

Optimizing the 15-Puzzle using Branch & Bound

Final Project - Introduction to Optimization 89589

Sara Spagnoletto

1. Introduction

The 15-Puzzle is a sliding puzzle developed by American puzzler Sam Loyd in the 1870s. In its original form, the puzzle consists of fifteen square tiles numbered 1 through 15, placed in a 4x4 tiles frame, leaving one unoccupied position. The puzzle may also be referred to as the 16-puzzle, alluding to its total tile capacity.

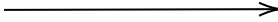
Tiles in the same row or column of the open position can be moved by sliding them horizontally or vertically, respectively. Therefore, a legal move consists of sliding a tile adjacent to the empty space towards the empty space. The solution state is where the 15 blocks are placed as shown in Figure 1, with the lower right corner left empty.

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	

Figure 1

Starting from a scrambled placement such as Figure 2, the objective of the puzzle is to use a sequence of legal moves to reach the solution state.

1	3	4	15
2		5	12
7	6	11	14
8	9	10	13



1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	

Figure 2

The 15-puzzle has been an object of great mathematical interest since its invention. The classic problem supplies modeling algorithms involving heuristics. Commonly used heuristics for this problem include counting the number of misplaced tiles and finding the sum of the taxicab distances between each block and its position in the goal configuration.

1.1 Project Objective

The goal of this project is to study heuristics towards an optimal solution to the 15-puzzle. The optimality of a particular sequence solution is directly proportional to the number of moves in such a sequence. When looking into the heuristic for solving the puzzle, two factors are considered:

- The time complexity of the method.
- The optimality of the solution outputted.

Ideally, the heuristic technique found is a balanced trade-off between a fast algorithm and a relatively short (=optimal) sequence solution.

The solution presented uses the method of Branch and Bound, with an iterative tree traversal. Both depth-first-search and breath-first-search will be examined and compared. The **branching** refers to the possible moves reachable from a given state. The **bound** attempts to find a minimal number of moves from the given state to the solution. For the bound, different heuristics have been developed over the years, of which two are compared and studied below.

2. The Solution Approach, Highlights and Insights

2.1 The Branch & Bound Method

The branch and bound method is a solution approach that can be applied to a number of different types of problems. It is an algorithm design paradigm for discrete optimizations. A branch-and-bound algorithm consists of a systematic enumeration of candidate solutions by means of state space search: the set of candidate solutions is thought of as forming a rooted tree with the full set at the root. The algorithm explores branches of this tree, which represent subsets of the solution set. The algorithm depends on the efficient estimation of the lower and upper bounds of regions/branches of the search space. If no bounds are available, the algorithm degenerates to an exhaustive search. It is particularly useful with typically exponential problems in terms of time complexity that may require exploring all possible

permutations in the worst case.

2.2 Why to use Branch & Bound for the 15-Puzzle

1. The 15-Puzzle is a discrete optimization problem. This means that some or all of the variables used are restricted to be discrete variables. Let us consider to be a feasible solution a sequence of moves from the starting state to the goal state, and the variables the possible moves for each state. Since the possible moves are 4 (moving the empty tile up, right, down, bottom), the 15-Puzzle is a discrete optimization problem.
2. The 15-Puzzle is a combinatory optimization problem. Combinatorial optimization consists of finding an optimal object from a finite set of objects, where the set of feasible solutions is discrete or can be reduced to a discrete set. Combinatory optimization is a process to find the maxima or minima for the objective function. In the 15-Puzzle the set of feasible solutions is all possible move sequences from the starting state to the goal state. It is clear that the set is finite as well as the objective to minimize the length of the sequence.
3. For the 15-Puzzle exhaustive search is not tractable. As mentioned above the Branch & Bound method is particularly useful for exponential problems, as it limits the amount of checks by bounding. The 15-Puzzle has an exponential set of feasible solutions, and therefore not tractable by exhaustive search or brute force. It requires a specialized algorithm that quickly rules out large parts of the search space or an approximation algorithm, such as the Branch & Bound.

2.3 How to use Branch & Bound for the 15-Puzzle

Branch

For the 15-Puzzle the branching on the search tree is based on the possible reaching states from a given one. Specifically, given a node representing some tiles' position, we will branch into the 4 new positions reachable from it. The children of the node x is the state reachable from x by one legal move. Consider the move as a move of empty space. Hence there are four possible moves, moving up, down, right and left.

There are three types of nodes involved in Branch and Bound:

1. Live node: is a node that has been generated but whose children have not yet been generated.
2. E-node: is a live node whose children are currently being explored. In other words, an E-node is a node currently being expanded.
3. Dead node: is a generated node that is not to be expanded or explored any further. All

children of a dead node have already been expanded.

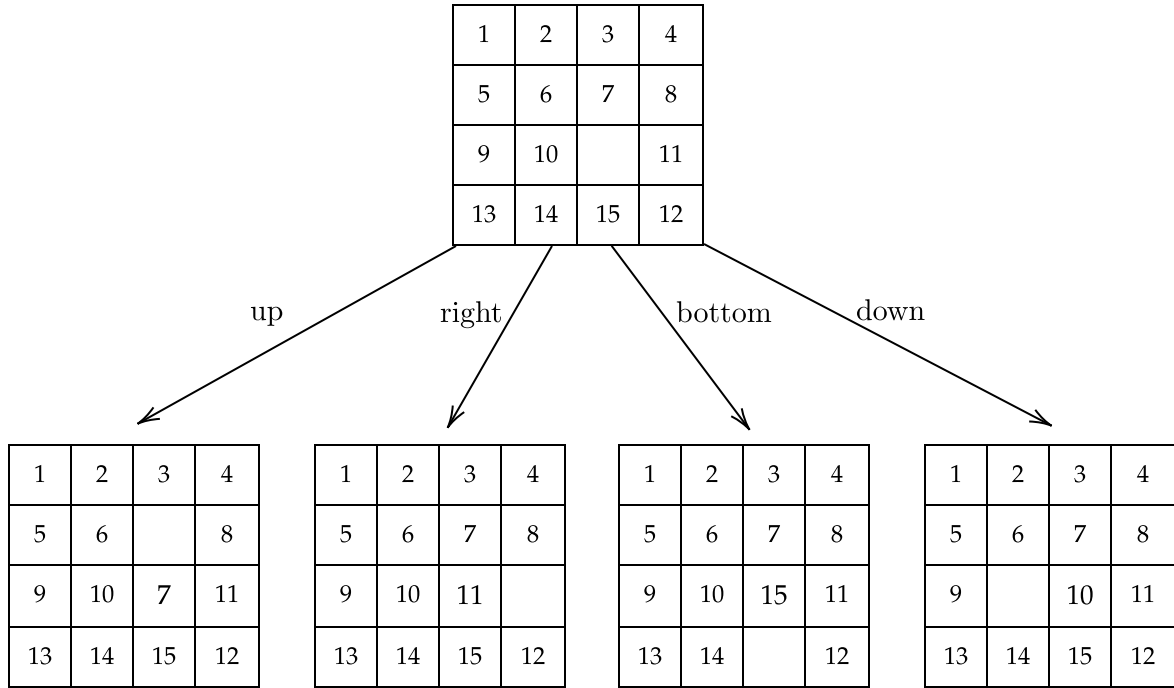


Figure 3

Bound

The goal is to use bounding functions to avoid generating subtrees that do not contain an answer node. Since the 15-Puzzle is a minimization problem, the bound given to a specific node refers to a minimum possible solution for that state. For a node representing some tiles' position, its bound will be the minimum number of moves needed to reach the goal state. Of course, this bound number doesn't necessarily describe the optimality of a node in a precise manner. Even if node x has a smaller bound than node y , node y could still ultimately be more optimal than node x when it comes to the final solution. This is to say that the bounds, in this case, are merely approximations that guide the tree into branching priorities, not absolute truths.

Given a node representing some tiles' position, the goal is to calculate a minimum to the number of moves needed to reach the goal position. For this purpose different heuristics can be used. The heuristics are based on the expected additional computational effort (cost) to reach a solution from the current live node. This can be achieved by using an estimate function $\hat{g}(x)$ instead of actually expanding the nodes.

Let $\hat{g}(x)$ be an estimate of the additional effort needed to reach an answer from node x . Node x is assigned a cost bound using a function $\hat{c}(\cdot)$ such that:

$$\hat{c}(x) = h(x) + \hat{g}(x)$$

Where $h(x)$ is the cost of reaching x from root.

Estimating $\hat{g}(x)$

As said above, the goal of $\hat{g}(x)$ is to supply a lower bound on the number of moves needed to reach a goal state from the current live node. Note that finding a lower bound is not a difficult task, for example $\hat{g}(x) = 0$ would always be correct. The difficulty of the task is to find a good approximation, as high as possibly close to the actual optimal solution.

There are various heuristics for calculating $\hat{g}(x)$ developed over the years:

- Manhattan Distance: For each tile the number of grid units between its current location and its goal location are counted and the values for all tiles are summed up. This heuristic is easy to compute but it is not very powerful.
- Walking Distance: Considerably more powerful than the Manhattan Distance while not using much memory.
- Pattern Databases: The 78 Pattern Database heuristic takes a lot of memory but solves a random instance of the puzzle within a few milliseconds on average. An optimal solution to the 80 moves antipodes takes a few seconds each.

For this project is used the Manhattan Distance, example in Figure 4.

Manhattan Distance = 8

1	3	4	15
2		5	8
9	10	11	12
13	7	6	14

Figure 4

Problem encountered with mentioned cost function

An unfortunate problem encountered running the simulation using the above cost function is staggering complexity. Since the cost is dependent on the depth of the level, the algorithm will mostly choose as E-node old vertices high on the tree.

The average optimal solution for a random tiles configuration is 60, which means it would

require 317,373,604,363 open nodes before reaching it. (and this is the optimal solution, an approximation could be more). This gigantic number is unapproachable both for time and memory complexity by a simple PC. Of course, it would be ideal to include the level as weight into the cost function. Expanding high-level nodes means checking more configurations overall, instead of diving deep into one subtree, providing a more optimal solution. For the purpose of this project, due to technological limits, the cost function used will be simply:

$$\hat{c}(x) = \hat{g}(x)$$

This research still provides interesting insights on this optimization problem, and the use of different traversal heuristics.

2.4 Practical Solution and Code

To put the theory above into practice it needs to be implemented a Tree structure where:

- After every expansion of the tree, calculate the cost for each new Live Node.
- For every step of the traversal, chose as E-node the one with minimum cost.

Pseudo-code

```
Branch_and_Bound(initial_tiles):
    liveNodes = PriorityDataStructure()
    boards_seen.append(initial_tiles)

    # create root node
    root_cost = calculate_cost(initial_tiles)
    root = Node(initial_tiles, root_cost)
    liveNodes.push(root)

    while not liveNodes.empty():
        min_node = liveNodes.pop()
        # if this node is the answer end
        if min_node.cost == 0:
            return

        # else do expansion
        for each new node x (up, right, left, bottom):
            x = Node(new_tiles, new_cost)
            if x not in boards_seen:
                liveNodes.push(x)
```

The technique and code above are quite straightforward. Instead, an interesting question to be asked is how to implement the Priority Data Structure, whether it should be a Stack or a

Queue. Note that since we are always popping the node with the minimum cost, the only difference is how will it react given a tie between costs.

- Stack - given a tie it will return the node entered lastly (i.e. on the lowest level in the tree).
- Queue - given a tie it will return the node entered first (i.e. on the highest level in the tree).

For ties between nodes on the same it purely random and has no effect on the study.

These two methods are in practice referring to whether the tree traversal should use BFS or DFS.

In general, branch & bound has no predefined method, and can be applied to both of them.

In this project both heuristics methods are examined and their results compared.

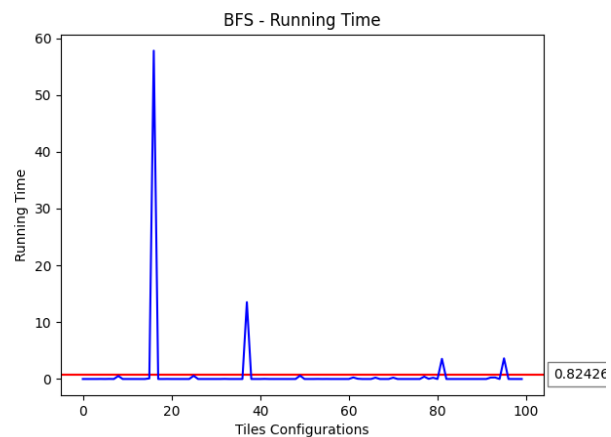
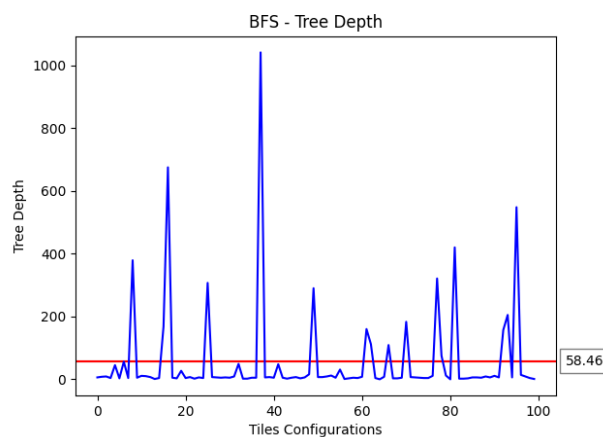
2.5 BFS vs DFS Solutions

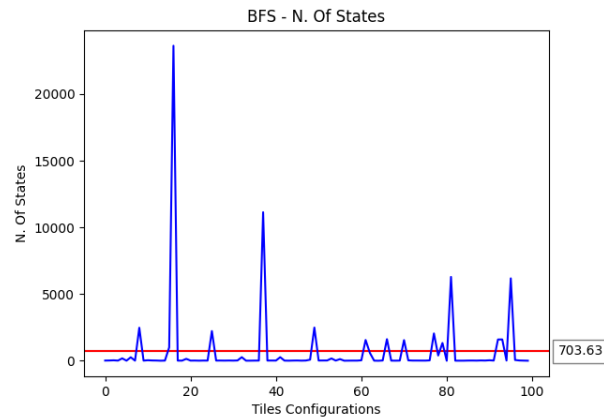
BFS

In the BFS test given a tie between node costs, the node on the lowest (the one entered later), was chosen, using a LIFO approach with a stack.

From a random sample of 100 starting tiles' position this method averaged:

- The depth of the tree before finding an answer: ~ 60 (i.e. the optimal answer has ~ 60 moves)
- The running time: $\sim 0.8s$
- The number of states visited: ~ 703



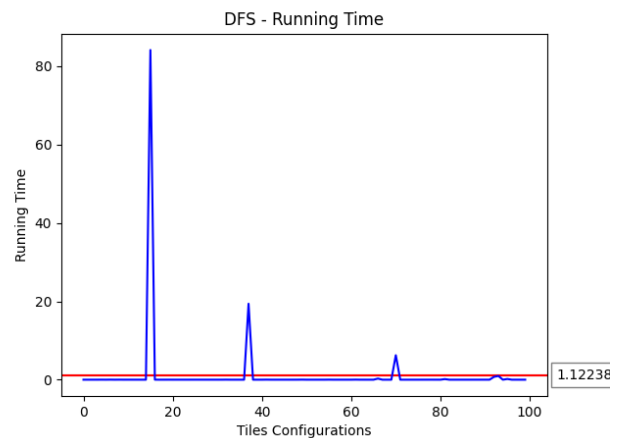
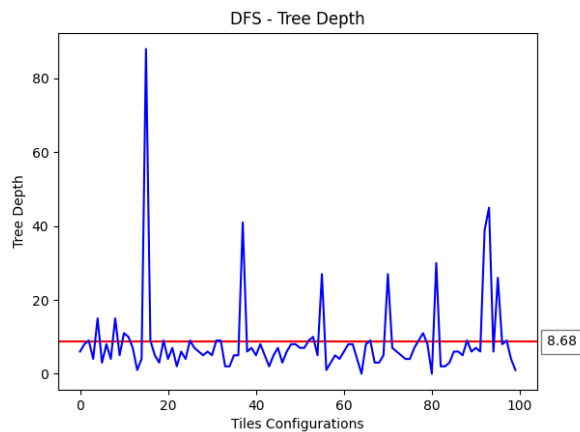


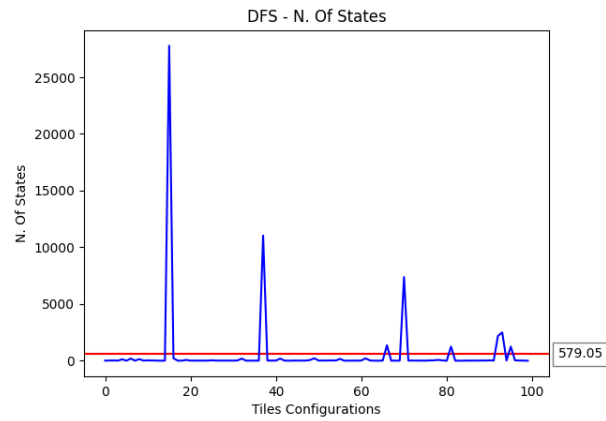
DFS

In the DFS test given a tie between node costs, the node on the highest (the one entered before), was chosen, using a FIFO approach with a queue.

From a random sample of 100 starting tiles' position this method averaged:

- The depth of the tree before finding an answer: ~ 8.7 (i.e. the optimal answer has ~ 8.7 moves)
- The running time: $\sim 1.1s$
- The number of states visited: ~ 580





Simple Implementation Example

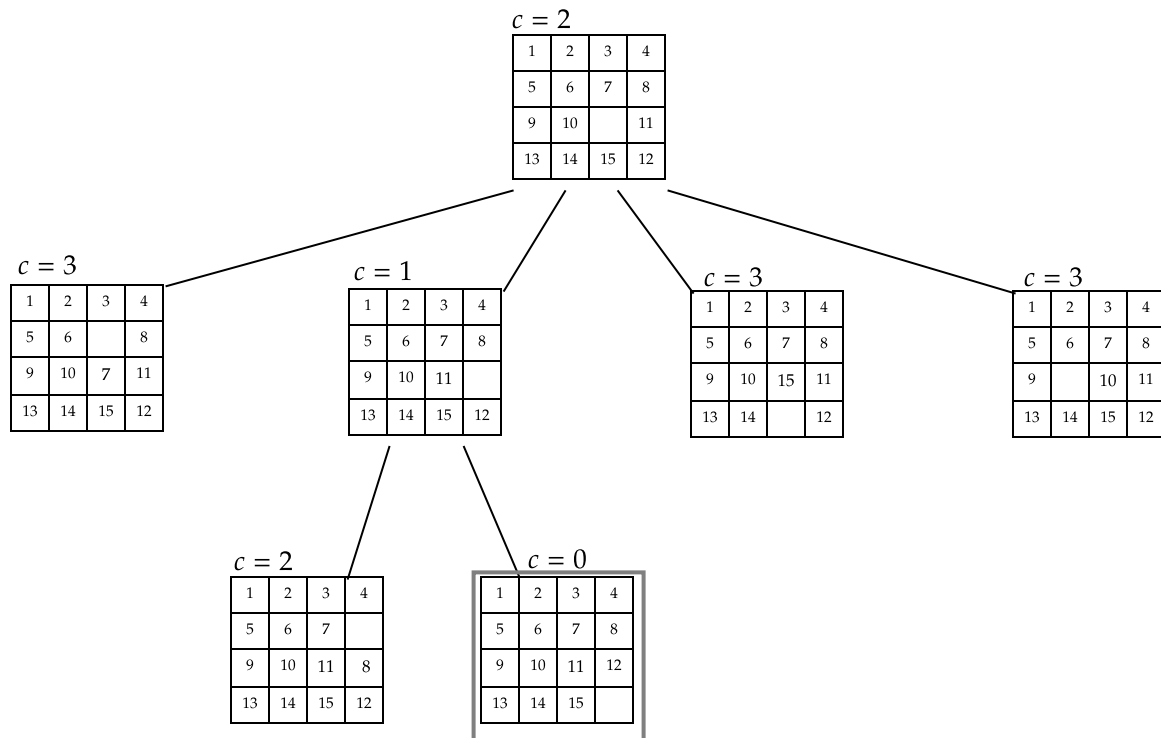


Figure 5

3. Conclusion

The main conclusion and interesting take out from the results of this project is the surprising

difference between using BFS and DFS. Indeed, the outcomes vary both in terms of the optimality of the solution and in terms of the time complexity used to reach that solution. The reason for this difference in results is due to the nature of the two methods.

The BFS method will always prefer to explore newly added nodes instead of nodes in previous levels. This entails that once an expansion of an E-node has started, it will likely continue going down that same subtree. Proceeding down without going back to upper levels will have benefits in terms of time complexity.

Since for the 15-Puzzle, an optimal solution (not approximated) takes an average of 60 moves, just to reach such a deep level using DFS will take an extraordinary amount of time. BFS traverses the tree quicker, which means it will likely find a correct solution eventually. The downside of this method is, of course, that the deeper the level of the solution is, the higher the number of moves to reach it, the less the solution is optimal.

The DFS method will prefer to explore old nodes instead of entering newly added levels. For this reason, the tree will more likely open horizontally quicker than vertically, therefore reaching deep nodes slower. As mentioned above this will impact negatively on the time complexity but it will improve the optimal solution. Since we are checking more options and more subtrees in parallel, the optimal solution will be better.

This difference also provides an insight into the trade-off of optimality and time complexity.

The project has taught me a lot both on a technical and on a personal level. On a technical level, I deepened my knowledge of the 15-Puzzle, how it works, how to solve it, how to find an optimal solution. In terms of the course subject, I learned a lot about the Branch & Bound technique for solving optimization problems. For example, when is best to use, coming up with the cost function, different heuristics advantages, and more.

On a personal level, I improved my researching and writing skills, merging a code implementation with researching ideas, testing the results, and drawing conclusions.

Sources

https://en.wikipedia.org/wiki/Branch_and_bound

<https://www.baeldung.com/cs/branch-and-bound>

https://en.wikipedia.org/wiki/15_puzzle

<http://www.cs.umsi.edu/~sanjiv/classes/cs5130/lectures/bb.pdf>

<https://www.geeksforgeeks.org/check-instance-15-puzzle-solvable/>

<http://kociemba.org/themen/fifteen/fifteensolver.html>

https://www2.seas.gwu.edu/~bell/csci212/Branch_and_Bound.pdf

https://scis.uohyd.ac.in/~wankarcs/index_files/pdf/Algo-2015-08.pdf