Sokoban Solved - Modeling Sokoban Puzzles as Search Problems

Matthew Days and **Yangzi Jiang**
`{madays,yajiang}@davidson.edu`
Davidson College
Davidson, NC 28035
U.S.A.

Abstract

Sokoban, a puzzle game invented in Japan, poses a challenging problem for studying the performance of search algorithms due to its enormous number of possible states and often irreversible moves. In this project, we test three search algorithms on the Sokoban domain one informed, two uninformedspecifically, Iterative Deepening A* (IDA*), Monte Carlo Tree Search (MCTS), and Breadth-First Search (BFS). We also investigate the impact of different heuristics on our informed search. Additionally, we investigate ways to optimize the performance of these algorithms using the knowledge that is specific to the domain of Sokoban. Our results show that IDA* yields a more move-optimal solution, whereas BFS executes faster on multiple puzzle sets. Meanwhile, we found that MCTS is not suited for this complex domain without heavy modification and optimization.

1 Introduction

Sokoban (literally Japanese for "warehouse keeper") is a Japanese puzzle game in which the player character attempts to push boxes to goal spaces. Created in 1981 by Hiroyuki Imabayashi for Thinking Rabbit Inc. (Thinking Rabbit 1981), Sokoban's mechanics are simple, but the process of solving the puzzle can be deceptively complicated—a fact which makes the puzzle a good testing ground for different search algorithms(Junghanns and Schaeffer 1997).

While solving a warehouse-management game may not seem to be a useful task, working in this simplified domain can enable a better look at the factors that affect a search algorithm's performance. These algorithms have multiple real-life uses, such as path-finding in video games and advance planning for autonomous vehicles. As such, gaining a better understanding of these kinds of search problems can prove invaluable to further research in machine reasoning.

The performance of various search algorithms, when applied to Sokoban, is affected primarily by the amount of information given to an agent beforehand. The amount of domain-specific information given to an agent, when combined with heuristics that allow that agent to estimate its proximity to a solution, can make or break an agent's performance.

In this paper, we present the results of our own implementations of Iterative Deepening A* (IDA*), Breadth-First Search (BFS), and Monte Carlo Tree Search (MCTS) as applied to the Sokoban puzzle. We detail the rules and complexity of the domain space, explain our experiment methods and unpack the results of applying the aforementioned algorithms to two sets of pre-made Sokoban puzzles. We then conclude our paper with a discussion of these results and other areas we would pursue to further our experiments.

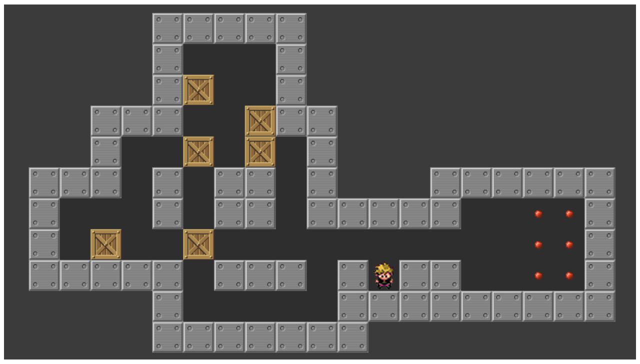


Figure 1: Initial state of Level-1 of the original Sokoban Image courtesy of `Thinking Rabbit`.

2 Background

Sokoban, Explained

The Rules The rules of Sokoban can be deceptively simple. It is played on a grid of squares, with each square acting as either a wall or floor square. Certain squares contain boxes, the player character, or markings that designate a spot as a goal position. The player itself can move vertically or horizontally, one empty square at a time (as such, the player may never pass through a wall or box). The player can also move into a box, pushing it to the space beyond. Boxes cannot be pulled, and they cannot be pushed into another box or or wall. Additionally, the number of boxes corresponds to the number of goal locations, and the game is considered solved when all of the boxes are pushed to those goal states.

Domain-Specific Information: Deadlocks From the typical layout provided by a given board state, some additional information can be derived that can reduce the number of states in the search space that our algorithms must traverse. We can supplement our Sokoban-playing agents by supplying them with greater capabilities to process state information specific to the Sokoban domain. One of the most important domain-specific additions we can provide is the ability to detect deadlocks, which are the unsolvable states that result from pushing boxes onto certain spaces that, due to the board configuration, render the box irretrievable without restarting the level. As such, we want our search algorithms to avoid those particular scenario if possible.

Generally speaking, there are two types of deadlocks: static deadlocks and dynamic deadlocks. Static deadlocks, sometimes called simple deadlocks, are static board positions that once a box enters in, it can never make it to a goal position. Dynamic deadlocks are the ones created based on the current configuration of the surrounding boxes and walls which render one or more boxes immovable. However, the task of detecting dynamic deadlocks can be an extremely complex one that could potentially involve every box. Thus, we look only for simple deadlocks—deadlocks that never change during game-play and, thus, can be detected ahead of time as the board layout is processed.

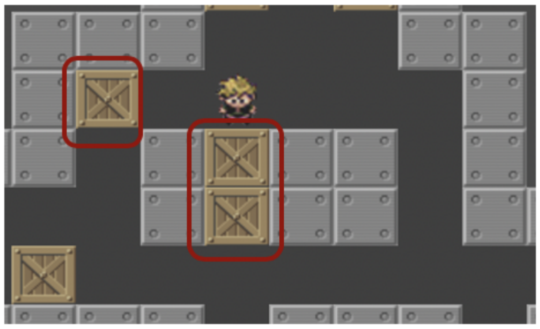


Figure 2: Example of a static deadlock (box in a corner) and a dynamic deadlock (frozen as a player cannot push two boxes together). Image courtesy of `Thinking Rabbit`.

A Complex Problem

Due to the existence of these deadlocks (in addition to the number of available move variations), Sokoban has a surprising complexity despite having relatively few components and mechanics. In fact, Sokoban has been proven to be both NP-hard and PSPACE-complete (Culberson 1997), which is generally assumed to be harder than NP-Complete problem. The implication of PSPACE-completeness is that there is no polynomial algorithm to verify whether a solution to a Sokoban puzzle has the least number of steps, not to mention finding a solution. Sokoban's surprising complexity but simple mechanics make it a good testing ground for various informed and uninformed search algorithms.

The Puzzle Sets Another benefit to using Sokoban to model search algorithms is the plethora of open source puzzle sets available. The community dedicated to designing and solving these puzzles is quite active, supplementing the original levels with a wealth of user-created, guaranteed-solvable levels. We tested our Sokoban agents on puzzles from two sets: the Microban set [1] and the SokWhole set [2]. Microban puzzles are generally designed to be easier, often targeted towards beginners and children (Skinner 2010). Usually consisting of smaller levels intended to demonstrate integral puzzle concepts or mechanics, the Microban set, designed by David W. Skinner, provided a good warm-up to start our agents against. The SokWhole puzzle collection, generated by the SokEvo application, had a higher complexity curve, ramping up rapidly in difficulty as the level number increased (Haywood 2011).

Search Algorithms

We chose to apply multiple search algorithms to our various Sokoban puzzles, using Breadth-First Search (BFS) and Monte Carlo Tree Search (MCTS) as examples of uninformed searches and Iterative Deepening A* (IDA*) to exemplify an informed search.

Monte Carlo Tree Search (MCTS) BFS is representative of uninformed searches in that it searches naively and resorts to visiting every possible state to eventually find the answer if it exists. Meanwhile, MCTS straddles the line between informed and uninformed searches. Unaltered, the basic MCTS searches essentially at random until it gathers enough information to find the most promising moves, a process that usually occurs only after a great deal of rounds (Chaslot et al. 2008). Such delay is inherent to a basic MCTS—also known as Pure Monte Carlo Tree Search—which is structured in four main steps that are repeated each round (See Figure 2). These steps consist of Selection, Expansion, Simulation, and Backpropagation. In selection, the algorithm starts from a root node and selects successive child nodes until a certain leaf. During expansion, that leaf node is expanded and child nodes are created. Then, during simulation, one of the child nodes is chosen as a starting node and the game is played out until a goal state is reached. Finally, during backpropagation, the result of the playout is used to update information on the nodes from one of the child nodes to the original root.

Memory-Enhanced Iterative Deepening A* (IDA*) Representative of informed search is IDA* (Reinefeld and

[1] `http://www.abelmartin.com/rj/sokobanJS/Skinner/David\%20W.\%20Skinner\%20-\%20Sokoban.htm`

[2] `https://leehaywood.org/games/sokoban/`

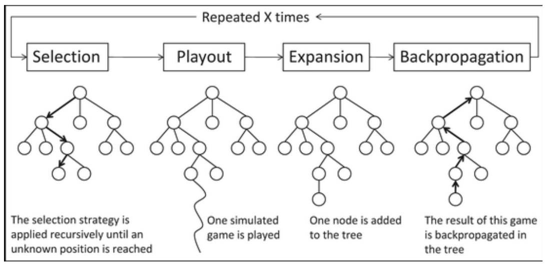

Repeated X times
Selection
Playout
Expansion
Backpropagation
The selection strategy is applied recursively until an unknown position is reached
One simulated game is played
One node is added to the tree
The result of this game is backpropagated in the tree

Figure 3: The Four Steps of MCTS, Image courtesy of `SpringerNature`.

Marsland 1994). This search algorithm acts as a cross between iterative deepening depth-first search (IDDFS) and basic A*. While IDDFS uses node depth as as a cutoff for each iteration, IDA* uses a more informative cost limit, which is based on a heuristic to guess the cost to travel from a node to the goal. This cost limit is given by the function $f(n) = g(n) + h(n)$, where $g(n)$ represents the cost to get to a node n and $h(n)$ represents that heuristic function. The algorithm uses this cost function to iteratively perform a DFS until it reaches a node whose cost exceeds the cutoff. Then, on the next iteration, the minimum cost of the values that exceeded the previous cutoff is chosen to be the new cutoff. Such a DFS approach is particularly beneficial when the problem is memory constrained since basic IDA* does not keep track of any node except the ones on the current path. However, the difficulties that Sokoban poses do not necessarily lie in a memory constraint; rather, they originate in the potential size of the search space. As such, we alter IDA* to maintain a transposition table of visited states, ensuring that the search algorithm does not revisit nodes. Additionally, IDA* is complete and optimal, meaning that it is guaranteed to find a goal state if it exists, provided that the heuristic is admissible.

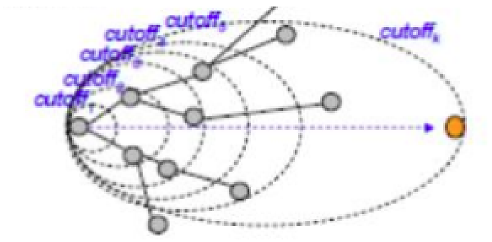


Figure 4: IDA* Cutoffs, Image courtesy of `SlideServe`.

Heuristics

As mentioned above, the heuristic function is crucial to the efficiency and efficacy of an informed search provided that it is admissible, that is, if it never overestimates the cost of reaching the goal, or rather, if it provides a lower-bound calculation for the cost to reach a goal state(Geffner and Haslum 2000). Should a heuristic function give no information, IDA* would essentially perform identically to IDDFS, potentially losing its optimality and completeness; meanwhile, if a heuristic provided a perfect cost estimation, IDA* would always follow the optimal path and never expand any unnecessary nodes. However, in terms of Sokoban, possible heuristic functions depend on what is defined as optimal. In the Sokoban community, the question of optimality takes on two meanings: one related to pushes, and one related to moves. For a Sokoban solution to be push optimal, it must minimize the number of pushes needed to move boxes towards the goal state. For a solution to a puzzle to be move-optimal, it must minimize the total number of moves needed.

Measuring Performance

Though not always indicative of the efficacy of heuristics and search algorithms, certain metrics can be calculated to compare performances. We record the number of nodes generated by a search algorithm, as, it follows, that the fewer nodes generated, the fewer nodes explored, and thus the performance of the algorithm is better. We also keep track of the number of nodes in the fringe, the number of nodes explored, and finally the run-time duration and the percentage of puzzles solved from the set.

3 Experiments

We looked to apply different types of search algorithms, consisting of both informed and uninformed searches, to the complicated domain of Sokoban, testing the performance of the different search algorithms and the different components of those algorithms that affect its performance (i.e. the heuristics used, the bound increment, and the amount of knowledge given to the algorithms). We implemented a brute-force BFS algorithm to find some baseline measurements and represent our uninformed search. We then worked to implement basic MCTS, which, unfortunately, is not well-suited for the complicated domain space of Sokoban. Finally, for our informed search, we implemented ME-IDA*.

We varied several components of our ME-IDA* implementation—specifically, the heuristic and the bound increment. In terms of heuristics, we calculated them based on distances between a box and the nearest goal state and the distance between that box and the player character, using both a Manhattan distance function and a Euclidean distance function. Due to the illegality of diagonal moves in Sokoban, the Manhattan distance function counts the sum of the horizontal and vertical distances between a box's position and the closest goal state. Similarly, we experimented with using a Euclidean distance as well, in which we calculated the direct distance between box and goal space and box and player.

Bound increment is the hyper-parameter we control to adjust how many levels deeper we allow the ME-IDA* to search in the tree in the next iteration, if the algorithm could not find the goal state under the current depth limit. Specifically, we experimented with three different settings: the default bound-increment = 1, the more aggressive bound-increment = 4, and the dynamically adjusted bound-increment that works as the following. As the number of iterations ME-IDA* has searched increases, the bound-increment decreases. The rational behind this conservative method is that the search space increases exponentially as the algorithm searches deeper and deeper into the tree. Searching Sokoban has a branching factor of 4 in the worst case. Therefore, ME-IDA* should be more conservative in later iterations. The comparison of the impacts from these bound increment values in shown in Table 1 & 2. As indicated in those two tables, ME-IDA* with bound-increment = 4 often outperforms the rest in running time and search space.

These search algorithms and variations were tested on puzzles from the two aforementioned sets. and improve run time, by reducing the search space as well as the branching factor of the search tree. Once deadlocks are detected, we removed them from the search spaces as we exclude these positions from the set of available moves.

Detecting Static Deadlocks

As discussed in Section 2, deadlocks generally fall into two categories, static deadlocks and dynamic deadlocks. We only detected the static deadlocks due to the complexity and performance hit of detecting the dynamically occurring deadlocks. The complexity comes from the numerous types of dynamical deadlocks, each can occur in many forms. For example, corral deadlocks is an area where a player cannot reach. This area can be create by very distant boxes and may not always cause the board to be unsolvable. The performance hit is due to the scanning of deadlocks for each unique state of the board, a node in the tree structure. On the other hand, static deadlocks only requires one scan when initializing the puzzle board.

Corner and Boarder Deadlocks Among the static deadlocks, we detected deadlocks that are in corners and boarders as they are the most common one. Corner deadlocks occur in one of the following configurations ($ denoting a box, and # indicating a wall).

```
##   #$#
#$   ###
```

Boarder deadlocks can exist on the boarder positions, along the edge of a puzzle. Our algorithms recognized that boarder positions are not deadlocks if they share either the same board column or board row with a goal. Boarder-like deadlocks may also rarely occur in the inner spaces but only for extremely large puzzle.

Here is the example of using our corners and boarders deadlock detection in a Sokoban puzzle *simple4*. We permit the trivial markings of deadlocks outside the board wall, as they will never be considered as movable positions in the search tree.

```
########
#x . .x#
#x   #x#
## # #x#
x#  $ x#
 #  $@x#
x#xxxxx#
 #######
```

Figure 5: The state of *simple4* after detecting static deadlocks (X - deadlock; @ - player; # - wall; $ - box; . - goal; (blank space) - space).

Impact of Deadlock Detection The impact of deadlock detection is significant. Without deadlock detection, in one trial, BFS spent 180.09 seconds to solve *simple4*, as 21,395 nodes were generated and 5,103 were explored. With both corner and board deadlocks detection, BFS solved it in only 12.22 seconds by exploring 590 nodes, while generating 1,899 nodes.

Because of this huge performance improvement, we decided to always employee deadlock detection to assist all the search algorithm we implemented. We intended to include the domain specific knowledge for all search algorithms as the focus of this project is not researching domain specific improvements, rather it is to compare search algorithms and heuristics.

Heuristics

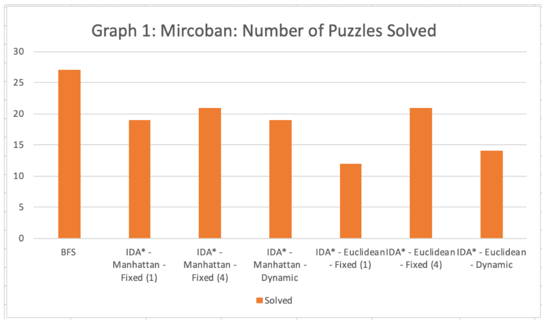
Manhattan and Euclidean Distance As discussed earlier, we implemented two heuristic functions, Manhattan distance and Euclidean distance, to guide ME-IDA* by accounting both the distance between the player and boxes as well as the distances between boxes and goal positions. In other words, we used heuristics to guide the searching on both player moves and box pushes.

Calculating Heuristic Value We computed the heuristic value for the current state of board. The method is to find the minimum distance, using Manhattan or Euclidean functions, from an object to a set of coordinates. When calculating the heuristic value between the player and boxes, it is the distance between the player and the ”nearest” box. Similarly, the heuristic value between boxes and goals is the sum of the distances between each box and its ”nearest” goal.

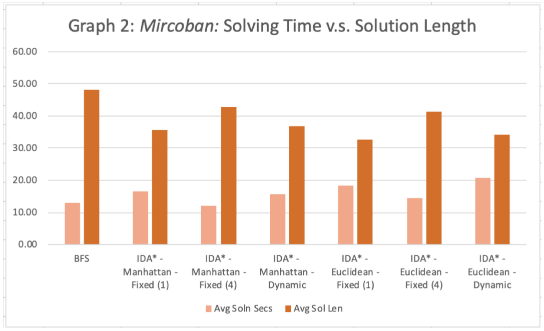
Impact As we expected, the heuristic function using Euclidean distance always yields an equal or less heuristic value than using Manhattan distance. The less the heuristic value undershoots the actual search cost, the better it guides the search algorithm. Consequently, as displayed in the results in Table 1 & 2 in the Appendix, ME-IDA* using Manhattan distance can always solve more Sokoban puzzles within two minutes. Also, for the puzzles that are solved by ME-IDA*-Euclidean, ME-IDA*-Manhattan can solve them faster with fewer generated and explored nodes.

4 Results

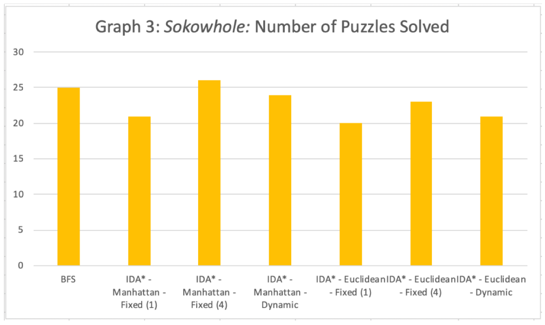
The results of our experiments are summarized below in Tables 1 and 2, and visualized in the following graphs.



In general, our experimental results aligned with our initial expectations. Our informed search (ME-IDA*) routinely generated fewer nodes than our uninformed search (BFS) but ended up exploring more, since, though it would only explore promising paths as dictated by our heuristic functions, it would have to iteratively explore those paths whereas the uninformed search would visit every node but explore comparatively fewer nodes. Additionally, as expected, our informed search generally found move-optimal solutions as compared to our uninformed search, which almost exclusively found non-optimal solutions.

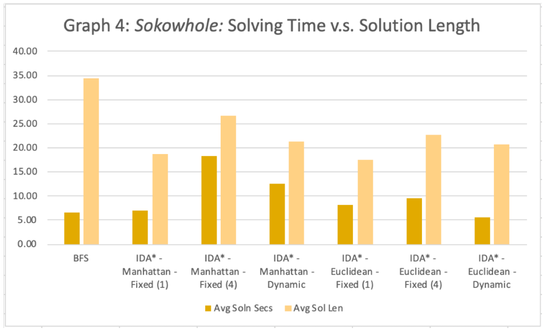


Still, there were a few results that were unexpected as well. Given the nature of ME-IDA* and BFS, we did not expect BFS to uniformly run faster than our ME-IDA*. Additionally, we did not expect BFS to have a better solved percentage than our ME-IDA*. These discrepancies are likely two-fold in nature. For one, the repeated evaluation of our heuristic functions could slow down ME-IDA*; then, the overall lack of complexity of the puzzle could also skew the results in favor of BFS. We chose puzzle sets that would generally be solvable for any non-expert Sokoban player, which likely enables the viability of the brute force



BFS. With more complex puzzles, our informed search will likely outstrip the uninformed brute force search in terms of performance. As was the case in our experiment, the experiment timeout could cause a serach to exit before it found a solution, so an unsolved puzzle does not necessarily reflect that the algorithm ”thinks” that no solution exists for that puzzle; rather it means that the algorithm could not find a solution in the given amount of time.

Finally, there was one result that proved quite problematic for the experiment—namely the runtime performance of MCTS. We knew that the MCTS was not well-suited for the complexity of Sokoban, but we still expected it to solve a puzzle in a reasonable amount of time. However, with the randomized playouts, our implementation of MCTS could not find solutions in an amount of time that made it feasible to run our experiments, which prompted our inclusion of BFS as a means to exemplify an uninformed search.



5 Conclusions

In this paper, we presented the results of modelling Sokoban as search problem in order to test various graph search algorithms, namely BFS, IDA*, and MCTS. Generally speaking, we uncovered a trade-off between runtime and solution optimality. The uniformed BFS outperformed the others in terms of runtime. However, IDA* consistently found the more optimal solutions. Within IDA*, we observed the same trade-off as we adjusted the bound-increment hyperparameter. However, the runtime of the basic MCTS was not experimentally viable in application to the Sokoban domain without being heavily optimized.

Sokoban as a stand-alone problem is worthy of further research. From the scope of our experiments alone, we found that there are numerous areas that could bear further exploration. Most obviously, we would perform those optimizations mentioned above to make MCTS more viable for experimentation. Additionally, we could optimize the performance of all of our search algorithms by enabling them to exploit more domain-specific knowledge than just the basic deadlock pre-processing—making deadlock detection more robust and encoding more goal-related and tunnel-related macros would likely be effective ways to drastically reduce the potential search space. Also, we could experiment with some slightly different heursitics, namely minmatching (see (Pereira, Ritt, and Buriol 2013)). Finally, we would look into exploring some directions novel directions as well, particularly ones that involve incorporating machine learning techniques with reinforcement learning algorithms, as even the best known Sokoban solvers today (that incorporate many of the optimizations discussed above) cannot solve every Sokoban puzzle in most testing sets.

6 Contributions

For the coding portion of the project, M.D. and Y.J. coded the Sokoban board implementation and worked on ME-IDA* together. Throughout the coding portion, M.D. focused more on implementing search algorithms, namely BFS, IDA*, and MCTS, while Y.J. dedicated more effort to the domain specific optimizations, deadlock detection and heuristics implementation. For the presentation and paper, M.D. led the effort on writing, while Y.J. spent more time on designing experiments and visualizing the results.

7 Acknowledgements

We would like to thank Dr. Raghu Ramanujan for his guidance on this project.

References

Chaslot, G.; Bakkes, S.; Szita, I.; and Spronck, P. 2008. Monte-carlo tree search: A new framework for game ai. In *AIIDE*.

Culberson, J. 1997. Sokoban is pspace-complete.

Geffner, P. H. H., and Haslum, P. 2000. Admissible heuristics for optimal planning. In *Proceedings of the 5th Internat. Conf. of AI Planning Systems (AIPS 2000)*, 140–149.

Haywood, L. 2011. Sokevo: Puzzle creator program. `https://leehaywood.org/games/sokoban/`. Retrieved on Apr. 16, 2019.

Junghanns, A., and Schaeffer, J. 1997. Sokoban: A challenging single-agent search problem. In *In IJCAI Workshop on Using Games as an Experimental Testbed for AI Reasearch*, 27–36. Universiteit.

Pereira, A. G.; Ritt, M. R. P.; and Buriol, L. S. 2013. Finding optimal solutions to sokoban using instance dependent pattern databases. In *Sixth Annual Symposium on Combinatorial Search*.

Reinefeld, A., and Marsland, T. A. 1994. Enhanced iterative-deepening search. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 16(7):701–710.

Skinner, D. W. 2010. Sokoban: Puzzles by david w. skinner. `http://www.abelmartin.com/rj/sokobanJS/Skinner/David\%20W.\%20Skinner\%20-\%20Sokoban.htm`. Retrieved on Apr. 16, 2019.

Thinking Rabbit, I. 1981. Sokoban. [NEC PC-8801].

Appendix

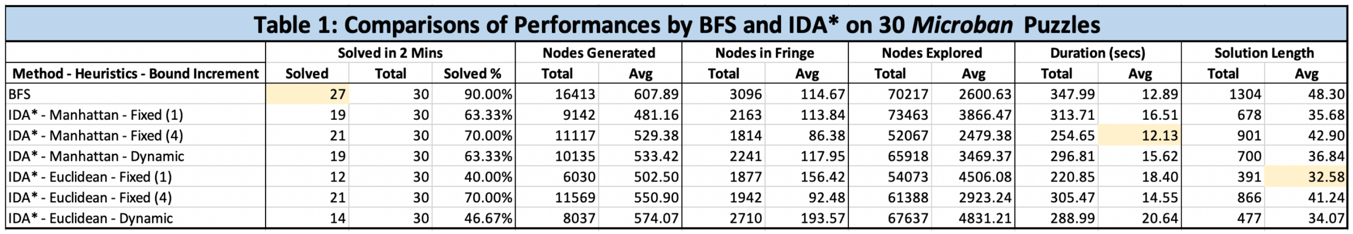


Table 1: Comparisons of Performances by BFS and IDA* on 30 *Microban* Puzzles

	Solved in 2 Mins			Nodes Generated		Nodes in Fringe		Nodes Explored		Duration (secs)		Solution Length	
Method - Heuristics - Bound Increment	**Solved**	**Total**	**Solved %**	**Total**	**Avg**	**Total**	**Avg**	**Total**	**Avg**	**Total**	**Avg**	**Total**	**Avg**
BFS	27	30	90.00%	16413	607.89	3096	114.67	70217	2600.63	347.99	12.89	1304	48.30
IDA* - Manhattan - Fixed (1)	19	30	63.33%	9142	481.16	2163	113.84	73463	3866.47	313.71	16.51	678	35.68
IDA* - Manhattan - Fixed (4)	21	30	70.00%	11117	529.38	1814	86.38	52067	2479.38	254.65	12.13	901	42.90
IDA* - Manhattan - Dynamic	19	30	63.33%	10135	533.42	2241	117.95	65918	3469.37	296.81	15.62	700	36.84
IDA* - Euclidean - Fixed (1)	12	30	40.00%	6030	502.50	1877	156.42	54073	4506.08	220.85	18.40	391	32.58
IDA* - Euclidean - Fixed (4)	21	30	70.00%	11569	550.90	1942	92.48	61388	2923.24	305.47	14.55	866	41.24
IDA* - Euclidean - Dynamic	14	30	46.67%	8037	574.07	2710	193.57	67637	4831.21	288.99	20.64	477	34.07

Figure 6: Comparison of Search Algorithms on *Microban*- the highlighted cell indicating the best performance in that category

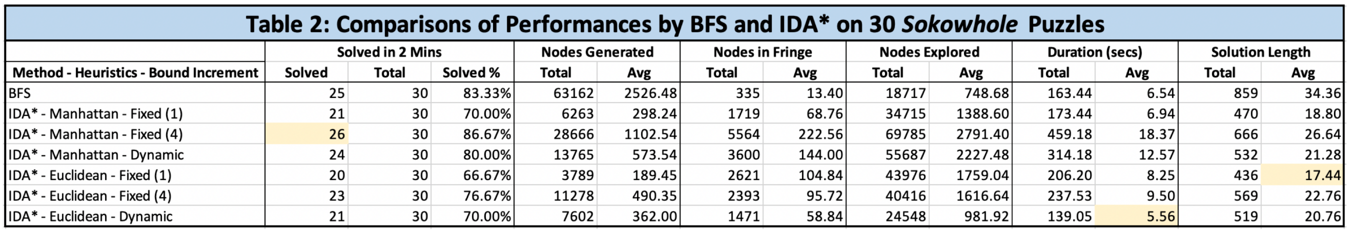


Table 2: Comparisons of Performances by BFS and IDA* on 30 *Sokowhole* Puzzles

	Solved in 2 Mins			Nodes Generated		Nodes in Fringe		Nodes Explored		Duration (secs)		Solution Length	
Method - Heuristics - Bound Increment	**Solved**	**Total**	**Solved %**	**Total**	**Avg**	**Total**	**Avg**	**Total**	**Avg**	**Total**	**Avg**	**Total**	**Avg**
BFS	25	30	83.33%	63162	2526.48	335	13.40	18717	748.68	163.44	6.54	859	34.36
IDA* - Manhattan - Fixed (1)	21	30	70.00%	6263	298.24	1719	68.76	34715	1388.60	173.44	6.94	470	18.80
IDA* - Manhattan - Fixed (4)	26	30	86.67%	28666	1102.54	5564	222.56	69785	2791.40	459.18	18.37	666	26.64
IDA* - Manhattan - Dynamic	24	30	80.00%	13765	573.54	3600	144.00	55687	2227.48	314.18	12.57	532	21.28
IDA* - Euclidean - Fixed (1)	20	30	66.67%	3789	189.45	2621	104.84	43976	1759.04	206.20	8.25	436	17.44
IDA* - Euclidean - Fixed (4)	23	30	76.67%	11278	490.35	2393	95.72	40416	1616.64	237.53	9.50	569	22.76
IDA* - Euclidean - Dynamic	21	30	70.00%	7602	362.00	1471	58.84	24548	981.92	139.05	5.56	519	20.76

Figure 7: Comparison of Search Algorithms on *Sokwhole* - the highlighted cell indicating the best performance in that category