First Assignment: Report

INTELLIGENT SYSTEMS

- **Víctor Pérez Piqueras**
- **José Javier Atiénzar González**

UCLM | ESIIAB

Content

Figures

Tables

1. Problem Description

The maze used in the assignment is composed of a matrix of cells and walls that separate them. There is an agent(hamster) located at the entrance of the maze (top row), that must reach the exit (bottom row). Besides that, the agent must visit every one of the cells containing a reward(cheese) and take(eat) them. Moreover, it must avoid visiting other special cells that contain cat.

The goal of the search is to find the shortest path (smaller cost), that holds the mentioned conditions. Also, it must be considered that:

- The hamster can only move to adjacent cells when there is no wall between them. Cannot move diagonally.
- The first time the hamster meets a cat, they fight, and the hamster gets damaged, so from that moment onwards, the cost of each movement is 2.
- The second time it meets a cat, the damage is total, so it is no longer able to carry out a movement.
- The hamster must eat all three cheeses in order to complete the path. So, it must reach the corresponding cells and carry out the action eat, that costs 1. Once eaten, the cheese disappears.

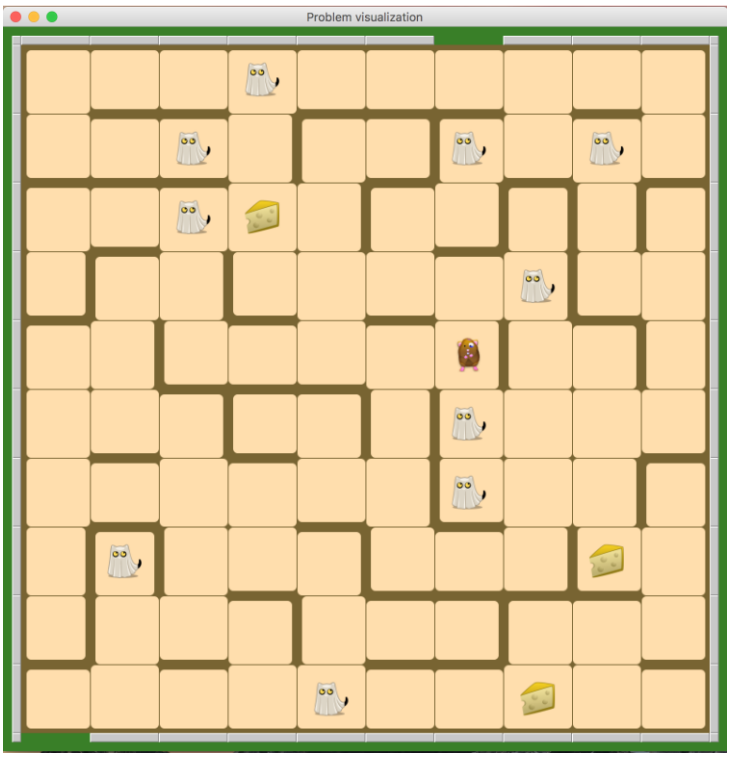


Figure 1: Maze

2. Problem formalization

This problem can be formalized as a search one taking into account these aspects:

- State space: the states that will be used for the problem definition are the states of the maze and its components at every move.
- Initial state: the hamster will be spawn in a cell, with the maze generated with all its components.
- Actions: the possible actions that the hamster is allowed to do that are the 4 moves (Left; Right; Up; or Down) and the eat action.
- Successor function: this function will return the successors obtained by applying an action over a state of the problem.
- Goal test: it determines if the hamster has reached the goal cell having the 3 cheeses been eaten.
- Path cost: is the cost obtained by adding all the moves and its corresponding cost, depending on the hamster's life.

Algorithms that will be used for the search problem:

- Non-informed search algorithms:
  - **Breadth-First:** always expands the sallowest open node, that may not be the optimal one, using a FIFO queue to manage this set of open nodes.
  - **Uniform-Cost:** expands the node with the lowest path cost, by using a priority queue ordered by cost.
  - **Depth-First:** expands the deepest node in the set of open nodes, by using a LIFO queue or stack.
  - **Depth-Limited:** it fixes a depth limit in a depth-first search.
  - **Iterative deepening:** Aims at finding the best depth limit, by increasing the maximum depth until the limit is found.

- Informed search algorithms:
  - **Greedy Best-First:** tries to expand the node that is closer to the goal, by evaluating nodes with the heuristic function.
  - **A*:** follows the best-first strategy but considering both the cost and heuristic to evaluate the nodes.

3. Description of the implementation

- Maze: Allows building and representing a maze with the characteristics described above. The implementation supports on objects of class *utils.Position*, that correspond to (x, y) coordinates, and represent the maze with three data structures that can be accessed (at package level):

    - cellGraph: contains the cells (position) that can be reached from each cell.
    - cheesePositions: contains set of positions where there is a piece of cheese.
    - catPositions: contains set of positions where there is a cat. The class also provides a series of public functions that return input and output positions, test if cells contain cheese or cats, etc.

- MazeState: Extends the class *search.State* to represent the states of the problem. In such a case, each state is defined by the coordinates (Y; X) where the hamster is, but it is also necessary to represent the damage level, or the pieces of cheese eaten. The attributes are:
    - Position (type Position)
    - Life(int)
    - Cheeses eaten (hash set of positions)

  The methods are:
    - Equals: compares if 2 states are the same.
    - hashCode: returns the hash value of the object.
    - toString: prints the values of the object.

- MazeProblem: Encapsulates objects of the class *Maze*. Implements the interface *SearchProblem* to provide the complete definition of the problem. Here we provide a pseudocode of the methods of the class:

    - *initialState*: returns the initial state of the problem, with the initial position of the hamster as input parameter.

    - *applyAction*: given a state and an action, it produces a new state:
        - If action == up→change position of the hamster to y-1.
        - If action == down→change position of the hamster to y+1.
        - If action == left→change position of the hamster to x-1.
        - If action == right→change position of the hamster to x-1.
        - If action == eat→add the cheese in the position to the set of eaten cheeses of the hamster.
        - For every action, if there is a cat in the current cell, the life of the hamster is updated.

- *getPossibleActions*: given a state, it returns a list with all the possible actions:
  - If the hamster's life==dead→returns no possible actions
  - If there is a reachable cell in any of the 4 directions it returns that direction.
  - If there is a cheese in the current position, returns that the action EAT can be performed as well.

- *cost*: given a state and an action returns the cost:
  - If the life==ok→returns cost 1
  - If the life==damaged→returns cost 2

- *testGoal*: given a state returns if it is a goal state or not:
  - checks if all cheeses are eaten
  - checks if the current position==final position

- *heuristic*: given a state, it returns a value with the heuristic related to it. We have defined 2 heuristics.

- MazeAction: Implements the interface *Action*. In the maze, and regardless of the state, the available actions are four: *{UP, DOWN, LEFT, RIGHT, EAT}*. Because of that, *MazeAction* is an enumeration (instead of a class).

- MazeView: Allow showing a maze and the effect of a sequence of actions. Makes use, and therefore depends, on the representation of the state, i.e., *MazeState*.

4. Performance evaluation

3.1 Between algorithms:

Each algorithm works in a different way, and so, each one will perform in a different manner. Applying them to the same problem we obtain the following information. These are the results obtained for the problem P1 with input data: *{maze size=10, number of cats=1, seed=0}* for the following algorithms:

- Breadth-First
- Depth-First
- Depth-Limited(tree)
- Depth-Limited(graph)
- Iterative-Deepening(tree)
- Iterative-Deepening(graph)
- Uniform Cost
- Greedy Best First
- A*

Data:

Table 1: maze size=10, number of cats=1, seed=0

Algorithm	Cost	Generated nodes	Expanded nodes	Explored nodes	Max. Size set open nodes	Max. Size set explored nodes	Time(ms)
Breadth First	30	2524	1055	2410	150	115	297
Depth First	299	788	329	669	124	120	15
Iterative Deepening(graph)	60	67270	27527	67291	63	43	72
Uniform Cost	30	1843	767	1697	152	147	16
Greedy Best First	78	1031	418	953	79	79	47
A*	30	1131	461	1010	150	122	16

- It is important to say that, for this problem, the depth-limited algorithm in both tree and graph versions and the tree version of the iterative deepening are not useful for this problem.

Depth Limited(tree)	SOLUTION TAKES TOO LONG FOR TREE VERSION
Depth Limited(graph)	NO SOLUTION (*)
Iterative Deepening(tree)	SOLUTION TAKES TOO LONG FOR TREE VERSION

- (*) Depth-Limited doesn't find solutions using a proper maximum depth, because whenever a state is explored (in graph search), it is never explored again, even if it is in a lower depth. For tree search, as we are not using a set of explored nodes, the total number of states raises to 5^{10}. We have only obtained solutions for a maze of size 4, because 5^4 is still reasonable, but for bigger mazes it is impossible.

- Iterative Deepening doesn't find a solution for the tree version for the same reason as Depth-Limited, but for the graph version it does. As it can be shown in the Figure 2, its efficiency is quite bad for this problem, so we haven't used it for other comparisons, because the range of nodes that it generates is so huge even for simple problems.

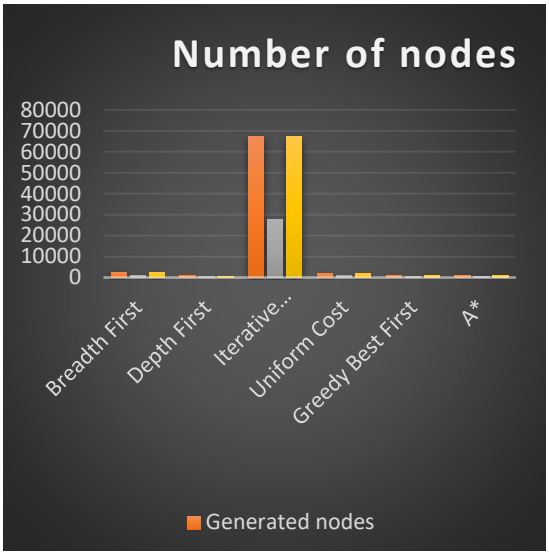


Figure 2: Number of nodes with Iterative Deepening

- For the following comparisons the Iterative Deepening algorithm will be discarded, as it is not useful to plot it with the other algorithms due to its high cost.

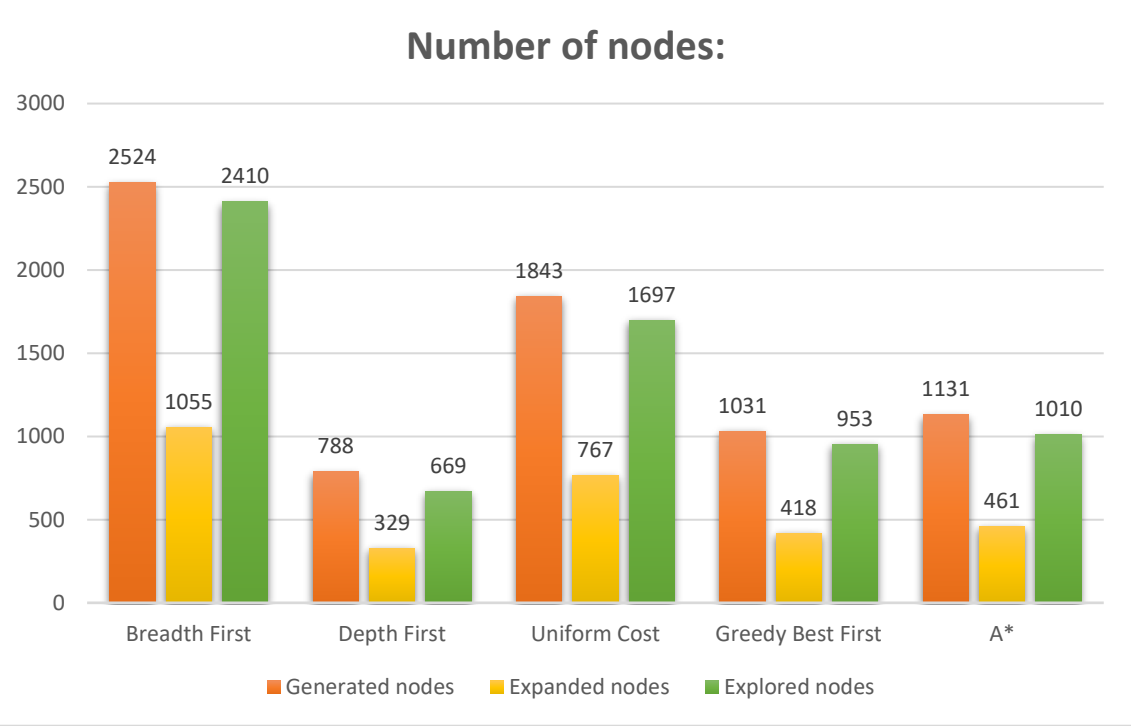


Figure 3: Number of nodes

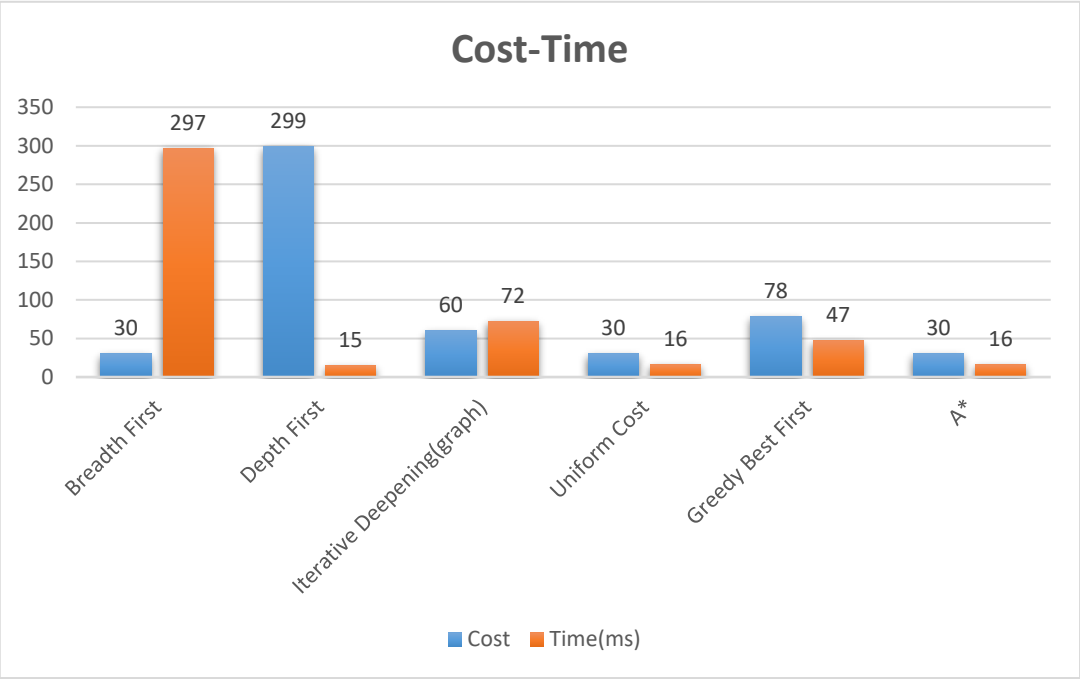


Figure 4: Cost-Time comparison

As we can see in the figure 4, while Breadth-First finds solutions with a good cost, it pays it by having the biggest execution time, because it generates all the successors from depth to depth until finding the solution.

On the other hand, Depth-First is quite fast, because doesn't need much memory to store nodes, but it returns a very bad solution. It depends on the problem to choose an algorithm faster but inefficient, or an efficient one but with a high cost in terms of execution time and memory.

For this particular problem, (apart from depth-limited that behaves more like the depth-first) these algorithms work more or less the same, returning a good cost solution in a very short time.

Maybe the greedy one has a bit higher cost, mainly due to the heuristic chosen for the problem, that in this case is less restrictive: the Euclidean distance between the mouse and the exit cell.

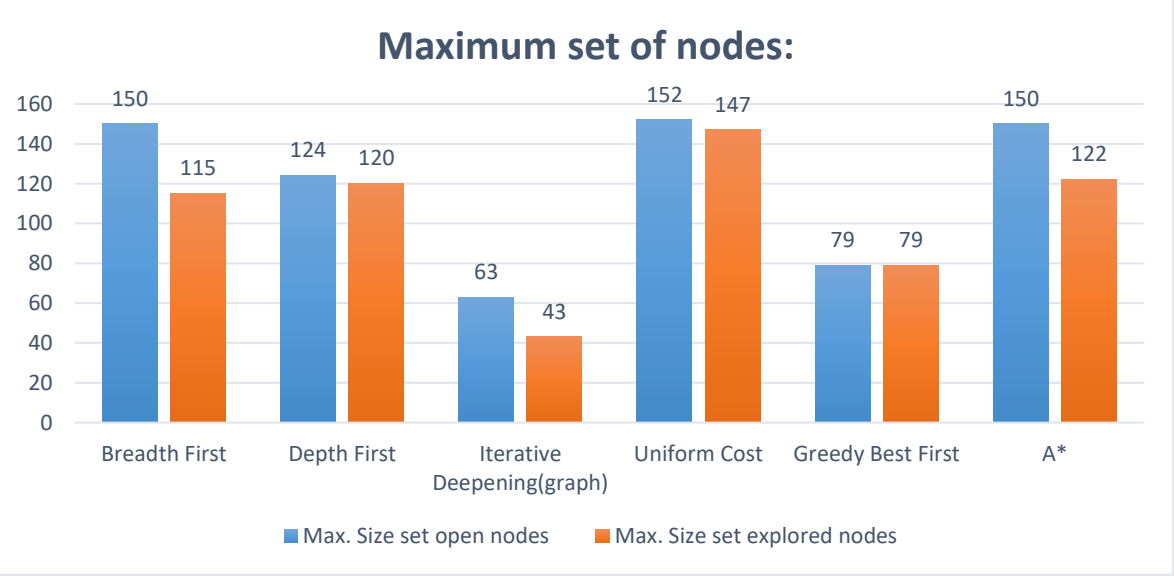


Figure 5: Sets of nodes

3.2 Increasing maze size:

Data:

Table 2: {Maze 10 10 0}

Algorithm	Cost	Generated nodes.	Expanded nodes.	Explored nodes	Time(ms)
Uniform Cost	85	1498	688	1492	19
Breadth First	85	1496	687	1482	12
Depth First	223	450	205	360	29
Greedy Best First (H2)	121	635	277	559	29
Greedy Best First (H1)	109	724	313	653	65
A Star (H2)	85	1296	586	1234	19
A Star (H1)	85	1437	653	1384	64

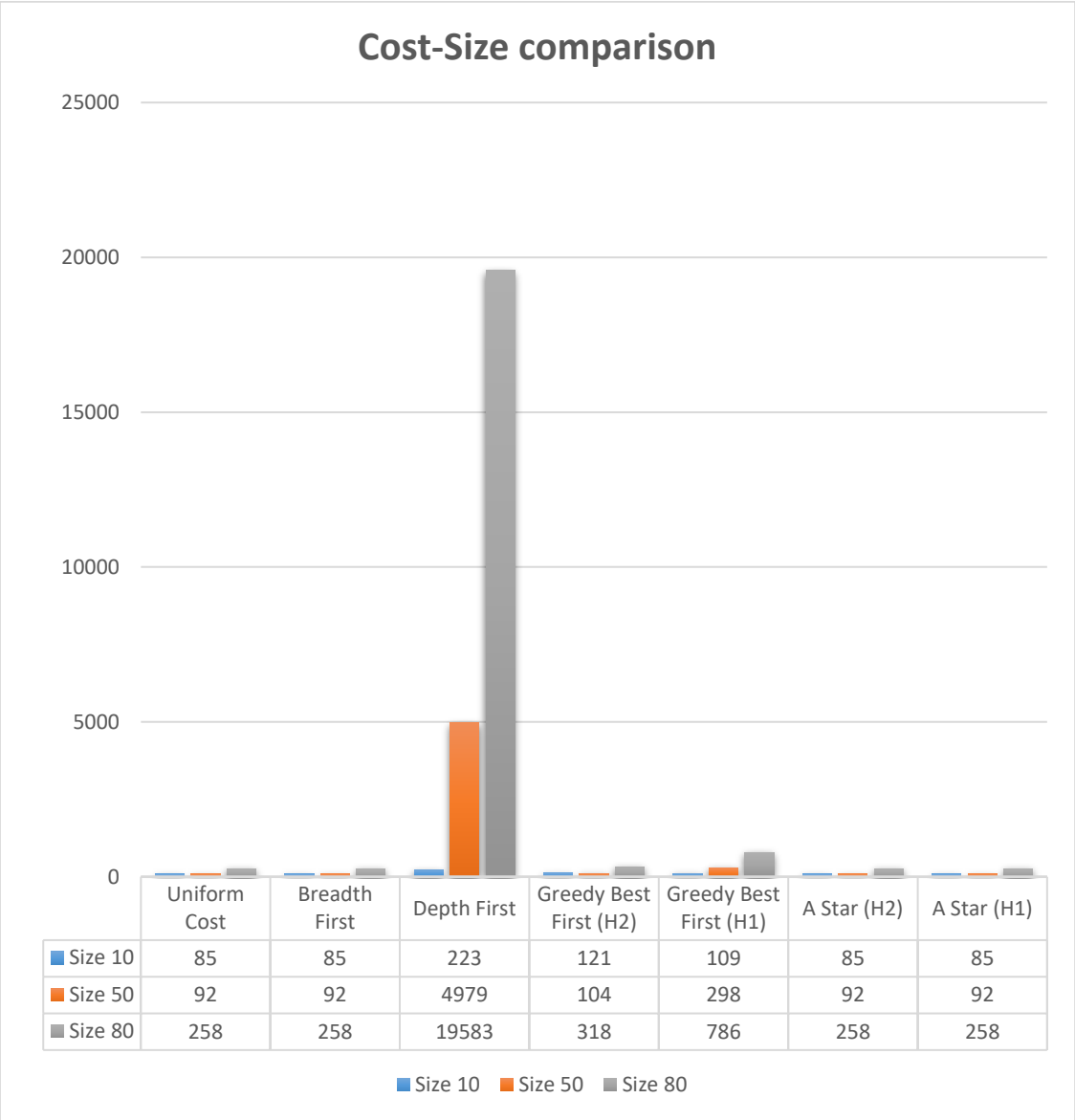
Table 3: {Maze 50 10 0}

Algorithm	Cost	Generated nodes.	Expanded nodes.	Explored nodes	Time(ms)
Uniform Cost	92	67203	26518	64376	3592
Breadth First	92	85291	33825	83730	12511
Depth First	4979	15602	6127	13041	833
Greedy Best First (H2)	104	1971	755	1738	85
Greedy Best First (H1)	298	37075	14713	36384	3869
A Star (H2)	92	3182	1228	2843	31
A Star (H1)	92	17959	6996	16391	1067

Table 4: {Maze 80 10 0}

Algorithm	Cost	Generated nodes	Expanded nodes	Explored nodes	Time (ms)
Uniform Cost	258	242755	95943	240652	680
Breadth First	258	257133	101707	256831	573
Depth First	19583	53233	20785	44063	101
Greedy Best First(H2)	318	3304	1277	2685	18
Greedy Best First (H1)	786	114368	45206	113825	195
A Star(H2)	258	37174	14597	34812	87
A Star (H1)	258	242755	95943	240652	680

Results:



	Uniform Cost	Breadth First	Depth First	Greedy Best First (H2)	Greedy Best First (H1)	A Star (H2)	A Star (H1)
Size 10	85	85	223	121	109	85	85
Size 50	92	92	4979	104	298	92	92
Size 80	258	258	19583	318	786	258	258

Figure 6: Cost-Size comparison

We can see that the bigger the maze the bigger the cost is, but specially for the Depth First algorithm the cost increases a lot. This is because this algorithm explores the leftmost part of the tree which results in an expensive solution in problems with a big size. The cost increase remains the same for the algorithms that find optimal solutions.

It is remarkable that the greedy algorithm obtained a better solution with a smaller maze. We have studied this behaviour over other maze problems and we have seen that for most of them, applying the more restrictive heuristic(H2), highly improves the performance. But for some cases, depending on the cat disposition and cells, it could happen that even with a better heuristic, the problem solution is not the best. But it only happens with the greedy algorithm, as it only takes the heuristic value. With A* for example, it will always take the better solution.

3.3 Increasing number of cats:

It is also interesting to study how increasing the cat number affects the problem solutions, and these are the data obtained for 2 sample problems:

Data:

Table 5: {Maze 80 41 0}

Algorithm	Cost	Generated nodes.	Expanded nodes.	Explored nodes	Max.size set open nodes.	Max.size set explored states.	Time(ms)
Uniform Cost	300	227614	90188	226215	1940	1400	289
Breadth First	512	230506	91375	229733	1256	774	19828
Depth First	17595	46211	18113	38071	8145	8141	84
Greedy Best First (H2)	618	44650	17699	44046	605	605	2274
Greedy Best First (H1)	1205	135086	53544	134497	776	590	5674
A Star (H2)	300	69909	27735	68466	1773	1464	243
A Star (H1)	300	166178	65797	164802	2326	1377	272

Table 6: {Maze 80 1 0}

Algorithm	Cost	Generated nodes.	Expanded nodes.	Explored nodes	Max.size set open nodes.	Max.size set explored states.	Time(ms)
Uniform Cost	234	182183	71954	180249	2394	1935	386
Breadth First	234	246072	97214	245086	2284	987	404
Depth First	20432	53690	20934	43700	9991	9991	73
Greedy Best First (H2)	292	2896	1122	2347	550	550	19
Greedy Best First (H1)	1136	123954	48985	123288	747	667	210
A Star (H2)	234	20848	8233	19979	904	870	53
A Star (H1)	234	96021	37803	94339	2048	1683	258

Results:

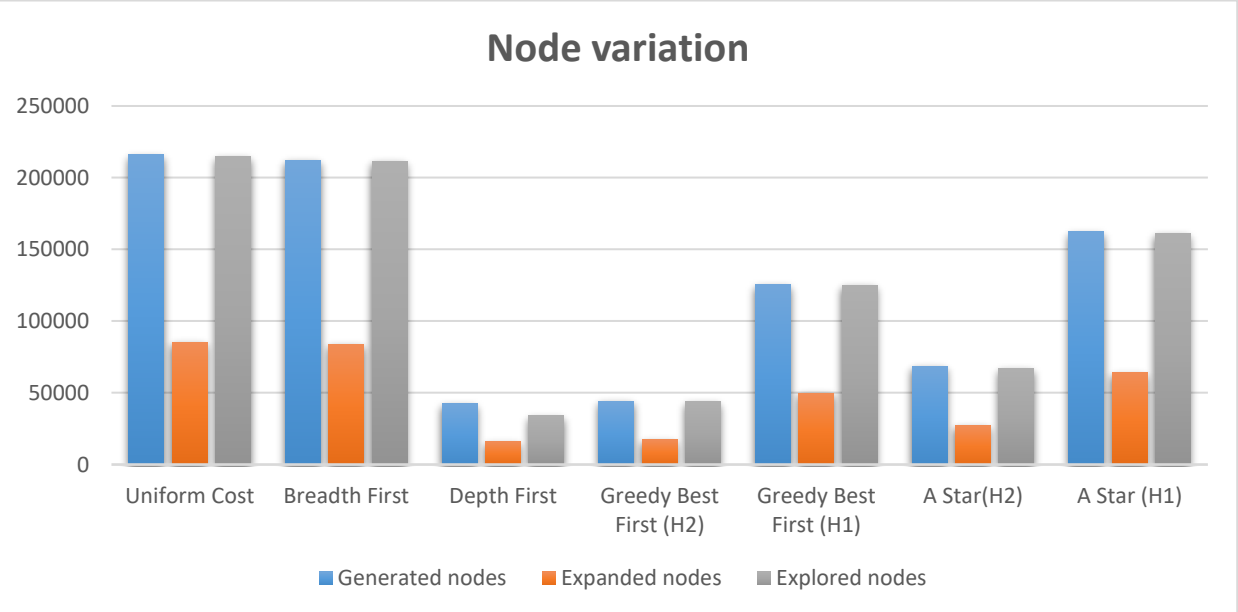


Figure 7: Node variation

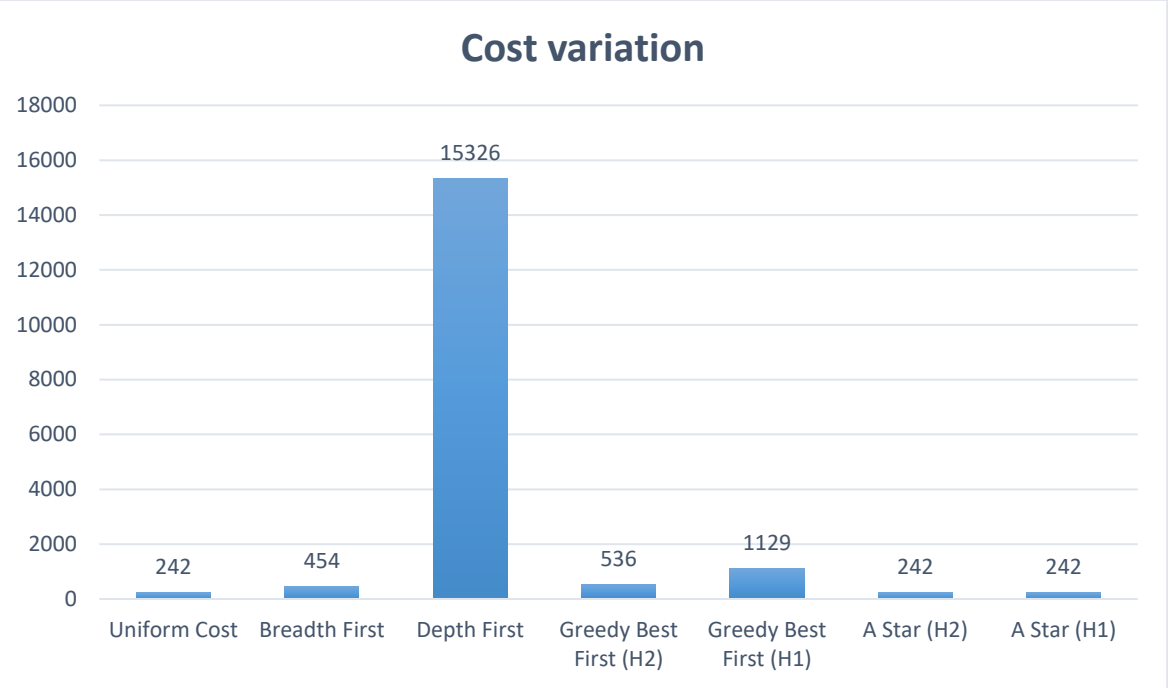


Figure 8: Cost variation

By comparing both difference in nodes and in cost we can see that for example using the **Uniform Cost** algorithm when adding cats the amount of nodes is increased a lot but the cost doesn't increase a lot, that's because it uses a priority queue which gives the minimum cumulative cost the maximum priority and expands it, which will eventually generate the most efficient solution in terms of cost. It's the best algorithm that doesn't use heuristics.

Breadth first: Works similarly to uniform cost but doesn't use the priority queue, it expands every breadth of the search which doesn't guarantee it to be the optimal solution in terms of cost just the shortest.

Depth First: Since it uses a LIFO-Queue the first element inserted and extracted is the first, which results in a tree which explores the leftmost branch first until it finds the solution. This means that it will generate much less nodes, but the solution will have a much higher cost.

Greedy Best First (H2): Uses the Heuristic function described before (page 3), since it is more restrictive, we can see that increasing the number of cats the cost doesn't increase too much, in fact it has the same cost as the other heuristic, but the number of generated nodes is much smaller since the heuristic is more restrictive.

Greedy Best First (H1): Uses the Heuristic function: Manhattan distance, which expands the node that is closer to the goal. Since it is greedy best first, it only considers the heuristic value and chooses the lowest, not considering the cost of the path. Therefore, the cost is higher than with the other heuristic.

A Star (H2): Uses the Heuristic function described before (page3), since it is more restrictive, we can see that increasing the number of cats the cost doesn't increase too much, in fact it has the same cost as the other heuristic, but the number of generated nodes is much smaller since the heuristic is more restrictive.

A Star (H1): Uses the Heuristic function: Manhattan distance, just as Greedy Best first but this algorithm also takes into account the cost of the path based on the A* Score which is equal to the cost + heuristic. And expands the node with the lowest A* Score, therefore, compared to the Greedy Best First using the same heuristic, it has a lower cost.

Overall, increasing the amounts of cats as we did with the size of the problem increases the number of nodes and cost.

3.4 Heuristics:

As we explained before, we have defined 2 heuristics that are:

- Heuristic 1: For a state, the value will be the Manhattan distance between its position and the final position, without taking care of any cheese. It is not very restrictive.

- Heuristic 2: For a state, checks every combination of the segments produced by the remaining cheeses and the exit, and selects the maximum segment, and given this, returns as value this minimum segment + distance from the agent to this cheese. If there is no cheese returns the Euclidean distance to the exit.

These are the data obtained to study how choosing different heuristics for a problem affect the solutions:

Data: For a maze with just 1 cat and seed 0. These charts show the data obtained with different sizes for the different algorithms and heuristics.

Table 7: Heuristic comparison

Greedy (H2)	Cost	Generated nodes	Expanded nodes	Explored nodes	Time(ms)
Size 10	36	263	100	187	13
Size 50	164	4259	1650	3894	207
Size 80	292	2896	1122	2347	146

Greedy (H1)	Cost	Generated nodes	Expanded nodes	Explored nodes	Time(ms)
Size 10	78	1031	418	953	144
Size 50	310	14098	5600	13743	1525
Size 80	1136	123954	48985	123288	11901

Difference:

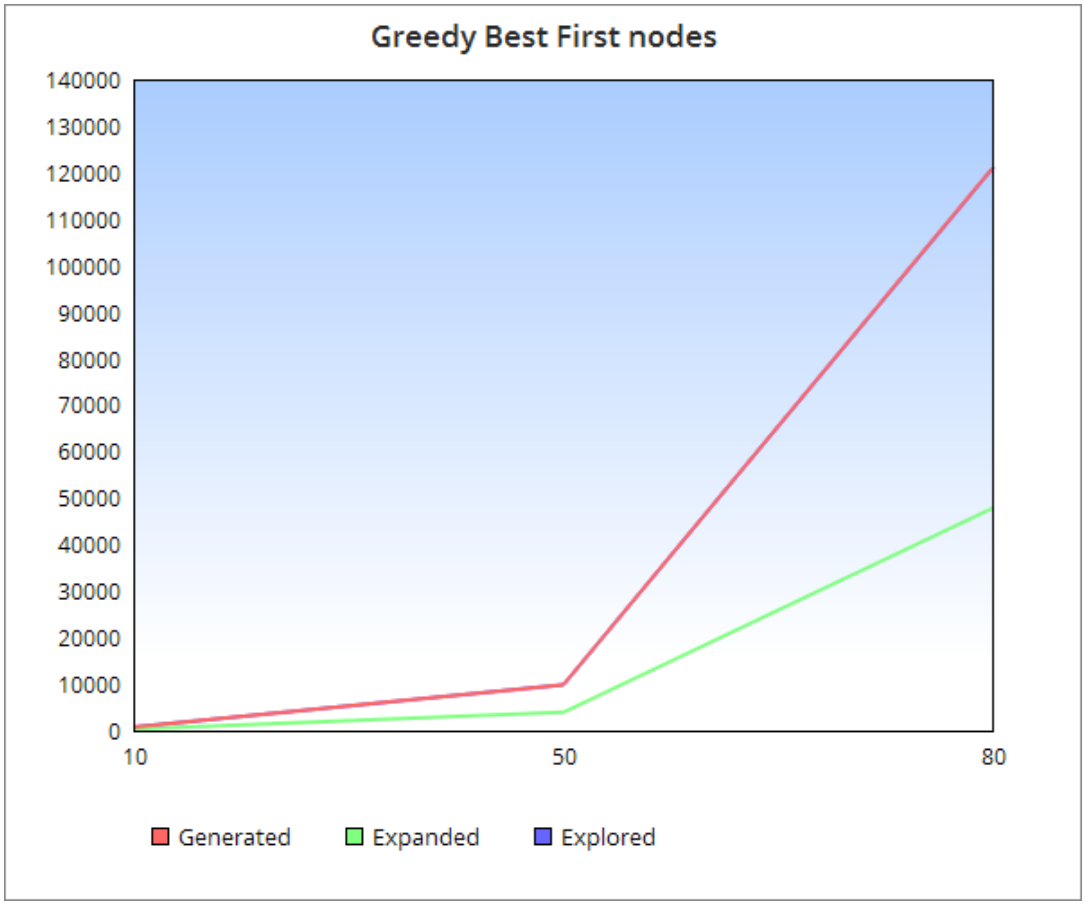


Figure 9: Greedy Best First nodes

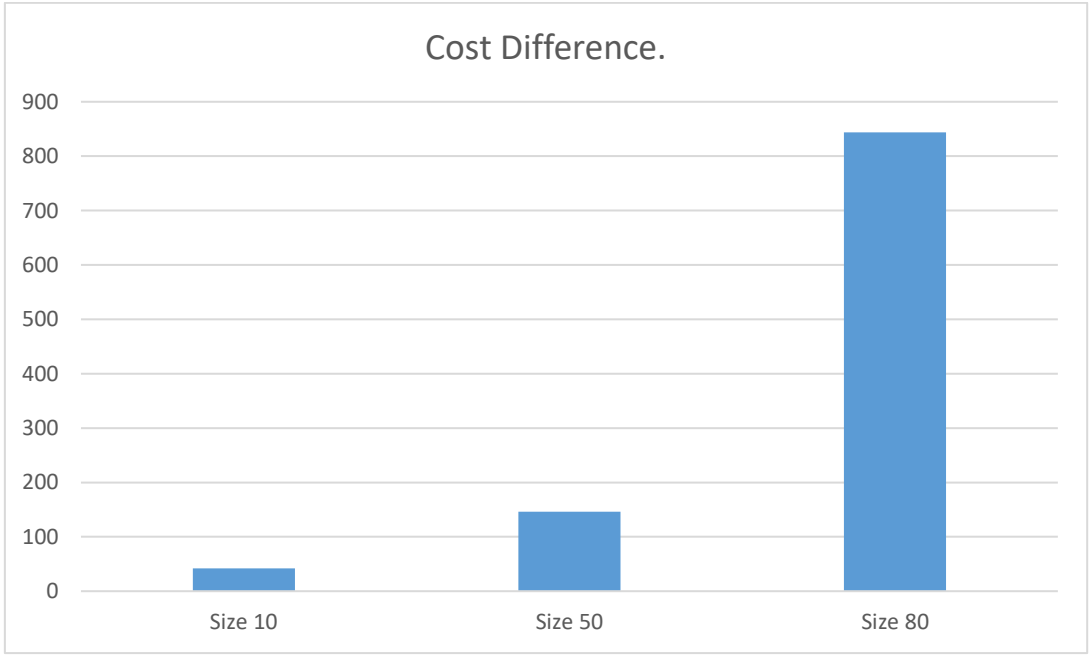


Figure 10: Greedy Best First Cost difference

We observe that as the size of the problem increases the cost and nodes increases as well, this is because a lot of nodes might have the same heuristic value and since this algorithm doesn't take into account the cost the solution is not optimal and that's why the cost increases a lot. We chose a good heuristic because comparing the result we see that the difference in cost and nodes is huge which indicates that there's a big improvement from using an heuristic to another.

Now using the A* Algorithm:

A*(H2)	Cost	Generated nodes	Expanded nodes	Explored nodes	Time(ms)
Size 10	30	454	180	370	13
Size 50	118	9576	3749	8967	47
Size 80	234	20848	8233	19979	122

A*(H1)	Cost	Generated nodes	Expanded nodes	Explored nodes	Time(ms)
Size 10	30	1131	461	1010	40
Size 50	118	23282	9147	22304	1421
Size 80	234	96021	37803	94339	5202

Difference:

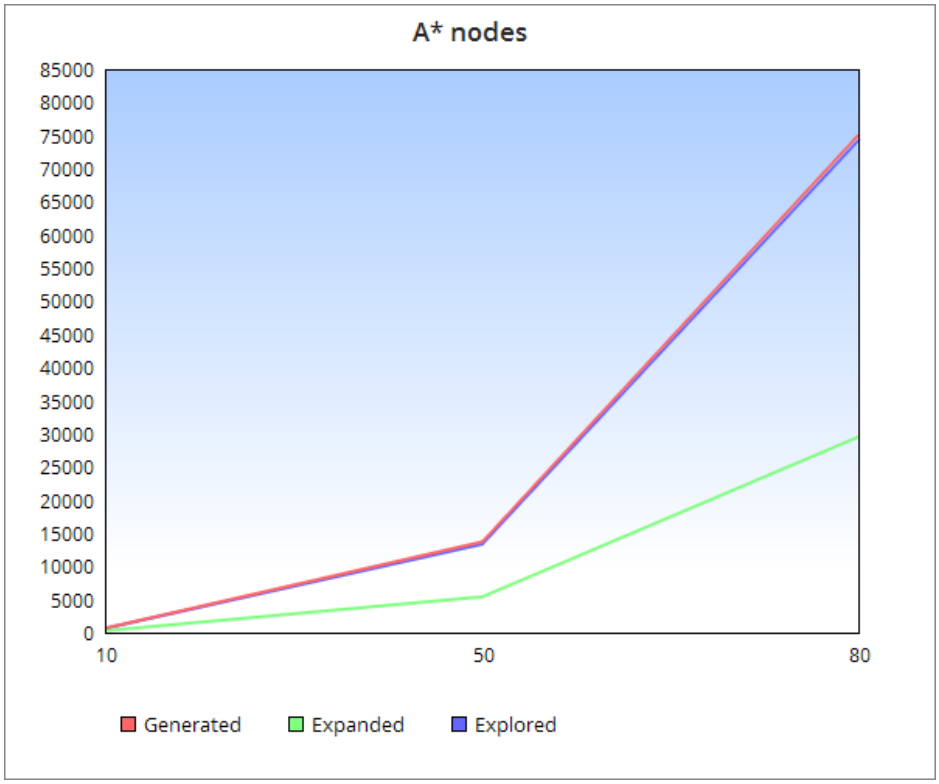


Figure 11: A nodes*

We can see that as the size increases the difference between heuristics increases, this is because the bigger the problem is, the worse the first heuristic is, since it will have to expand and generate a lot more nodes to arrive to the same solution that the algorithm finds using the other heuristic, that generates, expands and explores lesser nodes. With a small problem the difference was not so big, so we needed to try at least 2 more sizes to see if our heuristic was a good one and indeed it is. The cost of the solution is the same because it always finds the same solution.

3.5 Action priority:

Another important fact to take into account is the order in which the algorithms will check for possible actions. In order to improve the performance, we have to design an order of actions, giving more priority to the ones that we think might be more important for this problem. The set of actions is: {UP, DOWN, LEFT, RIGHT, EAT}. Given that in the problem the agent should eat a cheese whenever it is on it, it is clear that the EAT action should be the first selection. Moreover, the maze starting point is on the top side, and the maze end point is on the bottom, so it could be better to give preference to the DOWN movement than the UP, because it is more probable that the agent goes down more times than up since the exit is also down in the maze. LEFT and RIGHT actions don't seem to affect much at all.

To see how it affects the performance, the algorithm chosen is the Depth-First, as this algorithm takes the first action it sees. So, having a very bad ordering of actions will make the algorithm to follow unnecessary paths that could be avoided having other distribution.

- Depth-First with bad action set: {UP, RIGHT, LEFT, DOWN, EAT}
- Depth-First with better action set: {EAT, DOWN, RIGHT, LEFT, UP}

This is changed in the *getPossibleActions* method of the *MazeProblem*, considering that inserting an action into the set of possible actions using the add method puts it in the last position of the list. Because of this, the code of the selection is inverted to have the preferred action first taken.

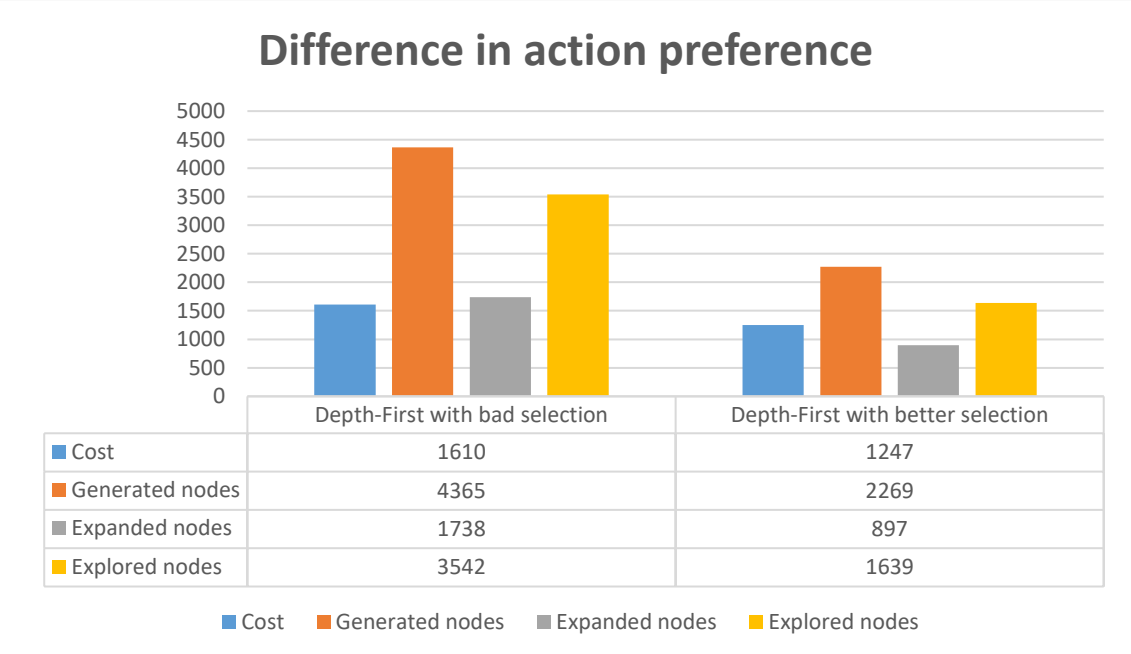


Figure 12: {Maze: 20, 0, 0}

5. Conclusions:

If we had to choose a non-informed search algorithm it would be the uniform-cost one since it has been proved that it's the best of them in terms of returning the least expensive solution, also in a short time, which is what we usually look for, even though it generates, explores and expands more nodes than the other algorithms.

If we take a look at the algorithms that use a heuristic, it is clear that the more restrictive the heuristic is, the better results we get. That's why in order to achieve the best performance it's very important to find the best and most restrictive heuristic. And that the best one is the A* algorithm, since it also takes into account the cost it provides us with the best solution.