# Brute-force and Heuristic Search Algorithms over the 8-Puzzle Problem

Miguel García García. UO245504@uniovi.es. Intelligent Systems. Bachelor's Degree in Software Engineering. EII. University of Oviedo. Campus de Los Catalanes. ES-33011. Oviedo.

**Abstract.** The 8-puzzle problem can be solved with different search algorithms applied in state spaces. This space can be modelled in various ways. Some of the most common search algorithms will be studied and analyzed, including uninformed brute-force algorithms such as Depth First Search or Bread First Search, and informed heuristic algorithms like A* or Static Weighting A*. For experimentation, the implementation of these algorithms included in the aima-java package will be used. Also, the practical results obtained will be compared with each other and with the theoretical expectations.

**Keywords:** Heuristics, 8-puzzle, Brute-force algorithms, Iterative Deepening, A*, Static Weighting.

## 1 Introduction

Search algorithms are widely used in all kinds of environments, from shortest routes calculation on maps to game-solving techniques. There are vast amounts of search algorithms but, depending on the situation, some can perform better than others. This performance is measured, basically, by the cost of the solution found, where the objective is to minimize said cost. Those solutions with minimum cost are known as optimal solutions, and are the ideal situations that we pursue. However, sometimes, these ideal solutions are extremely expensive if not impossible to achieve with conventional uninformed algorithms, so specific knowledge is needed to help our algorithms find said solution. Also, depending on the necessities, sub-optimal solutions could be accepted in some cases for the sake of performance.

All these scenarios will be covered in this paper, where some algorithms aiming to pursue the best possible results, each one prioritizing different parameters (runtime, optimality, …), are going to be discussed.

This document is structured as follows. In section 2, some of the most popular brute-force algorithms will be discussed. In section 3, three very common heuristic algorithms (A*, Static Weighting A*, and Greedy Best-First Search) will be explained. Lastly, the 8-puzzle problem which we're going to use for experimentation, its modelling, and the heuristics to be used will be detailed in section 4, while experiments over said problem with the algorithms from section 2 and section 3 are going to be put at work in section 5.

## 2 Brute-force Search Algorithms

Brute-force (also known as blind or uninformed) algorithms don't use any problem-specific knowledge. There are different algorithms, each with its own pros and cons. The most common are *Depth-first Search (DFS)*, *Breadth-first Search* (BFS) and *Uniform Cost Search (UCS).* All of these algorithms are based on two basic structures, usually called OPEN and A_TABLE, the former storing the known but not yet visited nodes and the latter storing information about each of the visited nodes (the cost from the initial node to the given node, and its predecessor). These two data structures are the main elements of the General Tree Search Algorithm, of which the algorithms mentioned above are particular implementations (the difference being the way each handles insertion on the OPEN table).

While expanding a node (taking its children), *DFS* inserts the child at the beginning of the OPEN table, that is, the algorithm prioritizes going all the way down through the left-most branch. If working with trees, this can lead to problems related to infinite branches (thus, not getting a solution in those cases, so it's not complete) and the A_TABLE storing information that sometimes can be unnecessary since we're adding all the children of a node and maybe never getting back to most of them. Nevertheless, memory consumption is linear to the depth of the tree.

*BFS*, however, inserts new nodes at the end of the OPEN table, thus working "level by level". This approach guarantees finding a solution if there is any (being then complete), but its memory consumption is exponential to the depth of the tree, since it stores all the nodes of the actual level in the A_TABLE, being this number of nodes exponentially higher as the algorithm goes down to the next level.

*UCS*, on the other hand, inserts each new node ordered by increasing cost (from the initial node to itself) into the OPEN table. Similar to the *BFS* algorithm, its memory consumption is exponential to the tree's depth, but the main advantage of *UCS* is its admissibility, since it ensures an optimal solution if it exists.

Other two relevant algorithms are *Depth Limited Search*, which is essentially a *DFS* limited by depth, and *Iterative Deepening Search*, which executes *DLS* iteratively, each time with increasing depth values, partially simulating the behavior of *BFS* but being in fact a *Depth-first* approach. For the former, a given limit is set, but coming up with a good limit is not an easy task. *IDS* is very helpful to overcome this fact.

In order to complement these general definitions, an in-depth analysis of each of the aforementioned algorithms will be made on 5.2.1.

## 3 Heuristic Search Algorithms

Heuristic search is a technique involving the use of heuristics for determining moves, being extremely useful especially for problems of exponential nature that would otherwise be enormously expensive or even impossible to solve.

Quoting [1], a heuristic is described as *"a criterion, method or principle for deciding which among several alternative courses of action provides to be the most* *effective in order to achieve some goal".* They're based on specific knowledge about the problem's domain, and they improve the search process on average. They've an associated cost, so they're only useful if the improvement pays off.

## 3.1 A* Algorithm

The A* algorithm was described in [2] by Peter Hart, Nils Nilsson and Bertram Raphael in 1968. It's a particularization of the *General Graph Search Algorithm* (which is itself a modification of the *General Tree Search Algorithm*), but employing heuristics to guide search.

It's formulated as $f(n) = g(n) + h(n)$, where $g(n)$ is the cost of the best path from the initial node to $n$ obtained so far, $h(n)$ (also called heuristic function) is a positive estimation of the cost of the shortest path from $n$ to the goal closest to $n$, so that $h(goal) = 0$, and being $f(n)$ then an estimation of the cost of the shortest path from the initial node to the goal, conditioned to go through $n$, and whose value is used to order the OPEN table ascendingly. Also, for each found node, $g(n)$ and $h(n)$ are registered in the A_TABLE.

Before going on, it's interesting to define some concepts, starting with consistence and monotonicity. If, for all pairs of nodes $n$ and $n'$, $h(n) \leq c(n,n') + h(n')$ where $c(n,n')$ stands for the cost of the rule used to go from $n$ to $n'$, we can say that the heuristic is consistent. Similarly, we can say it's monotone if $h(n) \leq k(n,n') + h(n')$ where $k(n,n')$ represents the minimum cost from $n$ to $n'$. Consistency and monotonicity are equivalent properties, and any consistent or monotone heuristic is always admissible. Therefore, A*, by another property, is admissible if it uses said heuristic.

If a heuristic is monotone, no rectification to the path from the initial node to already expanded nodes is needed, since the path found so far from the initial node to a node $n$ is optimal.

Lastly, we say that an admissible heuristic A dominates (or is more informed than) an admissible heuristic B if, for all states, A gives a higher value than B. That is, if $h_B(n) < h_A(n) \leq h^*(n)$ for all non-final $n$, where $h^*(n)$ is the optimal heuristic. If A dominates B, A is a better heuristic for the A* algorithm.

## 3.2 Static Weighting A*

As explained in [3], Static Weighting A* is a version of A* with $f(n) = g(n) + (1+\varepsilon)\, h(n)$, where the optimality of the solution is sacrificed in favor of performance, but controlling the quality loss of the solution with the parameter $\varepsilon$, which represents the maximum distance to the optimal cost. This approach is, then, not admissible, but it's said to be $\varepsilon$-admissible if it always finds a solution with a cost not exceeding $(1+\varepsilon)\, C^*$, where $C^*$ is the cost of the optimal solution.

## 3.3 Greedy Best-First Search

Greedy algorithms take decisions in an irrevocable manner. Thus, they're not admissible and, usually, they're also not complete. The benefit of these algorithms is their huge efficiency and, if they're guided by a good heuristic, their increased probability of obtaining a good solution.

Greedy Best-First Search algorithms are greedy algorithms where we select the node with the lowest *f(n)* value and never go back. They are formulated as *f(n) = h(n)* whereas, on the contrary, Uniform Cost Algorithms, explained in section 2, can be expressed as *f(n) = g(n)*. One problem of GBFS algorithms is that they could get lost into infinite search spaces but, if the search space is finite (i.e. a graph), although not admissible, they're very good.

## 4 The 8-puzzle Problem

The 8-puzzle is a smaller version of the 15-puzzle game invented by Noyes Palmer Chapman in the 1870s which reached such a huge popularity in the United States and Europe that even rewards were offered for those who were able to solve it from some given initial states.

### 4.1 Problem Description

The game is played on a 3-by-3 grid with 8 square blocks labeled 1 through 8 and a blank square. The goal is to rearrange the blocks so that they are in (a given) order. This can be done by sliding blocks horizontally or vertically into the empty square.

### 4.2 Modeling in the Frame of Search in State Spaces

This problem can be modelled as a state space consisting of states (a specification of each of the tiles in the board), an initial state (any state), a list of possible actions (different movements), a goal state, and path costs (which are always 1 for each movement).

The space can be thought of as a tree, where one state with a given arrangement, after applying a movement, leads to another state. This has a very big drawback in this problem: if we arrive at a state which we've already visited, we treat it just as a new one, consuming extra memory. This behavior can be improved by simply considering the space as a graph. This way, states that have already been visited don't take extra space in memory, increasing drastically the algorithms' performance over this problem. The differences between these two approaches will be clearly exemplified in section 5.

### 4.3 Heuristics

The heuristics we're going to employ in this problem are the following:

– $h_0(n)$ = no heuristic (thus $h(n)=0$, so A* works just as Dijkstra's algorithm)

– *$h_1(n)$* = number of blocks which in state *n* are out of their desired position in the goal state.
– *$h_{1\text{-}2}(n)$* = add one for each block which is not in its corresponding column in the goal, also add one for each block not in its corresponding row.
– *$h_2(n)$* = sum of the orthogonal distances (Manhattan distances) from each block to its position in the goal state.
– *$h_3(n)$* = 2 × number of blocks which are at an orthogonal distance of 2 from their position in the goal.
– *$h_4(n)$* = variation of $h_2$ following Fig. 2 of [4], defined as *$1.31h_2 - 2.19$.*

$h_0$, $h_1$, $h_{1\text{-}2}$, $h_2$ and $h_3$ are admissible, while $h_0$, $h_1$, $h_{1\text{-}2}$ and $h_2$ are also monotone. Moreover, it's easy to conclude that $h_{1\text{-}2}$ dominates $h_1$ and that $h_2$ dominates $h_{1\text{-}2}$, since the conditions for dominance specified at the end of 3.1 are fulfilled in both cases.

## 5 Experimental Study

In this section, different experiments will be made, using a varied set of examples and carefully measuring and comparing different algorithms and approaches for the 8-puzzle problem.

### 5.1 Sample Set

The sample set is comprised by 9 examples of initial states that can be solved optimally in 5 steps, 10 examples of cost 10, 10 examples of cost 15, 10 examples of cost 20, 5 examples of cost 25 and 5 examples of cost 30.

### 5.2 Experimental results

The results from multiple experiments will be shown, represented, compared and analyzed carefully in order to provide a better insight into the different algorithms and search strategies explained in the previous sections. It's important to highlight that only results from executions not surpassing 10 seconds were recorded for the sake of swiftness (in most cases, letting it run freely would lead to the program running out of memory and/or paging it, anyway), since we can already derive enough conclusions about the poor performance of said algorithms in those cases.

#### 5.2.1 Brute-force Search

The application of *Depth-first Search*, *Breadth-first Search* and *Uniform Cost Search*, both modelling the problem as a tree and as a graph for all the examples of the set yields the results shown below. The following table represents the mean number of expanded nodes with each algorithm, over a tree and over a graph, for each category of examples.

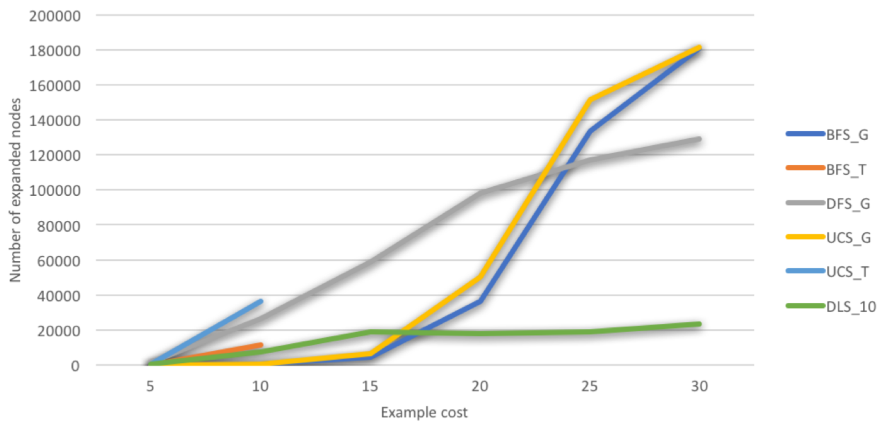

**Figure 1.** Number of expanded nodes with each algorithm over the 8-puzzle problem represented as a tree and as a graph for each subset of examples.

First of all, apart from the data in Figure 1, it's important to note that, among the three basic algorithms, in the 8-puzzle problem, both *BFS* and *UCS* always find the optimal solution, since the best sequence of states is always the shortest (and, as we've seen, the cost of all edges in this problem is always 1, so the least deep solution, i.e. the first one *BFS* and *UCS* get, is always optimal) while *DFS* doesn't.

Also, as Figure 1 exhibits, when representing the state space as a tree, all three algorithms show much worse results than when understanding it as a graph. More specifically, *DFS* is unable to find a solution for any example without running out of memory since, as explained in previous sections, apart from being a lot more states, there can be infinite paths *DFS* can get lost into. On the other hand, *BFS_T* can at least get the solution expanding sufficiently few nodes for the examples of cost 5 and 10. These amounts of expanded nodes are clearly higher than the ones we get applying the same algorithm to the problem represented as a graph, and this difference is much more pronounced as the cost of the examples increase. The same behavior as with *BFS* can be noticed in both versions of *UCS*, and this has its explanation in the way costs are managed in the particular case of the 8-puzzle, where all the edges have the same cost, 1, so *UCS* can't apply its characteristic ordering, leading then to simply introducing each new node at the end of the A_TABLE, having this way the exact same functioning *BFS* has.

Note here that the difference in expanded nodes between *BFS* and *UCS* is due to the implementation provided in aima-java, since for *BFS* it checks if a state is an objective before introducing it in the OPEN table, while for *UCS*, this check is done after, traversing then an extra level of the search space.

Following, some recursive approaches solely focused on the tree-like representation of the problem are discussed. First, if we apply a *Depth Limited Search* with a depth limit of 10, the results are obtained very quickly but, obviously, the algorithm only finds the optimal solution for examples whose optimal cost is the same as the maximum depth we set, that is, for the examples of optimal cost 10. For examples with lower optimal cost, the solution is not guaranteed to be optimal (because of the properties of *DFS* we talked about in previous sections). Besides, no solution can be found for examples of an optimal cost greater than the limit, therefore, it's not complete.

Applying this same algorithm with a limit of, for instance, 20, would give us optimal results for 20-cost examples, but the number of expanded nodes and, hence, the time, are much bigger than with a smaller limit (a hundred times more nodes than DLS_10 for 15-cost examples), making its use unreasonable even for examples of cost 20.

Anyway, this algorithm avoids one of the biggest problems of standard *DFS*, which is infinite branches, allowing us to find solutions expanding a more or less reasonable number of nodes for the simpler examples. Also, we can conclude from these two examples the importance of estimating a good depth limit according to our needs.

This necessity is covered by *Iterative Deepening Search*, which ensures optimal solutions in all cases, although for the higher-cost examples (and, thus, deeper iterations) the weaknesses of applying *DFS* over a tree start flourishing. Additionally, wasted computation is performed, since executions done with limits lower than the goal depth are discarded, as highlighted in [5], although this is not that big of a problem since the greatest portion of work is done at the deepest level. This algorithm has the admissibility advantage of *BFS* but with just a linear memory consumption as *DFS* (albeit its time complexity is exponential to the depth).

### 5.2.2 Heuristic search

Now, an experimental analysis is carried out with heuristic algorithms having the different heuristic functions described in 4.3 as protagonists.

| | **$H_0$** | **$H_1$** | **$H_2$_T** | **$H_2$** | **$H_{1-2}$** |
|---|---|---|---|---|---|
| **EXP. NODES** | 181365.8 | 104199.0 | 479860.5 | 6746.2 | 39212.8 |
| **TIME (MS)** | 819.2 | 1193.8 | 3893.75 | 39.2 | 328.2 |

**Table 1.** Mean number of expanded nodes and time taken with heuristics $h_0$, $h_1$, $h_{1-2}$ and $h_2$ over the 8-puzzle problem while performing graph search, and heuristic $h_2$ while performing tree search ($H_2\_T$). The measures are taken with 30-cost examples.

From the data displayed in Table 1, it's easy to see that $h_2$ is by far the best one of the four heuristics compared and is even able to find solutions in a reasonable time when performing the search over a tree, while the other heuristics take too long.

Going further into comparisons, $h_0$ is the worst heuristic (it's, indeed, a null heuristic). $h_1$ is better but still expands quite a big number of nodes. $h_{1-2}$ is an intermediate solution between our best heuristic, $h_2$, and $h_1$. In fact, as stated in section 4.3 we can prove that $h_{1-2}$ dominates $h_1$, in the same way that $h_2$ dominates $h_{1-2}$.

Keep in mind that these four heuristics are all monotone, so we can apply graph search without neither rectification nor reinsertion, as explained in section 3.1. We are going to see now what problems arise when trying this search using a non-monotone heuristic.

| | $H_2$_G | $H_2$_RECT | $H_2$_REIN | $H_3$_G | $H_3$_RECT | $H_3$_REIN |
|---|---|---|---|---|---|---|
| **EXP. NODES** | 6746.2 | 6732 | 6732 | *129483.2* | 127701.8 | 131468.6 |
| **TIME (MS)** | 39.2 | 41.2 | 38.2 | *1430.4* | 1613.4 | 1442.2 |

**Table 2.** Mean number of expanded nodes and time taken with heuristics *$h_2$* and *$h_3$* over the 8-puzzle problem while performing graph search, graph search with rectification and graph search with reinsertion. The measures are taken with 30-cost examples.

The non-monotone heuristic that will be tested is $h_3$. So, apart from the information shown in Table 2, the solutions obtained from the application of a graph search with $h_3$ are not optimal, getting, for instance, 32 as solution for one of the examples of cost 30 considered. Thus, the usage of node rectification or node reinsertion is necessary in order to get optimal solutions when using this kind of heuristic.

Now, heuristic *$h_2$* with static weighting and three different ε values will be compared with the standard version of the heuristic. The following measurements are the mean results of the application of graph reinsertion for examples of cost 30.

| | $H_2$ | $H_2$_0.01 | $H_2$_0.05 | $H_2$_1.0 |
|---|---|---|---|---|
| **EXP. NODES** | 6732 | 5292 | 1535.4 | 1399.6 |
| **TIME (MS)** | 38.2 | 62.4 | 12.6 | 12.2 |
| **SOLUTION COST** | 30 | 30 | 30.4 | 31.2 |

**Table 3.** Mean number of expanded nodes, time taken, and solution cost with heuristic *$h_2$* without static weighting and with static weighting with $\varepsilon = 0.01$, $\varepsilon = 0.05$ and $\varepsilon = 1.0$ over the 8-puzzle problem while performing graph search with reinsertion. The measures are taken with 30-cost examples.

In Table 3, the reduction in the number of expanded nodes is clear. Moreover, for this example, the static weighting version with $\varepsilon = 0.01$ still gives optimal solutions, but we have to keep in mind that it's not an admissible heuristic, so with different examples the results may not be optimal. Having a look to the other two versions, the one with $\varepsilon = 0.05$ seems to be the most balanced configuration, having a very limited spoilage on the results but expanding far less nodes than the first two options. The version with $\varepsilon = 1.0$ provides solutions with a significantly bigger error, and the number of expanded nodes is just slightly lower than with $\varepsilon = 0.05$, so it's not that good of a choice. The greatest difficulty in this kind of algorithm is finding a good value for ε, as we've just demonstrated.

Lastly, non-admissible algorithms with non-limited error are going to be analyzed. More specifically, A* with a non-admissible heuristic and a *Greedy Best-First Search* algorithm (non-admissible) with an admissible heuristic are compared in the following table.

| | GBFS | $H_4$ | $H_2$ |
|---|---|---|---|
| **EXP. NODES** | 218.6 | 3019.2 | 6746.2 |
| **TIME (MS)** | 42.8 | 35.8 | 39.2 |
| **SOLUTION COST** | 52.4 | 30.4 | 30 |

**Table 4.** Mean number of expanded nodes, time taken, and solution cost with Greedy Best-First Search over a graph with heuristic $h_2$, A* over a graph with node rectification with heuristic $h_4$ and A* over a graph with $h_2$. The measures are taken with 30-cost examples over the 8-puzzle problem.

As shown in Table 4, the benefit in performance provided by *GBFS* is remarkable, but the solutions obtained are far from being optimal. Focusing on the results obtained with a graph search with rectification using $h_4$, we can notice just a low deterioration on the solutions, but a number of expanded nodes that is much greater than with *GBFS* but still half the amount of standard $h_2$.

Comparing this results with the ones obtained for the static weighting variants of $h_2$ in Table 3, the solutions obtained with any of the three tested values of ε in static weighting are much closer to optimality than the ones obtained with *GBFS*, while the number of expanded nodes is greater in the weighted approach. On the other hand, $h_4$ produces the same low spoilage on the results as static weighting with $\varepsilon = 0.05$, but expanding double the number of nodes, thus being the solution shown in Table 3 better than $h_4$ in this situation.

## 6 Conclusions

This paper discussed two main kinds of search algorithms (brute-force and heuristic algorithms) over a given search problem, represented as a state space: the 8-puzzle.

With tree-like space representations, the performance decays because of reconsideration of repeated states while, when representing the space as a graph, this problem is avoided. With brute-force algorithms, the search process is uninformed, so the number of nodes visited by the algorithm is bigger. In order to reduce this number and, thus, runtime, heuristic algorithms with domain-specific knowledge are employed to help us choose the node which seems the most appropriate in each step, optimizing paths and, therefore, arriving faster to a solution. Sometimes, this is not enough, and we need better performance, while being willing to sacrifice some quality in the results in benefit of speed. This sacrifice can be limited or not, depending on the spoilage we want to allow in the results.

In informed search algorithms, being able to devise a good heuristic function for the problem is a crucial task that provokes a heavy impact on the performance of the algorithm. That’s why it’s essential to understand key properties such as consistency, monotony, admissibility and dominance to help us come up with a good heuristic.

At the end, the approach taken depends heavily in the particularities of the problem to be solved, and in our own objective, be it obtaining the best of the solutions, a sufficiently good one, or just the first solution we can get in the minimum time.

There’re different algorithms for different purposes, and knowing the differences and similarities of at least the most popular ones is key to understand other algorithms and to choose wisely which one is needed for each particular situation we face.

## Bibliography

1. Pearl, J.: Heuristics. Morgan Kauffman, San Francisco, CA (1983).
2. Hart, P., Nilsson, N., Raphael, B.: A Formal Basis for the Heuristic Determination of Minimum Cost Paths. IEEE Transactions on Systems Science and Cybernetics, vol. ssc-4, no. 2, pp. 100–107 (1968).
3. Sierra Sánchez, M.R.: Mejora de Algoritmos de Búsqueda Heurística mediante Poda por Dominancia. Aplicación a Problemas de Scheduling. Colección Tesis Doctoral-TDR no. 72, pp. 69–70, University of Oviedo, Oviedo (2009).
4. Stern, R et al.: Potential-based bounded-cost search and Anytime Non-Parametric A*. Artificial Intelligence 2014, pp. 7, Elsevier Science Publishers B.V. (2014).
5. Korf, R.: Depth-first Iterative-Deepening: An Optimal Admissible Tree Search. Artificial Intelligence 27, pp. 97–109, Elsevier Science Publishers B.V., New York (1985).