CS 4649/7649: RIP - Robot Intelligence - Planning

Project 1: Classical Sokoban Planner

Arvind Krishnaa Jagannathan, Zheng Yong, Luis Gustavo Souza Silva, Zhengyi Hu

1 Administrative

- This is a comprehensive report, with some plans incorporated into it. The plans are also available separately in the "*src/resources*" directory.
- We have a *README* file, which lists the directory structure of the attached *zip* file. It also has the instructions for running the program.
- We also have outlined the participation of each of the members in the README file. Here is a brief summary of them.

  1. **Luis**: PDDL definitions for Sokoban Part I for both the domain and each problem scenario. Also ensured that the planners work correctly on Windows.
  2. **Zheng**: Implemented the planning algorithm for Sokoban Part II, focusing on Branch optimization.
  3. **Zhengyi**: Implemented the planning algorithm for Sokoban Part II, focusing on state space compression and caching states into a hash map.
  4. **Arvind**: PDDL definitions for 3, 6, 10 disk Towers of Hanoi. Representation of domain and some minor fix-up to Sokoban Part II algorithm.

  Each of us have constructed the corresponding parts of this report. We have all read and reviewed each other's contribution to the report as well as the code.

2 Pre-Project: Towers of Hanoi

Planners Used

The two classical planners which we are using for the Towers of Hanoi problem are the Blackbox planner [1] (downloaded from `http://www.cs.rochester.edu/~kautz/satplan/blackbox/blackbox-download.html`) and the FF planner [2] (downloaded from `http://fai.cs.uni-saarland.de/hoffmann/ff/FF-v2.3.tgz`). The definition of the Towers of Hanoi domain, as well as the representation of the initial state of the problem (from Figure 1) are in the corresponding PDDL files, namely *hanoi-domain.pddl* and *hanoi-3.pddl*.

Figure 1: Towers of Hanoi with 3 disks

2.1 Questions

1. Explain the method by which each of the two planners finds a solution

BlackBox- The Blackbox planning algorithm essentially represents the PDDL representation of a problem as a set of Boolean satisfiability problems (SAT), which are then solved using a variety of SAT solvers (such as satz, walksat and chaff) to produce a plan. In order to construct the SAT problem, Blackbox uses the GraphPlan algorithm to construct a plan graph for the given problem representation. This plan graph is then "translated" to the SAT problem representation. This is outlined briefly below:

The GraphPlan algorithm works by constructing a planning graph out of a Strips representation, which will propagate actions/operators across different "layers" along with their "mutex" pre-conditions. So at every stage there will be a list of actions, which have mutually exclusive pre-conditions. The plan generated by GraphPlan will be a sub-graph of the plan graph such that, all the conditions of the initial and final state are incorporated without there being any "conflicting" actions. As described by Kautz et. al [3], each state in this solution graph can be encoded into propositional logic. This can be done by adding propositions of the form,

$$Precondition_i \implies Action_j$$

At each layer of the solution sub-graph, clauses/fluents can be resolved away to result in a compact propositional logic representation of the plan graph. This list of proposition corresponds to the "translation" of the plan graph into a SAT problem. Then any available SAT solver can be applied on this problem which (although NP-complete in theory) with reasonable assumptions be completed in exponential time. ($O(n^3)$)

FF Planner - The fast-forward planning algorithm utilizes GraphPlan as an admissible (and informed) heuristic to solve the planning problem using a search algorithm across the set of permissible states in the state space. FF planner works as follows: it sets up a relaxed solvable sub-problem ($S\prime$) of the original problem (S).Then GraphPlan is applied on the relaxed sub-problem; the length of the solution plan sub-graph is then treated as a heuristic to guide the state space search algorithm for a plan to the original problem. This takes into account the positive interactions between various facts in the problem.

FF uses the enforced hill climbing algorithm to search for valid solutions. Using the length of the relaxed GraphPlan solution as a heuristic, the enforced hill climbing algorithm evaluates the direct successors of a search state S. Until a state ($S\prime$) is found with a better heuristic evaluation than S, the search goes one step further.

Thus in summary both ***FF*** and ***Blackbox*** algorithms use the GraphPlan algorithm, however *Blackbox* uses it in a more direct way, to set up the SAT problem. In *FF*, GraphPlan is used simply as a heuristic measure for the enforced hill climbing algorithm. Both the planners generated the same plan, which is in Figure 2.

```
step    0: MOVE-DISK D1 D2 P1
        1: MOVE-DISK D2 D3 P2
        2: MOVE-DISK D1 P1 D2
        3: MOVE-DISK D3 P3 P1
        4: MOVE-DISK D1 D2 P3
        5: MOVE-DISK D2 P2 D3
        6: MOVE-DISK D1 P3 D2


time spent:    0.00 seconds instantiating 48 easy, 0 hard action templates
               0.00 seconds reachability analysis, yielding 18 facts and 38 actions
               0.00 seconds creating final representation with 17 relevant facts
               0.00 seconds building connectivity graph
               0.00 seconds searching, evaluating 21 states, to a max depth of 1
               0.00 seconds total time
```

Figure 2: Plan for Towers of Hanoi with 3 disks (generated from FF planner)

2. Which planner was fastest?

Both the algorithms were run on the same Linux box, and using the *time* command the time taken for them to produce plans were measure. Clearly the **FF** algorithm was the fastest among the two, with the following time measures

- FF - 0.004s
- Blackbox - 0.011s

This shows that **FF** is 2.75 times faster than **Blackbox**. This is pretty much expected from theory since the time complexity for each of the algorithms are,

$$\text{O}(\textit{Blackbox}) = \text{O}(\text{GraphPlan'}) + \text{O}(\textit{SAT Solver})$$
$$\text{O}(\textit{FF}) = \text{O}(\text{GraphPlan}) + \text{O}(\textit{Enforced Hill Climbing})$$
$$\text{O}(\textit{Enforced Hill Climbing}) = \text{O}(\text{iterations x successors}),$$
$$\text{O}(\textit{SAT Solver}) = \text{O}(2^n)$$

Clearly the SAT solver's exponential time complexity, as well as the time complexity of the complete GraphPlan algorithm for *Blackbox* vs. that of the relaxed GraphPlan as well as that of *Enforced Hill Climbing* for the *FF* make it obviously slower than FF; this has been shown by using the towers of hanoi state and domain PDDL description multiple times (to get an average measure) with Blackbox and FF.

3. Explain why the winning planner might be more effective on this problem

Its pretty obvious that the length of a plan in case of the Towers of Hanoi problem are a similar order of magnitude as that of the total number of states ($2^n - 1$ steps and 3^n states). In case of the **FF** planner, GraphPlan is used only as a heuristic and that too for a relaxed subset. *FF*'s major component is the local-search algorithm, which means it does not have to necessarily traverse all the states in order to obtain a solution.

Blackbox on the other needs to explicitly construct the mutex graph for every level until a solution is obtained. In case the state space is large (like 10 disks in the Towers of Hanoi problem), the construction of a complete plan graph is memory prohibitive and will not be effective on larger instances of this problem (it actually does not give plans for even 6 disks on my machine after 1 minute of execution). Another major issue with *Blackbox* is that several levels of the GraphPlan algorithm may lead to the same set of SAT problems, which will remain unsatisfiable. *Blackbox* is bound to be less effective in similar large state problems, since it seems to perform the same computation multiple times.

3 Project Part I: Sokoban PDDL

Note: We have made the assumption that the *planner can move any box to any target square.* This assumption is carried over to Part II as well.

The Sokoban's domain definition was created with the guideline idea of keeping it simple. Then, we followed the steps:

1. Identify the constants
2. Identify the fluents
3. Identify the actions

The first step is to identify the constants we evaluate possible elements that could be part of the domain. From this, constants like *robot*, *box* and *location* appeared. However the problem only have one robot and is more natural to think about each box by its location instead of any kind of specific identification. Because of this, we moved on with only the *location* constant.

The second step is to identify fluents related to this problem. We had a lot of possible fluents, most of them were related to the idea of using the constants already eliminated. However, to keep it simple, the fluents used were:

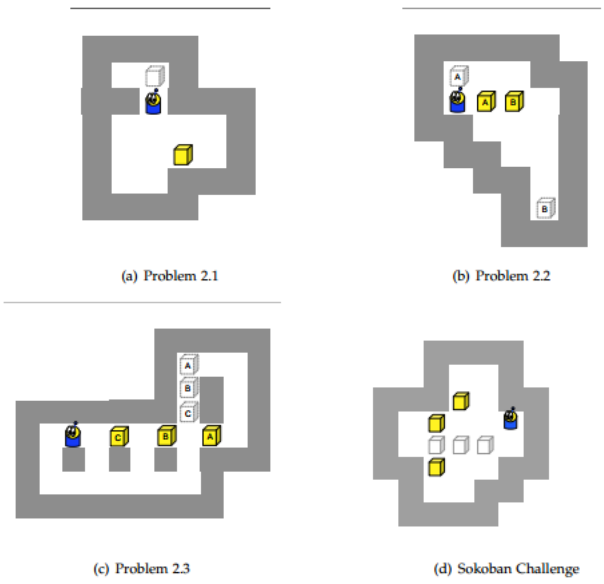


Figure 3: Sokoban Problems

- **in(?loc)** - true if the robot is at location loc;
- **boxAt(?loc)** - true if there is a box at location loc;
- **isLeft(?loc ?loc)** - true if loc(1) is at left of loc(2);
- **isRight(?loc ?loc)** - true if loc(1) is at right of loc(2);
- **isUp(?loc ?loc)** - true if loc(1) is at up of loc(2);
- **isDown(?loc ?loc)** - true if loc(1) is at down of loc(2).

Finally, the last step is to identify the actions. For the Sokoban domain, the robot can move or push a box in the up, down, right and left directions, provided that there is no box on the way. After defining each action in respect to its preconditions, add list and delete list we could define the following:

- **MOVE-<UP,DOWN,RIGHT or LEFT>(?loc-to ?loc-from)**;
- **PUSH-<UP,DOWN,RIGHT or LEFT>(?box-loc ?loc ?loc-to)**.

Done that, the next part of the question was to define each problem using this domain. For that, each grid position, from top to botton, and left to right, received a nickname of a alphabet letter. Then, all the constraints of linking these grid positions were defined using the fluent *isUp*, *isDown*, *isRight*, *isLeft* and the initial and goal state using the fluents *in* and *boxAt*.

3.1 Questions

1. Show successful plans from at least one planner on the three Sokoban problems in Figure 3 (1-3). The challenge problem is optional

Using the PDDL definitions for the domain, the FF planner was able to find solutions for all the problems and to the challenge. The result plan of each problem is shown on the Figures 4, 5, 6 and 7.

In order to make the solutions more readable and easier to verify, lets consider that when a robot moves for up, down, right or left direction it will be represented by a lowercase **u**, **d**, **r** or **l**, respectively. For the

```
ff: found legal plan as follows

step    0: MOVE-DOWN LOCATIONE LOCATIONC
        1: MOVE-RIGHT LOCATIONF LOCATIONE
        2: MOVE-RIGHT LOCATIONG LOCATIONF
        3: MOVE-DOWN LOCATIONK LOCATIONG
        4: PUSH-LEFT LOCATIONJ LOCATIONK LOCATIONI
        5: MOVE-UP LOCATIONF LOCATIONJ
        6: MOVE-LEFT LOCATIONE LOCATIONF
        7: MOVE-LEFT LOCATIOND LOCATIONE
        8: MOVE-DOWN LOCATIONH LOCATIOND
        9: MOVE-DOWN LOCATIONL LOCATIONH
       10: MOVE-RIGHT LOCATIONM LOCATIONL
       11: PUSH-UP LOCATIONI LOCATIONM LOCATIONE
       12: PUSH-UP LOCATIONE LOCATIONI LOCATIONC
       13: PUSH-UP LOCATIONC LOCATIONE LOCATIONB


time spent:    0.00 seconds instantiating 48 easy, 0 hard action templates
               0.00 seconds reachability analysis, yielding 38 facts and 48 actions
               0.00 seconds creating final representation with 37 relevant facts
               0.00 seconds building connectivity graph
               0.00 seconds searching, evaluating 47 states, to a max depth of 4
               0.00 seconds total time
```

Figure 4: Sokoban Problem 2.1

```
ff: found legal plan as follows

step    0: MOVE-UP LOCATIONA LOCATIOND
        1: MOVE-RIGHT LOCATIONB LOCATIONA
        2: MOVE-RIGHT LOCATIONC LOCATIONB
        3: PUSH-DOWN LOCATIONF LOCATIONC LOCATIONL
        4: MOVE-UP LOCATIONC LOCATIONF
        5: MOVE-LEFT LOCATIONB LOCATIONC
        6: MOVE-LEFT LOCATIONA LOCATIONB
        7: MOVE-DOWN LOCATIOND LOCATIONA
        8: PUSH-RIGHT LOCATIONE LOCATIOND LOCATIONF
        9: MOVE-DOWN LOCATIONK LOCATIONE
       10: PUSH-RIGHT LOCATIONL LOCATIONK LOCATIONM
       11: MOVE-LEFT LOCATIONK LOCATIONL
       12: MOVE-UP LOCATIONE LOCATIONK
       13: MOVE-UP LOCATIONB LOCATIONE
       14: MOVE-RIGHT LOCATIONC LOCATIONB
       15: PUSH-DOWN LOCATIONF LOCATIONC LOCATIONL
       16: MOVE-RIGHT LOCATIONG LOCATIONF
       17: PUSH-DOWN LOCATIONM LOCATIONG LOCATIONO
       18: PUSH-DOWN LOCATIONO LOCATIONM LOCATIONP
       19: PUSH-DOWN LOCATIONP LOCATIONO LOCATIONQ
       20: MOVE-UP LOCATIONO LOCATIONP
       21: MOVE-LEFT LOCATIONN LOCATIONO
       22: PUSH-UP LOCATIONL LOCATIONN LOCATIONF
       23: MOVE-RIGHT LOCATIONM LOCATIONL
       24: MOVE-UP LOCATIONG LOCATIONM
       25: PUSH-LEFT LOCATIONF LOCATIONG LOCATIONE
       26: MOVE-DOWN LOCATIONL LOCATIONF
       27: MOVE-LEFT LOCATIONK LOCATIONL
       28: PUSH-UP LOCATIONE LOCATIONK LOCATIONB
       29: MOVE-RIGHT LOCATIONF LOCATIONE
       30: MOVE-UP LOCATIONC LOCATIONF
       31: PUSH-LEFT LOCATIONB LOCATIONC LOCATIONA


time spent:    0.00 seconds instantiating 56 easy, 0 hard action templates
               0.00 seconds reachability analysis, yielding 45 facts and 56 actions
               0.00 seconds creating final representation with 42 relevant facts
               0.00 seconds building connectivity graph
               0.00 seconds searching, evaluating 135 states, to a max depth of 22
               0.00 seconds total time
```

Figure 5: Sokoban Problem 2.2

push actions of moving the box up, down, right or left, the capital letters **U**, **D**, **R** and **L** will be used. Thus, the solutions found can be represented as the strings shown below.

```
ff: found legal plan as follows

step    0: MOVE-RIGHT LOCATIONJ LOCATIONI
        1: MOVE-DOWN LOCATIONR LOCATIONJ
        2: MOVE-DOWN LOCATIONW LOCATIONR
        3: MOVE-RIGHT LOCATIONX LOCATIONW
        4: MOVE-RIGHT LOCATIONY LOCATIONX
        5: MOVE-UP LOCATIONS LOCATIONY
        6: MOVE-UP LOCATIONL LOCATIONS
        7: PUSH-LEFT LOCATIONK LOCATIONL LOCATIONJ
        8: PUSH-LEFT LOCATIONJ LOCATIONK LOCATIONI
        9: MOVE-RIGHT LOCATIONK LOCATIONJ
       10: MOVE-RIGHT LOCATIONL LOCATIONK
       11: MOVE-DOWN LOCATIONS LOCATIONL
       12: MOVE-DOWN LOCATIONY LOCATIONS
       13: MOVE-RIGHT LOCATIONZ LOCATIONY
       14: MOVE-RIGHT LOCATIONA1 LOCATIONZ
       15: MOVE-UP LOCATIONT LOCATIONA1
       16: MOVE-UP LOCATIONN LOCATIONT
       17: PUSH-LEFT LOCATIONM LOCATIONN LOCATIONL
       18: PUSH-LEFT LOCATIONL LOCATIONM LOCATIONK
       19: MOVE-RIGHT LOCATIONM LOCATIONL
       20: MOVE-RIGHT LOCATIONN LOCATIONM
       21: MOVE-UP LOCATIONF LOCATIONN
       22: MOVE-UP LOCATIOND LOCATIONF
       23: MOVE-UP LOCATIONA LOCATIOND
       24: MOVE-RIGHT LOCATIONB LOCATIONA
       25: MOVE-RIGHT LOCATIONC LOCATIONB
       26: MOVE-DOWN LOCATIONE LOCATIONC
       27: MOVE-DOWN LOCATIONG LOCATIONE
       28: MOVE-DOWN LOCATIONP LOCATIONG
       29: PUSH-LEFT LOCATIONO LOCATIONP LOCATIONN
       30: PUSH-LEFT LOCATIONN LOCATIONO LOCATIONM
       31: MOVE-DOWN LOCATIONT LOCATIONN
       32: MOVE-DOWN LOCATIONA1 LOCATIONT
       33: MOVE-LEFT LOCATIONZ LOCATIONA1
       34: MOVE-LEFT LOCATIONY LOCATIONZ
       35: MOVE-UP LOCATIONS LOCATIONY
       36: MOVE-UP LOCATIONL LOCATIONS
       37: PUSH-RIGHT LOCATIONM LOCATIONL LOCATIONN
       38: MOVE-LEFT LOCATIONL LOCATIONM
       39: MOVE-DOWN LOCATIONS LOCATIONL
       40: MOVE-DOWN LOCATIONY LOCATIONS
       41: MOVE-RIGHT LOCATIONZ LOCATIONY
       42: MOVE-RIGHT LOCATIONA1 LOCATIONZ
       43: MOVE-UP LOCATIONT LOCATIONA1
       44: PUSH-UP LOCATIONN LOCATIONT LOCATIONF
       45: MOVE-LEFT LOCATIONM LOCATIONN
       46: MOVE-LEFT LOCATIONL LOCATIONM
       47: MOVE-DOWN LOCATIONS LOCATIONL
       48: MOVE-DOWN LOCATIONY LOCATIONS
       49: MOVE-LEFT LOCATIONX LOCATIONY
       50: MOVE-LEFT LOCATIONW LOCATIONX
       51: MOVE-UP LOCATIONR LOCATIONW
       52: MOVE-UP LOCATIONJ LOCATIONR
       53: PUSH-RIGHT LOCATIONK LOCATIONJ LOCATIONL
       54: PUSH-RIGHT LOCATIONL LOCATIONK LOCATIONM
       55: MOVE-DOWN LOCATIONS LOCATIONL
       56: MOVE-DOWN LOCATIONY LOCATIONS
       57: MOVE-RIGHT LOCATIONZ LOCATIONY
       58: MOVE-RIGHT LOCATIONA1 LOCATIONZ
       59: MOVE-UP LOCATIONT LOCATIONA1
       60: MOVE-UP LOCATIONN LOCATIONT
       61: PUSH-UP LOCATIONF LOCATIONN LOCATIOND
       62: PUSH-UP LOCATIOND LOCATIONF LOCATIONA
       63: MOVE-DOWN LOCATIONF LOCATIOND
       64: MOVE-DOWN LOCATIONN LOCATIONF
       65: MOVE-DOWN LOCATIONT LOCATIONN
       66: MOVE-DOWN LOCATIONA1 LOCATIONT
       67: MOVE-LEFT LOCATIONZ LOCATIONA1
       68: MOVE-LEFT LOCATIONY LOCATIONZ
       69: MOVE-UP LOCATIONS LOCATIONY
       70: MOVE-UP LOCATIONL LOCATIONS
       71: PUSH-RIGHT LOCATIONM LOCATIONL LOCATIONN
       72: MOVE-LEFT LOCATIONL LOCATIONM
       73: MOVE-DOWN LOCATIONS LOCATIONL
       74: MOVE-DOWN LOCATIONY LOCATIONS
       75: MOVE-RIGHT LOCATIONZ LOCATIONY
       76: MOVE-RIGHT LOCATIONA1 LOCATIONZ
       77: MOVE-UP LOCATIONT LOCATIONA1
       78: PUSH-UP LOCATIONN LOCATIONT LOCATIONF
       79: MOVE-LEFT LOCATIONM LOCATIONN
       80: MOVE-LEFT LOCATIONL LOCATIONM
       81: MOVE-LEFT LOCATIONK LOCATIONL
       82: MOVE-LEFT LOCATIONJ LOCATIONK
       83: MOVE-DOWN LOCATIONR LOCATIONJ
       84: MOVE-DOWN LOCATIONW LOCATIONR
       85: MOVE-LEFT LOCATIONV LOCATIONW
       86: MOVE-LEFT LOCATIONU LOCATIONV
       87: MOVE-UP LOCATIONQ LOCATIONU
       88: MOVE-UP LOCATIONH LOCATIONQ
       89: PUSH-RIGHT LOCATIONI LOCATIONH LOCATIONJ
       90: PUSH-RIGHT LOCATIONJ LOCATIONI LOCATIONK
       91: PUSH-RIGHT LOCATIONK LOCATIONJ LOCATIONL
       92: PUSH-RIGHT LOCATIONL LOCATIONK LOCATIONM
       93: MOVE-DOWN LOCATIONS LOCATIONL
       94: MOVE-DOWN LOCATIONY LOCATIONS
       95: MOVE-RIGHT LOCATIONZ LOCATIONY
       96: MOVE-RIGHT LOCATIONA1 LOCATIONZ
       97: MOVE-UP LOCATIONT LOCATIONA1
       98: MOVE-UP LOCATIONN LOCATIONT
       99: PUSH-UP LOCATIONF LOCATIONN LOCATIOND
      100: MOVE-DOWN LOCATIONN LOCATIONF
      101: MOVE-DOWN LOCATIONT LOCATIONN
      102: MOVE-DOWN LOCATIONA1 LOCATIONT
      103: MOVE-LEFT LOCATIONZ LOCATIONA1
      104: MOVE-LEFT LOCATIONY LOCATIONZ
      105: MOVE-UP LOCATIONS LOCATIONY
      106: MOVE-UP LOCATIONL LOCATIONS
      107: PUSH-RIGHT LOCATIONM LOCATIONL LOCATIONN
      108: MOVE-LEFT LOCATIONL LOCATIONM
      109: MOVE-DOWN LOCATIONS LOCATIONL
      110: MOVE-DOWN LOCATIONY LOCATIONS
      111: MOVE-RIGHT LOCATIONZ LOCATIONY
      112: MOVE-RIGHT LOCATIONA1 LOCATIONZ
      113: MOVE-UP LOCATIONT LOCATIONA1
      114: PUSH-UP LOCATIONN LOCATIONT LOCATIONF

time spent:    0.00 seconds instantiating 104 easy, 0 hard action templates
               0.00 seconds reachability analysis, yielding 68 facts and 82 actions
               0.00 seconds creating final representation with 55 relevant facts
               0.00 seconds building connectivity graph
               0.00 seconds searching, evaluating 2104 states, to a max depth of 9
               0.00 seconds total time
```

Figure 6: Sokoban Problem 2.3

Problem 2.1 drrdLullddrUUU

Problem 2.2 urrDulldRdRluurDrDDDulUruLdlUru

Problem 2.3 rddrruuLLrrddrruuLLrruuurrdddLLddlluuRlddrruUllddlluu
RRddrruuUUddddlluuRlddrruUllllddlluuRRRRddrruuuddd
lluuRlddrruU

Challenge lldlluRRdddlUruulldRdrruruLLrddlluRuruulDDDlluRdrUrrd
LdLdlUrruullDRddlUruuruulDD

```
ff: found legal plan as follows

step    0: MOVE-LEFT LOCATIONH LOCATIONI
        1: MOVE-LEFT LOCATIONG LOCATIONH
        2: MOVE-DOWN LOCATIONL LOCATIONG
        3: MOVE-LEFT LOCATIONK LOCATIONL
        4: MOVE-LEFT LOCATIONJ LOCATIONK
        5: MOVE-UP LOCATIONE LOCATIONJ
        6: PUSH-RIGHT LOCATIONF LOCATIONE LOCATIONG
        7: PUSH-RIGHT LOCATIONG LOCATIONF LOCATIONH
        8: MOVE-DOWN LOCATIONL LOCATIONG
        9: MOVE-DOWN LOCATIONP LOCATIONL
       10: MOVE-DOWN LOCATIONS LOCATIONP
       11: MOVE-LEFT LOCATIONR LOCATIONS
       12: PUSH-UP LOCATIONO LOCATIONR LOCATIONK
       13: MOVE-RIGHT LOCATIONP LOCATIONO
       14: MOVE-UP LOCATIONL LOCATIONP
       15: MOVE-UP LOCATIONG LOCATIONL
       16: MOVE-LEFT LOCATIONF LOCATIONG
       17: MOVE-LEFT LOCATIONE LOCATIONF
       18: MOVE-DOWN LOCATIONJ LOCATIONE
       19: PUSH-RIGHT LOCATIONK LOCATIONJ LOCATIONL
       20: MOVE-DOWN LOCATIONO LOCATIONK
       21: MOVE-RIGHT LOCATIONP LOCATIONO
       22: MOVE-RIGHT LOCATIONQ LOCATIONP
       23: MOVE-UP LOCATIONM LOCATIONQ
       24: MOVE-RIGHT LOCATIONN LOCATIONM
       25: MOVE-UP LOCATIONI LOCATIONN
       26: PUSH-LEFT LOCATIONH LOCATIONI LOCATIONG
       27: PUSH-LEFT LOCATIONG LOCATIONH LOCATIONF
       28: MOVE-RIGHT LOCATIONH LOCATIONG
       29: MOVE-DOWN LOCATIONM LOCATIONH
       30: MOVE-DOWN LOCATIONQ LOCATIONM
       31: MOVE-LEFT LOCATIONP LOCATIONQ
       32: MOVE-LEFT LOCATIONO LOCATIONP
       33: MOVE-UP LOCATIONK LOCATIONO
       34: PUSH-RIGHT LOCATIONL LOCATIONK LOCATIONM
       35: MOVE-UP LOCATIONG LOCATIONL
       36: MOVE-RIGHT LOCATIONH LOCATIONG
       37: MOVE-UP LOCATIOND LOCATIONH
       38: MOVE-UP LOCATIONB LOCATIOND
       39: MOVE-LEFT LOCATIONA LOCATIONB
       40: PUSH-DOWN LOCATIONC LOCATIONA LOCATIONG
       41: PUSH-DOWN LOCATIONG LOCATIONC LOCATIONL
       42: PUSH-DOWN LOCATIONL LOCATIONG LOCATIONP
       43: MOVE-LEFT LOCATIONK LOCATIONL
       44: MOVE-LEFT LOCATIONJ LOCATIONK
       45: MOVE-UP LOCATIONE LOCATIONJ
       46: PUSH-RIGHT LOCATIONF LOCATIONE LOCATIONG
       47: MOVE-DOWN LOCATIONK LOCATIONF
       48: MOVE-RIGHT LOCATIONL LOCATIONK
       49: PUSH-UP LOCATIONG LOCATIONL LOCATIONC
       50: MOVE-RIGHT LOCATIONH LOCATIONG
       51: MOVE-RIGHT LOCATIONI LOCATIONH
       52: MOVE-DOWN LOCATIONN LOCATIONI
       53: PUSH-LEFT LOCATIONM LOCATIONN LOCATIONL
       54: MOVE-DOWN LOCATIONQ LOCATIONM
       55: PUSH-LEFT LOCATIONP LOCATIONQ LOCATIONO
       56: MOVE-DOWN LOCATIONS LOCATIONP
       57: MOVE-LEFT LOCATIONR LOCATIONS
       58: PUSH-UP LOCATIONO LOCATIONR LOCATIONK
       59: MOVE-RIGHT LOCATIONP LOCATIONO
       60: MOVE-RIGHT LOCATIONQ LOCATIONP
       61: MOVE-UP LOCATIONM LOCATIONQ
       62: MOVE-UP LOCATIONH LOCATIONM
       63: MOVE-LEFT LOCATIONG LOCATIONH
       64: MOVE-LEFT LOCATIONF LOCATIONG
       65: PUSH-DOWN LOCATIONK LOCATIONF LOCATIONO
       66: PUSH-RIGHT LOCATIONL LOCATIONK LOCATIONM
       67: MOVE-DOWN LOCATIONP LOCATIONL
       68: MOVE-DOWN LOCATIONS LOCATIONP
       69: MOVE-LEFT LOCATIONR LOCATIONS
       70: PUSH-UP LOCATIONO LOCATIONR LOCATIONK
       71: MOVE-RIGHT LOCATIONP LOCATIONO
       72: MOVE-UP LOCATIONL LOCATIONP
       73: MOVE-UP LOCATIONG LOCATIONL
       74: MOVE-RIGHT LOCATIONH LOCATIONG
       75: MOVE-UP LOCATIOND LOCATIONH
       76: MOVE-UP LOCATIONB LOCATIOND
       77: MOVE-LEFT LOCATIONA LOCATIONB
       78: PUSH-DOWN LOCATIONC LOCATIONA LOCATIONG
       79: PUSH-DOWN LOCATIONG LOCATIONC LOCATIONL

                    time spent:    0.00 seconds instantiating 86 easy, 0 hard action templates
                                   0.00 seconds reachability analysis, yielding 58 facts and 86 actions
                                   0.00 seconds creating final representation with 57 relevant facts
                                   0.00 seconds building connectivity graph
                                   0.04 seconds searching, evaluating 10010 states, to a max depth of 8
                                   0.04 seconds total time
```

Figure 7: Sokoban Challenge

2. Compare the performance of two planners on this domain. Which one works better? Does this make sense, why?

As an overview, the *FF*'s planner performed better than *Blackbox*. The *FF*'s planner was able to find results for all problems, including challenge, in few seconds. On the other hand, the *Blackbox* planner only managed to find the results from the first two problems and stopped on the others after using all the hardware resources. These results were expected since, as stated in Section 2.1, the *FF*'s planner have a lower asymptotic complexity and use an heuristic function to choose the best option of what expand next. Thus, is more likely to have better time and space complexities than the *Blackbox*.

3. Clearly PDDL was not intended for this sort of application. Discuss the challenges in expressing geometric constraints in semantic planning

While solving this problem is possible to notice that are two approaches. The first one requires a previous knowledge of a solution, and the developer states that a specific box goes to a specific location. The second approach, there is no kind of identification for the boxes, and this is the one we used as a solution. Independently of the approach, the domain contains a grid on which the robot should move. Thus, it is necessary to come up with a way to express when the robot can, or cannot, move between two locations if there is a wall or a box in between. Also, is necessary to express when the robot is allowed to do something, like moving between two empty adjacent squares. Thus, all these logic expressions were used to represent geometric constraints like collisions between the robot and the walls or boxes.

Also, we had to clearly discretize the continuous space (into a set of boxes on a grid), which may then introduce resolution errors (not in this problem, but discretization may result in "squares" which are obstacles although in the continous domain there would have been space for navigation).

4. In many cases, geometric and dynamic planning are insufficient to describe a domain. Give an example of a problem that is best suited for semantic (classical) planning. Explain why a semantic representation would be desirable

The classical planners have a variety of best suited problems, like in logistics, manofacturing, and management. One example is the DART system used for military logistics. Logistics are better expressed by a set of logical expressions to specify complex operations not related to the kinect motion.

4 Project Part II: Sokoban Planner

In this part of the project, we came up with our own representation of the Sokoban domain. The domain has the following specifications:

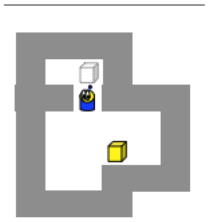
- **Geometry/Grid**: A world represented as a M x N grid. The Sokoban domain's geometry is defined by M rows and N columns, where each "cell" in the map has the same dimensions as the box.
- **Walls**: A wall is any of the grey boundary of the world, as well as a logical extension of the wall to fit the map. For instance, looking at the first problem in Figure 3, the top most row is entirely made up of walls (although there is some white space in the top right, that cannot be logically reached by the man and hence is a "wall". In our representation a wall is defined by the constant W.
- **Boxes**: In the grid for each problem, we define a constant B which corresponds to whether the box is present in that particular cell of the grid. We have assumed again that any box can be moved to any target square.
- **Target**: This is the target square where any box can be moved. Represented as D.
- **Man**: This represents the starting square in which the man is present. Represented in the grid as M.
- **Empty Squares**: These correspond to squares which are within the boundaries of the "walls" and can be occupied by a man or a box. Represented by $*$.

Representation of the Domain

Now that we have defined the components of the representation, we have defined the Sokoban domain for each of the 4 problems. Each problem has a "*input*" map file which has the following header:

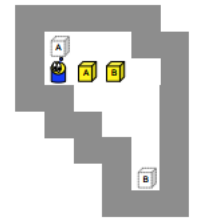
```
{Rows} {Columns} {Number of boxes}
```

Below the header is the grid based representation of the domain. The grid is a *Row* x *Column* matrix, where each element is a single character as defined previously. The 4 Sokoban problems (including the challenge problem) and their corresponding state representation as per our domain definition are presented in Figures 8, 9, 10, 11 respectively.



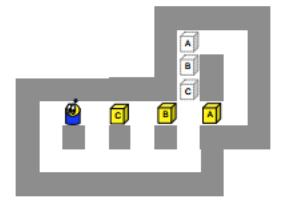
```
7 6 1
WWWWWW
W*D*WW
WWMWWW
W****W
W**B*W
W**WWW
WWWWWW
```

Figure 8: Sokoban Problem 2.1 and its corresponding representation



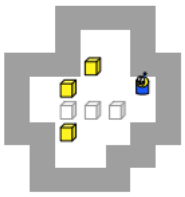
```
8 6 2
WWWWWW
WD**WW
WMBB*W
WW***W
WWW**W
WWWW*W
WWWWDW
WWWWWW
```

Figure 9: Sokoban Problem 2.2 and its corresponding representation



```
8 11 3
WWWWWWWWWWW
WWWWWWWD**W
WWWWWWWDW*W
WWWWWWWDW*W
W*M*B*B*B*W
W*W*W*W*WWW
W*******WWW
WWWWWWWWWWW
```

Figure 10: Sokoban Problem 2.3 and its corresponding representation



```
8 7 3
WWWWWW
WWW**W
WWB*WW
W*B**W
W*DDD*W
WWB**W
WW**WW
WWWWWW
```

Figure 11: Sokoban Challenge and its corresponding representation

Brief Outline of the Algorithm

The algorithm is a simple implementation of the breadth-first search algorithm. The state is represented by the position of all the boxes and the position of the man (as per the representation). These arguments are integrated into a single *long* integer.

At the beginning of search, there is only one element in the search queue which is the initial state. Every time we look at a state, we first check if it is the final state (target configuration, where boxes (B) are in the places where target squares (D) are placed initially). If it is the final state, the program terminates with the solution. If not, we see if we can make any of the boxes move. If one of the boxes can be moved one step

away in any direction, we take that move and add the new state into the queue.

Also, every time we add a new state into the queue we check if the state is already explored (already in the queue). This is done by using a hash map. After adding all possible states that can be derived from the state we are dealing with right now, we move on to the next state in the queue.

The program also terminates if the queue is empty, which indicates that there is no solution to the puzzle (and we report that).

Completeness, Speed and Efficiency

We believe that our planner is complete as well as fast. We discuss more about this in the questions sections (Questions 3 and 4). The state representation is pretty compact and really small in size. The processing on the input is also pretty fast since we cache the entire world state into a hash map. The maximum space complexity for the problem is $O(MXN)$ with M rows and N columns. However, this representation is pretty specific to the Sokoban domain. The only "strong" action in this domain is the PUSH action which we have incorporated into the state-space search. Our domain has nicely captured fluents such as LOCATION, BOX-AT and so on within the grid representation.

Miscellaneous Pieces of Information

We were able to capture most of the necessary pieces of information. We used a breadth-first search to explore the states, but we have not logged the explored states. So we are unable to report the number of states explored.

1. **Language Used**: C++
2. **Number of steps in plan**: We calculate the total number of steps it takes for every box to be moved to the target boxes. If there are N steps, then our plan would correspond to N world configurations, where the grid is represented as per the definitions stated above. There are currently some bugs in the printing logic, where in some steps, the man might make two or more movements as part of one step. After counting these mis-steps, our algorithm seems to take the same number of steps as FF. However, for transparency, we have the actual steps as reported by our algorithm below.
   - Problem 1: **14** (FF was 14)
   - Problem 2: **29** (FF was 32)
   - Problem 3: **91** (FF was 115)
   - Challenge Problem: **48** (FF was 79)
3. **Computation Time**: We have compared our planner against FF (which was only able to produce the solution for all the 4 problems). Both planners were executed on the same system and the computation time is shown in Table 1.

Table 1: Comparison of Computation Times

Problem	FF Time (in s)	Our Planner Time (in s)
1	0.005	**0.004**
2	0.007	**0.007**
3	0.021	**0.019**
Sokoban Challenge	0.043	**0.025**

Note: Although it appears that our algorithm is "faster" than FF, that might not necessarily be the case, since our algorithm prints to a file, whereas FF prints to console. Due to the length of the plans, which need to be printed to the terminal, FF takes more time. FF has a self-reporting "time spent" parameter, and by using that as a comparison, we get the comparison as shown in Table 2.

Table 2: Comparison of Computation Times (FF self-reported)

Problem	FF Time (in s)	Our Planner Time (in s)
1	0.00	**0.004**
2	0.00	**0.007**
3	0.00	**0.019**
Sokoban Challenge	0.04	**0.025**

4. **Machine Vitals**: The input representation for each of the problems was very small (of the order of 50-60 bytes), and the output was also suitably compact. Our program can compile and run on any C++ compiler conforming to ANSI standard, and does not require too much of memory to run. The only memory constraints are the fact that we are loading the entire world onto a hash map, but again this is not very significant for the Sokoban domain as well as the BFS which explores every possible reachable state.

5. **States explored**: While performing the BFS, there were a number of states which were explored during the course of finding a valid plan. For each of the problems, the number of states explored are shown below:

   - Problem 1: **14** (FF was 47)
   - Problem 2: **77** (FF was 135)
   - Problem 3: **344** (FF was 2104)
   - Challenge Problem: **954** (FF was 10010)

4.1 Questions

1. Give successful plans from your planner on the Sokoban problems in Figure 3 and any others

We have a successful plan for the first Sokoban problem depicted in Figure 12. The format for the solutions of all the Sokoban problems are the same as in the Figure. But for lack of space, we do not include all the plans here. However, the plans for all the problems (1, 2, 3 and the challenge problem) can be found in the "*resources/Our_Planner_Solutions*" directory.

Note that each step shows the world configuration with the position of the boxes, the man and "empty space" marked appropriately. We have an "accomplished" state, in which the boxes B need to be present where the the target squares D were present in the problem representation.

2. Compare the performance of your planner to the PDDL planners you used in the previous problem. Which was faster? Why?

As mentioned earlier, our planner was able to be almost as fast as the FF planner. Of course the time measure from Linux's *time* utility may not be a fair comparison, since FF prints a ton of output to the console, which will take more time than printing to a file, which is what we are doing in our algorithm. Taking FF's internal time metric into account, most Sokoban problem gave a solution as 0.00s (Note that their level of precision is two digits, whereas *time* has three digits of precision), except the last challenge problem, which FF reported to have taken 0.04s in total. However, we suspect FF might be not be as fast as our "specialized" Sokoban planner, since we have defined walls in the domain for this problem (as opposed to the case of FF where we did not). This definition enables our BFS to explore a lot less states than those explored by FF.

Of course, we did not try to define a fluent called Wall along with its effects and pre-conditions in the PDDL definition for FF. Had we done that, FF might have possibly been faster, but that would have bloated the domain definition more that what it is right now!

Figure 12: Sokoban Problem 2.1's Plan

3. Prove that your planner was complete. Your instructor has a math background: a proof "is a convincing argument." Make sure you address each aspect of completeness and why your planner satisfies it. Pictures are always welcome.

Basically our approach is Breadth First Search (BFS) with a little bit branch optimization, and it's easy to prove that it is complete. The state is represented by several arguments: the position of every box and the position of the person. And all these arguments are combined in one integer. We use shift operations in C++ to set and get a specific argument (For example, the position of the person), which saves a lot of space. Our algorithm searches for every possible move of the box, that is, as long as we can make a box move from the current state, we add this new state with the box moved to the new position. (And of course we assert that this new state is a state that we have not discovered yet. This is one of the branch optimizations) We keep on doing this until we hit the final state or we run out of states. If we hit the final state (with all boxes in designated positions), our algorithm ends and it prints out the plan we are looking for. If the set of states is empty and we still don't find a solution, the algorithm also terminates with no solution. Since BFS searches for every possible state that is reachable from the original state, if there exists a solution, we can always find it. If there is no solution, since our algorithm only search for possible states, it will terminate with no solution.

Now there are two conditions for BFS to be complete [4] and our definition of Sokoban domain satisfies both those conditions, namely

1. Finite branching factor: The Sokoban world is clearly finite, and there can only be a finite number of possible states which we can explore with respect to any given state. Since the domain has a finite branching factor, the first condition is satisfied.

2. Finite number of states: Clearly the Sokoban world has finite states. So there is no way for BFS to not converge to an answer if one exists (and will definitely report if one does not exist).

Also our algorithm is optimal in terms of box moves (i.e., every step of moving the box to a square has the same cost). This is because every time we add a new state, it is a state that is one box-move away from the previous state. Since we are using Breadth First Search, we can guarantee that our solution uses the least box moves.

4. What methods did you use to speed up the planning? Give a short description of each method and explain why it did or didn't help on each relevant problem

First of all we discard the new state that has already been explored. This branch optimization cuts out a lot of useless branches, and it asserts that our algorithm terminates when there is no solution.
Also when checking if the state has been explored, we use hash map. Specifically, we use a large integer to represent the state, and throw the explored states into the hash. So this saves the time needed for checking duplicate states.
Another branch optimization is that when we are in a state where one of the boxes has two adjacent sides facing the wall and this box is not in the designated position. This box cannot be moved anyway because we need the man on one side of the box and we need the opposite side of the wall to be clear so that we can move. (In this case at least one side of these two is wall) Therefore this state is itself a dead state. We will never make it to the final state in this case.

5 Post-Project: Towers of Hanoi Revisited

Constructing a PDDL representation for the N-disk towers of hanoi is pretty simple by utilizing the following simple structures, for initializing the problem state

1. Each disk labeled D_i is smaller than a disk labeled D_{i+1}. That is `(smaller` d_i d_{i+1}`)`.
2. Every disk is smaller than each of the three poles (by definition).
3. The smallest disk (i.e., D1), and the two other poles (P1 and P2) are clear.
4. Every disk D_i is *on* the disk D_{i+1}. The largest disk D10, is *on* the pole P3.
5. Every element D_i is a disk

The goal state is simply a conjunction (AND-ing) of all the states mentioned in Step 4 above, except that the largest disk D10, is *on* the pole P1.

The corresponding PDDL representations for the 6 disk and 10 disk towers of hanoi problem are present in the files "hanoi-6.pddl" and "hanoi-10.pddl".

Both **FF** and **Blackbox** planners were applied onto the PDDL representations, however only *FF* was able to produce valid plans for the 6 disk and 10 disk problem. *Blackbox* was unable to produce any results, due to relatively large nature of the state-space (I halted execution after 1 minute since the blackbox executable started running).

5.1 Questions

1. Give successful plans from at least one planner with 6 and 10 disks

The plan for the towers of hanoi problem with 6 disks, generated by *FF* is shown in Figure 13.

Clearly it has $2^6 - 1 = 63$ steps in the plan. Now the problem with 10 disks will have a plan of $2^{10} - 1 = 1023$ steps. The *FF* planner produces a valid plan with 1023 steps, but for the lack of space it is not produced here. The plan is present under the "resources/Hanoi Plans" directory as "hanoi-10-solutions".

```
step    0: MOVE-DISK D1 D2 P2
        1: MOVE-DISK D2 D3 P1
        2: MOVE-DISK D1 P2 D2
        3: MOVE-DISK D3 D4 P2
        4: MOVE-DISK D1 D2 D4
        5: MOVE-DISK D2 P1 D3
        6: MOVE-DISK D1 D4 D2
        7: MOVE-DISK D4 D5 P1
        8: MOVE-DISK D1 D2 D4
        9: MOVE-DISK D2 D3 D5
       10: MOVE-DISK D1 D4 D2
       11: MOVE-DISK D3 P2 D4
       12: MOVE-DISK D1 D2 P2
       13: MOVE-DISK D2 D5 D3
       14: MOVE-DISK D1 P2 D2
       15: MOVE-DISK D5 D6 P2
       16: MOVE-DISK D1 D2 D6
       17: MOVE-DISK D2 D3 D5
       18: MOVE-DISK D1 D6 D2
       19: MOVE-DISK D3 D4 D6
       20: MOVE-DISK D1 D2 D4
       21: MOVE-DISK D2 D5 D3
       22: MOVE-DISK D1 D4 D2
       23: MOVE-DISK D4 P1 D5
       24: MOVE-DISK D1 D2 D4
       25: MOVE-DISK D2 D3 P1
       26: MOVE-DISK D1 D4 D2
       27: MOVE-DISK D3 D6 D4
       28: MOVE-DISK D1 D2 D6
       29: MOVE-DISK D2 P1 D3
       30: MOVE-DISK D1 D6 D2
       31: MOVE-DISK D6 P3 P1
       32: MOVE-DISK D1 D2 D6
       33: MOVE-DISK D2 D3 P3
       34: MOVE-DISK D1 D6 D2
       35: MOVE-DISK D3 D4 D6
       36: MOVE-DISK D1 D2 D4
       37: MOVE-DISK D2 P3 D3
       38: MOVE-DISK D1 D4 D2
       39: MOVE-DISK D4 D5 P3
       40: MOVE-DISK D1 D2 D4
       41: MOVE-DISK D2 D3 D5
       42: MOVE-DISK D1 D4 D2
       43: MOVE-DISK D3 D6 D4
       44: MOVE-DISK D1 D2 D6
       45: MOVE-DISK D2 D5 D3
       46: MOVE-DISK D1 D6 D2
       47: MOVE-DISK D5 P2 D6
       48: MOVE-DISK D1 D2 P2
       49: MOVE-DISK D2 D3 D5
       50: MOVE-DISK D1 P2 D2
       51: MOVE-DISK D3 D4 P2
       52: MOVE-DISK D1 D2 D4
       53: MOVE-DISK D2 D5 D3
       54: MOVE-DISK D1 D4 D2
       55: MOVE-DISK D4 P3 D5
       56: MOVE-DISK D1 D2 D4
       57: MOVE-DISK D2 D3 P3
       58: MOVE-DISK D1 D4 D2
       59: MOVE-DISK D3 P2 D4
       60: MOVE-DISK D1 D2 P2
       61: MOVE-DISK D2 P3 D3
       62: MOVE-DISK D1 P2 D2

time spent:    0.00 seconds instantiating 231 easy, 0 hard action templates
               0.00 seconds reachability analysis, yielding 42 facts and 166 actions
               0.00 seconds creating final representation with 41 relevant facts
               0.00 seconds building connectivity graph
               0.00 seconds searching, evaluating 334 states, to a max depth of 4
               0.00 seconds total time
```

Figure 13: Plan for Towers of Hanoi with 6 disks

2. Do you notice anything about the structure of the plans? Can you use this to increase the efficiency of planning for Towers of Hanoi? Explain

One observation from the three plans generated (3, 6 and 10 disks) is that whenever there are odd number of initial disks, then the top most disk is moved onto the "destination" pole, and when there are even number of initial disks, then the top most disk is moved onto the "middle" pole.

Another noticeable aspect of the problem is that the towers of hanoi can be viewed as a simple recursive problem of moving the smaller $n-1$ disks from $P3$ to $P2$ then moving the largest disk from $P3$ to $P1$ followed by moving the $n-1$ disks from $P2$ to $P1$. This can be empirically verified - at every $(2^N-1)^{th}$ step, if the initial number of disks n are odd, then there will be $N-1$ disks on P2 and largest disk will be on P1. If the initial number of disks n are even, then there will be $N-1$ disks on $P3$ and the largest will be on $P2$. Of course here, $0 < N <= n$.

Thus the entire plan can be represented recursively (and efficiently) as:

Algorithm 1 Towers of Hanoi recursive definition

1: **function** HANOI-SOLVER($N, P3, P2, P1$) ▷ Move N disks from P3 to P1 (using P2 as intermediate)
2: HANOI-SOLVER($N-1, P3, P1, P2$)
3: MOVE($1, P3, P1$) ▷ Move the largest disk from P3 to P1
4: HANOI-SOLVER($N-1, P2, P3, P1$)

Thus this is a bottom-up approach in constructing a plan. All one needs to define are the "macro" propositions HANOI-SOLVER and MOVE (which is applied when there is just the largest disk remaining on $P3$). The planner can then use this base condition (a.k.a "macro" proposition) and basically "un-wind" the call stack to generate sub-plans. The correct plan is then obtained by simply reversing the "popped" elements of the call stack.

3. In a paragraph or two, explain a general planning strategy that would take advantage of problem structure. Make sure your strategy applies to problems other than Towers of Hanoi. Would such a planner still be complete?

A general recursive algorithm takes advantage of whenever there is a possibility of breaking down a problem into multiple smaller, but similarly structured problems, each if solved (and possibly in parallel if there is

sufficient independence) can be "combined" to get the solution to the original problem. The generic recursive algorithm is:

Algorithm 2 Recursive Planner

```
1: function RECURSIVE-PLANNER(Prob_N)                    ▷ Solve a sub-problem
2:     if Some Terminal Condition then
3:         TERMINAL-OPERATION(Prob_1)                    ▷ Some constant terminal operation
4:     RECURSIVE-PLANNER(Subset(Prob_N))                 ▷ Subset(Prob_N) might be Prob_{N-1}, Prob_{N/2} etc,..
```

There are a whole class of problems to which a recursive planner can be applied. For instance, this algorithm can be used in large scale map navigation problems, where loading an entire terrain in one go may be memory prohibitive.

There are two conditions for a planner to be complete,

1. **Produces a plan if there is one**: The recursive planner relies on all the sub-goals to complete their execution (or the recursion stack to be empty). So in case all the sub-goals reach their respective terminating condition, then all of them will reach completion (this is from an execution standpoint - different from the notion of completeness). If all the individual problems finish, then the top-level problem, which is basically a composition of these sub-problems will also produce a valid plan.

   However, it is possible that even though the entire problem has a solution, one of the sub-problems may never reach completion – this would cause the original problem to not produce a plan. Hence it cannot always be guaranteed that the planner produces a plan if there is a valid one.

2. **Reports that there is no plan if there is none**: The same sub-problem non-termination issue exists here as well. In case one of the sub-problems gets "stuck", there is no way for the original problem to terminate reporting that there is no plan.

However, if some sort of a terminating "*parameter*" can be used to force the sub-problems to halt/report no solution, then we can guarantee completeness. In such a case the "stack overflow" issue will be resolved and both the conditions of completeness will be met. Basically, we need to ensure that the sub-problems will be forcibly terminated which will ensure that the main problem will always terminate as well! (Can almost be viewed as a "stack unrolling" limited algorithm, much like depth-limited search).

References

[1] Henry Kautz and Bart Selman. Blackbox: A new approach to the application of theorem proving to problem solving. In *AIPS98 Workshop on Planning as Combinatorial Search*, volume 58260, pages 58–60, 1998.

[2] J. Hoffmann. The fast-forward planning system. *AI magazine*, 22(3), 2001.

[3] Henry Kautz, David McAllester, and Bart Selman. Encoding plans in propositional logic. *KR*, 96:374–384, 1996.

[4] Charles Leiserson Thomas Cormen and Ronald Rivest. *Introduction to Algorithms*. 1996.