Report on the first assignment

Artificial Inteligence(CC2006), FCUP – University of Porto

A Study of search algorithms using the 15-puzzle game as a base

Gonçalo Couto (201408558) & Tiago Bem (201504412)

& Nuno Rocha (197703976)

March 1, 2019

1 Introduction

1.1 Search Problems, definitions and descriptions of a search problem.

A search problem can be defined using a set of components such as, an initial state, actions/next state function, a goal state, and a path cost:

1. The **initial state**, in the 15-Puzzle game, represents the starting configuration of the board. It has all the tile placements in the desired places plus the empty space represented by the number zero {0}.

2. The **actions/next state function** are the set of moves in which a particular tile can move, in other words, given an x state, the function should return a list of Z states that contain y such that y can be reached through x.
In the 15-Puzzle game we would only move the blank space or {0} returning the set of moves, child-nodes, of this tile. In some cases as any regular tile based game the set would be limited as some positions would be unreacheable.

3. The **goal state**, the final configuration, determines whether a given state is a goal.
In the N-Puzzle game, we use this condition to check if a given puzzle configuration matches the given final configuration, it serves as a flag in a way but with a higher importance.

4. The **path cost**, a variable or function, that assigns a cost to every move possible, in a given set of moves.
In the 15-puzzle game this would corespond to a certain heuristic calculated during the program, in our case we used either the Manhattan distance or the number of tiles that were out of place in the board.

1.2 **Ways to solve search problems, search algorithms**

There are various ways one can use to solve a search problem but they mainly consist on using the same strategy. We identify our goal, in this case as explained above it should be the final state, and along the way to the goal child-nodes are being created, of this child-nodes the ones that have the better “cost”, in other words, the nodes that have a higher chance of taking us to the goal, are picked.

An optimal solution to our problem would be the solution with the least cost to do.

In the 15-puzzle game it very much works this way, we find the optimal solution to a problem by analizing the cost associated with each move or node and find the best path from starting configuration to end configuration. The best path would be the solution that requires the least amount of switches of the blank space or {0} with the rest of the tiles.

2 **Search Strategies**

Having defined the concept of a search problem in 1.1, and ways to solve search problems in 1.2 we now advance to some search strategies used in solving the 15-puzzle game.

As seen before, a node has a series of components:

1. A **starting configuration**, which has all of the tiles placements.
2. **Actions/next state function**, used to create child nodes.
3. **The goal state**, the board configuration we want to reach.
4. A **cost**, associated to each node.

In the 15-puzzle game, a node will consist of six components:

1. A **board**, representing the tile placements in that node.
2. A **father-node**, which is a pointer to the previous node that genereated the current node.
3. A **next-node**, which is a pointer to the next node, which is the node in the data structure that follows the current node.
4. A **cost**, that as seen before, represents the heuristic value of the node.
5. A **depth**, which represents the node placing in the graph, or state space.

The state space, in other words, all of the nodes that could be created throughout the search represent a graph with initial configuration being the root. We iterate over this graph in order to find our solution. If there is a solution in the state space we will eventualy find it, the reasoning behind the choice of which node to expand in each iteration depends on the search strategy being used.

2.1 **Blind search**

A **blind search**, also called uninformed search, are types of search strategies that have no information about the search space, they can only destinguish the current state up to a point in the search from the goal state.

This category of search algorithms includes the **BFS**(**Breadth-first search**), **DFS**(**Depth-first search**), **IDFS**(**Interative deepening depth-first search**).

2.1.1 **Breadth-first search**

A breadth-first search or BFS is a type of uninformed search strategy that goes through the tree, or graph, level by level, visiting all of the nodes starting with the top level, the root of the tree, and so on until it finds the solution.

This strategy is complete, in other words, if there is a solution to the problem it will be found, and optimal as long as the shallowest solution is the best solution.

A problem to this is that the breadth-first search when trying to find the solution keeps all of the leaf nodes in memory, which requires a substantial amount of memory when trying to do the search in anything more than a very small tree.

The time complexity of a breadth-first search is $O(n^d)$ where n is the branching factor and d is the depth of the solution.

We use a queue when using a breadth-first search.

An example on how the breadth-first search works is as follows:

1. First we remove node t from the top of the queue. This is our starting node.
2. Check if node t is the solution. If it is we return node t and stop the search, if not we go to step 3.
3. Expand node t
4. Insert node t's child nodes into the queue
5. Repeat step 1

2.1.2 **Depth-first search**

A depth-first search is a type of uninformed search that goes through the tree branch by branch, going all the way down to the leaf nodes at the bottom of the tree before trying the next branch.

This type of search strategy is much more efective, memory wise, than the breadth-first-search as it only needs to store a single path from the root of the tree down to the leaf node. However, it is incomplete, since it will keep going down one branch until it finds a dead-end, and non-optimal, if there is a solution in the at the second level in the second branch and a solution in the fifth level in the first branch, it will return the solution in the fifth level first.

The time complexity of the depth-first-search is $O(n^{md})$ where b is the branching factor and md is the maximum depth of the tree. It's space complexity is only b*md.

We use a stack when using a depth-first search.

An example on how the depth-first-search works, using a stack as a data structure, is as follows:

1. Remove node t from the top of the stack.

2. Check if node t’s state is the goal state. If it is, return node t and stop. If not, go to step 3.

3. Expand node t.

4. Insert node t’s child nodes into the top of the stack.

5. Go to step 1.

2.1.3 **Interative deepening depth-first search**

Iterative deepening depth-first search does repetitive depth-limited searches, it starts with a limit of zero and increments once each time.

It has the space-saving benefits of the depth-first search but is also optimal and complete, since it will visit every node on the same level first before continuing on to the next level in the next turn when the depth is incremented.

The time complexity in the iterative depth-first search is $O(b^d)$ where b is the branching factor and d is the depth of the solution. The space complexity of the algorithm is O(b*d).

2.2 **Informed search**

While an uninformed search has no kind of information regarding the search space, an informed search uses problem-specific knowledge in order to distinguish between any two different non-goal states. This way it can choose to expand nodes that have a higher probability of reaching a goal state.

The probability associated with each node is created using an evaluation function f(n) that tells the algorithm which node should be expanded first based on the current problem-specific knowledge.

We can proceed to find a solution using an informed search with a **priority queue** as a data structure, the difference between a priority queue and a regular queue is that

the objects associated to the priority queue have a priority associated with them, this will affect the way in which objects are inserted and ordered into the linked-list(priority queue), via a **comparator**.

A **comparator** in a priority queue is the function that will tell the list in which order to place the objects by analyzing a common trait associated with each object(node).

Every time we remove an object(node) from the list we do this by removing the head of the list. Since the best object(node) to remove will always be the one at the head of the list, we can say that every object(node) we remove is the best possible object(node) in order to reach the goal state.

An example on how this works would be:

1. Remove node t from the front of the priority queue
2. Check if node t is the solution, if it is stop the search. If not, go to step 3
3. Expand node t
4. Place node t's child nodes into the priority queue they will be ordered appropriately
5. Repeat step 1

2.2.1 **Heuristics, definitions and heuristic algorithms**

An heuristic is something that can be used to evaluate which possible course of action should be taken in order to find a solution to a problem, it may not be the best solution, but can be the fastest.

A heuristic is still a kind of an algorithm, but one that will not explore all possible states of the problem, or will begin by exploring the most likely ones.

Not always having an heuristic defined helps in finding the solution to the problem, in cases where we aren't searching for the best solution but for the solution that fits some constraint, a good heuristic can find the solution in a short time but fails if the solution is found in any course of action it chooses not to do.

In the 15-puzzle game we considered two heuristics:

1. The number of tiles excluding the empty space {0} that were out of place.

2. The Manhattan distance, which is the sum of the distances between the position of each tile and the position they should be(given to us by the final state or solution).

2.2.2 **Greedy best-first search**

A greedy best-first search is a search strategy that will pick the node that is closer to the solution and expand it, assuming that this will provide the quickest path to the solution. It evaluates nodes by using the heuristic function f(n) = h(n).

This algorithm is not optimal, is incomplete(it does not check repeated nodes throughout the path) and could go down an infinite tree branch until it hits a “dead-end” or until the cost of expanded nodes exceeds the cost of nodes waiting to be expanded, before changing its course.

This strategy can be useful in certain problems, if given a good heuristic it will produce good spacial and time complexities. Our implementation of this strategy for the 15-puzzle game used both the Manhattan distance as well as the number of tiles out of place as heuristic functions.

Spacial complexity: $O(b^m)$

Time complexity: $O(b^m)$

2.2.3 **A* search**

The A^* search evaluates node by combining the cost of reaching a node n, with the heuristic function h(n), thus producing, f(n) = cost + h(n)

Spacial complexity: $O(b^d)$

Time complexity: $O(b^d)$

In our implementation of the A^* strategy, we used both the same heuristics as in the greedy strategy.

3. **Implementation**

3.1 **Programming language used**

The programming language used by us during the course of this assignment was java, we choose this language mostly because of how used to it we are in comparison to other programming languages. Not only that but we found that it has a rich API when it comes to data structures and fortunately for us there were some good functions, some of which described during the course of this report, that really helped our implementation of the game.

3.2 **Data structures**

In our implementation of the 15-puzzle we used a priority queue, a regular queue and a stack. When trying to do the informed searches we considered using a hash-map structure to keep hold of visited nodes to try and lower redundancy. We later discarded this idea because this would imply using much of the system memory thus making the algorithm less dynamic, and, in certain cases, could fail.

3.3 **Code structure**

Our code is structured in a clean way, we have the main function in one file and each of the algorithms in its own different file. We choose to do this because this way everything is more understandable, we don’t have a lengthy code file and not only for the person reading our code, but for us it is much easier to find pieces of code and do debugs.

4 Results

For the tests preformed in this section we will use the test configurations given to us by the professor of the discipline.

The tests were performed on a Linux machine using 4Gb of RAM and an i5 processor clocked at 2.8 GHz.

We will use depths of 10, 12, 13.

Heuristic1 → number of tiles out of place.

Heuristic2 → Manhattan distance.

Test1:

Initial configuration = {6, 12, 0, 9, 14, 2, 5, 11, 7, 8, 4, 13, 3, 10, 1, 15}
End configuration = {14, 6, 12, 9, 7, 2, 5, 11, 8, 4, 13, 15, 3, 0, 10, 1}

Algorithm	Time(nanoseconds)	Memory space(Mb)	Solution Reached	Depth/ cost
BFS	69056350	6,8421	Yes	10
DFS	100106764	6,8420	Yes	10
I-DFS	83178804	6,8430	Yes	10
Greedy(Heuristic1)	37212551	2,5630	Yes	10
A*(Heuristic1)	34810669	2,5620	Yes	10
Greedy(Heuristic2)	40505178	2,5630	Yes	10
A*(Heuristic2)	37204197	2,5630	Yes	10

After the conclusion of the first test, by analizing the data we can see that the algorithms A* and Greedy have a lot better results than the rest. We can clearly say that an informed search has a greater advantage over an uninformed search, both memory and time wise.

Test2:

Initial configuration = {13, 11, 15, 4, 8, 9, 1, 5, 12, 14, 0, 2, 7, 10, 3, 6}

End configuration = {13, 11, 4, 5, 8, 0, 14, 15, 12, 1, 3, 2, 7, 9, 10, 6}

Algorithm	Time(nanoseconds)	Memory space(Mb)	Solution Reached	Depth/ cost
BFS	150614932	15,6450	Yes	12
DFS	164720095	15,7154	Yes	12
I-DFS	127103024	18,6984	Yes	12
Greedy(Heuristic1)	45390055	2,8421	Yes	12
A*(Heuristic1)	35532214	2,5628	Yes	12
Greedy(Heuristic2)	38859652	2,8421	Yes	12
A*(Heuristic2)	35958617	2,5627	Yes	12

Once again we see the informed search algorithms ruling the results with half the memory cost, and half the search time of the uninformed ones.

Test3:

Initial configuration = {6, 12, 0, 9, 14, 2, 5, 11, 7, 8, 4, 13, 3, 10, 1, 15}
End configuration = {14, 6, 12, 9, 7, 2, 5, 11, 8, 4, 13, 15, 3, 0, 10, 1}

Algorithm	Time(nanoseconds)	Memory space(Mb)	Solution Reached	Depth/ cost
BFS	166355900	23,5159	Yes	13
DFS	171053312	19,2838	Yes	13
I-DFS	145038156	24,6519	Yes	13
Greedy(Heuristic1)	30345658	2,5620	Yes	13
A*(Heuristic1)	31833883	2,5620	Yes	13
Greedy(Heuristic2)	27595863	2,5621	Yes	13
A*(Heuristic2)	28807378	2,5621	Yes	13

After concluding the last test(with the highest depth of them all) we can see once more that the informed search algorithms rule the table.

5 Conclusions

After testing we can conclude that the informed search algorithms are far superior than the uninformed ones in the resolution of the 15-puzzle game. With two simple heuristics, the misplaced tiles and the Manhattan distance, it is possible to reduce the time required to search the state space and find the solution with the added effect of decreasing the memory usage. Not only this, but as seen in the test results, with the increase of depth, uninformed algorithms struggle even more, whereas with the informed ones we see little to no difference between results.
We have concluded that the trophy goes to the A* algorithm in solving the 15-puzzle game, even though in some cases with the difference of heuristic it stood behind the greedy algorithm it showed the best memory usage and time consumption as well as nodes generated and nodes used to find the goal.

6 Bibliography

Dutra I. Estrategias não informadas de Procura/Busca; 2019.
Dutra I. Estrategias informadas de Busca; 2019.
https://www.ics.uci.edu/~welling/teaching/271fall09/InfSearch271f09.pdf
http://www.cs.cornell.edu/courses/cs4700/2011fa/lectures/03_InformedSearch.pdf
http://cse.unl.edu/~choueiry/S03-476-876/searchapplet/