Heuristic Analysis

Chyld Medford

Artificial Intelligence Nanodegree/May Cohort

Project III

Analysis

Optimal Plan

Problem 1:

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

Problem 2:

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

Problem 3:

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

Non-Heuristic Search Comparison

Breadth First, Depth First, Uniform Cost

In these metrics (Expansion, Goal Tests, New Nodes, Execution Time), Depth First was first, followed by Breadth First and last Uniform Cost. This was true except for one metric - Plan Length. It turns out that Depth First Search is fast and efficient, but its plan is not that good. So, if you're looking to compute a non-optimal plan quickly, then Depth First is a good choice. However, if you're needing an optimal plan, then choose Breadth First.

Heuristic Search Comparison

A* with Ignore Preconditions and Levelsum

In these metrics (Expansion, Goal Tests, New Nodes), A* with Levelsum was superior. However A* with Ignore Preconditions surpassed in Execution Time and tied with Plan Length.

Levelsum is a better, more knowledgeable algorithm than Ignore Preconditions. But although it is better and more efficient, it's also slower to make decisions because it takes a lot more time to compute than Ignore Preconditions, which is relatively simple.

Conclusion

Even though A* with Levelsum would appear to be the best overall search algorithm, I'd have to give the winner to A* with Ignore Preconditions. It's not as efficient as Levelsum, but it does generally point in the right direction and because it's a simple algorithm, it's quick to compute.

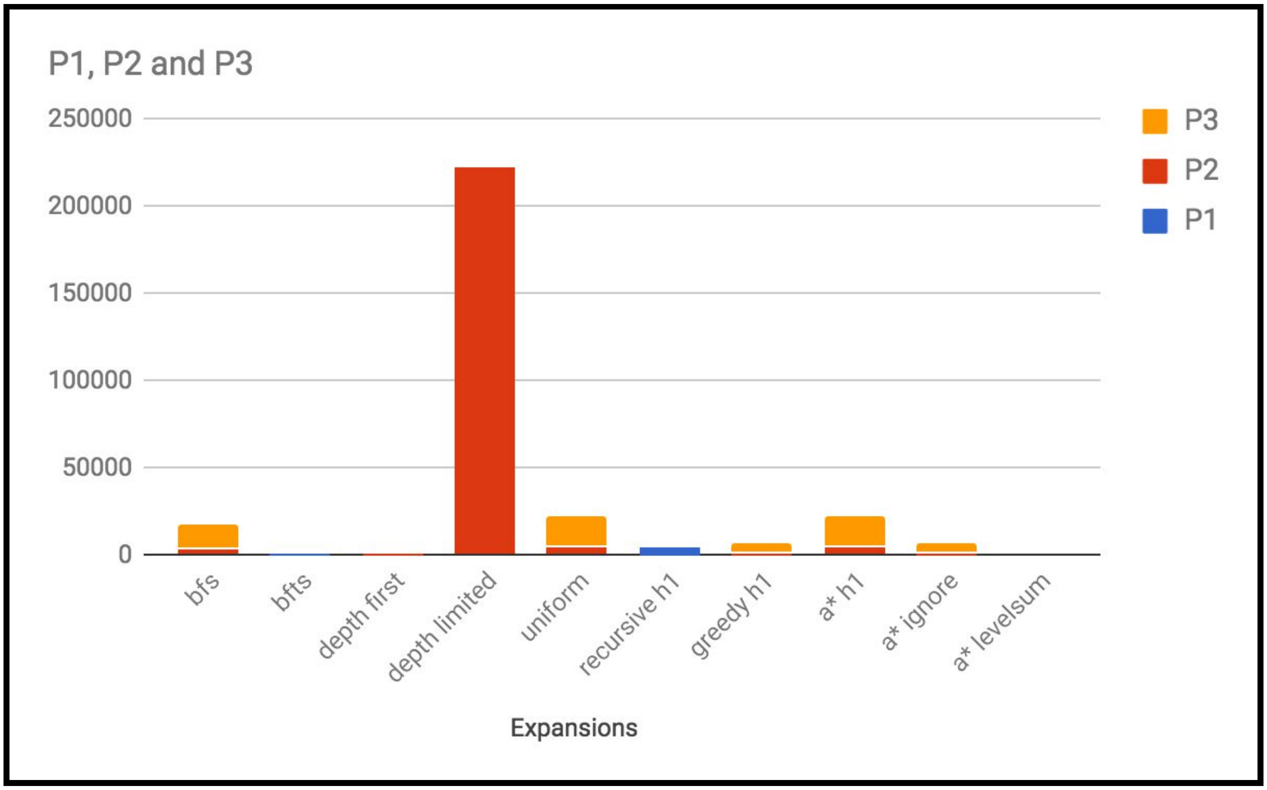
A* with Ignore Preconditions is the winner!

Collected Data

Expansions

Expansions	**P1**	**P2**	**P3**
bfs	43	3343	14663
bfts	1458		
depth first	21	624	408
depth limited	101	222719	
uniform	55	4853	18223

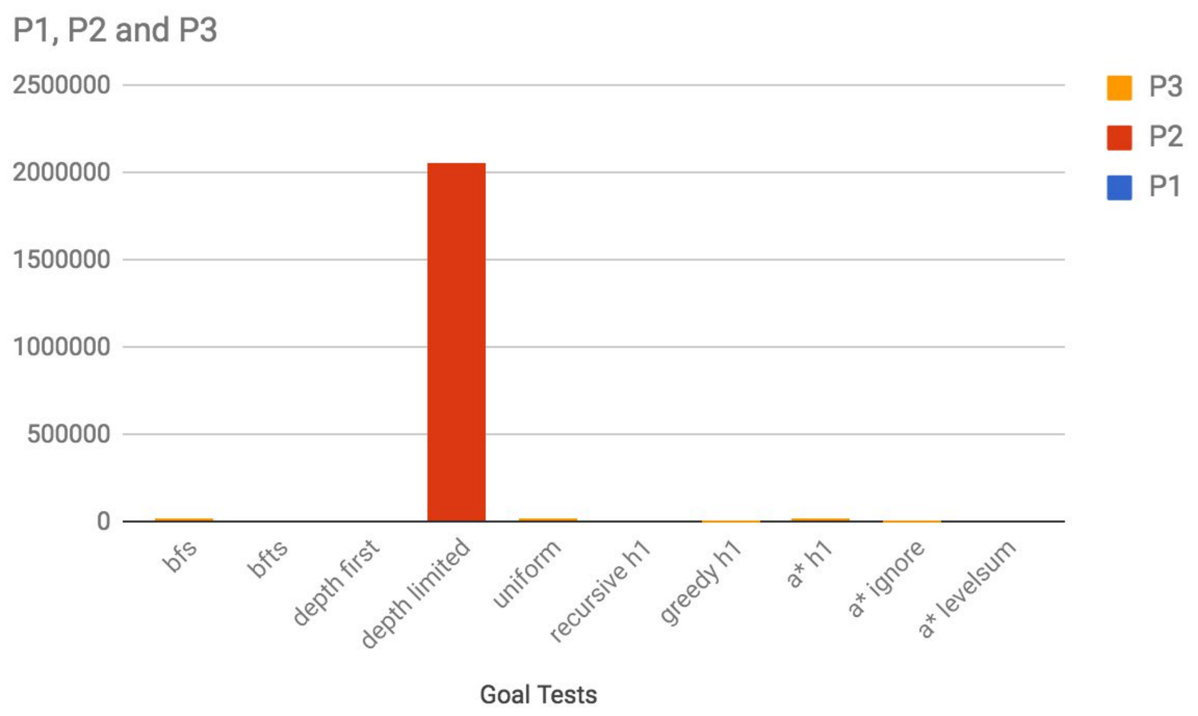
recursive h1	4229		
greedy h1	7	998	5578
a h1*	55	4853	18223
a ignore*	41	1450	5040
a levelsum*	11	86	325



Goal Tests

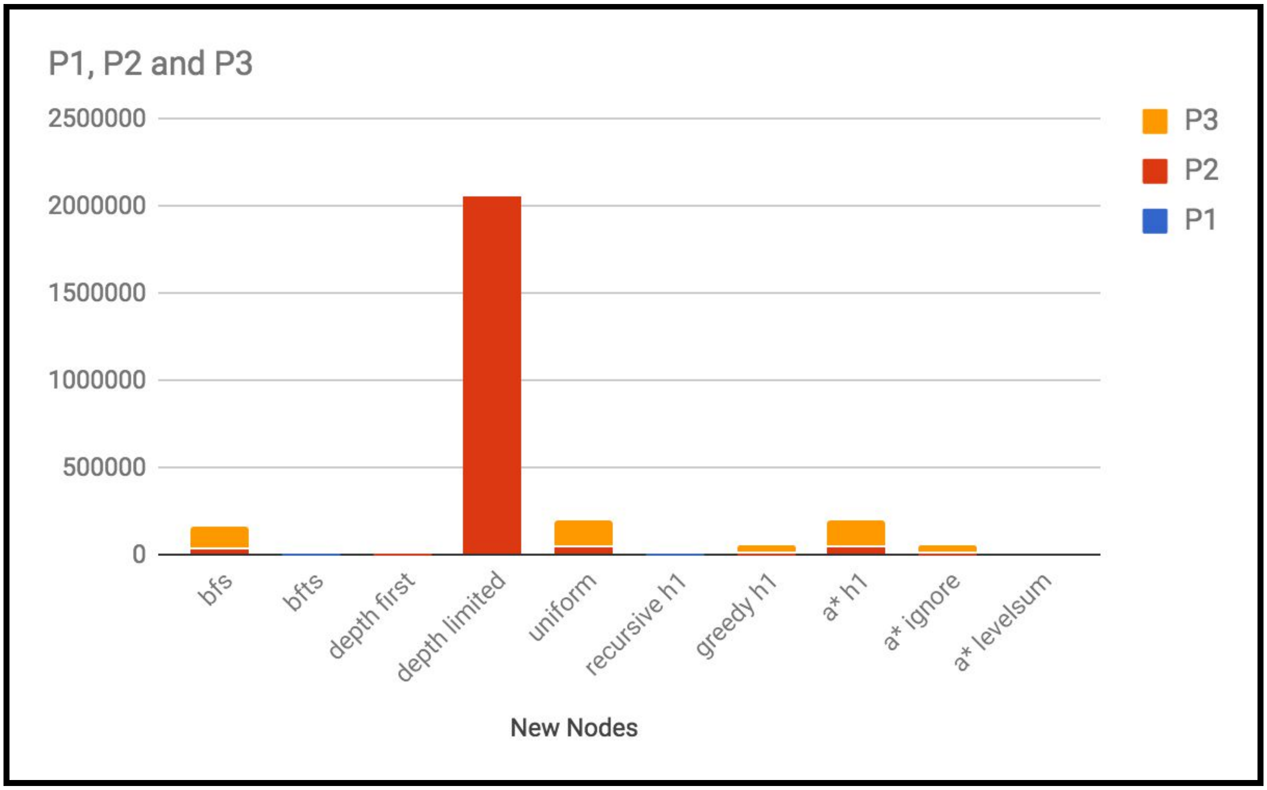
Goal Tests	**P1**	**P2**	**P3**
bfs	56	4609	18098
bfts	1459		
depth first	22	625	409
depth limited	271	2053741	
uniform	57	4855	18225
recursive h1	4230		
greedy h1	9	1000	5580
a h1*	57	4855	18225

a ignore*	43	1452	5042
a levelsum*	13	88	327



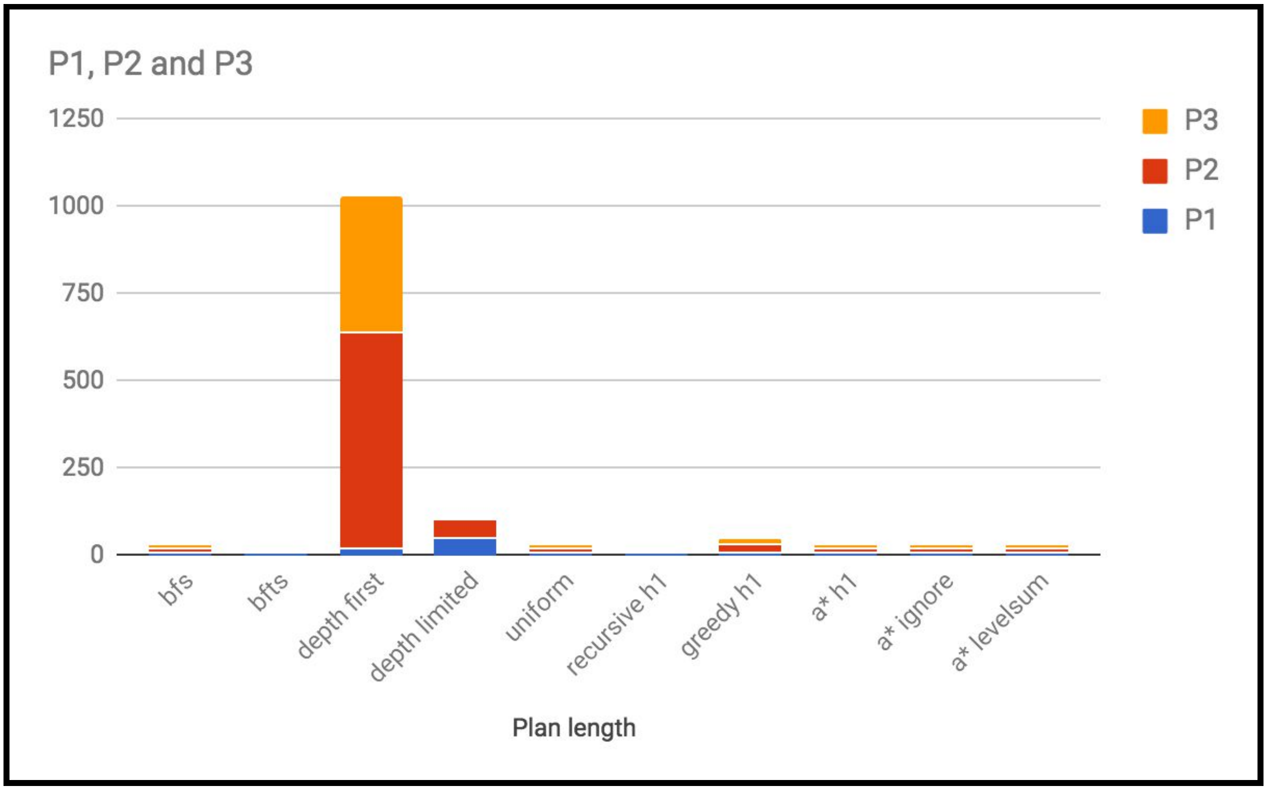
New Nodes

New Nodes	**P1**	**P2**	**P3**
bfs	180	30509	129631
bfts	5960		
depth first	84	5602	3364
depth limited	414	2054119	
uniform	224	44041	159618
recursive h1	17023		
greedy h1	28	8982	49150
a h1*	224	44041	159618
a ignore*	170	13303	44944
a levelsum*	50	841	3002



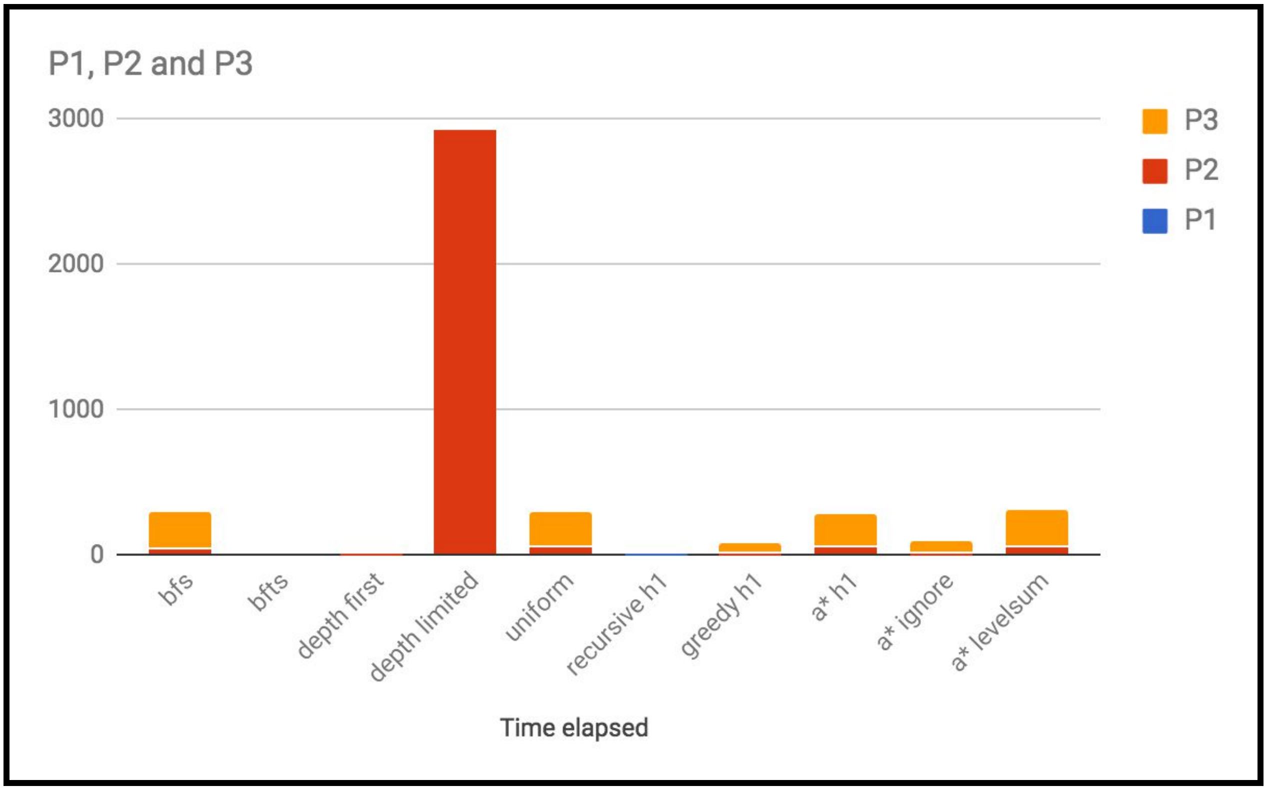
Plan Length

Plan length	**P1**	**P2**	**P3**
bfs	6	9	12
bfts	6		
depth first	20	619	392
depth limited	50	50	
uniform	6	9	12
recursive h1	6		
greedy h1	6	21	22
a h1*	6	9	12
a ignore*	6	9	12
a levelsum*	6	9	12

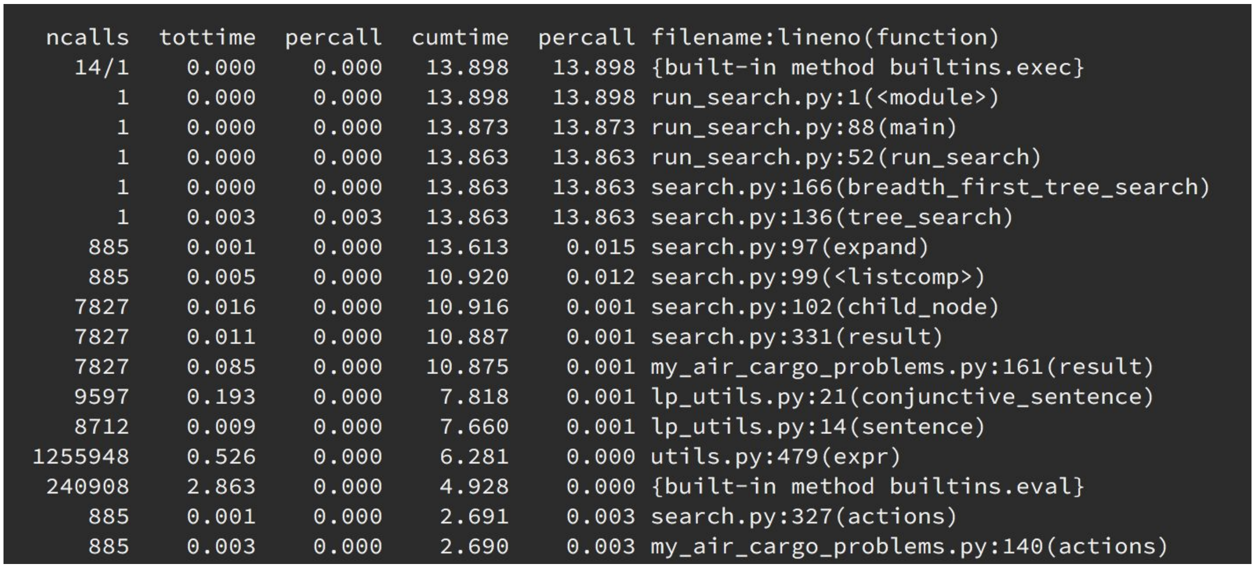


Time Elapsed

Time elapsed	**P1**	**P2**	**P3**
bfs	0.115329453	45.78239207	252.9254127
bfts	3.817701466		
depth first	0.053629791	8.433301724	5.50049988
depth limited	0.290445365	2930.118616	
uniform	0.145987641	54.6851305	237.7268489
recursive h1	10.98767507		
greedy h1	0.018589913	10.95879188	72.11773767
a h1*	0.152986179	53.58713161	234.3621965
a ignore*	0.13089709	19.08355863	81.87008337
a levelsum*	0.572419832	49.37726242	263.8445514



Profiler



```
 ncalls  tottime  percall  cumtime  percall filename:lineno(function)
   14/1    0.000    0.000   13.898   13.898 {built-in method builtins.exec}
      1    0.000    0.000   13.898   13.898 run_search.py:1(<module>)
      1    0.000    0.000   13.873   13.873 run_search.py:88(main)
      1    0.000    0.000   13.863   13.863 run_search.py:52(run_search)
      1    0.000    0.000   13.863   13.863 search.py:166(breadth_first_tree_search)
      1    0.003    0.003   13.863   13.863 search.py:136(tree_search)
    885    0.001    0.000   13.613    0.015 search.py:97(expand)
    885    0.005    0.000   10.920    0.012 search.py:99(<listcomp>)
   7827    0.016    0.000   10.916    0.001 search.py:102(child_node)
   7827    0.011    0.000   10.887    0.001 search.py:331(result)
   7827    0.085    0.000   10.875    0.001 my_air_cargo_problems.py:161(result)
   9597    0.193    0.000    7.818    0.001 lp_utils.py:21(conjunctive_sentence)
   8712    0.009    0.000    7.660    0.001 lp_utils.py:14(sentence)
1255948    0.526    0.000    6.281    0.000 utils.py:479(expr)
 240908    2.863    0.000    4.928    0.000 {built-in method builtins.eval}
    885    0.001    0.000    2.691    0.003 search.py:327(actions)
    885    0.003    0.000    2.690    0.003 my_air_cargo_problems.py:140(actions)
```

Due to the process taking hours to complete, I couldn't run Breadth First Tree Search and Recursive Best First Search (H1) for problems 2 and 3. Also couldn't run Depth Limited Search for Problem 3.

I profiled the execution of the program, above, to determine what the constraining factors were as it related to running time. I focused in on, specifically, the code that I wrote vs what was provided as part of the material.

As can be seen from the above image, the “result” and “actions” functions were - from my code’s perspective - taking some time to run. However, I looked at the code and it’s very minimal with no overhead. Therefore I didn’t notice any obvious bottlenecks with these particular functions.