Artificial Intelligence Nanodegree Project - Project 3: Air Cargo Planning Analysis

Summary

This article presents an analysis through several planning search implementations to solve the Air Cargo Transport System, a project in Artifical Intelligence Nanodegree performed by Udacity.

For more details about the project and source code, please visit https://github.com/rafaelrpaiva/AIND-Planning (https://github.com/rafaelrpaiva/AIND-Planning)

Optimal Plans for Given Problems

The project presents 3 planning problems in the Air Cargo domain that use the same **action schema**:

```
Action(Load(c, p, a),
    PRECOND: At(c, a) ^ At(p, a) ^ Cargo(c) ^ Plane(p) ^ Airport(a)
    EFFECT: ¬ At(c, a) ^ In(c, p))
Action(Unload(c, p, a),
    PRECOND: In(c, p) ^ At(p, a) ^ Cargo(c) ^ Plane(p) ^ Airport(a)
    EFFECT: At(c, a) ^ ¬ In(c, p))
Action(Fly(p, from, to),
    PRECOND: At(p, from) ^ Plane(p) ^ Airport(from) ^ Airport(to)
    EFFECT: ¬ At(p, from) ^ At(p, to))
```

Problem 1

Problem 1 works with two cargoes (C1, C2), two planes (P1, P2) and two airports (SFO, JFK). The initial state and goal are presented below:

```
Init(At(C1, SFO) ^ At(C2, JFK)
    ^ At(P1, SFO) ^ At(P2, JFK)
    ^ Cargo(C1) ^ Cargo(C2)
    ^ Plane(P1) ^ Plane(P2)
    ^ Airport(JFK) ^ Airport(SFO))
Goal(At(C1, JFK) ^ At(C2, SFO))
```

This problem is solved with an optimal plan of **6 actions**. One of the many options is presented below:

```
Load(C1, P1, SFO)
Load(C2, P2, JFK)
Fly(P1, SFO, JFK)
Fly(P2, JFK, SFO)
Unload(C1, P1, JFK)
Unload(C2, P2, SFO)
```

Problem 2

Problem 2 works with three cargoes (C1, C2, C3), three planes (P1, P2, P3) and three airports (SFO, JFK, ATL). The initial state and goal are presented below:

```
Init(At(C1, SFO) ^ At(C2, JFK) ^ At(C3, ATL)
    ^ At(P1, SFO) ^ At(P2, JFK) ^ At(P3, ATL)
    ^ Cargo(C1) ^ Cargo(C2) ^ Cargo(C3)
    ^ Plane(P1) ^ Plane(P2) ^ Plane(P3)
    ^ Airport(JFK) ^ Airport(SFO) ^ Airport(ATL))
Goal(At(C1, JFK) ^ At(C2, SFO) ^ At(C3, SFO))
```

This problem is solved with an optimal plan of **9 actions**. There are some different configurations to solve the problem, and one of the many options is presented below:

```
Load(C1, P1, SFO)
Load(C2, P2, JFK)
Load(C3, P3, ATL)
Fly(P1, SFO, JFK)
Fly(P2, JFK, SFO)
Fly(P3, ATL, SFO)
Unload(C3, P3, SFO)
Unload(C1, P1, JFK)
Unload(C2, P2, SFO)
```

Problem 3

Finally, Problem 3 works with four cargoes (C1, C2, C3, C4), four planes (P1, P2, P3, P4) and four airports (SFO, JFK, ATL, ORD). The initial state and goal are presented below:

```
Init(At(C1, SFO) ^ At(C2, JFK) ^ At(C3, ATL) ^ At(C4, ORD)
    ^ At(P1, SFO) ^ At(P2, JFK)
    ^ Cargo(C1) ^ Cargo(C2) ^ Cargo(C3) ^ Cargo(C4)
    ^ Plane(P1) ^ Plane(P2)
    ^ Airport(JFK) ^ Airport(SFO) ^ Airport(ATL) ^ Airport(ORD))
Goal(At(C1, JFK) ^ At(C3, JFK) ^ At(C2, SFO) ^ At(C4, SFO))
```

This problem is solved with an optimal plan of **12 actions**. There are some different configurations to solve the problem, and one of the many options is presented below:

```
Load(C1, P1, SFO)
Load(C2, P2, JFK)
Fly(P1, SFO, ATL)
Load(C3, P1, ATL)
Fly(P2, JFK, ORD)
Load(C4, P2, ORD)
Fly(P2, ORD, SFO)
Fly(P1, ATL, JFK)
Unload(C4, P2, SFO)
Unload(C3, P1, JFK)
Unload(C1, P1, JFK)
Unload(C2, P2, SFO)
```

Analysis of Non-Heuristics Search

The first block of problem, using uninformed search algorithms, since they don't work with additional information about the states. The algorithms used were: - `1. breadth_first_search` - `3. depth_first_graph_search` - `5. uniform_cost_search` - `7. greedy_best_first_graph_search h_1`

To compare the results between these algorithms, we used the outputs created by `run_search.py` execution, which reports execution time (in seconds), memory usage (in terms of nodes) and the path length:

Problem 1

Search Algorithm	Path Length	Execution Time (s)	Node Expansions
(1) Breadth First Search	**6**	0.03436s	43
(3) Depth First Graph Search	12	0.00917s	12
(5) Uniform Cost Search	**6**	0.04061s	55
(7) Greedy Best First Search	**6**	**0.00589s**	**7**

Problem 2

Search Algorithm	Path Length	Execution Time (s)	Node Expansions
(1) Breadth First Search	**9**	19.71453s	3343
(3) Depth First Graph Search	575	4.73562s	**582**
(5) Uniform Cost Search	**9**	18.37634s	4852
(7) Greedy Best First Search	17	**3.59012s**	990

Problem 3

Search Algorithm	Path Length	Execution Time (s)	Node Expansions
(1) Breadth First Search	**12**	157.00695s	14663
(3) Depth First Graph Search	596	**5.95305s**	**627**
(5) Uniform Cost Search	**12**	83.44421s	18235
(7) Greedy Best First Search	22	23.61275s	5614

Analysis of Uninformed Search Algorithms

Analysing the results from the three problems presented, we can say that **Breadth First Search** and **Uniform Cost Search** always solved with an optimal action plan. On the other hand, **Depth First Graph Search** never finds an optimal plan but is consistently faster and least-memory consumption, and thus can be a good solution to find quickly a solution, which can be useful in some situations (such as validation of a result).

Which one is best? If finding the optimal path is mandatory, **Breadth First Search** and **Uniform Cost Search** are good options: the three are very close in execution time and the first one is 20-30% better in node expansions. If not mandatory, it’s good to highlight that **Greedy Best First** algorithm, although not finding the optimal plan in Problems 2 and 3, is not that far from the optimal solution as **Depth First Graph Search**, but runs in a better execution time and memory consumption than the 3 strategies for the optimal plan and can be used for some situations where a bigger problem, with more variables, tends to expand memory consumption.

Analysis of Heuristics Search

Running with heuristic search is expected to find solutions more efficiently than previous strategies. about the states. The algorithms used were: - `8. astar_search h_1` - `9. astar_search h_ignore_preconditions` - `10. astar_search h_pg_levelsum`

To compare the results between these algorithms, we used the outputs created by `run_search.py` execution, which reports execution time (in seconds), memory usage (in terms of nodes) and the path length:

Problem 1

Search Algorithm	Path Length	Execution Time (s)	Node Expansions
(8) A* Search H1 Heuristic	**6**	0.04497s	55
(9) A* Search “Ignore Precond” Heuristic	**6**	**0.04449s**	41
(10) A* Search “Level Sum” Heuristic	**6**	1.52476s	**11**

Problem 2

Search Algorithm	Path Length	Execution Time (s)	Node Expansions
(8) A* Search H1 Heuristic	**9**	14.92789s	4852
(9) A* Search “Ignore Precond” Heuristic	**9**	**7.47146s**	1450
(10) A* Search “Level Sum” Heuristic	**9**	263.27003s	**86**

Problem 3

Search Algorithm	Path Length	Execution Time (s)	Node Expansions
(8) A* Search H1 Heuristic	**12**	76.36712s	18235
(9) A* Search “Ignore Precond” Heuristic	**12**	**26.46407s**	5040
(10) A* Search “Level Sum” Heuristic	**12**	1364.74363s	**318**

Analysis of Heuristic Search Algorithms

First of all, all heuristics found the optimal plan in all analysis (since all of them use an *h* (estimate to goal) function that doesn’t overestimate the distance), and in general they were faster than most of uninformed strategies.

The **A* Search h Ignoring Preconditions** uses the concept that every action becomes applicable in every state, and any single goal fluent can be achieved in one step (if there is an applicable action□□if not, the problem is impossible) [reference: AIMA - 10.2.3 - Heuristics for Planning]. That’s why the number os expansions is less than h1.

Finally, The **A* Search with Level Sum** uses the planning graph built and the sum of the level costs of the goals; this can be inadmissible but works well in practice for problems that are largely decomposable, as in this case [reference: AIMA - 10.3.1 - Planning Graphs for Heuristic Estimation]. This is why it is much more accurate than the other heuristics in terms of node expansions, but it suffers from execution time problems - the last execution didn’t attend the pre-requirements for a solution.

In general, **A* Search h Ignoring Preconditions** was the best heuristics: although consuming more memory than the “Level Sum” heuristics, this heuristic found all optimal plans in a very efficient time and with an average node expansions number. On the other hand, the most efficient in memory consuming was **A***

with Level Sum, occupying in the last example (Problem 3) 1 to 6% of memory from the other two heuristics. The inconvenient for this strategy is the very inefficient execution time (the last problem took near 23 minutes to finish), which can be prohibitive in certain scenarios.

Conclusions: Non-Heuristic vs. Heuristic Search Algorithms

Analysing the numbers above, and comparing only solutions the found the optimal plan, the heuristic **A* Search Ignoring Preconditions** was the winning strategy. Comparing with the most efficient in uninformed search (**Breadth First Search** and **Uniform Cost Search**), we can see better numbers in all 3 cases:

PROBLEM 1	Path Length	Execution Time (s)	Node Expansions
(1) Breadth First Search	**6**	0.03436s	43
(5) Uniform Cost Search	**6**	0.04061s	55
(9) A* Search “Ignore Precond” Heuristic	**6**	**0.04449s**	**41**

PROBLEM 2	Path Length	Execution Time (s)	Node Expansions
(1) Breadth First Search	**9**	19.71453s	3343
(5) Uniform Cost Search	**9**	18.37634s	4852
(9) A* Search “Ignore Precond” Heuristic	**9**	**7.47146s**	**1450**

PROBLEM 3	Path Length	Execution Time (s)	Node Expansions
(1) Breadth First Search	**12**	157.00695s	14663
(5) Uniform Cost Search	**12**	83.44421s	18235
(9) A* Search “Ignore Precond” Heuristic	**12**	**26.46407s**	**5040**

In all cases, the heuristic had the best performance in terms of speed and memory usage, enforcing the power of A* with a good *h* estimation function/heuristic.