REPORT OF AIP GROUP COURSEWORK (TEAM AITISSISSIMO)

Yilei Liang (K1764097) Yanpu Huang (K1763861) Xinchen Xu (K1763538) Yulai Gu (K1763700)

ABSTRACT

This report illustrates our conception of the best planner algorithm to run the **PDDL** planning tasks. In this project, we modified the beam search algorithm by adding restart function, evaluating the performance with Hill Climbing Search, Enforced Hill Climbing Search and Local Search Algorithm. We also developed a stochastic beam search algorithm so that it can reach the global optimum state occasionally. We also found a BUG in the provided HValue comparator of JavaFF: **Comparison method violates its general contract**.

1 INTRODUCTION

In this coursework, we first evaluate the efficiency for the algorithm, filters and successor selector we developed in Part 1, 2 and 3. After that, we decided to browse more algorithm such as simulated annealing – which is really similar to Hill Climbing Search + Roulette Successor Selector, Local Beam Search, Stochastic Beam Search and Genetic Algorithm. We decided to implement Local Beam Search for our satisfying task and Stochastic Beam Search for our optimizing task as they're easy to implement and able to provide fast heuristic searching. We also found that there is a BUG in $HValueComparator$ as when we want to use this comparator to sort our Linked List, we found that "Comparison method violates its general contract" might arise (occasionally).

2 PREVIOUS ALGORITHM ANALYSIS

2.1 Hill Climbing Search

In part 2 & 3, we developed the hill climbing search with different filters and successor selectors. To find a plan as soon as possible, we can consider to make a trade-off between memory usage and time complexity – we can keep multiple states rather than just the joint-best state – which bring us from HC to Beam Search

2.2 Local Search

In part 3.7, we developed a Local Searching Algorithm with restart mechanism. Although we didn't use this in our satisfying task, it do provide us a new idea to apply on our beam search – **Restarting**.

2.3 Filter and Successor Selectors

In previous parts, we applied some random stuffs such as $RandomThreeFilter$ and $RouletteSelectors$ – which can bring us to global optimise state occasionally. The Helpful filter can accelerate the hill climbing search (but not beam search as we are now consider m best successors). In optimising part, we modified the $RouletteSelectors$ to let it select m states rather than 1 state.

3 SATISFYING PLANNING

3.1 Modified Beam Search Algorithm

In this Beam Search algorithm, we first consider best m ($beamSize$) successors, stored the rest successors at $remainBeam$. If there is no unvisited successor benn added to $currentBeam$, it will restart from the $remainBeam$ of last iteration. This might violate the purpose of beam search – to save memory, but we found this solution can find a solution really quickly. This is a trade-off between memory and time. It's true that the $currentBeam$ will convergence to a small group of states – which might not able to find global optimum state. In Satisfying part, we just used this approach as we just need the local optimum state. In our optimizing part, we modified it by using Roulette Selector (now it select m states rather than one state) to select states randomly (so that we might able to find the global optimum state at some point. More details about Stochastic Beam Search will be discussed in Section 4.

3.2 Parameter setting in JavaFF

As we described in ***Algorithm 1***, the **modified beam search** algorithm need to take initial state S and Beam Size m (how many states we want to keep) for the parameter of the algorithm. We found that if we use a low $beamSize$ value, its performance will be good in small scale problem. But when facing a large scale problem, we found that it will be a good choice to use a larger $beamSize$. Therefore in our satisfying part, we start use a small $beamSize$ value and increment this value if it failed to find a solution by the current $beamSize$ value till 15 – which is a relative large value. Our evaluation result showed that in problem like Driverlog 2, it only took about 0.2 seconds to find a solution

Algorithm 1 Beam Search with Restart

```
  Input: States $S$, beamSize $m$
  repeat
    Initialize $currentBeam = \{\}$
    for $State\ s$ in $S$ do
      $successors = state.getSucessors()$
      for $Successors\ s'$ in $S'$ do
        if $needToVisit(s')$ then
          if $s'.goalReached()$ then
            $return\ s'$
          else
            $currentBeam.add(s')$
          end if
        end if
      end for
      if $currentBeam.isEmpty()$ then
        if NOT $remainBeam.isEmpty()$ then
          $S = remainBeam$
          continue      //Restart with remainBeam
        end if
      end if
      Sort($currentBeam$)
      if $currentBeam.size() \leq beamSize$ then
        $S = currentBeam$
      else
        $S = currentBeam[0 : beamSize]$
        $remainBeam = currentBeam[beamSize :]$
      end if
    end for
  until both $currentBeam$ and $remainBeam$ are $\emptyset$
```

Algorithm 2 Stochastic Beam Search with Restart

```
  Input: States $S$, beamSize $m$, Roulette Selector $rs$
  repeat
    Initialize $currentBeam = \{\}$
    for $State\ s$ in $S$ do
      $successors = state.getSucessors()$
      for $Successors\ s'$ in $S'$ do
        if $needToVisit(s')$ then
          if $s'.goalReached()$ then
            $return\ s'$
          else
            $currentBeam.add(s')$
          end if
        end if
      end for
      if $currentBeam.isEmpty()$ then
        if NOT $remainBeam.isEmpty()$ then
          $S = remainBeam$
          continue      //Restart with remainBeam
        end if
      end if
      $result = rs.selectMulti(currentBeam, m)$
      $currentBeam = result[0]$
      $remainBeam = result[1]$
    end for
  until both $currentBeam$ and $remainBeam$ are $\emptyset$
```

while it took more than 2 seconds to find a solution by using Hill Climbing Search.

In case that the beam search can not handle, it will start running Local Search – starting with depth 5 and increment it to 100, Enforced Hill Climbing Search and finally Best First Search.

4 OPTIMISING PLANNING

4.1 Stochastic Beam Search algorithm

As described in Section 1, we modified the roulette selector so that it can select multiple states rather than only one state. The return value of the updated selector is a Linked List with $size = 2$ containing $selectedStates$ (for $currentBeam$) and $unselectedStates$ (for $remainBeam$). The algorithm is described in **Algorithm 2**. The main difference between **Algorithm 1** and **Algorithm 2** is the previous one select best m states and the later one select m states randomly (the lower HValue will have greater probability to be selected).

4.2 Parameter setting in JavaFF

In optimizing task, we start with Enforced Hill Climbing Search – see if it can find a ”relative” good enough plan. Then we move to our local beam search, set the $beamSize$ to 15 as we don’t care about the speed now. Then, we do our Local Search with $RandomThreeFilter$ and $RouletteSuccessorSelector$, set its depthBound from 50 to 100, run it for 3 times. The next is our most important part: **Stochastic Beam Search**, by viewing the successors randomly and see if it can reach the global optimise state. The reason why I put this algorithm to hear is it can be very time consuming (we found that it sometime find a plan with cost over 1000 for driverlog 2 but it will convergence to 21 – which is the lowest cost plan we found).

5 Evaluation

5.1 Satisfying Evaluation

	Searching Algorithm		
Problem	Beam Search	HC + Helpful + Best	HC + Helpful + Roulette
Driverlog 2	0.212s	2.262s	3.125s
Driverlog 4	0.375s	2.041s	3.663s
Driverlog 8	1.142s	12.92s	9.666s
Driverlog 9	1.10s	18.544s	18.528s
Rover 6	1.256s	2.851s	3.163s
Depot 4	127.836s	Time Out	Time Out

In the table above, we demonstrated that the Beam Search do have better performance among these algorithms – especially in problems like Driverlog 8 and some complex problem like depot 4.

5.2 Optimising Evaluation

	Searching Algorithm	
Problem	Best Plan Found	HC + HelpfulFilter + BestSuccessor
Driverlog 2	21	21
Driverlog 3	12	12
Driverlog 4	16	16
Driverlog 5	19	29
Rover 6	36	37
Rover 7	18	20

In the table above, we demonstrated that our approach are able to find a better solution compare to our work in Section 1, 2 and 3 – although the improvement is not that obvious.

6 BUG Found and other issues

6.1 BUG Found

When we trying to use the provided comparator to sort our $currentBeam$, we found that sometime we will get **Comparison method violates its general contract**. We currently found this error only occurred in **Rover 9** & **Rover 10**. This might because of if some states have the same HValue, it will be compared by the hashed value – which might cause the cause if a >b, b >c, a == c (this might because of collision in hashing algorithm, if a.HValue() == b.HValue() == c.HValue())

6.2 Issue in Next Double Generator

When we developed our Roulette Selector, we used ***javaff.Javaff.generator.nextDouble()*** to generate a float number (so that we can simulate a continuous random process). However, we found that if we print out the random number generated by nextDouble(), their sequence will be the same (for example, no matter when I generate this number, it always start with 0.5xxxxx, and then 0.4xxxx – this might be able to solved by applying a seed with current time *System.currentTimeMillis()*).

6.3 Memory Using in Modified Roulette Selector

As we are now selecting the state randomly from all unseen successors, it might require quite a lot of memory and sometime it will cause OutOfMemoryException if the size of states is huge.

7 Conclusion

In our research work, we found that by keeping more states during our searching can give a faster searching speed as sometime the greedy approach cannot find a suitable solution. We also found that it might be time-consuming if we apply the helpfulFilter – this is because in helpfulFilter, we will perform the relaxed graph searching first and decide if this successor is helpful or not. We also found that the more random factor we apply in our searching, the more likely we will reach the global optimise state if we ran it with infinite times.