_label)
      node_list_dict[1] = identity(globalvars.node_label)
      node_list_dict[2] = identity(globalvars.node_label)
      node_list_dict[3] = identity(globalvars.node_label)
      node_list_dict[4] = head(globalvars.node_label)
      node_list_dict[5] = tail(globalvars.node_label)
      node_list_dict[6] = nil(globalvars.node_label)
      node_list_dict[7] = head(globalvars.node_label)
      node_list_dict[8] = tail(globalvars.node_label)
      node_list_dict[9] = apply(globalvars.node_label)
      node_list_dict[10] = zipping(globalvars.node_label)
      node_list_dict[11] = equal(globalvars.node_label)
      node_list_dict[12] = cons(globalvars.node_label)
      node_list_dict[13] = gaurd(globalvars.node_label)

      for node_i in node_list_dict.values():
        node_i.update_exp = 0

      self.g.add_node(node_list_dict[1])
      self.g.add_node(node_list_dict[2])
      self.g.add_node(node_list_dict[3])
    self.g.add_node(node_list_dict[4],node_list_dict[2])
    self.g.add_node(node_list_dict[5],node_list_dict[2])
    self.g.add_node(node_list_dict[6],node_list_dict[2])
    self.g.add_node(node_list_dict[7],node_list_dict[3])
    self.g.add_node(node_list_dict[8],node_list_dict[3])
    self.g.add_node(node_list_dict[9],node_list_dict[1],
    node_list_dict[4],node_list_dict[7])
    self.g.add_node(node_list_dict[10],node_list_dict[1],
    node_list_dict[5],node_list_dict[8])
    self.g.add_node(node_list_dict[11],node_list_dict[2],
    node_list_dict[6])
    self.g.add_node(node_list_dict[12],node_list_dict[9],
    node_list_dict[10])
    self.g.add_node(node_list_dict[13],node_list_dict[11],
    node_list_dict[6],node_list_dict[12])
    self.g.glinks = self.links

    for i in range(len(self.links)):
      self.g.initialnodes[i].links = (self.links[i],)

    output = self.g.terminalnodes[0].funct()
    self.data = output['data']
    self.world = output['world']
    #self.program_expression =
        self.g.terminalnodes[0].program_expression
    super().update_program_expression('recurse',self.data)
    del(node_list_dict)
    del(output)
    self.world_version =
        self.g.terminalnodes[0].world_version
        #self.world.version
  return {'data':pickle.loads(pickle.dumps(self.data,-1)),
  'world':self.world}
```

Implementation of aggregate operator in function class

```
class aggregate(node):

  def __init__(self,label, *links):
    node.__init__(self,label,'aggregate',2,
    {'function':{'input':['function','list'],
    'output':['any']}},links)
    #super().update_program_expression('aggregate',None)

  def funct(self):
    global globalvars
    #print(str(self.links[1].program_expression['data']))
    super().funct()
    node_list_dict ={}
    if self.data == None:
      self.g = Graph(globalvars.graph_label)
      node_list_dict[1] = identity(globalvars.node_label) #1
      node_list_dict[2] = identity(globalvars.node_label) #2
      node_list_dict[3] = head(globalvars.node_label) #5
      node_list_dict[4] = nil(globalvars.node_label)
      node_list_dict[5] = apply(globalvars.node_label) #4
      node_list_dict[6] = tail(globalvars.node_label) #6
      node_list_dict[7] = aggregate(globalvars.node_label) #7
      node_list_dict[8] = equal(globalvars.node_label)
      #node_list_dict[9] = apply(globalvars.node_label)
      node_list_dict[10] = gaurd(globalvars.node_label) #8

      for node_i in node_list_dict.values():
        node_i.update_exp = 0
      self.g.add_node(node_list_dict[1])
      self.g.add_node(node_list_dict[2])
      self.g.add_node(node_list_dict[3],node_list_dict[2])
      self.g.add_node(node_list_dict[4],node_list_dict[2])
      #self.g.add_node(node_list_dict[5],node_list_dict[1],
      node_list_dict[3])
    self.g.add_node(node_list_dict[6],node_list_dict[2])
    self.g.add_node(node_list_dict[7],node_list_dict[1],
    node_list_dict[6])
    self.g.add_node(node_list_dict[8],node_list_dict[6],
    node_list_dict[4])
    self.g.add_node(node_list_dict[5],node_list_dict[1],
    node_list_dict[3],node_list_dict[7])
    self.g.add_node(node_list_dict[10],node_list_dict[8],
    node_list_dict[3],node_list_dict[5])
    self.g.glinks = self.links
  #print(self.links)
    for i in range(len(self.links)):
      self.g.initialnodes[i].links = (self.links[i],)
    #print(self.g.terminalnodes[0].update_exp)
    #globalvars.recurse_temp_obj.append(self.g.nodes)
    output = self.g.terminalnodes[0].funct()
    self.data = output['data']
    self.world = output['world']
    #self.program_expression =
        self.g.terminalnodes[0].program_expression
    super().update_program_expression('aggregate',self.data)
    del(node_list_dict)
    del(output)
    self.world_version =
        self.g.terminalnodes[0].world_version
        #self.world.version
    update_node_type(self)
  return {'data':pickle.loads(pickle.dumps(self.data,-1)),
  'world':self.world}
```

REFERENCES

1. **Bird, R.** *et al.*, *Introduction to functional programming using Haskell*, volume 2. Prentice Hall Englewood Cliffs, 1998. 3, 12

2. **Burns, J.** and **J.-L. Gaudiot** (2002). Smt layout overhead and scalability. *IEEE Transactions on Parallel and Distributed Systems*, **13**(2), 142–155. 17

3. **Burton, F. W.** and **H.-K. Yang** (1990). Manipulating multilinked data structures in a pure functional language. *Software Practice & Experience*, **20**(11), 1167–1185. 16

4. **Chan, M.** (2016). cart-pole-balancing-with-q-learning. `https://medium.com/@tuzzer/cart-pole-balancing-with-q-learning-b54c6068d947`. Accessed: 2016-11-13. 10

5. **Erwig, M.** (2001). Inductive graphs and functional graph algorithms. *Journal of Functional Programming*, **11**(5), 467–492. 11

6. **Feo, J. T.**, **D. C. Cann**, and **R. R. Oldehoeft** (1990). A report on the sisal language project. *Journal of Parallel and Distributed Computing*, **10**(4), 349–366. 17

7. **Hutter, M.**, The fastest and shortest algorithm for all well-defined problems. International Journal of Foundations of Computer Science, 2002. 14

8. **Hutter, M.** (2007). Algorithmic information theory. *Scholarpedia*, **2**(3), 2519. Revision #186543. 3, 4

9. **Kashiwagi, Y.** and **D. S. Wise**, *Graph algorithms in a lazy functional programming language*. Computer Science Department, Indiana University, 1991. 17

10. **Legg, S.** and **M. Hutter**, Universal intelligence: A definition of machine intelligence. Minds and machines, 2007. 1

11. **Levin, L. A.** (1973). Universal sequential search problems. *Problemy Peredachi Informatsii*, **9**(3), 115–116. 4

12. **Li, M.** and **P. Vitányi**, *An introduction to Kolmogorov complexity and its applications*. Springer Science & Business Media, 2013. 5

13. **Looks, M.** and **B. Goertzel**, Program representation for general intelligence. *In Proceedings of the 2nd Conference on Artificiel General Intelligence (2009)*. Atlantis Press, 2009. 2, 16

14. **Paul, S. K.** and **P. Bhaumik**, A fast universal search by equivalent program pruning. *In 2016 International Conference on Advances in Computing, Communications and Informatics (ICACCI)*. IEEE, 2016. 14

15. **Paul, S. K.** and **P. Bhaumik**, A functional graph programming language for universal search based general intelligent agents. communicated. 1

16. **Schaul, T.** and **J. Schmidhuber**, Towards practical universal search. *In 3d Conference on Artificial General Intelligence (AGI-2010).* Atlantis Press, 2010. 1

17. **Schmidhuber, J.**, Optimal ordered problem solver. Machine Learning 54.3, 2004. 14

18. **Solomonoff, R. J.**, A system for incremental learning based on algorithmic probability. *In Proceedings of the Sixth Israeli Conference on Artificial Intelligence, Computer Vision and Pattern Recognition*. 1989. 5

19. **Solomonoff, R. J.**, Algorithmic probability, heuristic programming and agi. Atlantis Press, 2010. 1

20. **Sutton, R. S.**, **A. G. Barto**, *et al.*, *Introduction to reinforcement learning*, volume 135. MIT press Cambridge, 1998. 8

21. **Thórisson, K. R.**, **N. Nivel**, **B. R. Steunebrink**, **H. P. Helgason**, **G. Pezzulo**, **R. Sanz Bravo**, **J. Schmidhuber**, **H. Dindo**, **M. Rodríguez Hernández**, **A. Chella**, *et al.* (2014). Autonomous acquisition of natural situated communication. *IADIS International Journal on Computer Science And Information Systems*, **9**(2), 115–131. 16

22. **Wadler, P.**, Monads for functional programming. *In International School on Advanced Functional Programming*. Springer, 1995. 3