# Heuristic Analysis for an Air Cargo Problem

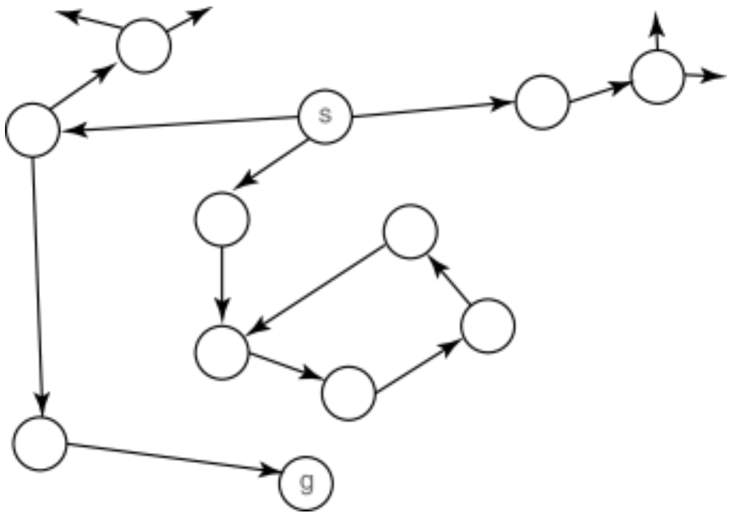
## Problem Scenario

The present project consist on an Air Cargo transport system, where several cargo wants to be moved from one city to another city's airport, having several planes to achieve it.

To do so, we implement a planning search agent to solve the problem with different approaches.

In a regular search algorithm, like for the adversarial search in the Isolation game, the problem-solving agent deals with atomic representation of states and thus needs good domain-specific heuristics to perform well. With first-order-logic we can build domain-independent heuristics based on the logical structure of the problem.

To measure the performance, we first run the already studied **uninformed non-heuristic search algorithms**. The caractheristics that makes this algorithms uninformed, is the fact that they do not have any information about the states beyond that the one provided in the problem definition. Therefore, they can only '*ask*' if each new state that the algorithm creates is the goal or not, and act in consequence by stopping if it is the goal, or creating new states if it is not.




An automatic **domain-independent heristics with A*** that searches on top o a planning graph has been created to compare the search efficiency agains the previously explained methods.

#### STATES AND ACTION SCHEMA

Planning Domain Definition Language - PDDL - allows to performe a **factored representation** of the world in which a state is represented by a collection of variables. This way, 4 things needs to be defined in the search problem:

- The initial state
- The actions that are available in the state
- The result of applying the action
- The goal state

The actions are described by a set of **Action schemas** that implicitly define the `ACTIONS(s)` and `RESULT(s,a)` functions needed to do a problem-solving search. Actions schemas are a lifted (from propositional logic to first-order-logic) representation that describes an action based on the preconditions for that action to occur, and the effects that action will produce. The action-schema for our current problem are the action of loading/unloading a cargo into a plane, and the plane to fly:

```
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))
```

As we said, we also need the Initial and Goal state for each of the 3 different problems proposed by Udacity, increasing the complexity:

#### ***Problem 1: 2 cargos, 2 cities, 2 planes***

```
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))
```

#### ***Problem 2: 3 cargos, 3 cities, 3 planes***

```
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))
```

#### ***Problem 3: 4 cargos, 4 cities, 4 planes***

```
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))
```

# Search Results

In this link (https://en.wikiversity.org/wiki/Search_techniques) there is a complete description of the search algorithm that had been used in this problem. The search algorithms code has been taken from the well known book: "*Artificial Intelligence, a modern approach*". The search algorithms can be divided into two subgroups:

### *Uninformed Search*

Uninformed search, also called blind search, is a class of general purpose search algorithms that operate in a brute-force way. These algorithms can be applied to a variety of search problems, but since they don't take into account the target problem.

### *Informed Search*

If information is available about the problem this could guide the search. Information is put in an evaluation function f(n) to be able to give a value to each state. Sometimes a heuristic function h(n) is used to guess the value if the information isn't perfect.

To run our search algorithms on the different problems we have to run the file run_search.py to which we can pass the parameteres included in that file as:

```
PROBLEMS = [["Air Cargo Problem 1", air_cargo_p1],
            ["Air Cargo Problem 2", air_cargo_p2],
            ["Air Cargo Problem 3", air_cargo_p3]]
SEARCHES = [["breadth_first_search", breadth_first_search, ""],
            ['breadth_first_tree_search', breadth_first_tree_search, ""
],
            ['depth_first_graph_search', depth_first_graph_search, ""],
            ['depth_limited_search', depth_limited_search, ""],
            ['uniform_cost_search', uniform_cost_search, ""],
            ['recursive_best_first_search', recursive_best_first_search
, 'h_1'],
            ['greedy_best_first_graph_search', greedy_best_first_graph_
search, 'h_1'],
            ['astar_search', astar_search, 'h_1'],
            ['astar_search', astar_search, 'h_ignore_preconditions'],
            ['astar_search', astar_search, 'h_pg_levelsum'],
            ]
```

## Uninformed Search

```
# Uninformed Search algorithms
def tree_search(problem, frontier):
    frontier.append(Node(problem.initial))
    while frontier:
        node = frontier.pop()
        if problem.goal_test(node.state):
            return node
        frontier.extend(node.expand(problem))
    return None


def graph_search(problem, frontier):
    frontier.append(Node(problem.initial))
    explored = set()
    while frontier:
        node = frontier.pop()
        if problem.goal_test(node.state):
            return node
        explored.add(node.state)
        frontier.extend(child for child in node.expand(problem)
                        if child.state not in explored and
                        child not in frontier)
    return None
```

The algorithms used following this approach are the following:

- Breadth First Search

```
def breadth_first_tree_search(problem):
  return tree_search(problem, FIFOQueue())
```

- Breadth First Tree Search

```
def breadth_first_search(problem):
  node = Node(problem.initial)
  if problem.goal_test(node.state):
      return node
  frontier = FIFOQueue()
  frontier.append(node)
  explored = set()
  while frontier:
      node = frontier.pop()
      explored.add(node.state)
      for child in node.expand(problem):
          if child.state not in explored and child not in fronti
er:
              if problem.goal_test(child.state):
                  return child
              frontier.append(child)
  return None
```

- Depth First Graph Search

```
def depth_first_graph_search(problem):
  return graph_search(problem, Stack())
```

- Depth Limited Search

```
def depth_limited_search(problem, limit=50):
  def recursive_dls(node, problem, limit):
      if problem.goal_test(node.state):
          return node
      elif limit == 0:
          return 'cutoff'
      else:
          cutoff_occurred = False
          for child in node.expand(problem):
              result = recursive_dls(child, problem, limit - 1)
              if result == 'cutoff':
                  cutoff_occurred = True
              elif result is not None:
                  return result
          return 'cutoff' if cutoff_occurred else None
  return recursive_dls(Node(problem.initial), problem, limit)
```

- Uniform Cost Search

```
def uniform_cost_search(problem):
  return best_first_graph_search(problem, lambda node: node.path
_cost)
```

- Recursive Best First Search

```
def recursive_best_first_search(problem, h=None):
  h = memoize(h or problem.h, 'h')

  def RBFS(problem, node, flimit):
      if problem.goal_test(node.state):
          return node, 0   # (The second value is immaterial)
      successors = node.expand(problem)
      if len(successors) == 0:
          return None, infinity
      for s in successors:
          s.f = max(s.path_cost + h(s), node.f)
      while True:
          # Order by lowest f value
          successors.sort(key=lambda x: x.f)
          best = successors[0]
          if best.f > flimit:
              return None, best.f
          if len(successors) > 1:
              alternative = successors[1].f
          else:
              alternative = infinity
          result, best.f = RBFS(problem, best, min(flimit, alter
native))
          if result is not None:
              return result, best.f

  node = Node(problem.initial)
  node.f = h(node)
  result, bestf = RBFS(problem, node, infinity)
  return result
```

- Greedy Best First Graph Search

```
def best_first_graph_search(problem, f):
  f = memoize(f, 'f')
  node = Node(problem.initial)
  if problem.goal_test(node.state):
      return node
  frontier = PriorityQueue(min, f)
  frontier.append(node)
  explored = set()
  while frontier:
      node = frontier.pop()
      if problem.goal_test(node.state):
          return node
      explored.add(node.state)
      for child in node.expand(problem):
          if child.state not in explored and child not in fronti
er:
              frontier.append(child)
          elif child in frontier:
              incumbent = frontier[child]
              if f(child) < f(incumbent):
                  # del frontier[incumbent]
                  frontier.append(child)
  return None
```

Based on Udacity advice somo of the algorithms were not run because of a long execution time:

- For poblem 2 Breadth Dirst Tree Search, Depth Limited Search and Recursive Best Search
- For problem 3: Breadth First Tree Search, Depth Limited Search, Uniform Cost Search, and Recursive Best First Search.

Results can be stored running the next commands:

```
python run_search.py -p 1 -s 1 2 3 4 5 6 7 >> problem1_uninformed.txt
python run_search.py -p 2 -s 1 3 5 7 >> problem2_uninformed.txt
python run_search.py -p 3 -s 1 3 5 7 >> problem3_uninformed.txt
```

***Problem 1***

| **Search Strategy** | **Optimal** | **Path Length** | **Execution Time (s)** | **Node Expansions** | **Goal Tests** | **New Nodes** |
|---|---|---|---|---|---|---|
| Breadth First Search | **Yes** | **6** | 0.034 | 43 | 56 | 180 |
| Breadth First Tree Search | **Yes** | **6** | 1.045 | 1458 | 1459 | 5960 |
| Depth First Graph Search | No | 12 | **0.009** | 12 | 13 | 48 |
| Depth Limited Search | No | 50 | 0.089 | 101 | 271 | 414 |
| | | | | | | |

| | | | | | | |
|---|---|---|---|---|---|---|
| Uniform Cost Search | **Yes** | **6** | 0.038 | 55 | 57 | 224 |
| Recursive Best First Search | **Yes** | **6** | 3.084 | 4229 | 4230 | 17029 |
| Greedy Best First Graph Search | **Yes** | **6** | **0.01** | **7** | 9 | 29 |

***Problem 2***

| Search Strategy | Optimal | Path Length | Execution Time (s) | Node Expansions | Goal Tests | New Nodes |
|---|---|---|---|---|---|---|
| Breadth First Search | **Yes** | **9** | 12.847 | 3343 | 4609 | 30509 |
| Breadth First Tree Search | -- | -- | -- | -- | -- | -- |
| Depth First Graph Search | No | 575 | 4.055 | 582 | 583 | 5211 |
| Depth Limited Search | -- | -- | -- | -- | -- | -- |
| Uniform Cost Search | **Yes** | **9** | 18.379 | 4853 | 4855 | 44041 |
| Recursive Best First Search | -- | -- | -- | -- | -- | -- |
| Greedy Best First Graph Search | **Yes** | **9** | **1.47** | **399** | 401 | 3617 |

***Problem 3***

| Search Strategy | Optimal | Path Length | Execution Time (s) | Node Expansions | Goal Tests | New Nodes |
|---|---|---|---|---|---|---|
| Breadth First Search | **Yes** | **12** | 74.815 | 14663 | 18098 | 129631 |
| Breadth First Tree Search | -- | -- | -- | -- | -- | -- |
| Depth First Graph Search | No | 596 | **4.511** | **627** | 628 | 5176 |
| Depth Limited Search | -- | -- | -- | -- | -- | -- |
| Uniform Cost Search | **Yes** | **12** | 92.658 | 18223 | 18225 | 159618 |
| Recursive Best First Search | -- | -- | -- | -- | -- | -- |
| Greedy Best First Graph Search | No | 22 | 28.939 | 5578 | 5580 | 49150 |

**Analysis**

If we consider the most important point to reach the optimal solution within the constraint of 10 minutes, only Breadth First Search and Uniform Cost Search algorithms perform that well.

However, Depth First Graph Search seems to be the fastest (despite for the problem 2 Greedy Best First Graph Search performed amazingly fast) and also seems to need the least number of node expansions i.e. less memory use. However, it didn't find the optimal path at any of the problems.

Therefore, we can only keep Depth First Search and Uniform Cost Search as they are the only ones which always find the optimal path, and between this two, **Depth First Search performs a little bit better than Uniform Cost Search in the three cases**.

Only in the cases where the optimal path is not the criteria to determine which algorithm to use, the Greedy Best First Graph Search will be the best choice. It's execution time is more than aceptable and it only didn't find the optimal path in the most complex problem (3). It did find 22 instead of 12, which is not that bad if the look at Depth First Graph which is the fastest but found a path of length 596.

## Informed Search with A*

As we have mentioned prevously, informed search uses domain-specific knowledge and can find the solutions more efficiently thanks to knowledge.
3 different heuristics will be implemented for the A* algorithm.

```
def h_1(self, node: Node):
        # note that this is not a true heuristic
        h_const = 1
        return h_const

    def h_pg_levelsum(self, node: Node):
        '''
        This heuristic uses a planning graph representation of the prob
lem
        state space to estimate the sum of all actions that must be car
ried
        out from the current state in order to satisfy each individual
goal
        condition.
        '''
        # requires implemented PlanningGraph class
        pg = PlanningGraph(self, node.state)
        pg_levelsum = pg.h_levelsum()
        return pg_levelsum

    def h_ignore_preconditions(self, node: Node):
        '''
        This heuristic estimates the minimum number of actions that mus
t be
        carried out from the current state in order to satisfy all of t
```

```
he goal
        conditions by ignoring the preconditions required for an action
to be
        executed.
        '''
        # TODO implement (see Russell-Norvig Ed-3 10.2.3  or Russell-No
rvig Ed-2 11.2)
        # Bring the knowledge base of locial expressions
        kb = PropKB()
        # Add the possitive sentence of the current state
        kb.tell(decode_state(node.state, self.state_map).pos_sentence()
)

        count = 0
        # Iterate over all the goals in the problem
        for clause in self.goal:
            # If the goal is not already among the positive states - wh
ich means
            # we have no reach the goal yet - then increase the counter
            if clause not in kb.clauses:
                count += 1

        return count
```

***Problem 1***

| Search Strategy | Optimal | Path Length | Execution Time (s) | Node Expansions | Goal Tests | New Nodes |
|---|---|---|---|---|---|---|
| A* Search with h1 heuristic | **Yes** | **6** | 0.043 | 55 | 57 | 224 |
| A* Search with Ignore Preconditions heuristic | **Yes** | **6** | **0.039** | 41 | 43 | 170 |
| A* Search with Level Sum heuristic | **Yes** | **6** | 5.10 | **7** | 9 | 28 |

***Problem 2***

| Search Strategy | Optimal | Path Length | Execution Time (s) | Node Expansions | Goal Tests | New Nodes |
|---|---|---|---|---|---|---|
| A* Search with h1 heuristic | **Yes** | **9** | 18.371 | 4853 | 4855 | 44041 |
| A* Search with Ignore Preconditions heuristic | **Yes** | **9** | **6.270** | 1428 | 1430 | 13085 |
| A* Search with Level Sum | No | 21 | 249.784 | **97** | 99 | 906 |

| | | | | | | |
|---|---|---|---|---|---|---|
| heuristic | | | | | | |

--- This last one is raising an error ---

***Problem 3***

| Search Strategy | Optimal | Path Length | Execution Time (s) | Node Expansions | Goal Tests | New Nodes |
|---|---|---|---|---|---|---|
| A* Search with h1 heuristic | **Yes** | **12** | 91.983 | 18223 | 18225 | 159618 |
| A* Search with Ignore Preconditions heuristic | **Yes** | **12** | **27.898** | 5040 | 5042 | 44944 |
| A* Search with Level Sum heuristic | **No** | 21 | 333.905 | **71** | 73 | 687 |

--- This last one is raising an error ---

**Analysis**

The first and very important point of these approaches is that all of them led to the optimal path (except level sum in problem 3 - level sum may have a implementation bug).

However, it is clear that **Ignore Preconditions heuristic outperform the others** if we look at the execution time.

The Level Sum heuristic on the other hand has expanded way less nodes that the other heuristics. Then, if memory usage is the main criteria, this would be the heuristic to use, with the disadvantage of being vey slow. The low speed is a consequence of having to explore the graph and check in which level the goal is.

## Uninformed Search vs Informed Search

If we compare the winning strategy of each block:

***Problem 1***

| Search Strategy | Optimal | Path Length | Execution Time (s) | Node Expansions |
|---|---|---|---|---|
| Breadth First Search | **Yes** | **6** | **0.034** | 43 |
| A* Search with Ignore Preconditions heuristic | **Yes** | **6** | 0.039 | **41** |

***Problem 2***

| **Search Strategy** | **Optimal** | **Path Length** | **Execution Time (s)** | **Node Expansions** |
|---|---|---|---|---|
| Breadth First Search | **Yes** | **9** | 12.847 | 3343 |
| A* Search with Ignore Preconditions heuristic | **Yes** | **9** | **6.270** | **1428** |

***Problem 3***

| **Search Strategy** | **Optimal** | **Path Length** | **Execution Time (s)** | **Node Expansions** |
|---|---|---|---|---|
| Breadth First Search | **Yes** | **12** | 74.815 | 14663 |
| A* Search with Ignore Preconditions heuristic | **Yes** | **12** | **27.898** | **5040** |

It looks clear how A* outperforms Bread First Search, and clearer when the problem gains complexity. This shows the **benefits of informed search over uninformed search** where the results are achieved using less memory and in less time. Furthermore, informed search allows to customize a trade-off between speed and memory by customizing the different heurisitics that can not be done with uninformed search strategies.