RIP HW 1

Inder Dhir, Ajmal Kunnummal, Sagar Laud, James Liu, Richard Stauffer

October 6, 2014

1 Problem 1

(a) **Compare the Methods of the Two Planners:**
Blackbox Planner
This planner operated by turning the given problem into a set of boolean satisfiability problems through two mechanisms. The front end is a graphplan technique where instead of state nodes and edges representing possible traversal, we have nodes of actions/facts and edges from facts $\rightarrow$ actions or from actions $\rightarrow$ facts affected by said action. Arranged in an alternating fashion: Facts, to possible actions, back to facts. It also uses backward chaining and iterative depth probing to keep from exploring too many extraneous nodes.

VHPOP Planner
VHPOP is a partial-order planner. This means it generates a lists of actions necessary to get to the goal, only constraining their order when absolutely necessary (One has preconditions, an earlier action must synthesize) Based off POCL it eliminates flaws in a partial plan until all preconditions for each action are satisfied. Its given list of features (From the VHPOP website) are as follows:

- Can plan with either ground or lifted actions.
- Can enforce joint parameter domain constraints when using lifted actions.
- Has several different plan ranking heuristics to choose from that can be combined into complex plan ranking functions.
- Efficiently implements all common flaw selection strategies, including LCFR and ZLIFO, and many novel ones, and allows flaw selection strategies to be specified using a concise notation.
- Several flaw selection strategies can be used simultaneously.
- Implements A*, IDA*, and hill climbing search.

(b) **Which Planner was Fastest:**
Both took very little time, as this is a relatively small problem in terms of search space, but the Blackbox planner was still undoubtedly faster at 12 milliseconds versus the 5160 milliseconds of the VHPOP planner.

(c) **Why Might this Planner be Faster?**
Much of A*'s speed relies on finding a good heuristic. Since VHPOP uses several generalizable heuristics it might have been able to save time if given a problem specific heuristic. We have seen that in later problems (Sokoban) Blackbox actually performs very poorly. This is probably due to an effective result to small state spaces, however for longer plans it seems to wasting a lot of work by first exploring all plans on step size 1, then all plans of size 2, 3, 4, 5, and so on. So it terminates quickly for plans less than say n but becomes very slow later on. Comparably, VHPOP might beat out Blackbox on long step problems.

2 Problem 2

(a) **Solutions for Sokoban Problems:**
Solutions given by FF Planner
For Black box solutions (2.1, and 2.2) please look in "Folder Problem 2"

2.1
step 0: MOVE SOUTH BOT B4 B3
1: MOVE EAST BOT B3 C3
2: MOVE EAST BOT C3 D3
3: MOVE SOUTH BOT D3 D2
4: PUSH WEST BOT BLOCK1 D2 C2 B2
5: MOVE NORTH BOT C2 C3
6: MOVE WEST BOT C3 B3
7: MOVE WEST BOT B3 A3
8: MOVE SOUTH BOT A3 A2
9: MOVE SOUTH BOT A2 A1
10: MOVE EAST BOT A1 B1
11: PUSH NORTH BOT BLOCK1 B1 B2 B3
12: PUSH NORTH BOT BLOCK1 B2 B3 B4
13: PUSH NORTH BOT BLOCK1 B3 B4 B5

2.2
step 0: MOVE NORTH BOT A5 A6
1: MOVE EAST BOT A6 B6
2: MOVE EAST BOT B6 C6
3: PUSH SOUTH BOT BLOCKB C6 C5 C4
4: MOVE NORTH BOT C5 C6
5: MOVE WEST BOT C6 B6
6: MOVE WEST BOT B6 A6
7: MOVE SOUTH BOT A6 A5
8: PUSH EAST BOT BLOCKA A5 B5 C5
9: MOVE SOUTH BOT B5 B4
10: PUSH EAST BOT BLOCKB B4 C4 D4
11: MOVE WEST BOT C4 B4

12: MOVE NORTH BOT B4 B5
13: MOVE NORTH BOT B5 B6
14: MOVE EAST BOT B6 C6
15: PUSH SOUTH BOT BLOCKA C6 C5 C4
16: MOVE EAST BOT C5 D5
17: PUSH SOUTH BOT BLOCKB D5 D4 D3
18: PUSH SOUTH BOT BLOCKB D4 D3 D2
19: PUSH SOUTH BOT BLOCKB D3 D2 D1
20: MOVE NORTH BOT D2 D3
21: MOVE WEST BOT D3 C3
22: PUSH NORTH BOT BLOCKA C3 C4 C5
23: MOVE EAST BOT C4 D4
24: MOVE NORTH BOT D4 D5
25: PUSH WEST BOT BLOCKA D5 C5 B5
26: MOVE SOUTH BOT C5 C4
27: MOVE WEST BOT C4 B4
28: PUSH NORTH BOT BLOCKA B4 B5 B6
29: MOVE EAST BOT B5 C5
30: MOVE NORTH BOT C5 C6
31: PUSH WEST BOT BLOCKA C6 B6 A6

2.3
This answer was too long to include. Please refer to the file "Problem 2/sokoban-2.3-FF.txt" for the complete solution

(b) **Compare two Planners:**
The FF planner always rounded down to 0.0 seconds for these problems but blackbox took .05 seconds, 12 minutes, and would not solve the 3rd one without manually allowing for larger step size solutions. FF might have performed better due to a breadth first search approach to enforced hill climbing. They both used variations of graphplan so it is safe to assume that blackbox's issues arose with its other half: the boolean constraint satisfiability solver. This is a rather large search space to be defining boolean constraints between objects and that might have been its hang up, it also searches all plans of each length before finding the correct one whereas depth first search has a (small) chance of finding a correct path in its first try.

(c) **Challenges of Expressing in PDDL:**
Semantic (or rather; logical/numerical based) planning require carefully spelled out relationships between items. In a graph we might have a relationship where if the x coordinate of one item is the same as the x coordinate of another, they are on the same column. Meanwhile geometric constraints are much trickier to define. For instance, parallelism: To do this we have to express each line in formula and determine the slope. A much more involved process.

PDDL (more importantly in the domain file) contains actions with a precondition and an effect. If we're manipulating geometric objects in a "real" world it

quickly becomes difficult to cover all the factors changed in the state. After the action, are all previous lines that were parallel still parallel? Are we occupying the same space as another object? Are triangles still congruent to one another?

(d) **Describe a Problem to be used in Semantic Planing:**
Say we had a world with a lot of propositions.$P \rightarrow Q, Q \rightarrow R, \bar{Q} \rightarrow S$ A Forward chaining approach to determine if we can get from X$\rightarrow$Y would be faster than considering the domain in a physical space and figuring out connections therein. In other words, deterministic, fully observable, and stochastic. Thus classical planners are good at solving simple games. Consider solitaire with well defined states and goals. (but without the ambiguity of the cards, basically allow the computer to cheat and undo moves if necessary) Stochastic would be good at iterating through possible states and finding a solution if there exists one.

3 Problem 3

(a) **Solutions for Sokoban Problems:**
Our planner was written in Java:
2.1
Found goal in 14 steps
Path: DOWN RIGHT RIGHT DOWN LEFT UP LEFT LEFT DOWN DOWN RIGHT UP UP UP
Time taken: 0m 0.008695s
No of states explored: 88

2.2
Found goal in 32 steps
Path: UP RIGHT RIGHT DOWN UP LEFT LEFT DOWN RIGHT DOWN RIGHT LEFT UP UP RIGHT DOWN RIGHT DOWN DOWN DOWN UP LEFT UP RIGHT UP LEFT DOWN LEFT UP RIGHT UP LEFT
Time taken: 0m 0.056724s
No of states explored: 1067

2.3
Found goal in 89 steps
Path: RIGHT DOWN DOWN RIGHT RIGHT UP UP LEFT LEFT RIGHT RIGHT DOWN DOWN RIGHT RIGHT UP UP UP UP UP RIGHT RIGHT DOWN DOWN DOWN LEFT RIGHT UP UP UP LEFT LEFT DOWN DOWN DOWN LEFT LEFT DOWN DOWN RIGHT RIGHT UP UP UP UP DOWN DOWN LEFT LEFT DOWN DOWN LEFT LEFT UP UP RIGHT RIGHT RIGHT LEFT DOWN DOWN RIGHT RIGHT UP UP UP DOWN DOWN DOWN LEFT LEFT LEFT LEFT LEFT LEFT UP UP RIGHT RIGHT RIGHT RIGHT RIGHT LEFT DOWN DOWN RIGHT RIGHT UP UP
Time taken: 0m 1.074816s

No of states explored: 8021

Sokoban Challenge
Found goal in 76 steps
Path: DOWN LEFT LEFT LEFT LEFT UP RIGHT RIGHT DOWN DOWN DOWN LEFT UP RIGHT UP UP LEFT LEFT DOWN RIGHT DOWN RIGHT RIGHT RIGHT UP RIGHT UP LEFT LEFT RIGHT DOWN DOWN LEFT LEFT UP RIGHT UP RIGHT UP UP LEFT DOWN DOWN DOWN LEFT LEFT UP RIGHT DOWN RIGHT UP RIGHT RIGHT DOWN LEFT DOWN LEFT RIGHT UP UP LEFT LEFT DOWN RIGHT DOWN DOWN LEFT UP RIGHT UP UP RIGHT UP UP LEFT DOWN DOWN
Time taken: 1m 31.956295s
No of states explored: 47077

(b) **Compare our Planner vs Previous Planners:**
Our planner fell somewhere in the middle of the two planners we already tried. While FF ran in less than .01 seconds for each problem, blackbox ran anywhere from .05 seconds to 12 minutes. Out planner generated solutions in .008 seconds,.05 seconds ,1 second, 1 minute 31 seconds respectively. So while it was perhaps tied with FF for the first problem, it performed worse for the rest. And in all cases, better than blackbox. This might be due to the already seen high efficiency of FF on sokoban problems as explained in question 2.1 (b). Our planner was fast due to our speed improvements on A* explained below in d). (As well as being able to choose a heuristic that we knew would be effective on the problem)

(c) **Proof of our Planner's Completeness:**
Our planner uses an A* search with some pruning. It is complete because the search space contains every unique state, except the pruned states, which are guaranteed to never lead to the goal state. This is because we have accounted for this specific problem's geometric constraints having to do with rendering a box unreachable for the rest of the problem. So while a less informed planner will explore states with a unreachable box for a long time, ours never even explores those paths. Since our planner visits every state that is not the goal or a terminating state, we are guaranteed to reach some sequence of states leading to the goal if such a path exists. (E.G. if n states are necessary in the plan: A* is guaranteed to reach state n-1 and then explore all its neighbors. Inductively we then see we can reach all states 1$\rightarrow$n in consecutive order at some point.)

(d) **How did we Speed up our Planner?:**
We used A* search because it is much faster than BFS as it expands the states from the open set that are most likely to reach the goal state first before the other states. It does this with the help of a heuristic.

Our heuristic (rip.hw1.planner.sokoban.SokobanHeuristic) calculates the sum of the manhattan distances between each box and its goal. As this is simply a

relaxation of the problem, the heuristic is admissible and thus the algorithm always gives an optimal solution.

In addition to the heuristic, we also prune the search tree that improves performance a lot, especially for the challenge problem. We make the observation that if you 'jam' one of the boxes anywhere on the map except a goal position, there is no way a solution can be found, no matter what you do. We also make the observation that a box is jammed when there is a wall or another jammed box on at least two of its adjacent sides. As we only remove states that will never lead to the goal state, the algorithm will still be complete.

The relevant methods that implement this are in rip.hw1.planner.sokoban.SokobanHeuristic. The method names are canReach, jammed and recJammed. canReach checks weather any of the boxes in the state are 'jammed'. jammed and recJammed checks weather a particular box is jammed. Because to know whether any box is jammed, you need to know whether a neighboring box is jammed, the algorithm is recursive.

4 Problem 4

(a) **Solutions for Large Hanoi Problems:**
Solutions given by FF Planner
Since the solutions for tower of hanoi problems are $2^n - 1$ length, where n is the number of discs, it didn't seem prudent to include a text form in this document. Please instead refer to the problem 4 folder of our code, it contains a hanoi10output.txt and a hanoi12output.txt with the solutions enclosed.

(b) **Notes on the Structure of the Plan:**
The Tower of Hanoi plan is actually rather simple. And it is very very repetitive. For instance: imagine a tower n high in position 3 and a goal of a tower n high in position 1. Then, logically due to the rule of only smaller numbers on larger numbers. In order to move disc n to 1, 1 has to be empty and there can't be any discs on top of disc n. Therefore there is a stack of n-1 in position 2. So there exists some number of moves to stack n-1 in position 2, then a move of disc n from position 3 to 1, followed by the the same number again of moves to get n-1 from position 2 onto disc n.

In short if we consider our position numbers flexible we repeat our pattern constantly. Even with n=5. We move 4 discs onto position 2, which requires moving 3 discs onto position 1, which requires moving 2 discs onto position 2, and moving 1 disc onto position 3. Using these precondition states for actions we could then repeat our as-of-yet partially discovered plan and swap in the correct position locations.

(c) **General Planning Strategy using Problem Structure:**
As already explained there is a lot of repetition in Tower of Hanoi. If we saved our states in sort of a dynamic programming solution we could save a lot of time by comparing previous found solutions. So we save our states and our minimum cost paths for all explored states in a plan. But, especially in this instance, we

save them in a generalizable manner that we can plug in variable names at the end. So we might consider the state 001 and 010 to be the same state with the same sequence of actions to get to it. However in this sequence of actions the position names might be left blank and we can fill those in as needed in our plan.

We do need to be careful to consider the relative pattern compared to the start space, because in that instance we cannot consider the moves to get from all discs on p3 to p1 (many moves)as the same as getting all discs on p3 onto p3 (0 moves). Anyways, we would take our goal state, consider the n-1 disc problem and assemble our solution as Solve(N)=Solve(n-1) + move disc n + Solve(n-1).

This should be complete because it returns a solution unless one of the subproblems is impossible. And if we can't complete a precondition for some necessary action in the final solution, then there exists no solution.

5 Problem 5

(a) **HTN Planning Problem for Hanoi:**
Please refer to the folder "Problem 5" for our HTN Planing problem.

(b) **Describe Encoded Domain Knowledge:**
We knew that the Hanoi problem can be solved recursively, as moving larger stacks of disks was simply solving two smaller Hanoi instances, with one or two more moves. Thus we could create a compound task "move" that moves recursively breaks down the task into smaller and smaller Hanoi instances that ultimately breaks down into a single disk move from one peg to another, which is left as the "move_disk" primitive task.

(c) **Solve HTN for 3 12 discs:**
Look at folder "Problem 5" for the encoded py files and the subfolder "HtnPlanner" for outputted solutions.

(d) **Compare with non HTN Planner:**
Both of the planners produced optimal solutions ($2^n - 1$ total steps, where n is the number of disks). The HTN planner, pyhop, did notably better on large instances Hanoi (0.293s on Hanoi-12) when compared to FF (5.4s on Hanoi-12). Both were run on the same machine.

(e) **Observations:**
Even with a interpreted language like python, it did notaby faster than FF, as it took full advantage of the problem structure. FF, while implemented in a faster language, C, is still less informed about the problem than pyhop, which is why pyhop was faster for larger instances of Hanoi.