An Evaluation of Uninformed and Informed Search Algorithms on the k-puzzle Problem
AY19/20 Semester 2 CS3243 Project 1, Group 37

Lim Fong Yuan, Zhuang Xinjie, Otto Alexander Sutianto, and Mario Lorenzo

School of Computing, National University of Singapore

1 Problem Specification

Fix $n \in \mathbb{Z}_{\geq 2}$. Let $k = n^2 - 1$. A valid state v is a $n \times n$ array with the entries $v_{(x,y)}$ containing a permutation of the integers $[0, k]$. V is the set of valid states. Each valid state has a unique coordinate $e = (x_e, y_e)$ such that $v_e = 0$. Call v_e the empty cell of v. The goal state g is where $g_{(x,y)} = (x - 1)n + y$ for all $(x, y) \in (\mathbb{Z}_{[1,n]})^2$, except for $g_{(n,n)}$ which is 0. Let $A = \{\texttt{up} = (-1, 0), \texttt{down} = (1, 0), \texttt{left} = (0, -1), \texttt{right} = (0, 1)\}$ be the set of actions. The transition function $T : V \times A \to V$ is where $T(v, a) = v'$ with v' being identical to v except with $(v'_{e-a}, v'_e) = (v_e, v_{e-a})$.

The problem is to determine if g is reachable from a given initial state s, and if so, specify a sequence of actions $p \in A^*$ (called a solution) that leads from s to g.

Throughout this report, the following variable definitions are used:

Let d be the depth of the shallowest goal node g.

Let b be the maximum branching factor of the problem space. $b = 3$, see A.1.

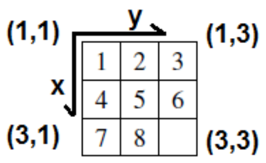

Fig. 1. The coordinate system on the goal state, in the case of $n = 3$.

2 Technical Analysis

2.1 Breadth-First Search (BFS) (Uninformed)

BFS explores all reachable nodes of depth d' before it starts exploring nodes of depth $d' + 1$. Hence, there are at most $\sum_{i=0}^{d} b^i = \frac{b^d - 1}{b-1} = \frac{3^d - 1}{3-1} \in O(3^d)$ nodes that BFS will explore before it explores g.

Correctness and Optimality The algorithm first checks for solvability (see Appendix A.2), terminating if it is not. If a goal node g exists, BFS explores a finite number of nodes before finding the shallowest goal node g. Then, it backtracks through its search tree to derive the sequence of actions that leads from s to g. All actions that BFS has explored are valid, and the sequence leads from s to g, therefore BFS returns a correct sequence. BFS finds an optimal solution because it explores all nodes of depth $d-1$ before it explores nodes of depth d. If there were another solution at depth $d-1$ or closer, BFS would have found it first.

Efficiency Each node is explored in constant time, thus BFS has $O(3^d)$ time complexity. BFS tracks all explored and seen nodes in memory, Which amount to at most $\sum_{i=1}^{d+1} b^i \in O(b^{d+1}) = O(b \cdot b^d) = O(3 \cdot 3^d) = O(3^d)$, thus BFS has $O(3^d)$ space complexity.

2.2 A* Search (Informed)

A* explores the node v with the least estimated cost $f(v) = g(v) + h(v)$ to the shallowest goal node g, where $g(v)$ is the cost from the initial state s to v, and $h(v)$ is an admissible estimate of the cost from v to g.

Let $d(v, v')$ be the optimal distance between any two nodes v and v'.

Assuming g exists, let p be an optimal path from s to g. For any node v in p, note that $f(v) = g(v)+h(v) \leq g(v)+h^*(v) = d(s,n)+d(n,g) = d(s,g) = d$. Also note that at least one node in p will be in the frontier of A* at any given point, hence there is always at least one node v in the frontier with $f(v) \leq d$. Therefore, A* will explore at most all reachable nodes v such that $f(v) \leq d$ before finding g. In particular, it will search at most all nodes v of depth $d' = d(s,v) = g(v) \leq g(v)+h(v) = f(v) \leq d$. Hence, there are at most $\sum_{i=0}^{d} b^i = \frac{b^d-1}{b-1} = \frac{3^d-1}{3-1} \in O(3^d)$ nodes that A* will explore before it explores g.

Correctness and Optimality Same as with BFS above.

Efficiency Same as with BFS above, A* has $O(3^d)$ time and $O(3^d)$ space complexity, in the worst case (i.e. constant heuristic). The time complexity may be improved depending on the heuristic, generally taking on the form $O(b^{d-h(s)})$, or $O((b^*)^d)$ (where b^* is the effective branching factor of A* using h, which has to be empirically measured), see [2]. At best, under the optimal heuristic h^* (where $h^*(v)$ returns the actual cost from v to g), A* runs in $O(d)$ time.

Heuristics For any (valid) state v and any successor v' of v, their difference is the position of a single tile T (the empty cell is not counted), which has swapped places with the adjacent empty cell. Their absolute difference in cost $d(v, v') = 1$. Let $d_h(v, v') := h(v') - h(v)$ be their difference in heuristic cost.

h_1: Sum of Manhattan distances of each (non-empty) tile from its final position. This heuristic is consistent.

h_2: Number of misplaced (non-empty) tiles. This heuristic is consistent.

h_3: Linear conflicts. [1] This heuristic is a refinement of h_1. A pair of tiles are in horizontal (resp. vertical) *linear conflict* if they are in the same row (resp. column), both their goal positions are also in the same row (column), and they have to cross over one another in order to reach their goal positions. In order to swap the two tiles, one of the tiles has to minimally move out of the row (column) to let the other tile pass, then move back into the row (column) after the two tiles have moved past each other, incurring an additional cost of at least 2.

In a row (or column) with multiple conflicts, the tile with the most conflicts is identified and moved out, incurring a cost of 2. The remaining row is then scanned for conflicts again and the tile with the most conflicts is again identified and removed. This process continues until the row has no more conflicts. This process identifies the minimum number of tiles needed to be displaced in order to resolve all conflicts in the row.

Thus, $h_3(v) := h_1(v) + h'_3(v)$, where $h'_3(v) = 2\sum_{i=1}^{n}(l(r_i) + l(c_i))$, where r_i (resp. c_i) denotes the i-th row (resp. column) in state v, and $l(r)$ (resp. $l(c)$) is the total number of tiles that are displaced in order to clear row r (resp. column c) of conflicts.

Proof of consistency: For each move, T either moves out of its goal row (or column), stays out of its goal row, stays in its goal row, or moves into its goal row. If T moves out of its goal row, it necessarily moves farther away from its goal position, thus $d_{h_1} = 1$. T may or may not have been a tile that was identified for removal. If it was, then $d_{h'_3} = -2$, thus $d_{h_3} = 1 - 2 = -1$. If not, then $d_{h'_3} = 0$, thus $d_{h_3} = 1 + 0 = 1$. If T stays out of or in its goal row, it neither creates nor resolves any linear conflicts. It may either move closer to ($d_{h_3} = d_{h_1} = -1$) or farther away from ($d_{h_3} = d_{h_1} = 1$) its goal position. If T moves into its goal row, it necessarily moves closer to its goal position, thus $d_{h_1} = -1$. T may or may not introduce new linear conflicts. If it does, then it increases the number of tiles that need to be removed to resolve all conflicts by just 1, thus $d_{h'_3} = 2$, thus $d_{h_3} = -1 + 2 = 1$. If it does not, then $d_{h'_3} = 0$, thus $d_{h_3} = -1 + 0 = -1$. In all cases, $|d_{h_3}(v, v')| \leq 1 \implies h_3(v) \leq 1 + h_3(v') = d(v, v') + h_3(v')$, $\therefore h_3$ is consistent. □

Trivially, h_3 dominates h_1.

3 Experimental Setup

The performances of BFS and A*-search with each of the three heuristics have been experimentally measured, and consolidated in Table 1 and plotted in Figure 2.

Inputs are randomly generated from the set of $n \times n$ initial states. Unsolvable states are discarded. The optimal depth d is measured and the initial state is categorized accordingly. The metrics used are the absolute time taken to obtain an output, and the number of nodes seen. These metrics directly reflect the time and memory requirements of each algorithm respectively.

The algorithms were coded in Python 2.6.4 and the experiments were run on NUS's `sunfire` server. Absolute time was measured with Python's `time` package.

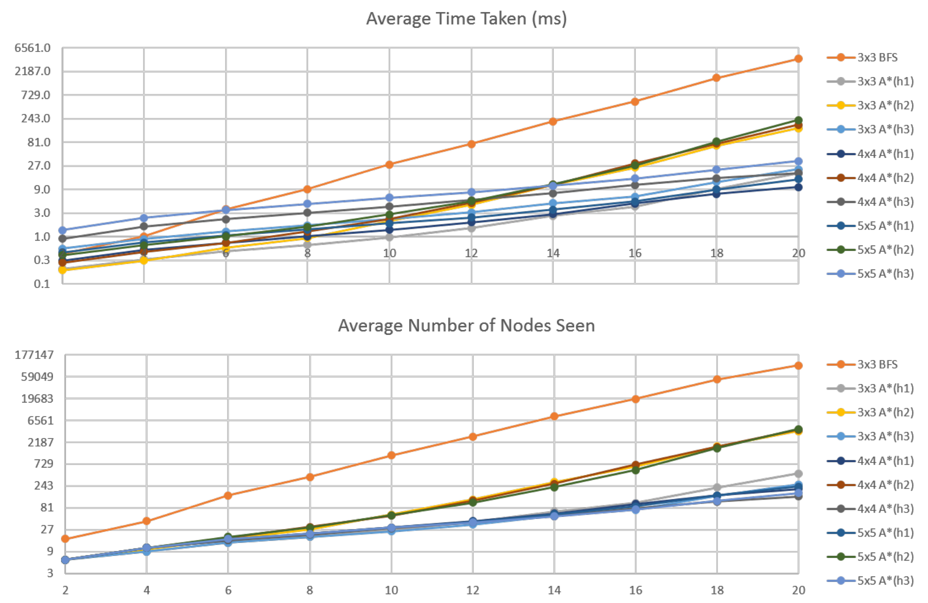

Fig. 2. Plot of the performance results of the proposed algorithms. The x-axis is the depth of the optimal solution. A logarithmic scale (base 3) is used for the y-axes so that the results are more visible.

4 Results and Discussion

The time and space complexities for all algorithms are all $O(3^d)$. The empirical results in Figure 2 supports this analysis, although some are even more efficient.

Figure 2 also shows that the algorithms in order of most to least time-efficient is A* with $h_3 \approx$ A* with $h_1 >$ A* with $h_2 >$ BFS. This agrees with our theoretical analysis regarding the dominance of one heuristic over another (where BFS may be regarded as A* with a heuristic that returns 0 everywhere).

Even though h_3 dominates h_1, A* with h_3 takes more time to execute than A* with h_1. This is because it is more expensive to calculate h_3 than h_1. However, h_3 always explores less nodes than h_1, and past a certain point (perhaps around $d \approx 25$), h_3 starts to become more time-efficient than h_1 as well.

Table 1. Performance results of the proposed algorithms. B stands for BFS, while h_i stands for A* with the h_i heuristic.

3x3		Avg time taken (ms)				Avg #nodes seen			
Depth d	#Trials	B	h_1	h_2	h_3	B	h_1	h_2	h_3
2	1115	.441	.225	.209	.571	17	6	6	6
4	1738	1.02	.351	.329	.889	42	9	10	9
6	2280	3.57	.506	.591	1.27	150	14	17	14
8	2112	9.15	.686	.941	1.68	382	19	27	19
10	1321	28.6	.969	2.05	2.23	1136	26	58	25
12	743	74.5	1.50	4.37	3.09	2881	40	122	35
14	312	212	2.64	11.0	4.73	7944	68	298	54
16	126	534	4.05	24.7	6.45	19076	103	650	74
18	34	1623	9.09	69.2	12.8	51102	224	1750	148
20	9	3933	18.8	155	23.0	104154	452	3850	264

4x4		Avg time taken (ms)			Avg #nodes seen		
Depth d	#Trials	h_1	h_2	h_3	h_1	h_2	h_3
2	528	.328	.297	.909	6	6	6
4	1337	.549	.493	1.62	11	11	11
6	1831	.753	.752	2.25	16	18	16
8	1989	1.01	1.27	3.04	22	31	21
10	1786	1.38	2.26	4.08	30	55	29
12	1266	1.94	4.79	5.46	42	117	39
14	720	2.85	11.2	7.49	61	269	54
16	284	4.69	30.3	11.0	99	712	80
18	107	7.25	76.3	15.1	152	1744	111
20	27	10.1	182	19.2	210	4087	143

5x5		Avg time taken (ms)			Avg #nodes seen		
Depth d	#Trials	h_1	h_2	h_3	h_1	h_2	h_3
2	478	.478	.425	1.35	6	6	6
4	1039	.768	.675	2.39	11	11	11
6	1578	1.06	1.01	3.44	17	19	17
8	1925	1.39	1.59	4.60	22	30	22
10	1891	1.86	2.82	6.10	30	56	30
12	1396	2.44	5.23	7.88	40	105	39
14	872	3.51	11.4	10.7	59	229	53
16	469	5.17	27.6	15.0	87	543	75
18	178	9.04	83.9	22.7	151	1698	114
20	64	14.5	228	33.8	238	4237	170
22	16	27.7	672	54.0	450	12083	273

A Appendix: Proofs

A.1 Maximum and Average Branching Factors

When determining the branching factor, backtracking actions that lead right back to the previous state are discounted, as that state has already been explored. This reduces the possible number of actions in all cases by 1, except for the initial state. However, the initial state incurs only a constant factor with regards to complexity analysis of the search tree, thus it can be safely ignored.

Maximum branching factor. Given any valid state, there are up to $|A| = 4$ possible actions. After discounting backtracking actions, the maximum branching factor $b = |A| - 1 = 3$. $\square$

Note: In the case $n = 2$, each cell is a corner cell and thus has only 2 actions, and 1 of them backtracks and thus is discounted. Thus $b = 1$ in this case. The complexities evaluated as $O(b^d)$ become wrong in this case as $O(1^d) = O(1)$, and they should be $O(d)$ instead.

Average branching factor. Given n, there are 4 corner cells of 2 branches each, $4(n-2)$ side cells of 3 branches each, and $(n-2)^2$ center cells of 4 branches each. Discount the backtracking action from each case. Assuming all states are equally likely to occur, the average branching factor b_{avg} is thus $\frac{1\cdot 4+2\cdot 4(n-2)+3\cdot(n-2)^2}{n^2} = \frac{4+8n-16+3n^2-12n+12}{n^2} = \frac{3n^2-4n}{n^2} = 3 - \frac{4}{n}$. $\square$

Table 2. Average branching factor for some values of n.

n	2	3	4	5	6	7	8
b_{avg}	1	5/3	2	2.2	7/3	17/7	2.5

A.2 Solvability of an Initial State [3]

Definition 1. *For any given state v, write out its entries in a sequence $S(v)$, ignoring the empty cell. i.e. $S(v)_i = v_{(\lceil i/n \rceil, ((i-1) \bmod n)+1)}$ for $i \in \mathbb{Z}_{[1,n^2]}$, then remove the empty cell from $S(v)$ (so it is now indexed from 1 to k). An* ***inversion*** *is a pair of indices (i, j) such that $i < j$, but $S(v)_i > S(v)_j$.*

Every state v has a well-defined number of distinct inversions, denote it $I(v)$. $I(v)$ can be calculated by tallying how many distinct pairs in v are inversions, which takes $O(k^2)$ time as there are ${}^kC_2 = \frac{k(k+1)}{2}$ pairs to check.

Given a state v, let $e_r(v) = n - x_e(v)$ be the row of the empty cell as counted from the bottom, and starting at 0.

Theorem 1. *An initial state s is solvable if and only if one of the following (mutually exclusive) cases is true:*

1. *n is odd and $I(s)$ is even.*
2. *n is even, and $I(s)$ has the same parity as $e_r(s)$.*

($\Longrightarrow$) A solvable state is one that can reach the goal state g in a finite number of actions. Since each action is reversible by an opposite action, this can be restated as: A state is solvable if and only if it is reachable from g in a finite number of actions.

Now given a solvable initial state s, consider a sequence of actions taken to move from g to s, and how $I(v)$ changes across it. For each action, consider its start state v and result state v'.

If an action moves horizontally, $S(v) = S(v')$, thus $I(v') = I(v)$.

If an action moves vertically, then $n-1$ swaps between adjacent elements occur from $S(v)$ to $S(v')$, and each swap either introduces or resolves an inversion, which flips the parity of $I(v')$.

If n is odd, then the number of swaps $n-1$ is even and thus the parity of $I(v')$ is always the same as $I(v)$. Thus, the parity of $I(v)$ stays the same over the entire path from g to s. Therefore, $I(s)$ has the same parity as $I(g)$. And $I(g) = 0$ is even.

If n is even, then $n-1$ is odd and thus the parity of $I(v')$ is always different from $I(v)$ (when the action is vertical).

If $e_r(s)$ is even (resp. odd), then the number of vertical actions taken to get to g is even (resp. odd) as well. Thus, the parity of $I(v)$ flips an even (resp. odd) number of times over the entire path from g to s. Therefore, $I(s)$ has the same (resp. different) parity as $I(g) = 0$, which is even. $\square$

($\Longleftarrow$) This is essentially a tedious proof by construction, and the proof is deferred. The general strategy is to move 1 to the right place and never move it again, then do the same with 2, then 3, and so on, up until the puzzle is solved. $\square$

References

1. Hansson, O., Mayer, A., Yung, M.: Criticizing solutions to relaxed models yields powerful admissible heuristics. Information Sciences **63**(3), 207–227 (Sep 1992). https://doi.org/10.1016/0020-0255(92)90070-O, https://doi.org/10.1016/0020-0255(92)90070-O
2. Russell, S.J., Norvig, P.: Artificial Intelligence: A Modern Approach, pp. 98,103. Prentice Hall, 3rd edn. (2010)
3. Ryan, M.: Solvability of the tiles game, https://www.cs.bham.ac.uk/ mdr/teaching/modules04/java2/TilesSolvability.html