Artificial Intelligence and Expert Systems Project

Game of 15

Abstract

In this paper we analyze and compare the performance of common informed and uninformed search algorithms applied on the Game of 15

Matteo Belenchia

902765@edu.p.lodz.pl

Contents

The problem[1]

The game of 15 (or also the tile-game or 15 puzzle) is a sliding puzzle that consists of a frame of numbered squares in random order with one missing tile. The purpose of the player is to rearrange these tiles in an ordered state. There might be different definitions for a "ordered state", namely they differ about where the blank tile should be; in my implementation the ordered state has the blank tile in the last position of the board, right after the biggest number. The game also exists in a number of variants, which differ by the number of tiles. The program I developed can work with frames of any size, although for frames bigger than 4x4 it can become extremely slow or run out of memory more often.

The game itself is not always solvable, so a rule[2] must be implemented to distinguish solvable from non-solvable states, and it such cases the program returns -1. In the game of 15 there are $16! \sim 20.000 \cdot 10^9$ possible states, but only half of them are solvable.

Finding a solution to the puzzle is not necessarily hard, but finding the optimal, shortest, solution is NP-Hard, and the solution lengths range from 0 to 80 single moves (for the easier 3x3 game, this range is reduced to 0 to 31 single moves).

The time and space complexity of the algorithms employed on this problem depend on the branching factor, that is the number of successors generated by each state/node. In the game of 15 we have 4 corners from which only 2 moves are available, 8 edges from which 3 moves are available and 4 central tiles from which 4 moves are available. Since there are in total 16 tiles, we have on average a branching factor of:

$$\frac{4 \cdot 2 + 8 \cdot 3 + 4 \cdot 4}{16} = 3$$

This value will come up later, when we analyze the different algorithms.

The program

I developed this project in Python 3, mainly for the speed of development and to get to learn the language better, but in retrospect using a lower level language like C might have been better to achieve faster execution times (even though Cython, a Python implementation with C-like performance, also exists for this purpose).

The software can be installed on a UNIX/LINUX environment by running "`python3 setup.py install`" or "`python3 setup.py develop`" on the terminal, which will install the required dependencies (it is just the sortedcontainers library from which I use the SortedSet class)

[1] https://en.wikipedia.org/wiki/15_puzzle
[2] https://www.cs.bham.ac.uk/~mdr/teaching/modules04/java2/TilesSolvability.html

In the AI2018 folder the following files can be found:

Algorithms.py	It contains all the implemented algorithms
Board.py	It contains the board (state) implementation
Heuristics.py	It contains the heuristics, $g(n)$ and $f(n)$ functions
Main.py	This is the program entry point
Node.py	It contains the functions to create and expand a node
nodeSMA.py	It's a node implementation specific for SMA Algorithm
verifier.py	This is the second program, to verify any solution on any board

The program can be launched by running "`python3 main.py`" which will show a useful help screen with all the available commands. Then, after launching it again with the appropriate parameters, it will ask the user to input the required initial state information.

The second program is developed in `verifier.py` and can be launched by running "`python3 verifier.py`", at which point it will ask the user for the initial state information and for the solution to verify. At this point it will show a help screen of the available commands and wait user input for what to do next. The commands are briefly explained here:

step	It executes the next instruction on the solution string
jump n	It executes the next n instructions on the solution string
exec	It runs all the remaining instructions on the solution string
help	It shows again the help screen
exit	It terminates the program

The data structures

The suggested data structure for the project was a hash table which would contain, for a 4x4 puzzle, a 64bit key that would represent the board in binary encoding and the move letter to be the corresponding entry. This is enough to build the solution string and to check if a node has already been visited. On the other hand, while this practice is indeed very memory efficient, it also introduces a significant overhead to the application, which must convert the 64 bit long back to a sequence of integers in order to generate child nodes or, in order to avoid conversion, specify every function to operate on such kind of data representation at the cost of obfuscating further the code and being in general still slightly slower. It is also worth mentioning that this doesn't scale well with boards of different sizes, which may need more or less bits and thus won't fit nicely in a long variable

Given the limited speed of the hardware that I can use to run this project, I dropped this data structure in favor of a more memory expensive but faster to execute one. The Board class generates objects to represent a particular board. The board itself is represented by a tuple of its numbers in order top to bottom left to right. Additionally, the coordinates of the empty tile are saved, to speed up board operations. A node of the tree (Node class) contains a pointer to a Board object and a character to represent the last move applied that got us there from the previous node. When running certain algorithms, each node also contains the G and F value of the node, that is distance from start node to the current node and the value of the function $F(n) = g(n) + h(n)$ where $g(n)$ is the aforementioned G value (which in case of Best-First Search is omitted) and $h(n)$ is the value of a heuristic function. These attributes are saved in slots to decrease the memory footprint of the node object. In the case of SMA* algorithm, even more attributes are present in a single node, but in this case memory isn't much of a concern because the algorithm is supposed to run with a limited number of nodes in memory space anyway.

When an algorithm needs to keep track of already visited nodes, a hash map in form of a dictionary is used, which stores as key the tuple contained in the board object and as value the last move applied that is contained in the node itself, plus eventually additional information that must be retained thorough the execution. The practical difference here from the suggested data structure is that the key of this hash table is already in an integer form instead of a bit representation.

The heuristics[3]

The heuristic function is a way to inform the search about the direction to a goal.[4]

A heuristic function is said to be admissible if it never overestimates the cost to reach the goal or in other terms if

$$0 \leq h(n) \leq h^*(n)$$

Where $h(n)$ is the value of the heuristic function on a node n and $h^*(n)$ is the value of the actual cost from node n to goal.

A heuristic function is said to be consistent (or monotonic) if, for every node n and each of its successors n', the estimated cost of reaching the goal from n is no greater than the cost of getting to n' plus the estimated cost of reaching the goal from n' or in other terms if

[3] http://ai.stanford.edu/~latombe/cs121/2011/slides/D-heuristic-search.pdf
[4] https://artint.info/html/ArtInt_56.html

$$h(n) \leq c(n, n') + h(n')$$

I have implemented 3 different heuristics, in addition to the obvious $h(n) = 0$:

The Number of Misplaced Tiles heuristic, also known as Hamming Distance counts the number of tiles in the wrong position, ignoring the blank space. It is obvious that it can never overestimate the cost to the goal, because to put all tiles in order you have to at least move each wrong positioned tile once. This heuristic is both admissible and consistent.

The Manhattan Distance heuristic sums for each non-blank tile the number of squares that lie between its current position and its position in the goal state. In mathematical terms it computes the sum of the L1-norm of each tile. For the same reasons as above, it clearly results that this metric never overestimates the cost to the goal, because tiles can only move square by square and cannot move diagonally. This heuristic is both admissible and consistent too.

The Sum of Inversions heuristic returns the number of inversions on a given board, by summing the inversions of each tile. The number of inversions in respect of a given tile is the number of tiles that ought to precede it but are currently subsequent to it, ignoring the blank tile. This heuristic is not admissible and the proof follows by contradiction. Assume the following state:

1	2	3	4
5	6	7	0
9	10	11	8
13	14	15	12

The tiles [9], [10] and [11] each have an inversion, because [8] is currently subsequent to them, additionally [13], [14] and [15] also each have an inversion because [12] is subsequent for a total of 6 inversions in this board, but it is only 2 moves away from the goal. So, the heuristic is overestimating the path cost to goal and thus is not admissible neither consistent.

The $h(n) = 0$ heuristic always returns 0 and it obviously never overestimates the cost to the goal, making it admissible and consistent, but also a rather poor choice.

How do these heuristics compare between themselves? For A* Search algorithm the lower the estimate to the goal, the more nodes the algorithm has to expand. With the $h(n) = 0$ heuristic it has to expand the largest number of nodes and it decays into Dijkstra's algorithm by accounting only for the $g(n)$ function. The Number of Misplaced Tiles heuristic is less accurate than Manhattan Distance, i.e. it underestimates more, and thus will expand a larger number of nodes and take more time, while Manhattan heuristic is actually the most accurate yet still underestimating heuristic and so it is the preferred one for the problem. The Number of Inversions heuristic, being sometimes overestimating, is expected to expand fewer nodes and take less time than the previous two heuristics.[5]

[5] http://theory.stanford.edu/~amitp/GameProgramming/Heuristics.html

The algorithms[6]

We can distinguish 2 main classes of algorithms: uninformed and informed search.

Uninformed search algorithms use a brute-force approach without using any domain specific knowledge[7]. In this project, we have 3 of them: Breadth-First Search, Depth-First Search and Iterative Deepening Depth-First Search.

Informed search algorithms try to reduce the amount of search that must be done by making intelligent choices for the nodes that are selected for expansion. This implies the existence of some way of evaluating the likelihood that a given node is on the solution path. In general this is done using a heuristic function.[8] In this project we have again 3 of them: Best-First Search, A* Search and Simplified Memory Bounded A* Search.

Breadth-First Search

Breadth-First Search searches the graph by expanding all nodes at the given depth before going deeper, until the goal state is reached. In the worst case the algorithm will expand all nodes up to depth d where it will find the shallowest solution, with a time complexity of $O(b^d)$ where b is the branching factor. Thus, in our specific case the time complexity is $O\left(3^d\right)$. All nodes have to be stored until the point in which their children are generated, therefore the space complexity is proportional to the number of nodes at the deepest level that will be reached (the level of the solution), which is $O(b^d)$ and so $O\left(3^d\right)$. Breadth-First Search in the game of 15 is always complete and optimal, so if there is a solution it will always find it and it will always be the shortest possible solution, but it will usually run out of memory for larger problems. In practice though the algorithm becomes extremely slow even before that. In the program the algorithm makes use of a queue data structure to hold the nodes the visit in a FIFO order, and new nodes are added only if they have not been already visited (this is done by checking the hash map mentioned earlier). To respect the ordering of children set by the user, nodes are already generated using that order in the Node class, so we can just add them to the queue in that same order.

Depth-First Search

Depth-First Search searches the graph by always expanding a child of the latest explored node until, if set, a maximum depth m is reached, at which point it will start back-tracking, or a solution is found. The time complexity is $O(b^m) = O(3^m)$ and thus it depends on the maximum depth set for the algorithm, while the space complexity is $O(bm) = O(3m)$ because the algorithm only needs to store the path from the initial node to the current node and the siblings of each node encountered. We can notice that the time complexity is terrible if $m \gg d$ i.e. if the maximum depth set is much larger than the depth of the shallowest solution, or any solution in general.

In our case, the algorithm is complete if we set a depth limit $m > d$ where d is the depth of the shallowest solution. If we do not set a depth limit, then the algorithm is still not complete if it finds an infinite path with no goal states, unless we keep track of visited nodes. The algorithm is also generally not optimal, but

[6] Taken from "Artificial Intelligence: A Modern Approach", Chapter 3

[7] http://cs.lmu.edu/~ray/notes/usearch/

[8] http://people.sju.edu/~jhodgson/ugai/infsrch.html

it can be if we set a depth limit equal to the depth of the shallowest solution. Given the vastness of the problem, the algorithm will eventually finish the memory when a limit is not set and it starts exploring a path with a too deep goal state.

In the program this algorithm uses a stack data structure, in practical terms a list visited in LIFO order, to hold the nodes to visit. Since we are using a depth limit, in this case a node can be revisited if it was last visited at a deeper level, otherwise we could be blocking path to possible solutions, especially if such nodes are at depth $memoryLimit + 1$ and thus are not even analyzed and just thrown away. For this purpose, in this algorithm the hash map also contains an entry to register the last depth a node was discovered at. To respect the ordering of children set by the user, nodes are already generated using that order in the Node class, but since this algorithm works in a LIFO order, we explore these nodes in reversed order.

Iterative Deepening Depth-First Search

Iterative Deepening Depth-First Search combines the advantages of both Breadth-First search and Depth-First search, by performing a Depth-First search iteratively for each increasing level of depth. The time complexity is asymptotically the same as Breadth-First search $O\left(b^d\right) = O(3^d)$ while the space complexity is as modest as Depth-First search at $O(bd) = O(3d)$. In our problem, this algorithm is also complete and optimal.

In the program this algorithm basically iteratively calls a DFS procedure with increasing depth limits. There is no need to set an iteration limit for this algorithm at all, as we could just set an infinity and wait until it eventually finds a solution, but for consistency I wanted to let the user set a limit for this algorithm as well. The same implementation details described for DFS also apply here.

Best-First Search

Best-First Search expands the node which believes to be closest to the goal, on the grounds that is likely to lead to a solution quickly. It achieves this by evaluating each node according to a heuristic function that should be a good estimate of the distance between the node and the goal. This kind of approach makes it a greedy algorithm, because it simply takes the best available choice at every step. The space and time complexity are $O(b^m) = O(3^m)$ with m being the maximum depth of the search space. The heuristic function has a primary role in pruning away unpromising nodes, reducing both complexities by operating on the branching factor b which a good heuristic will substantially reduce. In this project, this algorithm is complete because we keep track of visited states, but it is not optimal.

In the program, the algorithm uses a binary heap to store the nodes according to their F value, which in this case is the value given by the heuristic function only. The top of the heap is always the least F-value node in memory.

A* Search

Being a variant of Best-First search, A* Search is a widely known algorithm that instead of only using the heuristic function to evaluate nodes, it additionally takes into account the cost to reach the given node. In other words, it will evaluate each node according to the function $f(n) = g(n) + h(n)$ with $g(n)$ being the cost to reach node n. The $f(n)$ function can be understood as the estimated cost of the cheapest solution through n. The time and space complexity are the same as Best-First search but given a heuristic function that satisfies certain conditions it is both complete and optimal. In our case, this is true if we use a consistent heuristic function. Additionally, if the algorithm uses such heuristic it is also optimally efficient, i.e. there is no other optimal algorithm that can guarantee to expand fewer nodes than A*.

In the program this algorithm again uses a binary heap to store nodes according to their F-value, but the checking for already visited nodes is done in a slightly different way: a node is added to the visited nodes upon expansion instead of when it is being generated. This means that the heap might at some point contain the same states, which can differ by their $g(n)$ value. Only the best out of these nodes will be popped from the heap and promptly added to the visited states, so all the other less promising copies are discarded right away after being popped. The algorithm is not guaranteed to return the optimal solution if we use an inconsistent heuristic, but it can be modified in order to do so, by adding another data structure to keep nodes already expanded and eventually modify their G-value and re-visit them.

Simplified Memory Bounded A* Search

Simplified Memory Bounded A* Search is a A* search variant that will use only the available memory given to find a solution. It will proceed like A* search for as much the memory allows, and then when it is full it will discard the worst leaf to make room for more promising nodes, but it will save the best $f(n)$ value of the forgotten children of a node to eventually re-expand them if they turn out to be the most promising path. The algorithm has space complexity equal to the given memory and time complexity comparable to A* but subject to the effect of node re-expansion, with the more nodes being regenerated the more the time complexity increases. It is complete if there is enough memory to store the path from root to goal node, and in our scenario this is enough to make it optimal as well.

My implementation of SMA* works if it is given enough memory, in the same way as A* Search. It uses the SortedSet class which implements a set-like data structure which is ordered, likely by using balanced binary search trees. This data structure, which I will call frontier from now on, holds the nodes currently considered to be in memory, and it is the only structure the algorithms needs, as it does not keep track of visited nodes.

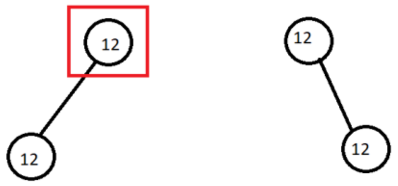
For some reason I couldn’t fix, the algorithm starts cycling once it hits the memory limit, while the algorithm is supposed to find a depth n solution with a n+1 nodes wide memory. Also, sometimes the algorithm might crash when the memory finishes, because it needs to distinguish the nodes exactly, i.e. checking if they have the same state is not enough but it also needs to check if two nodes also have the same sequence of moves from the starting state. This is because the algorithm sometimes needs to check if the father of a node is still in memory, and a father must be identified in this exact way. To avoid this crash it is sufficient to remove this comment from the NodeSMA class:

```
return self.gameBoard.gameBoard == other.gameBoard.gameBoard #and self.sequence == other.sequence
```

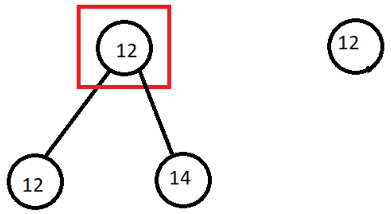
By doing this the algorithm won’t behave like A* anymore though, because now duplicate children can be added to the frontier and the algorithm will need much more memory and time to compute, as it does not keep track of already visited nodes at all.

Here’s an example of the cycling it is experienced:

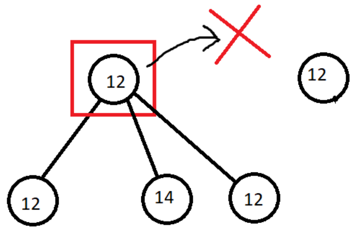
1) Suppose a memory that supports 4 nodes, and have at a certain point this situation. The marked node is selected for expansion.



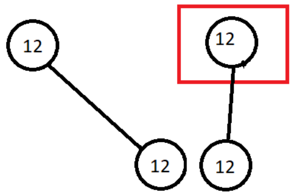
2) In order to make room for the new node, we delete the worst, oldest and deepest leaf.



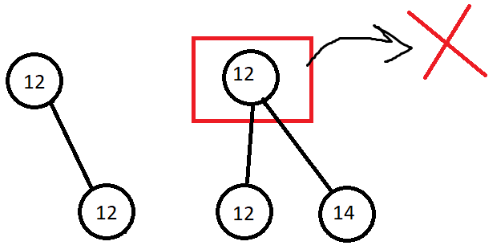
3) The same node is expanded again, and since all of its children are in the memory it is also deleted and there's no need to forget any node.



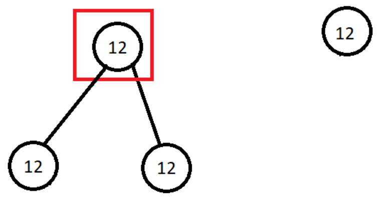
4) Now the right node is selected for expansion, but to make room for his child we need to delete a node on the left, the worst one. By deleting it, we are required to put again its father in the memory, and to make room for the father as well, we are required to delete another node.



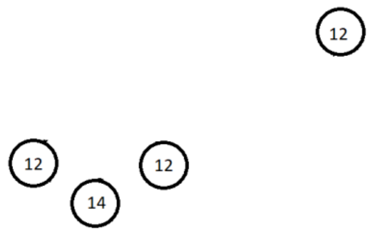
5) The same node is expanded again, and since it has generated all its children it is also deleted



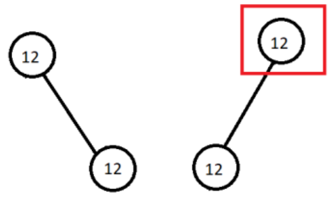
6) Now the node on the left is selected for expansion, and its leftmost child is generated again. We need to deleted a leaf on the right, and after we do so we have to put its father back into memory, so to make room for it as well we delete the other leaf on the right as well.



7) The same node is expanded again, and since all its children are in memory it is also deleted. Notice that we are in the same situation as step 3.



8) Now the childless node on the right will be expanded again, deleting the fatherless leaves on the left, which will reintroduce their father in the memory and which will be eventually re-expanded again in a never-ending cycle, without ever reaching the nodes at depth 4 (the father of the 2 uppermost nodes is assumed to be the root).



I have tried many pseudo-code implementations of SMA and all of them suffer from this endless cycling.

Tests

The following tests have been run on an AMD A12-9720P CPU with 8GB of DDR4 RAM available. Sadly, the CPU is able to modulate its clock speed dynamically regardless of my wishes, so the following measurements are most likely dependent on which clock speed was being used in that time frame. The initial states are generated by applying random moves to the goal state, checking that no move is ever followed by its opposite. This is not enough to generate initial states with an exact distance to the goal state, as cycles in the moves applied or better paths can still appear. The bigger the number of moves applied to the goal state, the harder it is to generate even deeper initial states because cycles or alternative paths to goal become increasingly more common. Because of this, in some graphs it is only specified the number of moves applied to the goal state instead of the exact distance from initial to final state.

Comparison of uninformed search algorithms

In the following pictures lies a comparison of the uninformed search algorithms BFS, DFS and IDFS. All the tests have been run using the same order preference "DURL". Additionally, the depth limit for DFS was set at the length of optimal solution. Figure 1

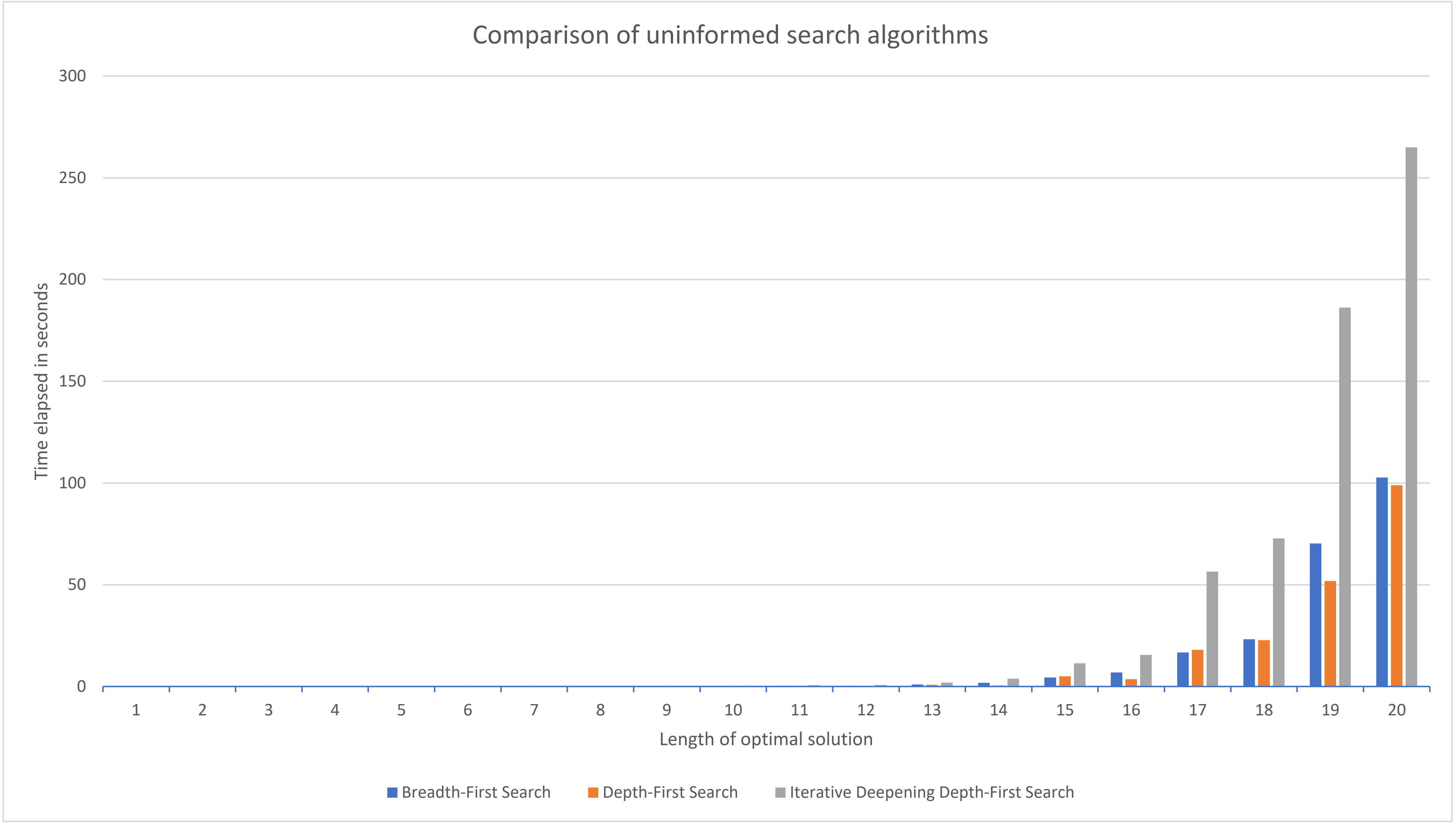


Figure 1

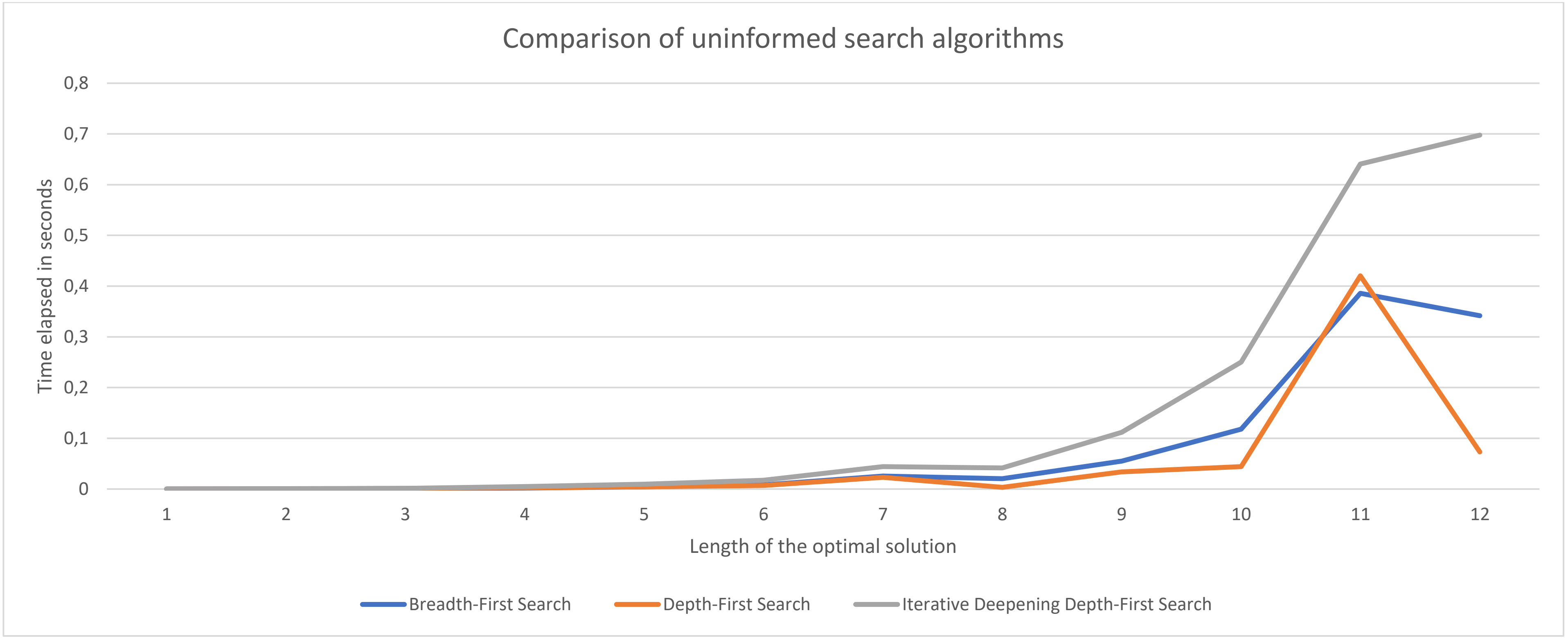


Figure 2: Detail of Figure 1

We can notice that both the BFS and IDFS algorithm get exponentially slower more quickly than DFS, with a runtime of 264 seconds to solve a problem with a depth 18 solution for IDFS. DFS, by applying a depth first approach, is generally quicker to find a solution, but only because we set the depth at the same level where we expect a solution, an information which we usually don't have. IDFS runs obviously slower than BFS, because it has to generate more nodes than BFS, but this overhead isn't particularly significant because having on average 3 children per node, the number of nodes at depth n is equal to twice the number of nodes in all previous levels combined.

In the Figure 3 we can evaluate the performance of DFS on a given problem with optimal solution at depth 10. The algorithm is tested by setting increasingly higher depth limits, up to 20. It is interesting to notice how elapsed time increases exponentially with the depth limit even though the problem is still the same, and how at increasing depths new, longer, solutions are being found. In some cases, finding a longer solution brings the elapsed time down again, because the algorithm converges the quickest when it finds a solution on the current max depth.

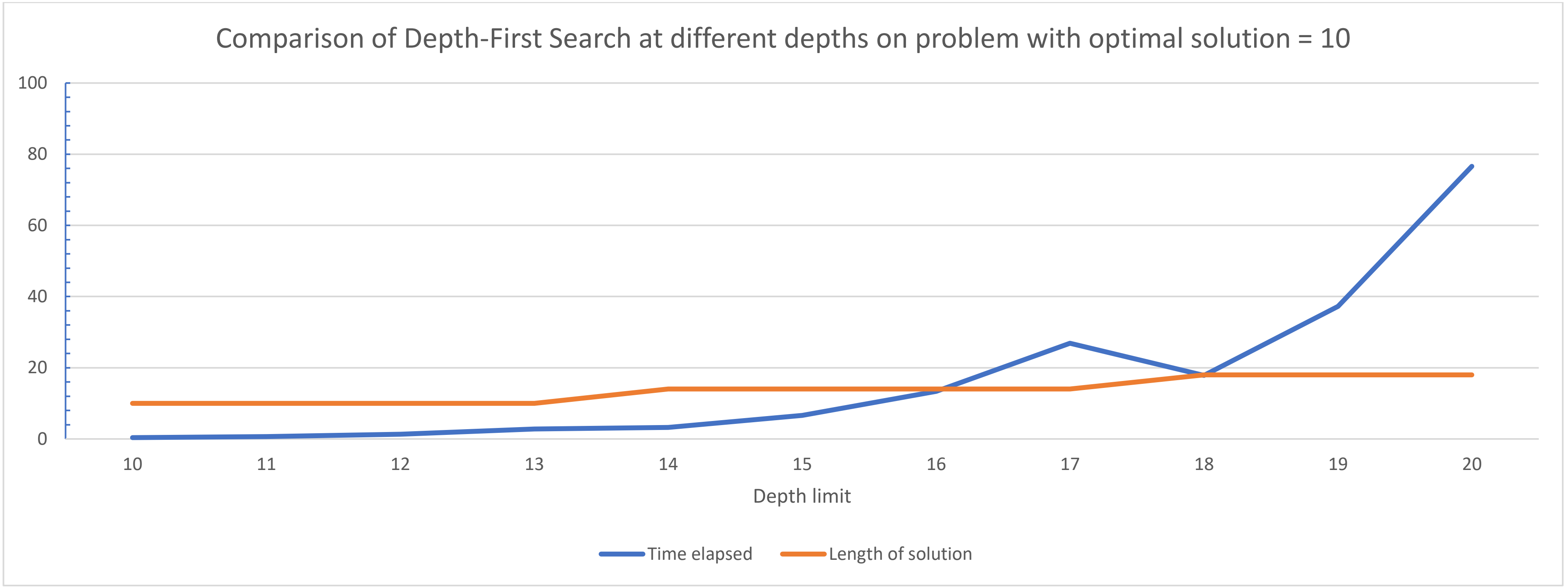


Figure 3

Next, it follows a comparison of all the possible ordering preferences on a given problem with optimal solution “LDRURRULLL” at depth 10.

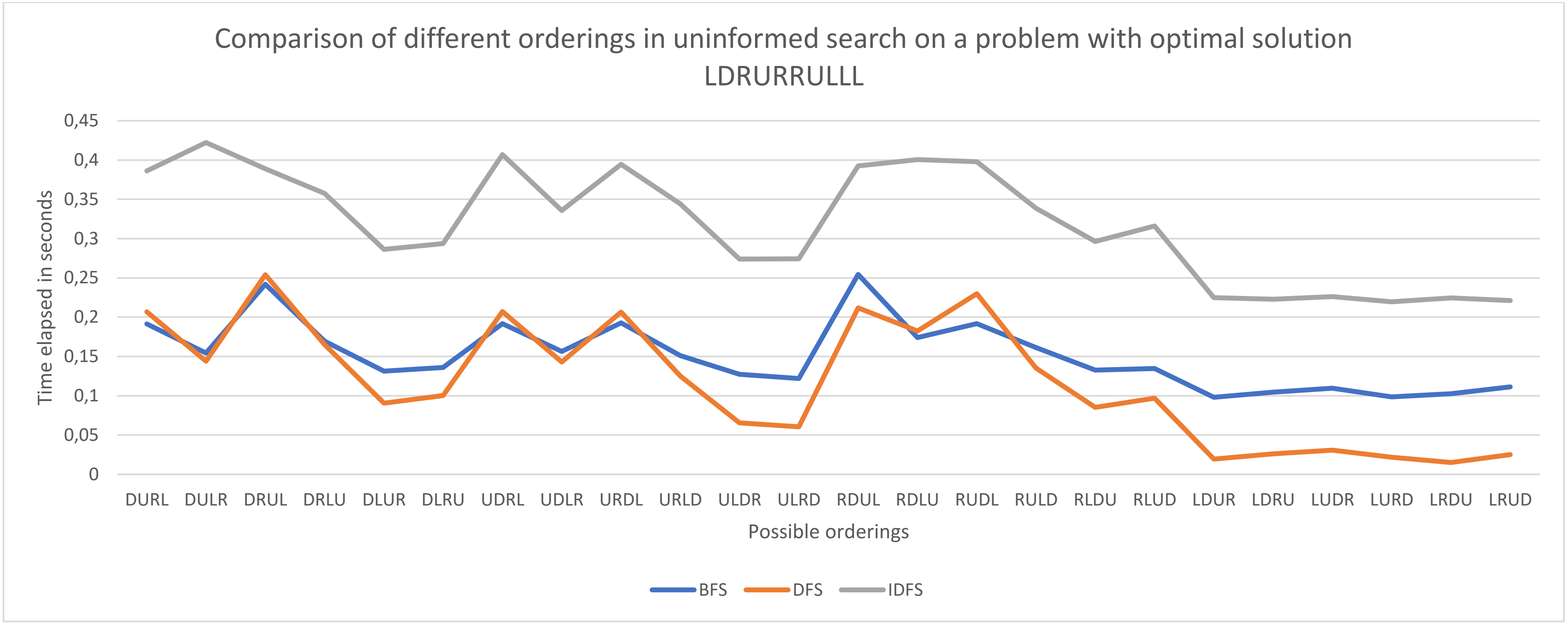
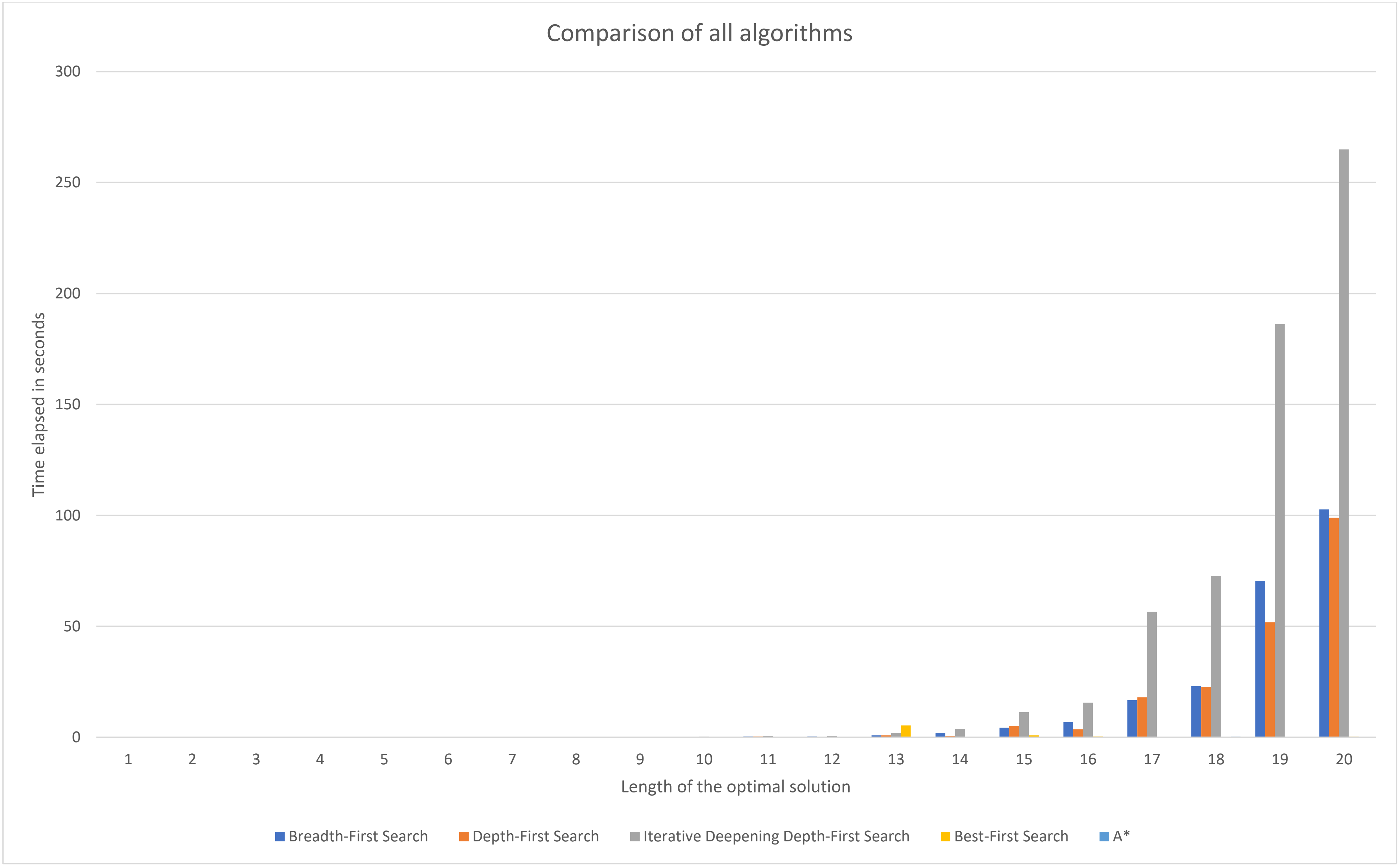


Figure 4

It can be noticed that all algorithms perform significantly better in all orderings that start with L and that the worst runtimes happen when the ordering features the L at the last priority. The move "L" is not only the most common in the solution, but it is also repeated three times in the three last moves, thus making it particularly significant for Depth-First Search and IDFS. For Breadth-First Search the runtime differences are slighter but prioritizing "L" made it check the "L" nodes earlier than in other cases and leading to a faster convergence.

Comparison of all algorithms

A comparison of all algorithms for the first 20 levels of depth of the optimal solution is shown but given the speed of informed search algorithms it is barely possible to see their values.



Comparison of all algorithms
Time elapsed in seconds
0
50
100
150
200
250
300
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
Length of the optimal solution
Breadth-First Search
Depth-First Search
Iterative Deepening Depth-First Search
Best-First Search
A*

Figure 5

Comparison of informed search algorithms

In the following graphs Best-First Search and A* Search are being compared. In these tests, both algorithms use the Manhattan Heuristic.

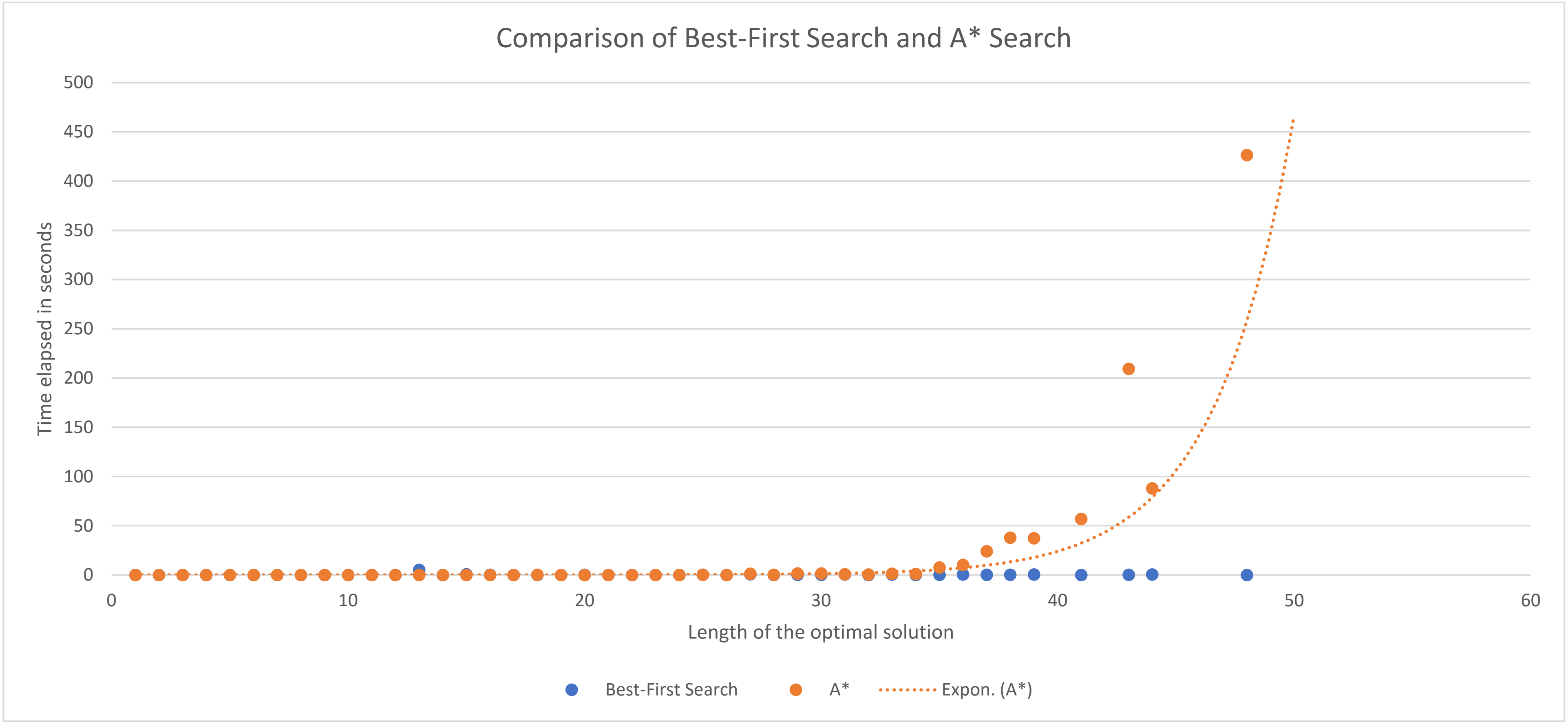


Figure 6

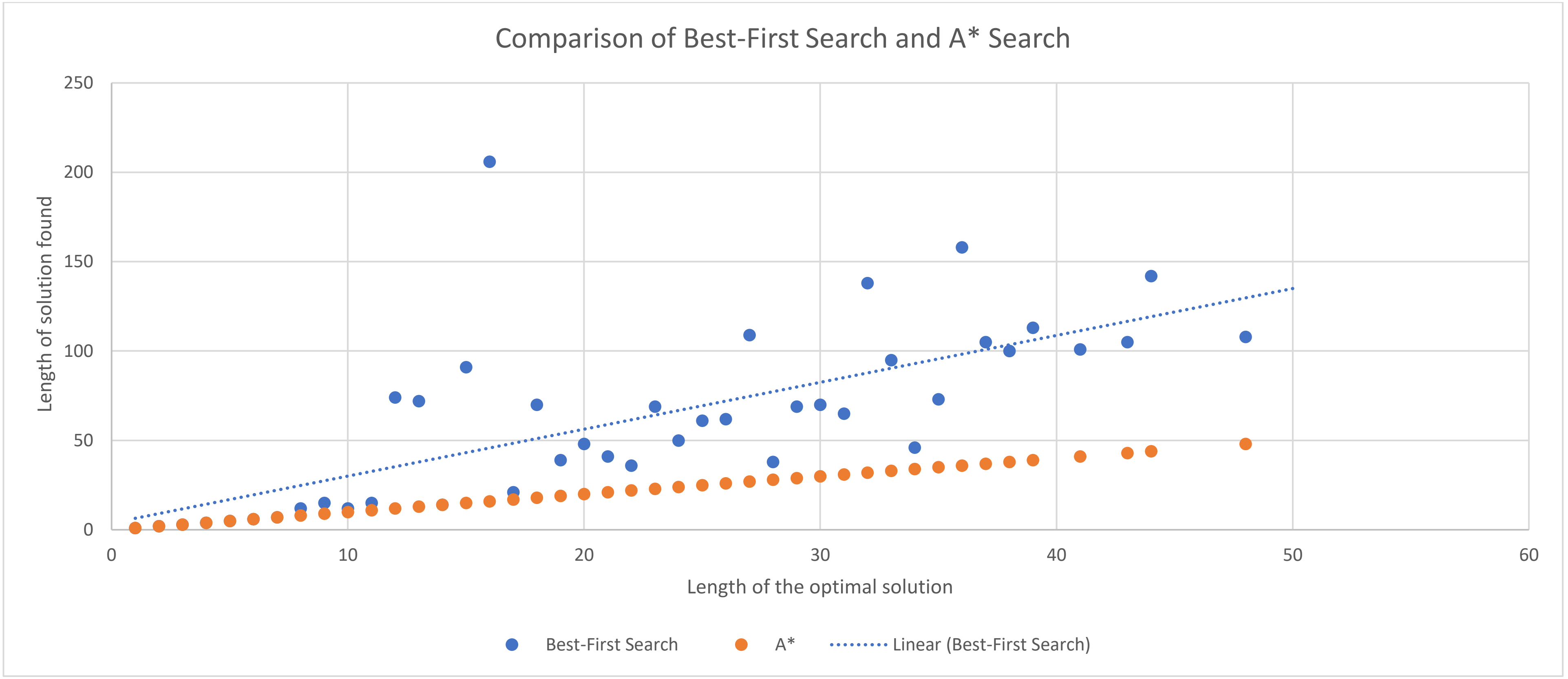


Figure 7

In Figure 6, we compare the speed of both algorithms when run on problems of optimal solution up to depth 48. We can start seeing the exponential runtime of A* Search from depth 35 onwards, while Best-First Search seems mostly untouched by the increased depth of the solutions. It is good to remember now that finding an optimal solution is NP-Hard and A* has to search a bigger number of nodes in order to account for the $g(n)$ function as well, while finding a solution, any solution, can be done in a reasonable amount of time, as the greedy Best-First Search algorithm shows.

Figure 7 shows that, in fact, the solutions found by Best-First Search are definitely longer, sometimes up to 2 or 3 times the length of the optimal one. In some cases (not shown here) they might even be in the thousands of lengths. For real life applications these solutions might not be good enough even though they can be obtained in such a short time: imagine playing the game and applying hundreds of moves when the actual solution might require only a dozen. It can be seen though that even for Best-First Search the length of the solutions found grows somewhat linearly with the length of the optimal solution. A* Search, by using a consistent heuristic with a $g(n)$ function, is guaranteed to find to always find an optimal solution.

The capabilities of Best-First Search are further evaluated inFigure 8, which shows the application of the algorithm to initial states generated by applying a number of moves in the range of 40 to 500 to the goal state (which in no way guarantees a depth between 40 and 500!). The algorithm is shown to be still pretty fast, with spikes that go no further than 20 seconds of runtime, and which are directly proportional to the number of nodes visited and thus on the length of the solution found.

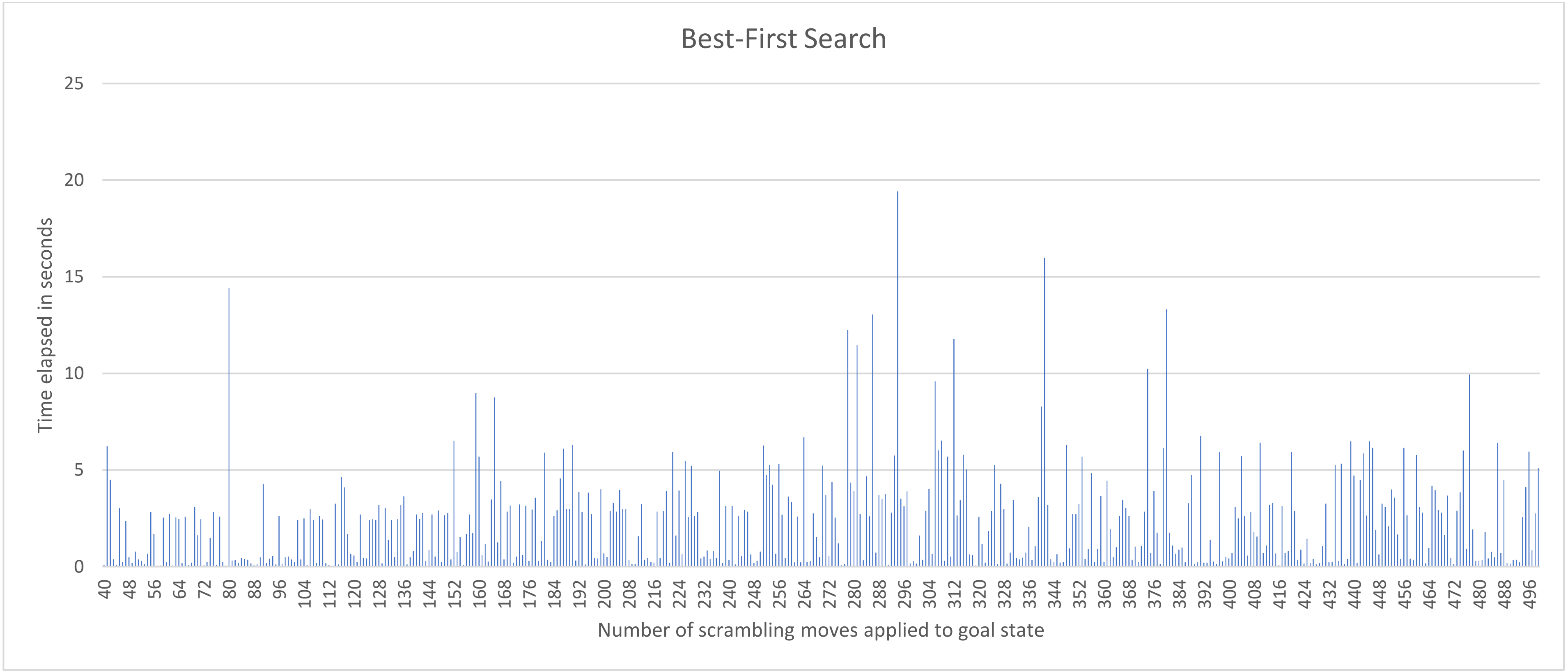


Figure 8

Comparison of heuristics

In the following section with compare the three available non-zero heuristics on 4 different problems using both Best-First Search and A* Search.

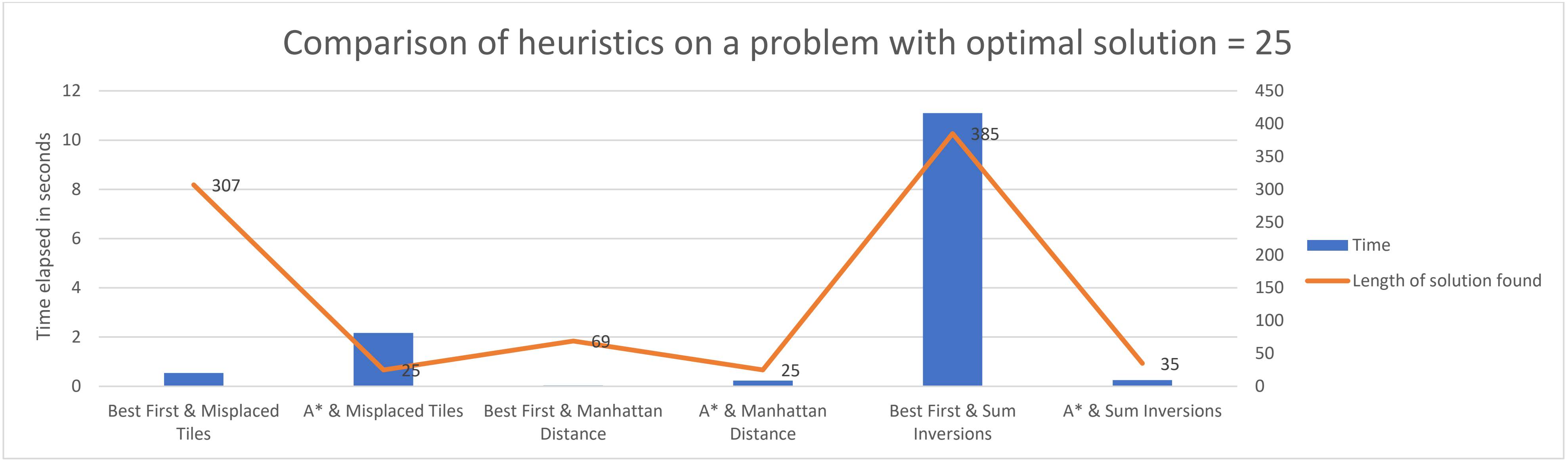


Figure 9a

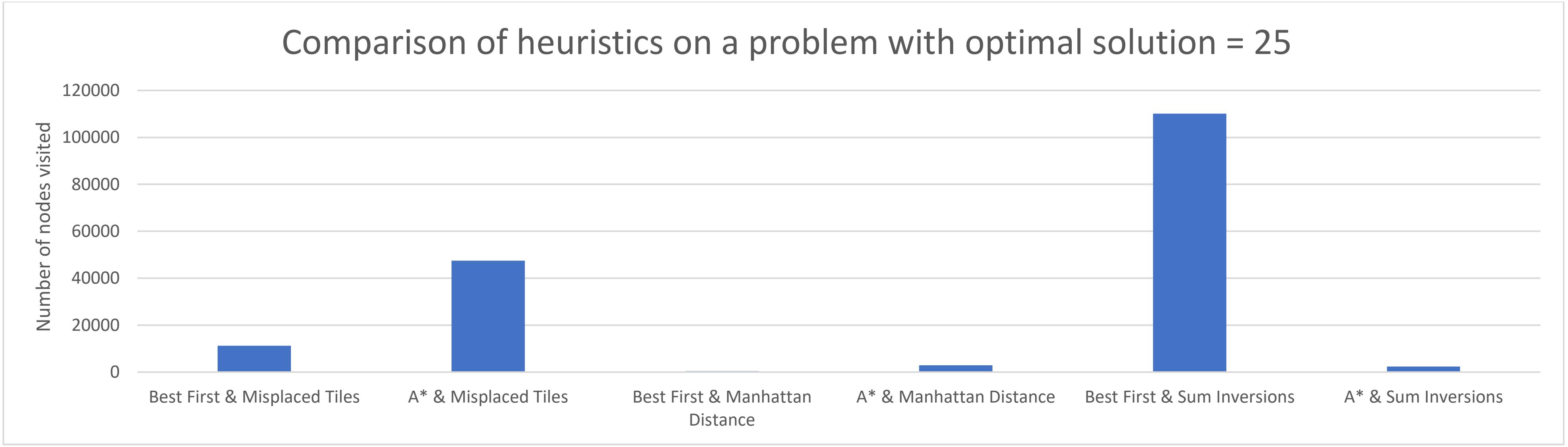


Figure 9Figure 9b

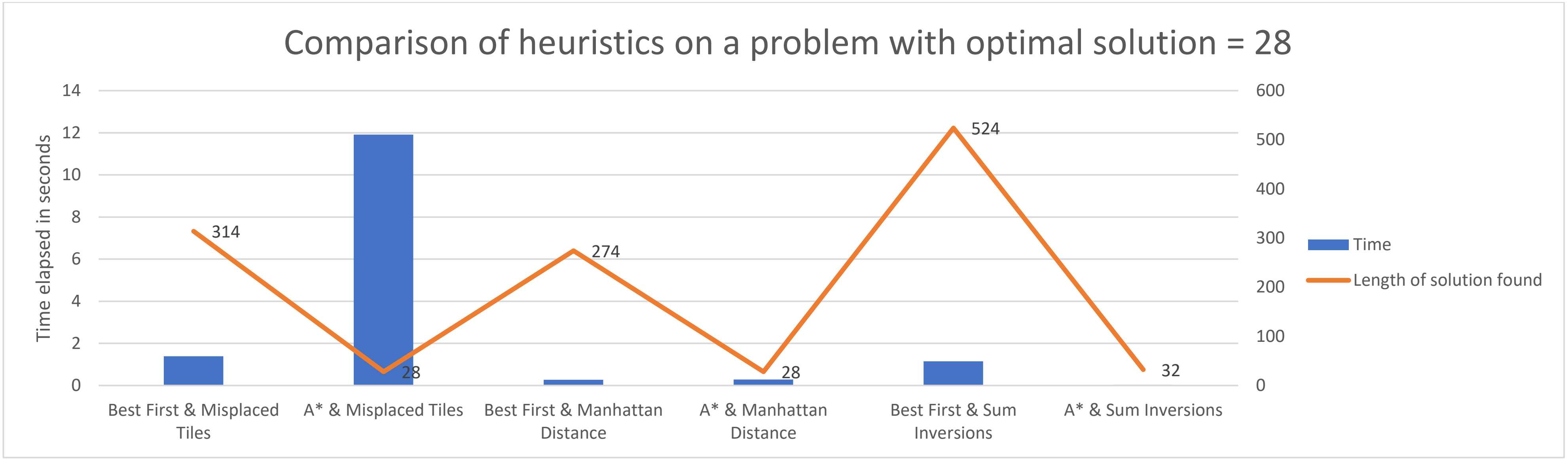


Figure 10a

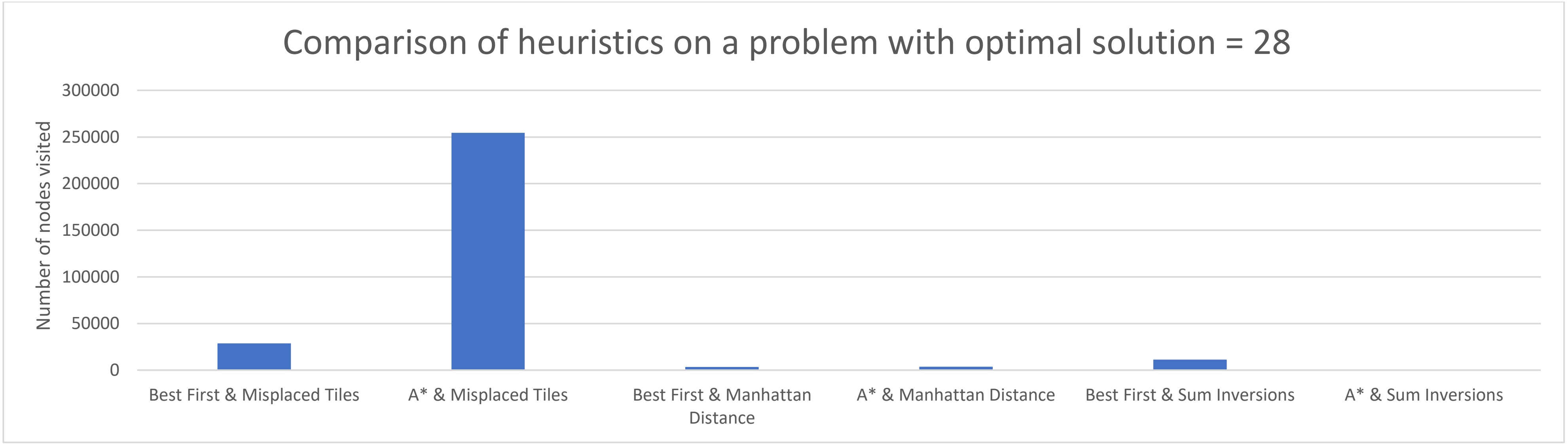


Figure 10b

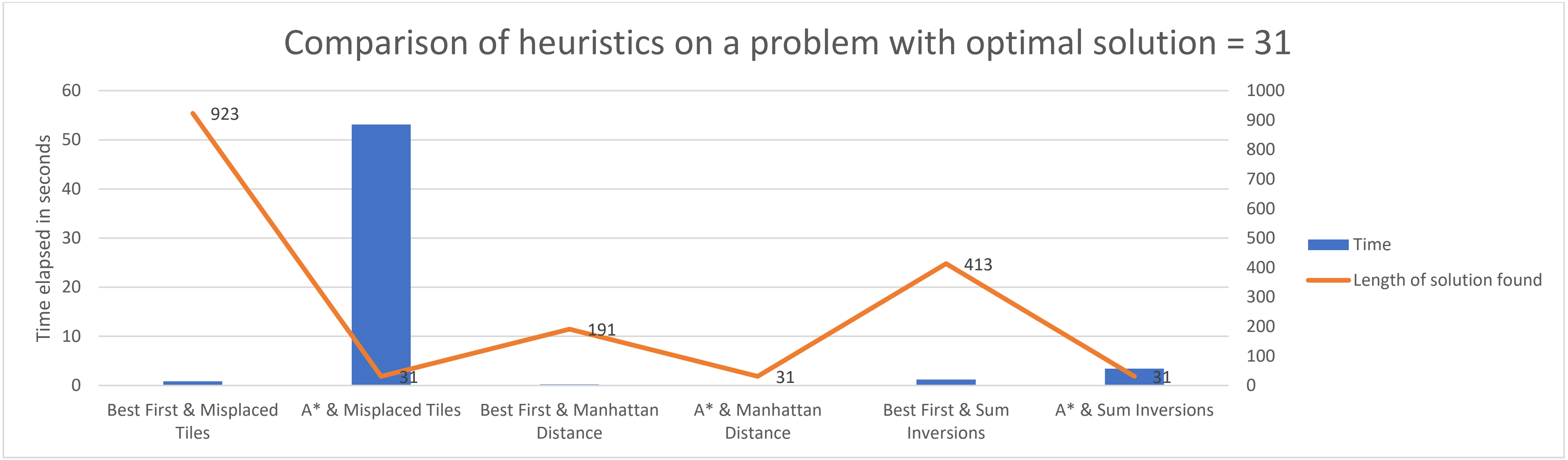


Figure 11a

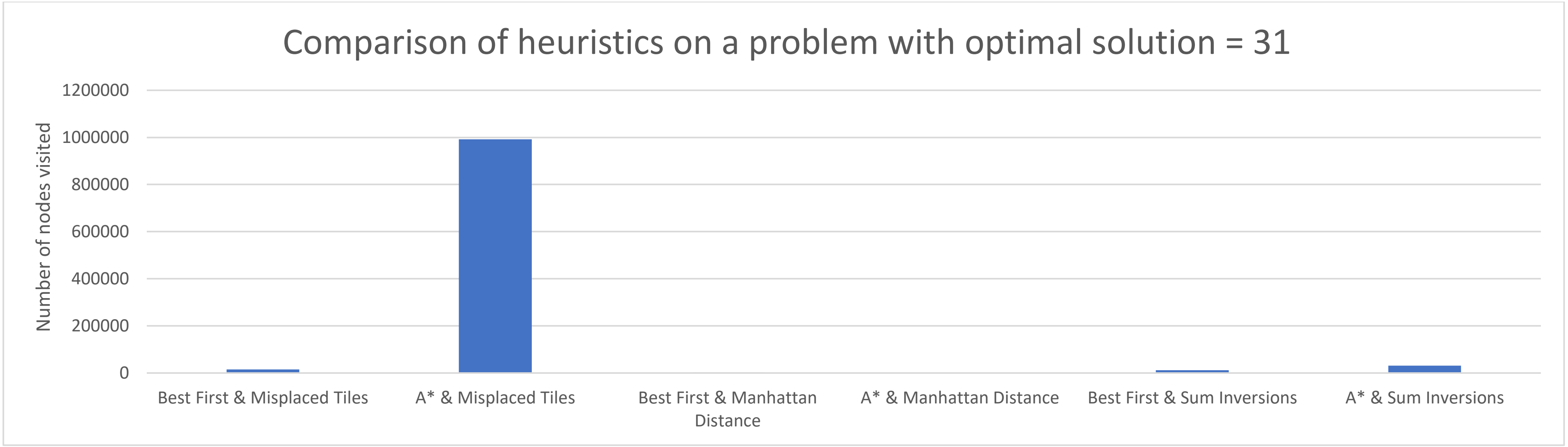


Figure 11b

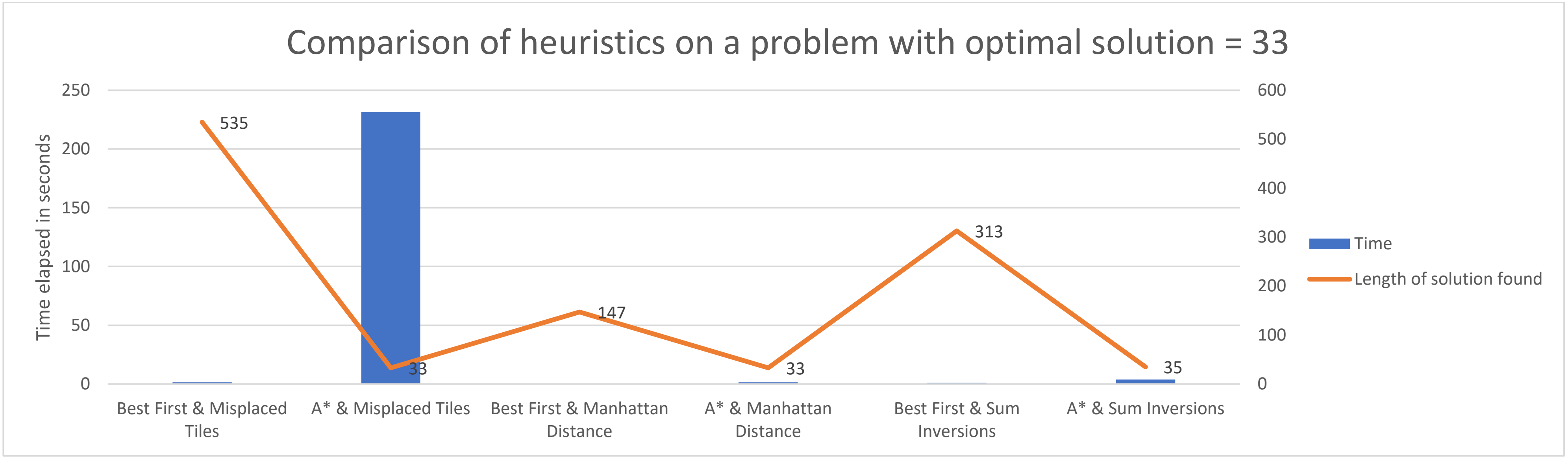


Figure 12a

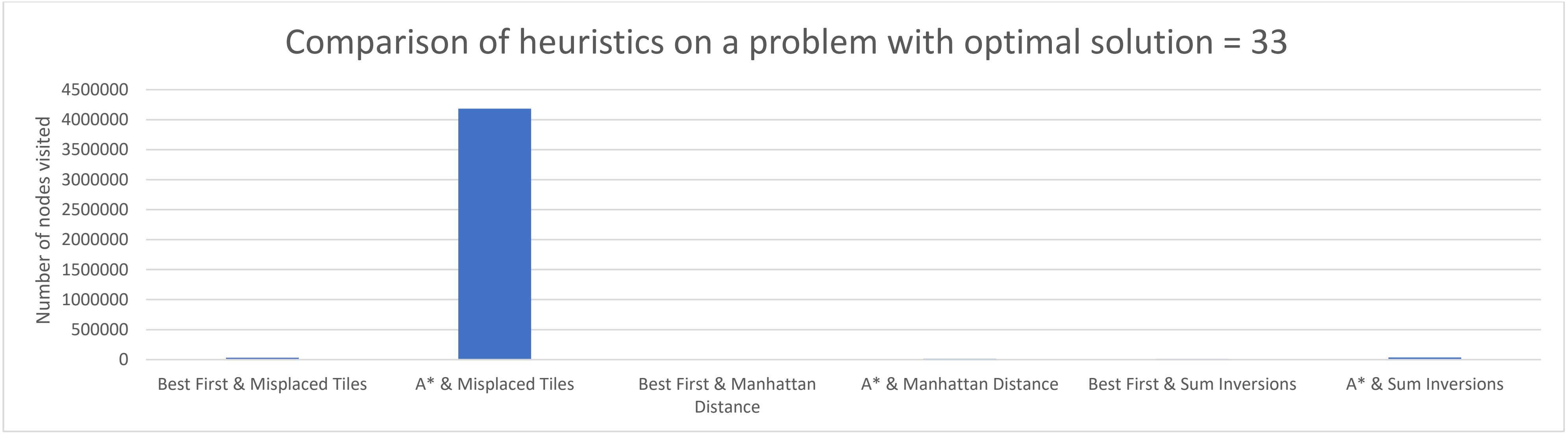


Figure 12b

Regarding Best-First Search, we can notice from Figure 9b that using the Sum Inversions heuristic it visited a large number of nodes, an occurrence which has experimentally shown to be rather uncommon for the algorithm as a whole. We can also notice that the shortest solutions are found by the Manhattan Distance heuristic. With the exception of Figure 9b, the heuristic which leads to expanding the highest number of nodes is the Number of Misplaced Tiles, while on the opposite side the heuristic which expands the least nodes is the Manhattan Distance. In Figure 9b the algorithm converged only after 395 nodes, using this heuristic. Finally, the consistent heuristics (Number of Misplaced Tiles and Manhattan Distance) generally lead to less nodes being expanded in Best-First Search compared to A* Search, while the opposite is true for the Sum of inversions heuristic.

For A* Search we notice that the Number of Misplaced Tiles heuristic is far slower than the other two, taking exponentially longer time for harder problems and leading to a huge number of nodes expanded, especially compared to the number of nodes expanded by the other two heuristics on this same algorithm. In Figure 12b this heuristic led to the expansion of a staggering 4.186.000 nodes. On the other hand, this heuristic is consistent, so the solution found is always an optimal one. The Manhattan Distance heuristic is the best one for this algorithm as well, leading always to an optimal solution because it’s consistent but it is also much faster and expands less nodes than the previous heuristic. Finally, the Sum of Inversions heuristic, while having generally the same speed/node expanded performance as the Manhattan Distance heuristic, it is not consistent so in some cases it leads to a non-optimal solution, such as in Figures 9a, 10a and 12a.

Comparison of the $h(n) = 0$ heuristic

Using the $h(n) = 0$ heuristic, Best-First Search can decay in either a Random Search or a Breadth-First Search, depending on the behavior of the data structure it uses to store the nodes to visit. Usually a binary heap is being used, and heaps are not stable, meaning that the order of elements with the same priority isn’t guaranteed to be retained. In this case, Best-First Search decays into Random Search and will most likely keep running until it finishes the memory without any satisfying result. On the other hand, if we somehow make our binary heap stable (by numbering its entries for example), then Best-First Search behaves in the exact same way as Breadth-First Search, and the similarities between the two can be seen in Figure 13. Obviously Best-First Search is the least performing of the two, due to the heap functions that still have to operate.

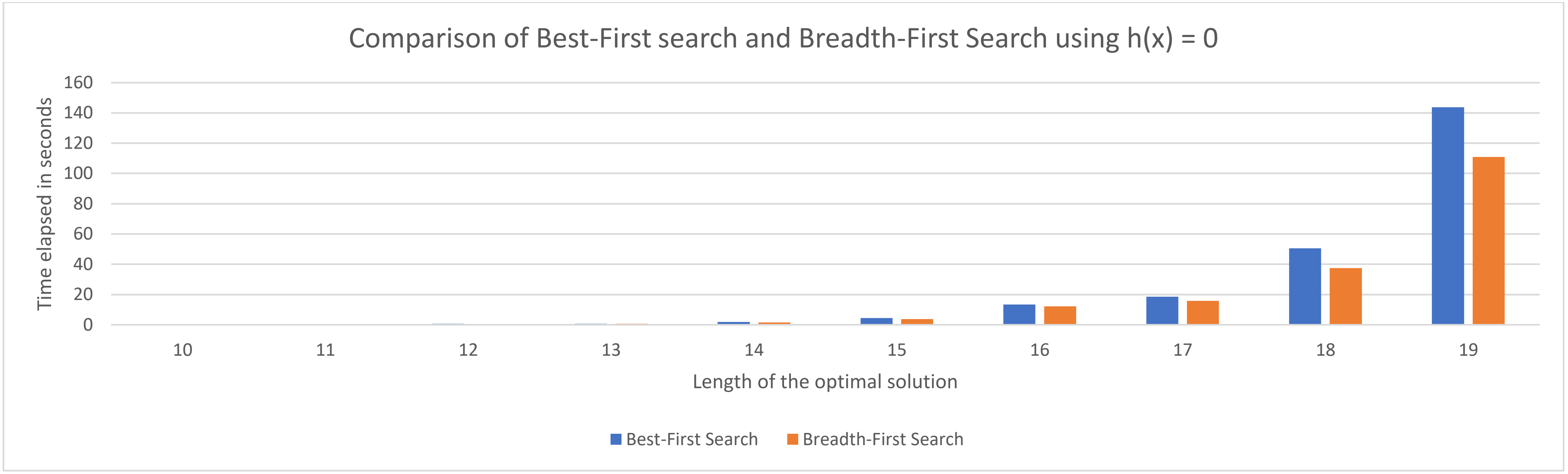


Figure 13

The A* Search algorithm, given the heuristic in question, can only rely on the $g(n)$ function, and decays into Dijkstra's Algorithm, but noticing that the cost of a path between two nodes in the game of 15 is 1, it becomes essentially the same as Breadth-First Search and the Best-First Search algorithm with a stable heap. The runtime of both algorithms can be compared in Figure 14, and it can be noticed that it is roughly the same, if not somewhat better for A* because of simpler heap operations.

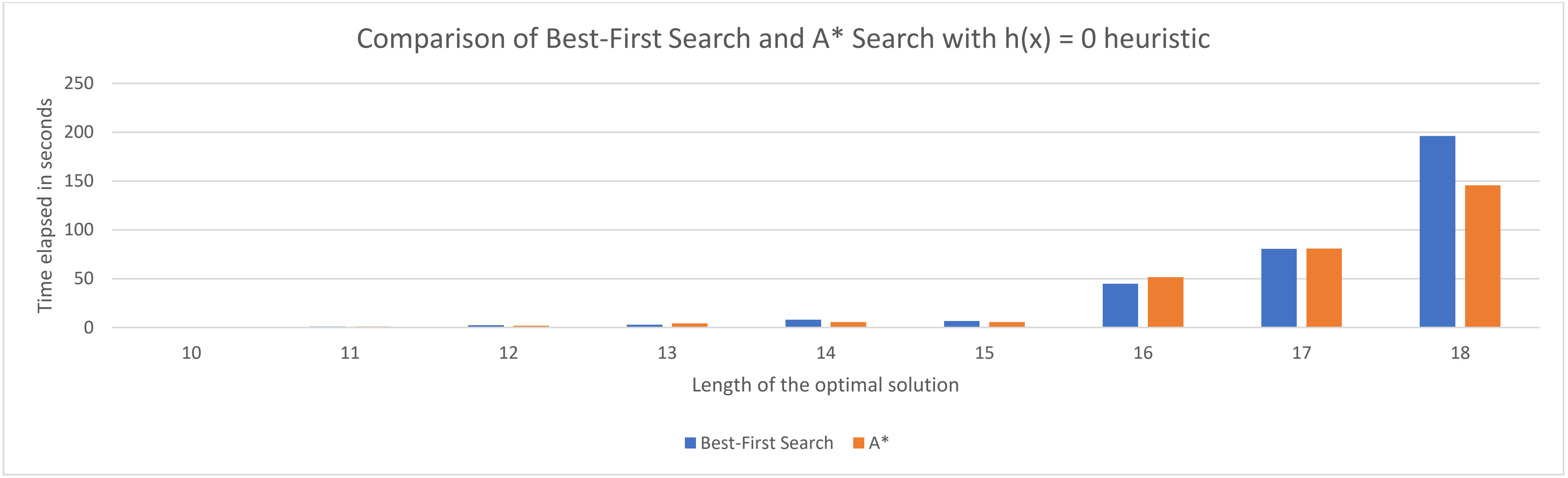


Figure 14

Conclusion

We have compared uninformed and informed search strategies, noticing that the informed search strategies vastly outperform the former. For easy problems the difference is negligible, but in the long run, for realistic, more difficult problems, we are forced to use some heuristic strategies to guide our search for a solution. For this purpose A* Search is the ideal algorithm, which coupled with a high quality heuristic can reliably be expected to do the job. For very hard problems we are faced with a choice. If we have enough memory and time, we can still use A* Search and get an optimal solution. If we lack space, we can resort to SMA* Search and still have an optimal solution at cost of increased time cost. If we lack both space and time, a greedy approach like Best-First Search can give us a solution of doubtful quality, but we get it with a low time and memory cost.

Uninformed strategies can still be useful, too. If we have some knowledge about the solution we can set an optimal depth for DFS or change the preferred order used to evaluate nodes, which will speed up the execution and expand a fewer number of nodes.