AI Homework 1 - Heuristic Agent
Submitted by Anusha Annengala

1. Implement an agent module/class *Agent* that uses Heuristic Search for choosing an action to take given a state. The agent uses an independent module/class *Game* for functions defining state transitions and states neighbourhood and a module/class *Heuristics* for definitions of heuristic functions.
2. Choose two different problems/games/puzzles and provide a statistical experimentation on several problem instances for comparatively evaluating the agent performance using different search algorithms and different heuristics

---

Game 1 - Eight Puzzle Solver

An instance of the 8-puzzle game consists of a board holding 8 distinct movable tiles, plus an empty space. For any such board, the empty space may be legally swapped with any tile horizontally or vertically adjacent to it. In this assignment, the blank space is going to be represented by the number 0. Given an initial state of the board, the search problem is to find a sequence of moves that tran-sitions this state to the goal state; that is, the configuration with all tiles arranged in ascending order 0,1,2,3,4,5,6,7,8.

Uniform Cost Search:

This search is essentially a breadth-first search using a priority queue. This is because each time a state expands, its g(n) cost (the cost of getting to the current node from the initial node) increases by 1 (we did not define an expansion weight). In comparison with the other two algorithms, this is the slowest search method and takes up the largest space.

A* Misplaced Tile Heuristic:

This algorithm uses the combined values of g(n), number of misplaced tiles between the initial and current state, and h(n), the number of misplaced tiles between the current state and the goal. This algorithm performs in the middle of the other two, more efficient than the uniform cost and less so than the A* using the Euclidean heuristic.

A* Euclidean Distance Heuristic:

This algorithm uses the combined values of g(n) and h(n). These functions represent the euclidean distance between the initial state and the current state, and the euclidean distance between the current state and the goal. This algorithm performed most efficiently when compared to the other two when the input is significantly more difficult to solve.

Design:

The code of line that calls main is the **agent**, a main() function which is the **game** calls greets and prompts you to select either a default 3x3 puzzle, or enter your own custom puzzle. If you opt for the default puzzle, you are then asked to choose from a pool of premade initial states (trivial, very easy, easy, doable, oh boy, and impossible).
After you select your desired difficulty, the program asks you to select an algorithm. The following algorithms are available:

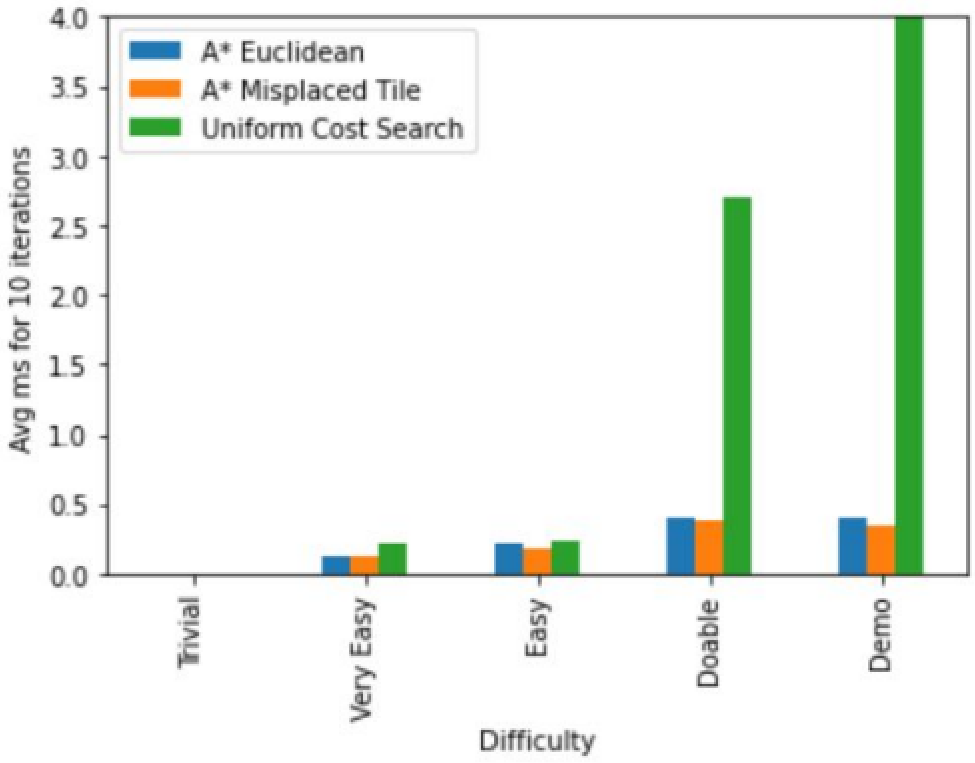
- **Simple Uniform Cost Search**
- **A* Misplaced Tile Heuristic**
- **A* Manhattan Distance Heuristic**
- **A* Euclidean Distance Heuristic**

After you select an algorithm, the program begins to search for a solution. It subsequently displays a traceback of the solution nodes, the number of expansions, max nodes in the queue at any time, and the execution time.
The main class in this project is the Node class. Each Node has a number of useful features including its parent node (if not root), its puzzle state, the cost from start (g(n)), the cost to goal (h(n)), and a list of children. One of the most important methods in this class is the expand() method. Here, the algorithm finds 0 in its board state, and proceeds to generate a list of possible swaps. The states in this list are then used in the tree_search() function where they are pushed to a heap queue depending on whether they were previously seen (either in the heap queue, or the explored set).
The heart of this program lies in the tree_search() function. The algorithm is mostly the same as the pseudocode provided in the project manual. One of the challenges that I faced was that I needed my heap queue to be able to easily sort each node according to its f-value. Remember that the board states are wrapped inside the Node class where only g(n) and h(n) are stored. So, my heap queue had no idea how to compare these objects. You can instruct a heap queue to compare different complex objects by defining **lt**(self, other) in my class where we compare each node's cost_from_start + cost_to_goal. This way, whenever I push a node to my heap queue, it is automatically sorted by its f-value.

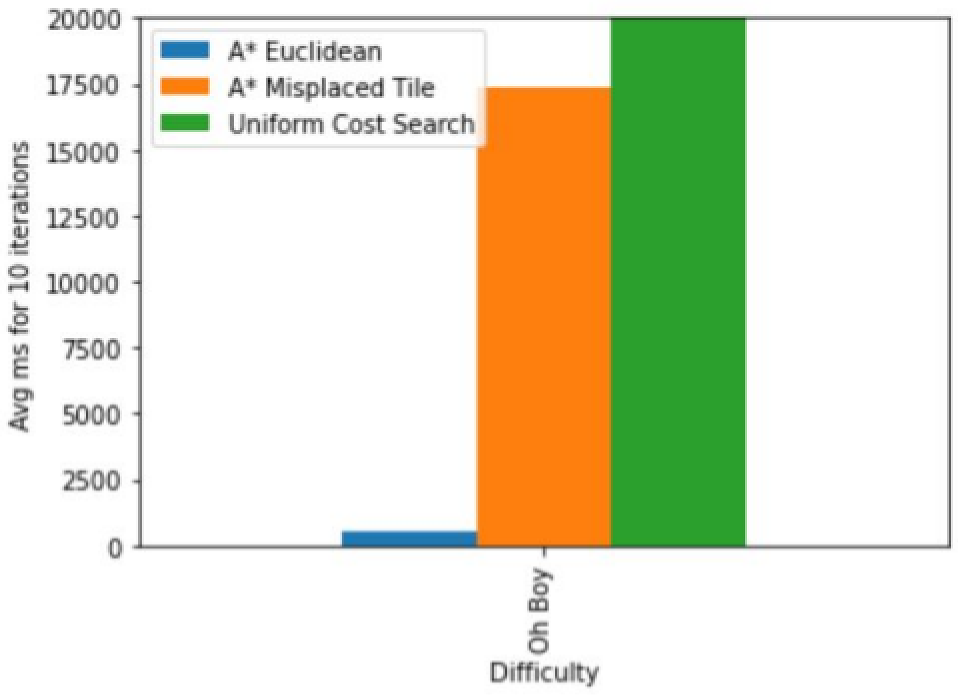
Performance and analysis:

Here, you may find a comparison of the three main algorithms when tested on the following initial state puzzles. I ran each algorithm on each puzzle state and averaged their performance over 10 iterations each. The testing notebook is named as performance-graphs-eight-puzzle-solver.ipynb.

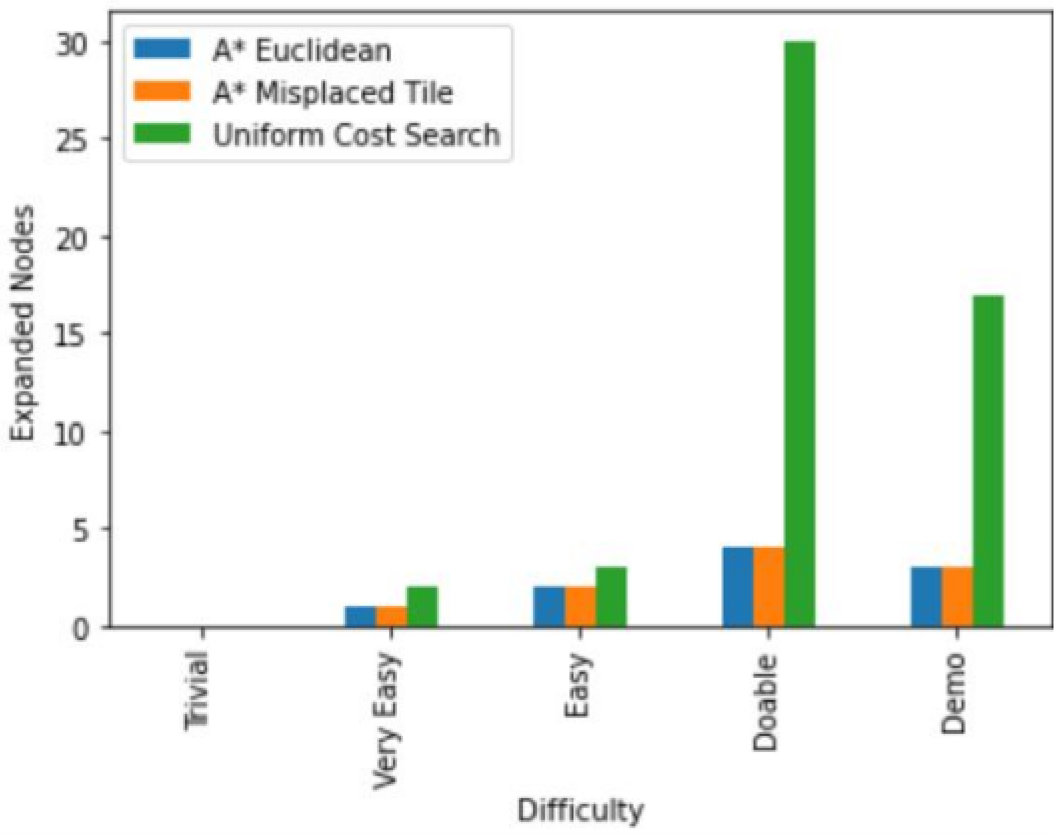
```
Trivial     Easy        Oh Boy      Very Easy   Doable      Demo
+-----+     +-----+     +-----+     +-----+     +-----+     +-----+
|1 2 3|     |1 2 0|     |8 7 1|     |1 2 3|     |0 1 2|     |1 0 3|
|4 5 6|     |4 5 3|     |6 0 2|     |4 5 6|     |4 5 3|     |4 2 6|
|7 8 0|     |7 8 6|     |5 4 3|     |7 0 8|     |7 8 6|     |7 5 8|
+-----+     +-----+     +-----+     +-----+     +-----+     +-----+
```



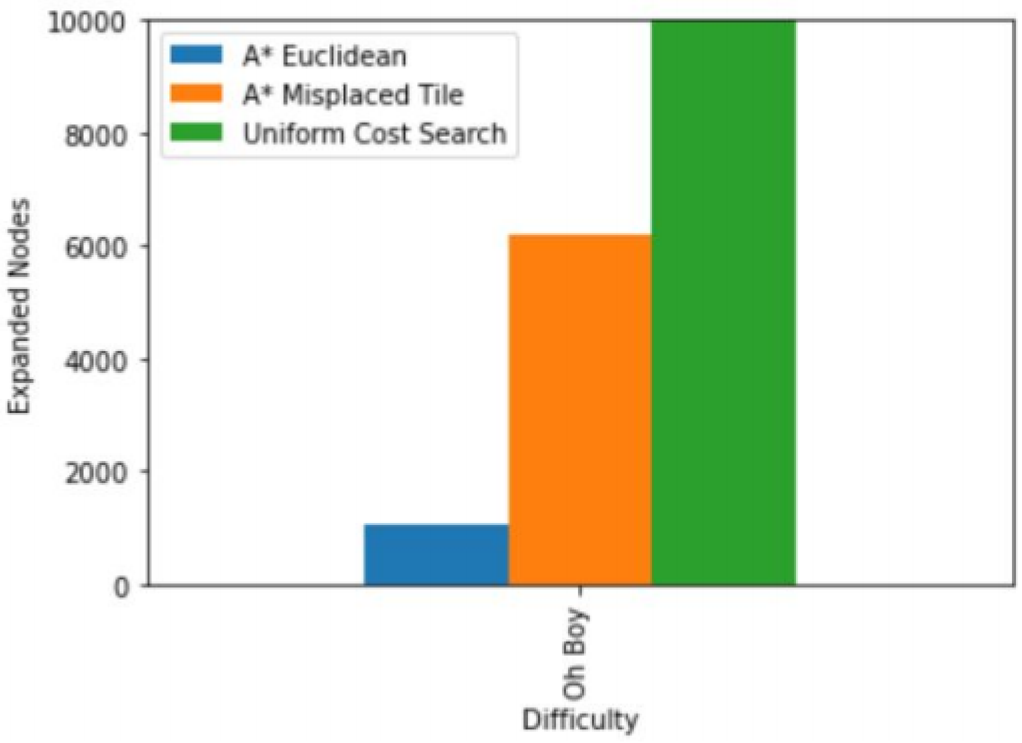
The uniform cost search takes the longest to finish. Moreover, the A* misplaced tile heuristic outperforms the euclidean heuristic for now.
Let's measure the time it takes for all algorithms to solve the "Oh Boy" puzzle difficulty:



The A* Euclidean which was previously outperformed by the A* misplaced tile is now significantly more efficient. Uniform cost search takes much longer to complete and hence does not even fit on the same scale as A* euclidean! Let's investigate how many nodes each algorithm expands:

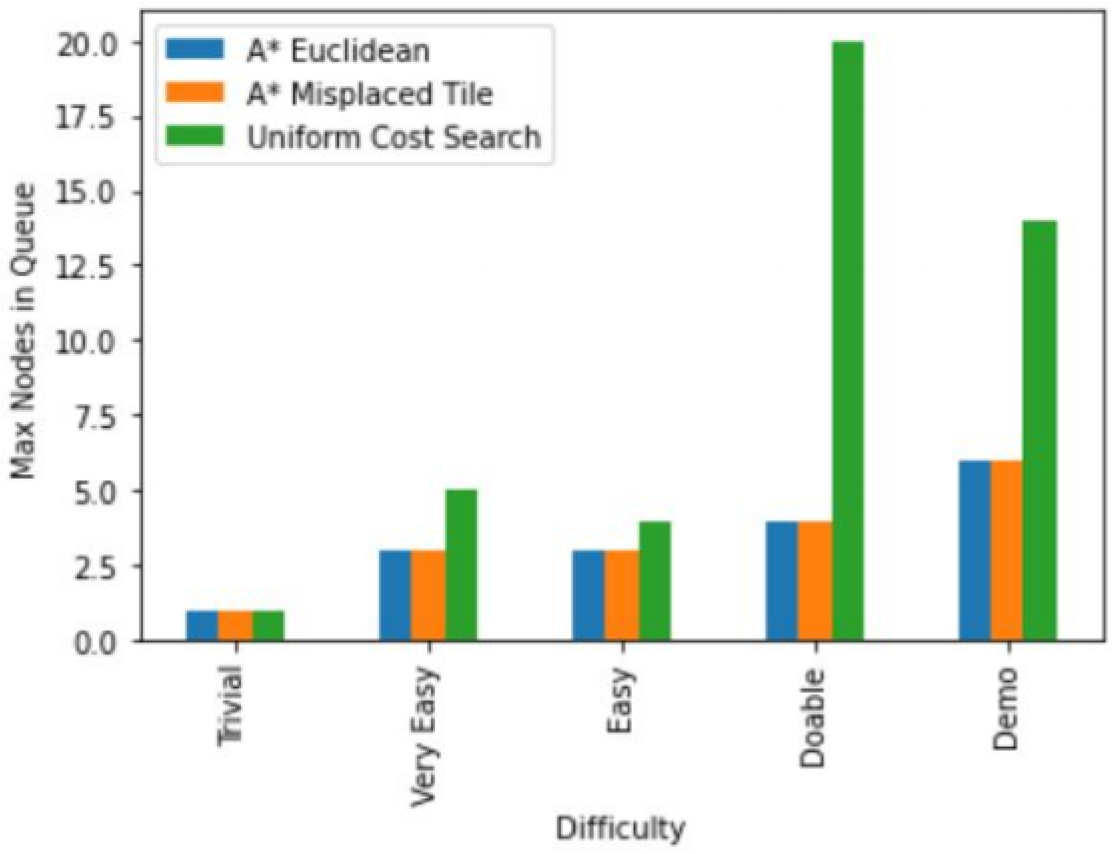


You may notice that the A* search outperforms the uniform cost search in terms of space requirement as well. Turns out using an admissible heuristic makes everything better.
Since the Oh Boy state is so difficult, it gets its own chart:

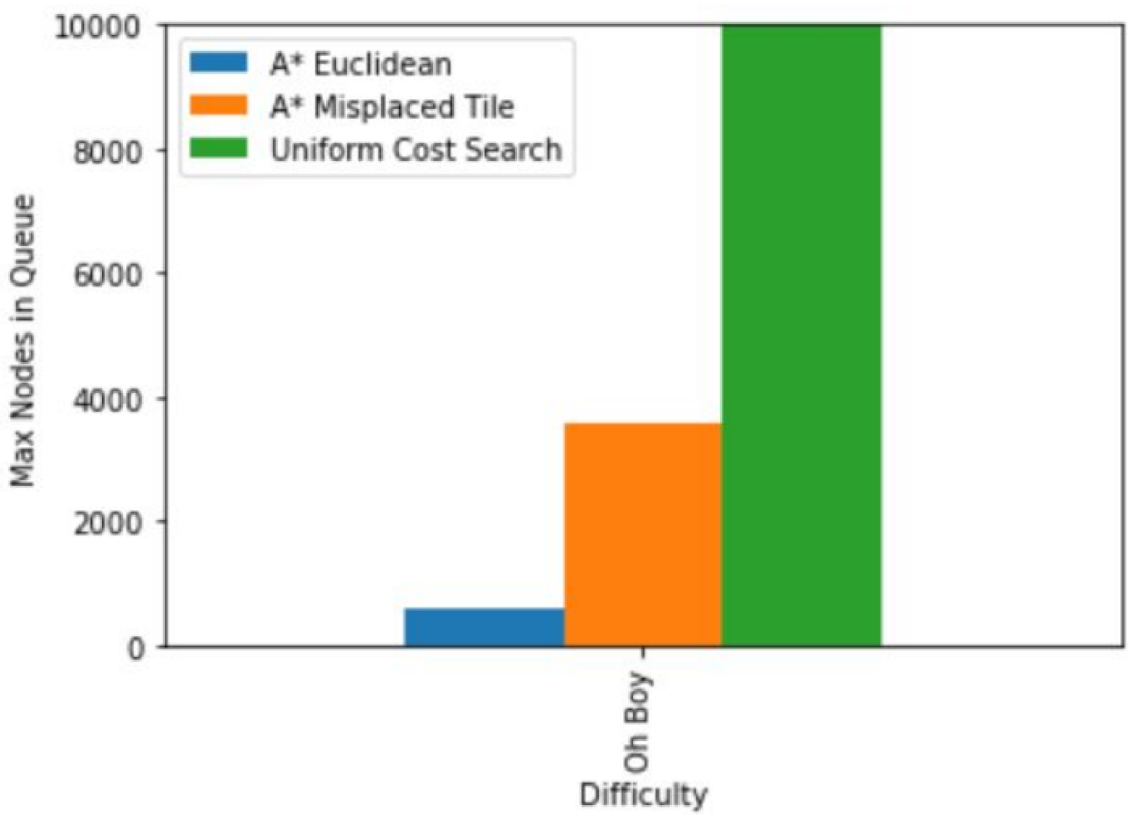


We see that again, A* search outperforms the other two by a huge margin. Uniform cost search just takes too much space that it would outscale A* euclidean if fully drawn.

Here’s another chart that shows the number of maximum nodes in our queue at any given time:



Again, the uniform cost search requires much more space when it comes to increasingly difficult puzzles. Here’s the Oh Boy difficulty chart for max nodes in queue:



Conclusion:

Based on my empirical results concerning the three search algorithms of uniform cost, A* misplaced tile, and A* euclidean heuristics, it can be said that:

1. Using a good heuristic for the job saves us a lot of space and time. For example, uniform cost takes a substantial amount of time to complete the same job compared to A*. In our case, for the uniform cost search, since h(n) was 0, and g(n) was always equal to the depth of a node in the tree, it degenerated into a breadth-first-search, one that uses a priority queue. The time and space complexity for a breadth-first-search is O(bd) where b is the branching factor and d is the depth.
2. Using the A* search significantly improved the time and complexity compared to the uniform cost. Between the two heuristics that we used to A*, euclidean distance outperformed missing tiles in both time and space when the puzzle was more difficult. The missing tiles heuristic slightly outperforms euclidean distance in terms of time when our puzzle is simple while it occupies the same space.

Program output:

```
Welcome to 8-puzzle solver.
Select an option:
[1] Default 3x3 puzzle
2 Custom puzzle
(Press enter to default to [1])
Default puzzle: enter the difficulty (1 to 7):
[1] demo
2 trivial
3 very easy
```

```
4 easy
5 doable
6 oh boy
7 impossible
5
Selected doable
Display state traceback at the end? (may crash your notebook if puzzle is too difficult) [y]/n
Select algorithm:
1 Uniform Cost Search
2 A* Misplaced Tile Heuristic
3 A* Manhattan Distance Heuristic
[4] A* Euclidean Distance Heuristic (fastest)
Selected A* Euclidean Distance Heuristic
The best state to expand with g(n) = 0 and h(n) = 0 is...
[0, 1, 2]
[4, 5, 3]
[7, 8, 6]
The best state to expand with g(n) = 1 and h(n) = 3 is...
[1, 0, 2]
[4, 5, 3]
[7, 8, 6]
The best state to expand with g(n) = 2 and h(n) = 2 is...
[1, 2, 0]
[4, 5, 3]
[7, 8, 6]
The best state to expand with g(n) = 3 and h(n) = 1 is...
[1, 2, 3]
[4, 5, 0]
[7, 8, 6]
The best state to expand with g(n) = 4 and h(n) = 0 is...
[1, 2, 3]
[4, 5, 6]
[7, 8, 0]
Found a solution!
Traceback:
[[1, 2, 3], [4, 5, 6], [7, 8, 0]]
[[1, 2, 3], [4, 5, 0], [7, 8, 6]]
[[1, 2, 0], [4, 5, 3], [7, 8, 6]]
[[1, 0, 2], [4, 5, 3], [7, 8, 6]]
[[0, 1, 2], [4, 5, 3], [7, 8, 6]]
5 nodes traced.
To solve this problem the search algorithm expanded a total of 4 nodes.
The maximum number of nodes in the queue at any one time: 4
--- 0.0067656999999599066 milliseconds ---
```

Game 2 - Pacman

Pac-man is a maze arcade game, the player supervises Pac-man, who must eat all the dots inside an surrounded maze while keeping away from the four coloured ghosts. Eating huge flashing dots called power pellets changes the ghosts to turn blue, allowing Pac-Man to consume them for bonus points. Pac-Man was a critical and commercial victory, and has a commercial and cultural legacy.

The main aim is to build an intelligent pacman agent which finds optimal paths through the maze to find a particular goal such as a particular food position, escaping from ghosts. For that, we have implemented AI search algorithms like Depth first search, Breadth first search, A*search, Uniform cost search.

BFS:

This algorithm has a queue as its frontier and a set as it's explored.

- At first we add the initial state into the frontier and then pop it from the queue.
- Then all possible actions from initial values which does not explored before, will add to the frontier
- and next we will redo these steps until we find the goal state.

BFS is an optimal search algorithm.

IDS:

This algorithm has a stack as its frontier (we used dequeue data structure and its pop method) and a set (set of ExploredNode) as it's explored.

- We have a loop to increase depth in every iteration of it. Actually at every iteration we increase depth and run inner loop which is a limited dfs.
- At every iteration we reset all of the data structures to initial values.
- In limited dfs :
    - We pop first node from stack
    - If it's depth is our limit we didn't add it's children to frontier
    - Else we add children to frontier

This IDS is an optimal search algorithm

Heuristic:

Our heuristic is minimum Manhattan distance from all foods + [number of remaining food after eating nearest food / 2] .
We will compute it in the manhatan_heuristic method of InformedPacmanWorld class. In this method, first we find all of the world 's food positions, then find the number of all foods. After that we returns minimum computation of heuristic from P foods and Q foods + number of remained foods / 2

A*:

- At first we add the initial state into frontier and then pop it from the queue.
- Then all possible actions from initial values which does not explored before, will add to the frontier
- and next we will redo these steps until we find the goal state.

This Astar search is an optimal search algorithm. Because it uses an admissible and consistent heuristic.

Comparing algorithms:

At First we run every test case and compute all variables into a pandas data frame. These variables are Answer Cost, Total Seen States, Total Separate Seen States, and Execution Time.

Optimality:

All of these algorithms return optimal answers. BFS is optimal because we move in depth. IDS is optimal too because we limit the depth of DFS. In A* if we have an admissible and consistent heuristic, we can say it is an optimal algorithm. Here we have a heuristic with these characteristics which is proven above.

Time Complexity:

For `BFS` if we assume that we have b-ary tree of depth d time complexity is the number of nodes in that which is $O(b^d)$.

For A*Search time complexity is the number of nodes which heuristic expands.

As you see in above tables time order is IDS > A* > BFS. Although number of seen states is in IDS > BFS > A* but time of A* is a little more than bfs. It is because of computing heuristic overheads. For bigger maps which this overhead can be ignored we can say time of A* is less than BFS; on the other hand, IDS have a lot of more time complexity in this problem. Because we have a big depth for solutions. It is always slower than BFS and A*.

Space Complexity:

Space complexity of BFS is like its time complexity, which is the number of its nodes. ($O(b^d)$).

IDS have less space for computing. It has a space complexity of $O(bd)$ and that's why people use IDS instead of BFS.

A* also like BFS have an exponential space complexity but because of less nodes which are seen in this algorithm we can say we hope it got less space.