Daniel Pickem, Andrew Melim, Jarius Tillman, Matheus Svolenski



Deliverables include the following:

- A 4-5 page PDF summary that describes your work, results and answers to the questions posed below. Thoroughness in analysis and answers in the reports are the primary component of your grade!
- A page in your summary describing what each group member did to participate in the project.
- A zip le that includes your code and results in an organized form.
- A README file that describes how to run the code.
- Print the summary and bring it to the following class.

## Problem 1 - Pre-Project: Towers of Hanoi

1. **Explain the method by which each of the two planners nds a solution.**
   To test the Towers of Hanoi problem, we compared two diferent planners, named BlackBox and Fast Forward (FF).
   - **BlackBox (see [2])**: BlackBox is a planning system that works by converting problems specified in STRIPS notation into Boolean satisfiability problems. It employs a graph plan system, later converted, that is solved by any of a variety of fast SAT engines.
   - **Fast Forward (see [3])**: FF relies in a forward search in the state space, guided by a heuristic that estimates goal distances by ignoring delete lists. It uses an enforced hill-climbing search, that searches for the direct successors of a path, until finding a value that is better than the one it started with. It performs a complete breadth first search (BFS) for a state with strictly better evaluation. However, this method depends on the structure of the problem. If a local search fails, FF will skip everything done so far and switch to a complete best-first algorithm that simply expands all search nodes by increasing order of goal distance evaluation. This was the case in the Towers of Hanoi problem.
2. **Which planner was fastest?**
   The fastest planner was the Fast Forward (FF). Its performance can be analysed through Table **??**, that shows how long it took for each solution to be found and in how many steps this happend. In most cases, BlackBox was not able to find a solution, giving up after 50 iterations. The outputs are all included in the .zip file that accompains the project.

Table 1: Performance Comparison of BlackBox and FF (timing results in seconds).

| **Problem Instance** | **Fast Forward** | **BlackBox** | |
|---|---|---|---|
| Hanoi 3 Disks [time] | 0.00 | 0.01 | |
| Hanoi 6 Disks [time] | 0.01 | - | |
| Hanoi 6 (other team) Disks [time] | 0.01 | - | |
| Hanoi 10 Disks [time] | 0.89 | - | |
| Hanoi 3 Disks [steps] | 7 | 7 | |
| Hanoi 6 Disks [steps] | 63 | - | |
| Hanoi 6 (other team) Disks [steps] | 63 | - | |
| Hanoi 10 Disks [steps] | 1023 | - | |

3. **Explain why the winning planner might be more effective on this problem.**
   A big difference between the planners is that FF uses a good heuristic function to constantly analyse the best available options, this reduces considerably the nodes that need expansion and, therefore, helps finding a solution faster. When the problem is simple, like the 3 disks problem for the Towers of Hanoi, there are not many nodes to expand, so both planners perform a good work, however, when the problem grows in dimension the possible nodes expand exponencially, making a lot more difficult to BlackBox to expand through those states.

## Problem 2 - Project Part I: Sokoban PDDL

1. **Show successful plans from at least one planner on the three Sokoban problems in Figure 2 (1-3). The challenge problem is optional.** The plans for problem 2.1 to 2.3 as well as the challenge problem are shown in the appendix.

2. **Compare the performance of two planners on this domain. Which one works better? Does this make sense, why?**
   An overview of the numerical results is given in Table **??**. We compared three planners, namely Fast Forward (FF), Conformant-FF, and BlackBox. FF and Conformant-FF seem to use a very similar underlying architecture while BlackBox relies on a different representation of the problem as a satisfiability problem.

   - **BlackBox (see [2])**: BlackBox converts the PDDL description of the problem into a Boolean satisfiability problem and solves it using a variety of satisfiability engines. The front-end of BlackBox relies on the Graphplan system.
   - **Fast Forward (see [3])**: FF is a forward chaining heuristic state space planner. The heuristic relaxes the task at hand into a simpler task by ignoring the delete list of all operators. That relaxed problem is solved using a GraphPlan-style algorithm and the solution to the relaxed problem is used as heuristic to guide the search. The relaxed plans are also used to prune the search space. those produced by members of the respective

Table 2: Performance Comparison of BlackBox, FF, and Conformant-FF (timing results in seconds).

| **Problem Instance** | **FF (default)** | **Conformant-FF** | **BlackBox (default)** | |
|---|---|---|---|---|
| Problem 2.1 [time] | 0.0 | 0.0 | 0.27 | |
| Problem 2.2 [time] | 0.0 | 0.02 | - | |
| Problem 2.3 [time] | 0.02 | 0.12 | - | |
| Problem Challenge [time] | 0.7 | 34.54 | - | |
| Problem 2.1 [steps] | 12 | 15 | 14 | |
| Problem 2.2 [steps] | 31 | 31 | - | |
| Problem 2.3 [steps] | 110 | 88 | - | |
| Problem Challenge [steps] | 77 | 77 | - | |

relaxed solution. FF employs a slightly more elaborated form of this heuristic, which we call helpful actions pruning. The simple architecture described so far already solves most of the available benchmarks extremely efficiently. Problematic cases are when there are dead ends — states from which the goals get unreachable — or goal orderings. In the presence of the latter phenomenon, like in the Blocksworld, the local search sometimes proceeds too greedily, and gets trapped. To overcome this, we have integrated the Goal Agenda algorithm (first proposed by Jana Koehler), as well as a simple goal ordering technique of our own, based on the relaxed solutions. In order to deal with dead end states, which can cause the search to fail entirely, we have chosen a simple safety-net solution: if local search fails, then we skip everything done so far and switch to a complete best-first algorithm that simply expands all search nodes by increasing order of goal distance evaluation.

As expected (see Table **??**), FF and Conformant-FF achieve much better results and generally solve the problems in much shorter times. BlackBox takes orders of magnitude longer and fails to compute plans for problems that take more than 25 steps to solve (i.e. the solver times out). The number of steps BlackBox can handle seems to be fairly low. The first up to 20 levels of the graph are computed fairly fast, but any level in the GraphPlan graph after that take on the order of seconds or minutes. Clearly the number of levels is too low to solve a Sokoban level like the ones shown in problem 2.2 to 2.4.
It seems that, unlike FF, BlackBox does not use any heuristic to prune the search tree or search in a more goal-directed fashion. FF prunes large parts of the search space using the heuristic described above, which allows FF to handle much larger problem instances and compute a plan much faster.

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

   - PDDL can't handle continuous worlds, the world has to be discretized.
   - There is no inherent notion of adjacency or neighborhood. Such relations have to be expressed explicitly. E.g. if squares are used, for each square PDDL requires 4 adjacency clauses.

- In PDDL each square has to be given a label instead of just saying if the robot moves one step to the right, its x-coordinate increased by one. This leads to a large amount of labels and adjacency relationships in the problem description.
- It needs to be explicitly stated whether a square is empty or contains a box or robot in order to detect an obstacle. Each action taken must ensure to update the 'empty' property of a square. This is a classical manifestation of the frame problem.
- Even for small problem instances as the ones shown in the examples it was too tedious to write down the PDDL instantiation by hand. What could be described easily as a matrix required roughly 100 lines of PDDL code.

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.**

   - Clearly, geometric constraints can't be efficiently expressed in PDDL. Therefore, all problem types that have a physical interpretation and an embodiment in a physical and geometric world are not well suited for PDDL. Domains with geometric constraints that encode for example positions of objects or distances between objects (in continuous or discrete space) are best handled by geometric planners.
   - Problems that represent an abstract world, however, where neither states nor actions are tied to a geometric interpretation of the world are well suited for PDDL and classical semantic planning. In these types of problems dimensions/size of objects do not matter and actions are not tied to geometric constraints. In this case constraints define the abstract rules of the domain rather than a geometric setup.
   - A very classic problem in semantic planning is for example the 'change tire problem' (see [1]) that describes how to change a flat tire on a car. The *move* action moves objects from one abstract location to another (e.g. from the trunk to an axle). These locations hold no geometric information and are essentially just labels for objects in the world. The dimensions of objects do not matter either. Such a domain where the problem has been abstracted away from the geometric constraints are well-suited for a semantic description such as PDDL.

## Problem 3 - Project Part II: Sokoban Planner

1. **Give successful plans from your planner on the Sokoban problems in Figure 2 and any others.** See the attached file **sokoban astar sols** for the solutions.

2. **Compare the performance of your planner to the PDDL planners you used in the previous problem. Which was faster? Why?**
   Our results (see Table **??**) shows two different cost functions for our A* implementation. The column on the left shows utlizes the cost of the actions already taken by the robot in addition

Table 3: Performance results of our A* algorithm (timing results in seconds).

| **Problem Instance** | **A* with action cost** | **A* without action cost** |
|---|---|---|
| Problem 2.1 [time] | 0.01 seconds | 0.0 seconds |
| Problem 2.2 [time] | 0.16 seconds | 0.08 seconds |
| Problem 2.3 [time] | 3.24 seconds | 0.62 seconds |
| Problem Challenge [time] | 411.14 seconds | 5.45 seconds |
| Problem 2.1 [steps] | 14 steps | 14 steps |
| Problem 2.2 [steps] | 32 steps | 36 steps |
| Problem 2.3 [steps] | 93 steps | 102 steps |
| Problem Challenge [steps] | 76 steps | 120 steps |

to the Manhattan distance of boxes from their starting location. The column on the right compares with just using the box movement cost.

The results are quite illuminating in that we can see the trade off between efficiency and plan quality. With the action cost included, while the code can solve the simple problems fairly efficiently, the time to find a solution grows quite significantly as the difficult of the problem grows. This approach does actually find high quality solutions on par with FF and Conformant-FF.

On the other hand, we found significant speed up in computation time when ignoring the robot action cost with the trade off of longer plans. This makes conceptual sense as it's easy to realize that a method which ignores robot movement could take extra steps to reduce box movement. Furthermore, the expansion of the search space is reduced since the algorithm runs the box quickly to the goal, ignoring better possible paths. However, we still were not able to match FF's combination of efficiency and solution quality.

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.**
Our A* implementation keeps track of an open and closed set of expanded states. Each iteration we expand all possible states from the current state with the lowest cost from the open set. For all possible expansions we check to see if the same state configuration already exists in the closed set (that is, the set of states already expanded), if not it's added to the open set with it's cost. If an expanded state results in a goal condition, the algorithm returns the state and all actions taken to get to that state.
The process of searching the open set will check all feasible state configurations. The closed set guarantees that no state will be expanded multiple times. Therefore, assuming a bounded world which will not expand off into infinity, once the open set is exhausted the algorithm will terminate. This shows that the algorithm is complete since it is guaranteed to exhauste the open set or find a solution given a bounded world. Additionally, since the search will expand all possible state configurations and they are chained by legal robot moves, when a goal state

is found in the search, it's guaranteed that a solution is found. The solution is recovered by backtracking up the search tree to the root goal.

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.**
As mentioned above, the use of different cost functions found a trade off between efficiency and solution quality. A bi-directional A* search, where two simultaneous searchs start from both the initial state and a goal state, would improve the planning speed. It's possible that a bi-directional A* with a cost function including action costs could improve the efficiency while maintaining the better solutions found in our slower results.

## Problem 4 - Post-Project: Towers of Hanoi Revisited

1. **Give successful plans from at least one planner with 6 and 10 disks.**
The solution for both the 6 disk and 10 disk problem were solved using FF. The 6 disk solution took 0.00 seconds to compute and was solved in 63 steps. The 10 disk solution took 0.36 seconds to compute and was solved in 1023 steps. The 6 disk solution is provided in the appendix. Both the 6 disk and 10 disk solution are attached in the .zip submission

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.**
The plans were very repetitive. For instance, the plan for 10 disks contains in some form a similar plan for 6 disks multiple times throughout the entire solution. This is because in order to move the next largest disk off of the pole, another pole needs to be completely empty. This scenario requires all smaller sized disks to be stacked on the third pole in the correct order. The most efficient way for this to occur is to reuse the solution already created. In general, in order to improve the logic of the plan it would be efficient to group these steps into sub plans that can be reexecuted quickly. There would need to be some reasoning to determine which pole to move to first, but after that the rest of the movements are predetermined.

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? +**
One possible strategy would be to create a planner that could self expand. For instance, it could state with a single disk tower of hanoi, moving from pole x to pole z. Then solve a 2 disk game moving a stacked tower on pole x to a stacked tower on pole z. This program would continually iterate and hopefully through some form of machine learning detect repeatedly grouped steps. For instance, the optimal steps for moving a 2 disk stack from pole x to pole y: move-stack(disk2,pole-x,pole-y) can be simplified to 3 substeps: move(disk1,pole.x,pole.z), move (disk2,pole.x,pole.y),move(disk1,pole.z,pole.y). The next move-stack action move-stack (disk3,pole.x,pole.y) would simply involve move-stack (disk2,pole.x,pole.z), move (disk3,pole.x,pole.y), move-stack (disk2,pole.z,pole.y). Hopefully this program would be able to detect other progressively complex tasks that require similar recursion methods.

# Summary

This section summarizes the contribution of each team member.

1. Matheus Svolenski
   - Implemented the parser class to properly extract the Sokoban world information.
   - Ran BlackBox and FF planners on the Towers of Hanoi problems.
2. Jarius Tillman
3. Andrew Melim
   - Wrote project build system
   - Implemented state class and expansion of states for Sokoban domain.
   - Code debugging and verification via unit testing for A* solver
   - Created PDDL description for problem 2
4. Daniel Pickem
   - Implemented the class structure of the planner from problem 3.
   - Implemented the world class for the planner that describes and stores the environment and Sokoban map.
   - Implemented the A* algorithm.
   - Created PDDL description for problem 2.
   - Ran BlackBox and FF planners on the Towers of Hanoi problem 1 and problem 4.
   - Wrote Matlab tool to autogenerate Tower of Hanoi PDDL description.
   - Wrote Matlab tool to autogenerate Sokoban PDDL description.

## References

[1] Russell, Stuart J. and Norvig, Peter, *Artificial Intelligence: A Modern Approach*, Pearson Education, 2010

[2] BlackBox Planner, *http://www.cs.rochester.edu/ kautz/satplan/blackbox/#super*

[3] Fast Forward Planner, *http://fai.cs.uni-saarland.de/hoffmann/ff.html*

## Problem 2 - Project Part I: Sokoban PDDL - Successful Plans

All generated plans can be found in the attached .zip file, specifically in the results folder. The relatively short plans for problem 2.1 and problem 2.2 are shown below.

- Problem 2.1 (*sokoban_problem_2.1.txt*):
    1. : MOVE R S-3-3 S-3-4 UP
    2. : MOVE R S-3-4 S-4-4 RIGHT
    3. : MOVE R S-4-4 S-5-4 RIGHT
    4. : MOVE R S-5-4 S-5-5 UP
    5. : PUSH R B1 S-5-5 S-4-5 S-3-5 LEFT
    6. : MOVE R S-4-5 S-4-4 DOWN
    7. : MOVE R S-4-4 S-3-4 LEFT
    8. : MOVE R S-3-4 S-2-4 LEFT
    9. : MOVE R S-2-4 S-2-5 UP
    10. : MOVE R S-2-5 S-2-6 UP
    11. : MOVE R S-2-6 S-3-6 RIGHT
    12. : PUSH R B1 S-3-6 S-3-5 S-3-4 DOWN
    13. : PUSH R B1 S-3-5 S-3-4 S-3-3 DOWN
    14. : PUSH R B1 S-3-4 S-3-3 S-3-2 DOWN
- Problem 2.2 (*sokoban_problem_2.2.txt*):
    1. : MOVE R S-2-3 S-2-2 DOWN
    2. : MOVE R S-2-2 S-3-2 RIGHT
    3. : MOVE R S-3-2 S-4-2 RIGHT
    4. : PUSH R B2 S-4-2 S-4-3 S-4-4 UP
    5. : MOVE R S-4-3 S-4-2 DOWN
    6. : MOVE R S-4-2 S-3-2 LEFT
    7. : MOVE R S-3-2 S-2-2 LEFT
    8. : MOVE R S-2-2 S-2-3 UP
    9. : PUSH R B1 S-2-3 S-3-3 S-4-3 RIGHT
    10. : MOVE R S-3-3 S-3-4 UP
    11. : PUSH R B2 S-3-4 S-4-4 S-5-4 RIGHT
    12. : MOVE R S-4-4 S-3-4 LEFT

13. : MOVE R S-3-4 S-3-3 DOWN
14. : MOVE R S-3-3 S-3-2 DOWN
15. : MOVE R S-3-2 S-4-2 RIGHT
16. : PUSH R B1 S-4-2 S-4-3 S-4-4 UP
17. : MOVE R S-4-3 S-5-3 RIGHT
18. : PUSH R B2 S-5-3 S-5-4 S-5-5 UP
19. : PUSH R B2 S-5-4 S-5-5 S-5-6 UP
20. : PUSH R B2 S-5-5 S-5-6 S-5-7 UP
21. : MOVE R S-5-6 S-5-5 DOWN
22. : MOVE R S-5-5 S-4-5 LEFT
23. : PUSH R B1 S-4-5 S-4-4 S-4-3 DOWN
24. : MOVE R S-4-4 S-5-4 RIGHT
25. : MOVE R S-5-4 S-5-3 DOWN
26. : PUSH R B1 S-5-3 S-4-3 S-3-3 LEFT
27. : MOVE R S-4-3 S-4-4 UP
28. : MOVE R S-4-4 S-3-4 LEFT
29. : PUSH R B1 S-3-4 S-3-3 S-3-2 DOWN
30. : MOVE R S-3-3 S-4-3 RIGHT
31. : MOVE R S-4-3 S-4-2 DOWN
32. : PUSH R B1 S-4-2 S-3-2 S-2-2 LEFT

- Problem 2.3: See attached .zip file (*sokoban_problem_2.3.txt*).
- Problem 2.4 Challenge: See attached .zip file (*sokoban_problem_challenge.txt*).

## Problem 4 - Post-Project: Towers of Hanoi Revisited

- Problem 4.1.a: 6 Disks Using FF

1. MOVE-DISK D1 D2 P2
2. MOVE-DISK D2 D3 P1
3. MOVE-DISK D1 P2 D2
4. MOVE-DISK D3 D4 P2
5. MOVE-DISK D1 D2 D4
6. MOVE-DISK D2 P1 D3

7. MOVE-DISK D1 D4 D2
8. MOVE-DISK D4 D5 P1
9. MOVE-DISK D1 D2 D4
10. MOVE-DISK D2 D3 D5
11. MOVE-DISK D1 D4 D2
12. MOVE-DISK D3 P2 D4
13. MOVE-DISK D1 D2 P2
14. MOVE-DISK D2 D5 D3
15. MOVE-DISK D1 P2 D2
16. MOVE-DISK D5 D6 P2
17. MOVE-DISK D1 D2 D6
18. MOVE-DISK D2 D3 D5
19. MOVE-DISK D1 D6 D2
20. MOVE-DISK D3 D4 D6
21. MOVE-DISK D1 D2 D4
22. MOVE-DISK D2 D5 D3
23. MOVE-DISK D1 D4 D2
24. MOVE-DISK D4 P1 D5
25. MOVE-DISK D1 D2 D4
26. MOVE-DISK D2 D3 P1
27. MOVE-DISK D1 D4 D2
28. MOVE-DISK D3 D6 D4
29. MOVE-DISK D1 D2 D6
30. MOVE-DISK D2 P1 D3
31. MOVE-DISK D1 D6 D2
32. MOVE-DISK D6 P3 P1
33. MOVE-DISK D1 D2 D6
34. MOVE-DISK D2 D3 P3
35. MOVE-DISK D1 D6 D2
36. MOVE-DISK D3 D4 D6
37. MOVE-DISK D1 D2 D4
38. MOVE-DISK D2 P3 D3
39. MOVE-DISK D1 D4 D2

40. MOVE-DISK D4 D5 P3
41. MOVE-DISK D1 D2 D4
42. MOVE-DISK D2 D3 D5
43. MOVE-DISK D1 D4 D2
44. MOVE-DISK D3 D6 D4
45. MOVE-DISK D1 D2 D6
46. MOVE-DISK D2 D5 D3
47. MOVE-DISK D1 D6 D2
48. MOVE-DISK D5 P2 D6
49. MOVE-DISK D1 D2 P2
50. MOVE-DISK D2 D3 D5
51. MOVE-DISK D1 P2 D2
52. MOVE-DISK D3 D4 P2
53. MOVE-DISK D1 D2 D4
54. MOVE-DISK D2 D5 D3
55. MOVE-DISK D1 D4 D2
56. MOVE-DISK D4 P3 D5
57. MOVE-DISK D1 D2 D4
58. MOVE-DISK D2 D3 P3
59. MOVE-DISK D1 D4 D2
60. MOVE-DISK D3 P2 D4
61. MOVE-DISK D1 D2 P2
62. MOVE-DISK D2 P3 D3
63. MOVE-DISK D1 P2 D2