Robot Intelligence - Planning CS 4649/7649

Group Homework 1: Classical Planning

October 6, 2014

Team 1: Mark McCurry, Vivek Agrawal, Sen Hu, Ryan Kerwin, Nehchal Jindal

1 Warming up: Towers of Hanoi

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

Blackbox:

The basic functioning of the Blackbox planner is that it takes inputs in the form of PDDL (or similar first-order logic languages, such as STRIPS), then uses a combination of SATPLAN and Graphplan methods to find a solution. Given a problem in propositional logic, Blackbox creates a 'plan graph' for the problem, which can then be pruned to improve search time. This representation is then converted to a Boolean satisfiability problem in conjunctive normal form (CNF), which a planning method such as SATPLAN can solve quickly. The combination of these two approaches allows Blackbox to leverage the strengths of both methods--the pruning and setup of Graphplan and the quick search time of SATPLAN--making for a planner that outperforms both.[1]

VHPOP:

VHPOP (Versatile Heuristic Partial Order Planning) is a planner that improves upon the basic Partial Order Planner (POP) paradigm by introducing a heuristic into the plan-space search. While traditional POP planners search the space in an uninformed manner (such as breadth-first search or similar), the addition of a heuristic element means that VHPOP performs an informed search (specifically an A* search), where it actively seeks a valid solution.[2]

b) *Which planner was fastest?*

Blackbox Version 43:	Average **4ms**	←	Fastest
VHPOP 2.2.1:	Average **200ms**		

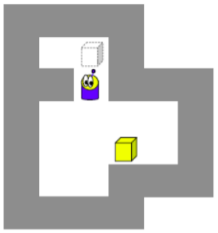
c) *Explain why the winning planner might be more effective on this problem?*

In general, a plan-space planner such as VHPOP would be expected to run more quickly than a state-space planner such as Blackbox, since it has a lower branching factor and deals only with partial plans, rather than full world states (refer to slide 28 from September 2nd's class slides). However, this is a very small problem with a low branching factor and small world state to begin with; the overhead required to run a POP approach seems to be excessive for such a case (forget the heuristic element - that's overkill), and Blackbox can find a solution very quickly from its internal CNF representation of the problem using SATPLAN.

1:(move-disk d1 d2 p1)
2:(move-disk d2 d3 p2)
3:(move-disk d1 p1 d2)
4:(move-disk d3 p3 p1)
5:(move-disk d1 d2 p3)
6:(move-disk d2 p2 d3)
7:(move-disk d1 p3 d2)

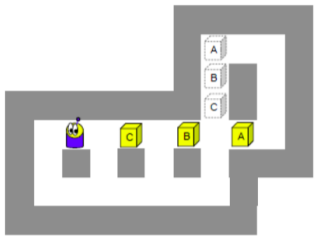
Optimal solution found by both the Blackbox and VHPOP planners.

2 Sokoban PDDL

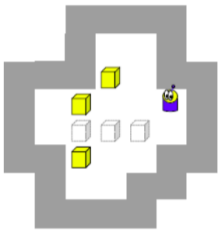


(a) Problem 2.1

(b) Problem 2.2



(c) Problem 2.3



(d) Sokoban Challenge

a) *Show successful plans from at least one planner on the three Sokoban problems in Figure 2(1-3*).

Refer to file “README_ALL_SOLUTIONS” in “Problem2” folder

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

	FF Planner	SatPlan
Problem2.1	0 s	0.04 s
Problem2.2	0 s	1.3 s
Problem2.3	0.01 s	144.58 s
Sokoban Challenge	0.04 s	Takes too long to run

The FF planner runs far more quickly than the SatPlan planner in this domain. This makes sense: the Sokoban problem includes relationships between many different objects (robot, boxes, goals, walls, empty spaces). The SatPlan solver has to encode all of this information as a state and then search the space created by this representation. As the size of the problem grows, this blows up significantly (as evidenced by the increasingly poor runtimes of SatPlan, culminating in a failure to halt in the Challenge problem.

On the other hand, the FF planner encodes the information as a graph and then uses heuristic information to walk towards a goal. This does not require the entire world state to be encoded, and the search is directed (by an A*-like algorithm), so the planner finds a solution much more quickly.

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

Based on our experience, PDDL does not appear to be a suitable approach for applications such as solving Sokoban puzzles as there are significant challenges in representing the different geometric constraints such as wall, obstacle, block and robot positions, along with the open and navigable spaces given the different degrees of freedom of movement for the robot assembly. A notable problem with defining the Sokoban puzzle with PDDL was the need to represent each and every state condition in the form of initialization or pre-conditions which made the PDDL very verbose and also prone to errors.

While solving the Sokoban puzzle with PDDL specifications, we were able to create abstractions of the geometric constraints in the symbolic plan, primarily since the problem space was 2 Dimensional and limited in worst case to a 11*9 grid size. In more involved applications, this representation will become increasingly difficult.

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

A good example of a problem that is best suited for semantic (classical) planning is the Warehouse Robot that loads and unloads merchandise from the warehouse to delivery trucks. Classical Planning based on Semantic Representation is a very effective way of solving problems that can be easily characterized by atomic initial and goal conditions and sequence of actions that have structured representations. Also, the underlying assumptions of finite, deterministic, observable and static state space have implications on the Semantic Representation.

These considerations make the Warehouse Robot a problem best suited for semantic (classical) planning.

3 Sokoban Challenge

a) *Give successful plans from your planner on the Sokoban problems in Figure 2 and any others.*

The plans can be found on the GitHub repository. The timings are as follows:

	Team_1 Planner
Problem2.1	0.004 s
Problem2.2	0.002 s
Problem2.3	0.014 s
Sokoban Challenge	0.018 s

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

(Refer to the table in question 2 to see the outputs from the other planners)

Our planner clearly runs much more quickly than the SATPLAN planner. This is because SATPLAN needs to encode the entire state of the world for each state in order to solve the problem; which for large problems such as the Sokoban Challenge blows up quickly (to the point that the SATPLAN planner took too long to run on this problem.

Our planner only has a marginal advantage over the FF planner we used in problem 2. In fact, in all but the challenge problem, the timing on the FF planners results was essentially equivalent to ours. On the challenge problem, which is more complex than the rest, our planner is more than twice as fast as the FF planner. The reason for the similar runtimes is that the FF planner approaches the problem in a very similar manner to the way that our planner approaches it: it encodes the problem as a graph and then uses some heuristics to expand the graph until a goal state is reached. However, the reason that our planner outperforms the FF planner on the challenge problem (and likely larger problems, too) is that our planner was built specifically to operate in the Sokoban domain. We had the luxury of choosing how our state graph would be built and tuning our heuristics towards solving Sokoban. Given that, it's impressive that the FF planner kept up with our planner on the smaller problems; but it is unlikely that it would stay competitive for long if we expanded our tests to larger and larger problems.

c) *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.*

For the Sokoban problem, the defined planner P1 is a complete algorithm. First let's prove the completeness of another algorithm P2. This algorithm will look at each possible up/down/left/right movement of the robot. Let a state be defined as the tuple of the robot position and the set of box positions. Let the directed graph G=(V,A) define which state transitions exist with unitary weighted edges. When A* is applied to G with cost heuristics which are pure functions of the underlying map and the state, then each node may only be visited a finite number of times as a node is only visited if it is done so with a lower cost as the triangle inequality holds under this unitary weighted graph. This upper bound is given by O(|A|). Thus given the above constraints, all connected states will be visited in O(|A|) worst case and by detecting if the goal state has been reached or not, it can be said if the problem has any solution.

With this established, the used algorithm is a restricted version of above which produces a reduced state directed graph G2=(V2,A2) which maintains connectivity with respect to the goal state. The first graph simplification is to remove any subgraphs which cannot feasibly reach the goal state. In all presented problems it is assumed that the number of boxes is equal to the number of goals. Thus if one or more boxes enters a configuration where a box at a non-goal space cannot decrease its distance to a goal under all permutations of movable boxes, then the goal state cannot be reached. Thus this subgraph originating at the vertex representing this state cannot be connected to the goal and eliminating it does not alter the graph connectivity.

The second graph simplification is to merge states under which the robot exists within the same subspace of the map and the boxes are at identical positions. Let the new state S2 be the subspace of the robot and the box positions. Let G3=(V3,A3) be G2 with vertices merged when their S2 representation is identical and with duplicate directed edged in A3 eliminated. If there was a connected path in G2, then there must be a connected path in G3 as the path can be expressed by merging identical path steps from the solution in G2 and by definition the merged edges will exist. If there was not a connected path in G2, then there must not be a path in G3, as that would require an edge A to connect the subgraph that the solution exists in and the subgraph that is accessible in G2. No such edge exists in A and by definition any two nodes that are merged in G3 are connected, thus there cannot exist an edge A in G3.

Given the proof of completeness of the full problem under A* and the lemmas of identical connectivity with respect to the starting state and the goal state, this algorithm is complete.

d) *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.*

Our two primary expediting measures are introduced in part (c) - we ignored states that led to boxes getting stuck (thus indicating a dead end path); and we treated states where the boxes were in identical locations and the robot was contained in the same subspace of the map as equivalent states.

For the first simplification, this sped up our code execution because it prevented our planner from searching paths that we could already determine would end in failure. This was effectively accomplished by skyrocketing the cost of states where we detected that a box had become stuck. Since our planner was using A* search, these high-cost states were always ignored when the planner chose the next state to explore.

For the second simplification, we perceived that the most important element in moving towards a goal state was the movement of the boxes, and that when the robot moved without pushing a box, nothing was accomplished. We leveraged this fact and explored the search space by generating states only when the robot pushed a box. Since we could track the robot's position each time it pushed a box, we were able to reproduce the path that the robot must have taken to move between pushes. This simplification allowed us to ignore all of the interstitial states between box pushes, cutting down the search space significantly; which in turn improved our run time.

competition stats) *To give you feedback on how well you accomplished these objectives we conduct an informal competition for who can produce the most optimal planner for the 4th problem(Figure 2.(d)). Purportedly this is a difficult problem for humans to solve - have a go at it before you let your planner do the work! If your planner can solve it, include the solution and report the following pieces of information:*

- *Computation time for Sokoban Challenge*: 0.018 s
- *Number of steps in your plan*: 197 iterations; 19 'nodes' in the path; 76 robot steps
- *Number of states explored*: 265 found, 197 explored
- *Language used*: C++
- *Machine vitals*: Intel i5-4300U CPU @ 1.90GHz × 4, 7.7 GB RAM

4 Towers of Hanoi Revisited

a) *Give successful plans from at least one planner with 10 and 12 discs*.

Refer to file "README_ALL_SOLUTIONS" in "Problem4" folder

b) *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.*

Yes, there is clearly a structure in the plans for different number of discs in the Hanoi problem space. This structure derives from the fact that each additional disk requires establishing interrelationships between the other discs in the problem set and also adds a layer of recursive reference in case of problem solving within the domain definition.

There might be scope for exploiting this structure along with some advanced heuristics implemented by different planners, but for our implementation, we did not realize any appreciable improvement in performance using the different planners. The run time for the planners was proportional to the increase in the number of discs, as can be seen from the test run results.

c) *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?*

In any problem where the structure is such that the problem can be broken down recursively into equivalent subproblems, we can find an optimal solution by solving the smallest (lowest level of recursion) subproblem, then using this result as a starting point for solving the next smallest subproblem, etc. until we have discovered the global solution. This is the essence of dynamic programming: a problem is decomposed into equivalent subproblems and the solutions to these are recursively used to compute a solution to the initial problem. In the case of towers of hanoi, this would take the form of using the solution of the (trivial) 1-disk Towers of Hanoi problem to solve the 2-disk Towers of Hanoi problem; then using this to solve the 3-disk case; and so on until a solution is reached for the 10- or 12-disk case. This is not constrained only to the Towers of Hanoi problem. Any problem for which a solution to a subproblem ensures a solution to a larger problem can be solved this way (such as in textbook dynamic programming problems, such as the knapsack problem).

A planner that takes advantage of this equivalent substructure property *would* still be optimal (assuming it was optimal to begin with). Consider the case when the planner has reduced the problem to its smallest subproblem; in this case it would need to find a solution without the aid of any recursion. By the definition of our problem class (the subproblem is equivalent to the larger problems), we can use our solution to the lowest-level subproblem as a starting point to solve the next level up (think of using the solution to the 1-disk Towers of Hanoi to solve the 2-disk version). By induction, we could say that to solve the *k*th level problem, we can use the *k*-1st level solution, all the way up to the top-level problem. Therefore, if the planner is complete on the base-level problem, it will be complete on the original problem.

5 Towers of Hanoi with HTN planning

a) *Formulate a HTN planning problem for the Towers of Hanoi.*

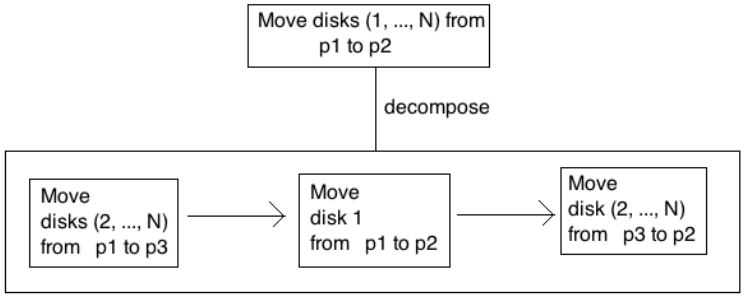
HTN Planning problem consists of initial state and the higher level task. Initial state can be represented as set of predicates. Higher level task, is decomposed into subtasks according to description of domain. Here the higher level task is to move stack ([disk_1, …, disk_N]) from Pole 1 to Pole 2. The HTN planning problem for 3 disks has been formalized below.

Planning Problem (using 3 disks)

```
DOMAIN: tower_of_hanoi_domain
INIT: [ (smaller d1 p1) (smaller d2 p1) (smaller d3 p1) (smaller d1 p2)
      (smaller d2 p2) (smaller d3 p2) (smaller d1 p3) (smaller d2 p3)
      (smaller d3 p3) (smaller d1 d2) (smaller d1 d3) (smaller d2 d3)
      (clear p2) (clear p3) (clear d1)
      (disk d1) (disk d2) (disk d3)
      (on d1 d2) (on d2 d3) (on d3 p1) ]
TASK: moveStack( [d3, d2, d1], p1, p2, p3)/ see description in part(b)
```

b) *Describe the domain knowledge you encoded and resulting planning domain(i.e. primitive tasks, compound tasks, and methods) in detail.*

The problem of shifting stack from pole 1 to pole 2 can be decomposed recursively into ordered subtasks. The domain has only one primitive task and that is to move disk from top of one pole to another.



Problem-Domain (tower_of_hanoi_domain):

```
PREDICATE( disk(x) )                    // x is a disk
PREDICATE( smaller(disk1, disk2) ) // disk x1 is smaller than disk2
PREDICATE( clear(disk) )                // disk is cleared
PREDICATE( on(disk1, disk2) )       // disk1 is on disk2

ACTION( moveDisk (disk, belowDisk, newBelowDisk),
   PRECOND: [clear(disk), clear(newBelowDisk), on(disk, belowDisk)],
   ADD-LIST: [ on(disk, newBelowDisk), clear(belowDisk) ],
   DEL-LIST: [ !on(disk, belowDisk), clear(newBelowDisk) ] )

ACTION( moveStack(list[1, ..., n], belowDisk, newBelowDisk, emptyDisk),
   PRECOND: [ on(list[i] list[i-1]) for all i = 2, ..., n,
         clear(list[n]), clear(newBelowDisk), clear(emptyDisk)
         on(list[1], belowDisk) ],
   ADD-LIST: [ on(list[1], newBelowDisk), clear(belowDisk) ],
   DEL-LIST: [ on(list[1], belowDisk), !clear(newBelowDisk) ] )


DECOMPOSE( moveStack(list[1, ..., n], belowDisk, newBelowDisk,
                                    emptyDisk),
   STEPS: [ moveStack(list([2, ..., n], list[1], emptydisk,
                                 newBelowDisk)),
        moveDisk(list[1], belowDisk, newBelowDisk),
        moveStack(list[2, ..., n], emptyDisk, list[1], belowDisk)  ] )
```

c) *Solve your HTN problem for the cases with 3 ~12 discs (use SHOP/SHOP2 or whatever HTN planner available to you).*

We have used SHOP2 planner to solve the above problem. Domain and problem file were written using format specified in SHOP2 documentation. Refer to part (d) for results.

d) *Solve the same cases with a non-HTN planner of your choice, and compare the results with (c).*

Num of disks	Time (HTN) (in secs)	Num of actions(HTN)	Time (FF) (in secs)	Num of actions (FF)
3	0.557	7	0	7
6	0.629	63	0	63
10	34.40	1023	0.42	1023
12	Couldn't complete (Heap exhausted)	4095 (expected)	5.80	4095

e) *Describe your observations and discuss about them.*

FF planner runs significantly faster than HTN planner. Also, for larger number of disks, FF generates paths of lesser length as compared to HTN. FF is domain-independent planner while domain information is used HTN planner. FF is heuristic based state space planner.

We expected the HTN planner to perform better than FF, as domain knowledge often helps the planner to perform better. But results show the opposite. This may be due to the recursive hierarchical formulation of problem. The formulation is complete and optimal, but the memory overhead is too large as implemented in the SHOP2 planner. HTN planner has to recurse deeply, which means exploring exponential number of more nodes as the size of the problem grows. In FF, a good heuristics helps it search for the goal more directly, meaning it uses less extra memory and operates much faster.

References:

[1] Henry Kautz and Bart Selman (1999). Unifying SAT-based and Graph-based Planning. *16th International Joint Conference on Artificial Intelligence (IJCAI-99),*Stockholm, Sweden, 1999.

[2] Håkan L. S. Younes and Reid G. Simmons. 2003. VHPOP: Versatile heuristic partial order planner. *Journal of Artificial Intelligence Research* 20: 405–430.

Group Member Contributions:

Mark McCurry

- A* Implementation for use with Sokoban Challenge
- Encoding of impossible states for sokoban
- Initial Cost functions for sokoban
- Sokoban solver optimization
- Got the solver to work on both a simple test case and level one of the flash game version
- Proof of completeness for reduced A* state space used in sokoban solver
- Debugging of tower of honoi domain
- Rewrite of tower of hanoi domain HTN SHOP code
- Ran some test cases for SHOP planner

Nehchal Jindal

- Hierarchical decomposition of tasks for Tower of Hanoi domain using SHOP2 planner.
- Formulation of planning problem for Tower of Hanoi problem using SHOP2 planner.
- Writing of problem and domain files for shop2 planner.
- Ran tests for shop2 planner
- Description and discussion of results of shop2 and FF planner for Tower of Hanoi problem

Sen Hu

- Experimented with different planners for Sokoban and Hanoi
- Troubleshoot Sokoban and Hanoi PDDLs
- Gathered and analysed results from different planners
- Contributed to answers to Tower of Hanoi and Sokoban question

Vivek Agrawal

- Generated domain and problem PDDLs for the 10 and 12 discs Hanoi problems
- Generated domain and problem PDDLs for the 4 scenarios in Sokoban problem set
- Troubleshoot Sokoban and Hanoi PDDLs
- Contributed answers to Tower of Hanoi and Sokoban conceptual and derivation based questions

Ryan Kerwin

- Improved heuristics for Sokoban challenge coding problem (calculated actual distance, rather than Euclidean distance and optimized weightings for cost function by experimentation)
- Expanded Sokoban planner to read input files for generic levels
- Recovered the robot path from a solution state in the planner
- Contributed answers to Tower of Hanoi and Sokoban questions