Uninformed search
Four search methods were explored: breadth_first_search, breadth_first_tree_search, depth_first_graph_search, uniform_cost_search
It has been considered that optimal solution(s) are those having the shortest plan length in the options.
Tests details are shown in the table below:

Problem	Search	Expansions	Goal_tests	New nodes	time (s)	plan length	Optimal
P1	breadth_first_search.	43	56	180	0.03564195113	6	**Yes**
P1	breadth_first_tree_search...	1458	1459	5960	1.067126252	6	**Yes**
P1	depth_first_graph_search...	12	13	**48**	**0.01018714131**	12	No
P1	uniform_cost_search...	55	57	224	0.04641878733	6	**Yes**
P2	breadth_first_search.	3343	4609	30509	17.0811178	9	**Yes**
P2	breadth_first_tree_search...	died					No
P2	depth_first_graph_search...	582	583	**5211**	**6.166585127**	575	No
P2	uniform_cost_search...	4852	4854	44030	13.59925451	9	**Yes**
P3	breadth_first_search.	14663	18098	129631	120.9480345	12	**Yes**
P3	breadth_first_tree_search...	died					No
P3	depth_first_graph_search...	627	628	**5176**	**4.438608638**	596	No
P3	uniform_cost_search...	18235	18237	18237	67.79124268	12	**Yes**

Optimal sequence of actions for each problem:

Problem 1:
Load(C2, P2, JFK)
Load(C1, P1, SFO)
Fly(P2, JFK, SFO)
Unload(C2, P2, SFO)
Fly(P1, SFO, JFK)
Unload(C1, P1, JFK)

Problem 2:
Load(C2, P2, JFK)
Load(C1, P1, SFO)
Load(C3, P3, ATL)
Fly(P2, JFK, SFO)
Unload(C2, P2, SFO)
Fly(P1, SFO, JFK)
Unload(C1, P1, JFK)
Fly(P3, ATL, SFO)
Unload(C3, P3, SFO)

Problem 3:
Load(C2, P2, JFK)
Load(C1, P1, SFO)
Fly(P2, JFK, ORD)
Load(C4, P2, ORD)
Fly(P1, SFO, ATL)
Load(C3, P1, ATL)
Fly(P1, ATL, JFK)
Unload(C1, P1, JFK)
Unload(C3, P1, JFK)
Fly(P2, ORD, SFO)
Unload(C2, P2, SFO)
Unload(C4, P2, SFO)

From the results above we can see that breadth_first_tree_search (BFS) and uniform_cost_search (UCS) have consistently obtained the optimal plan in uninformed searches.
In terms of speed, Depth First Search (DFS) is by far the fastest and the difference with BFS and UCS becomes more evident.
Memory wise, DFS again is the best solution compared to the other 3.
On the other hand, breadth_first_tree_search has an unworkable space expansion (quantified as 'New nodes'), as a consequence Problem 1 expanded to 5960 nodes (compared to 48 of DFS). Problems 2 and 3 were aborted, elapsed over 30 mins.
Overall BFS appears to be the best search strategy among those tested, it is optimal with very similar execution time and slightly better memory consumption than UCS. DFS and BFTS are not optimal and are clearly not competitors to BFS in uninformed search for this problems.

Informed search
Two heuristics were implemented in this section: ignored preconditions and level_sum. They were tested with a* search on the 3 problems.

Problem	Search	Expan sions	Goal_tests	New nodes	time (s)	plan length
P1	a* with ignore_preconditions	41	43	170	**0.04440838851**	6
P1	a* with h_pg_levelsum	**11**	**13**	**50**	4.416750415	6
P2	a* with ignore_preconditions	1450	1452	13303	**9.564189926**	9
P2	a* with h_pg_levelsum	86	88	841	**450.2093706**	9
P3	a* with ignore_preconditions	5040	5042	44944	**35.71356842**	12
P3	a* with h_pg_levelsum	**318**	**320**	**2934**	3134.868269	12

Looking at the above results there are no differences when looking for optimality differences between these 2 heuristics.
They are both optimal but level_sum finds the same plan using a lot less resources:
Level_sum consumes less memory, by expanding up to 94% less than ignore preconditions.
This is logical since ignore preconditions is a good simplistic estimate that comes with a high memory cost (great amount of node expansions)

In problem 1, BFS being the better performer of uninformed heuristics presents a much smaller execution time than h_pg_levelsum but the later consumes a lot less memory to get to optimal path.
Problem 2 presents a similar situation when comparing BFS and h_pg_levelsum both achieve optimality with a plan length of 9. Even though, h_pg_levelsum
takes a higher execution time (450s. over 17s.) the number of nodes expanded are smaller than BFS by 2 orders of magnitude.
Having Problem 3 a higher branching factor (one more cargo and airport to branch on) the memory usage is much shorter, but execution time is so large that could
become unworkable for real life use, applications consuming this as a backend service could time-out.

Why did we chose BFS for uninformed search?
As explained above even though DFS presented great performance in memory and speed of execution compared to all other uninformed heuristics it was not chosen as the preferred one, BFS.
The reason is that DFS is great at vertically traversing thru its children to quickly find ***a*** solution, not the best solution, since it will stop when the set of literals satisfies the goal but will clearly leave other branches unexplored that might (and in this case have) better options to reach the goal. At the expense of longer time and more nodes explored BFS and UCS find the optimal solution even in the absence of heuristics (Lesson8: Search, slide 23).

Why do we chose h_levelsum in Informed Search?

To get the best of forward search using concrete space we need to define really good heuristics. Also, these heuristics have to be possible to automate to test as many options as possible. In this problem domain, the fact we have the problem encoded in the logical form based on propositional logic allows a program to ‘create’ heuristics by relaxing the problem (Lesson 9: Implement a planning search, slide 17).
On the other hand h_levelsum is generally inadmissible but works well if the problem definition is highly independent (S. Russell & P. Norvig, 2010). This is the case of all our 3 problems, have high subgoal independence (for instance, moving C1 to JFK to P1 does not impact moving C2 to the same airport in the same plane). Also, since we record a limited number of negative actions, it could make the overall level_sum optimistic in terms of ignoring the negative effects that a certain goal plan could have on the others.

More specifically, when we perform ignore preconditions at each level we are performing a large number of (unfiltered) actions that are simple to calculate, hence the large number of nodes and short time. In contrast h_levelsum chooses only the action nodes that comply with the preconditions (established as action parents) and follows thru each goal literal in the set list of states then adding the level cost for each one as the estimate. This is more constraint but at the same far more nested operation and consequently longer than ignore preconditions calculations.

If longer execution time is not a deal-breaker level_sum does a better job than ignore preconditions and no heuristic at all.