An Experimental Study of Search Algorithms in Sokoban

Saurab Mishra
I2 Maths
Indian Institute of Science Education and Research Thiruvananthapuram
India
saurab23@iisertvm.ac.in

1

Abstract—**Sokoban is a challenging PSPACE-complete puzzle in which a player must push boxes to designated goal locations on a grid while carefully avoiding deadlocks and irreversible moves. Even small instances can generate a rapidly expanding state space, making the problem computationally demanding and well suited as a benchmark for evaluating search algorithms. The interaction between spatial constraints and long-term planning requirements makes Sokoban particularly useful for studying the strengths and limitations of different search strategies.**

In this work, we model Sokoban as a formal state-space search problem by clearly defining its states, actions, transition dynamics, goal condition, and path cost. We implement five classical search algorithms: Breadth-First Search (BFS), Depth-First Search (DFS), Uniform Cost Search (UCS), Greedy Best-First Search, and A* Search. Their performance is evaluated on a benchmark level using solution path length, number of nodes expanded, and wall-clock runtime as comparison metrics. All experiments are conducted within the same controlled computational environment to ensure a fair and consistent evaluation.

The empirical results reflect theoretical expectations. Uninformed approaches such as BFS and UCS guarantee optimal solutions but incur substantial computational overhead. DFS explores quickly but does not ensure optimality. Greedy Best-First Search reduces exploration through heuristic guidance, though without guarantees. A* Search, using an admissible Manhattan-distance heuristic, achieves optimal solutions while significantly reducing search effort, demonstrating a practical balance between efficiency and solution quality.

Index Terms—**Search algorithms, Sokoban, Artificial Intelligence, styling, insert**

I. Introduction

Sokoban is a well-known planning problem widely used as a benchmark in artificial intelligence research. The problem has been proven to be PSPACE-complete [2], implying that the number of reachable configurations grows exponentially with the number of boxes. As a result, brute-force exploration quickly becomes computationally infeasible for non-trivial instances. The presence of irreversible moves, constrained movement, and potential deadlock states further increases its complexity and makes it a suitable testbed for evaluating search strategies.

In this study, we formally model Sokoban as a graph-search problem by defining its state representation, action space, transition model, goal condition, and path cost function. We implement five classical search algorithms, including both uninformed and heuristic-based approaches, and analyze their behavior on a benchmark level. To ensure a fair comparison, all experiments are conducted under the same computational conditions within a controlled environment. Performance metrics such as solution path length, number of nodes expanded, and wall-clock runtime are recorded systematically. The goal is to empirically compare these algorithms and assess how their observed performance aligns with their theoretical properties, particularly optimality and completeness.

[1]Resources and code implementations can be found here: https://github.com/SaurabMishra12/Search-Algorithms-in-Sokoban

II. Methodology

Problem Formulation. We model Sokoban as a deterministic state-space search problem. A state is represented as a tuple (p, B), where p denotes the current position of the player on the grid and B represents the set of box positions. The environment consists of fixed walls and designated goal locations G. The action space is defined by the four cardinal movements $\{\mathrm{UP}, \mathrm{DOWN}, \mathrm{LEFT}, \mathrm{RIGHT}\}$, each incurring a uniform cost of 1.

The transition function enforces the rules of Sokoban: the player may move freely into empty cells, cannot pass through walls, and may push a box only if the adjacent cell in the direction of motion is free. Moves that result in wall collisions or invalid box pushes are discarded. The goal test is satisfied when all box positions coincide exactly with the predefined goal positions, that is, $B = G$.

Heuristic Design. For the informed search strategies, we employ the Sum of Minimum Manhattan Distances heuristic defined as

$$h(s) = \sum_{b \in B} \min_{g \in G} d_{\text{Manhattan}}(b, g).$$

This heuristic estimates the remaining cost by summing, for each box, the Manhattan distance to its closest goal. It is admissible because each box must travel at least its Manhattan distance to reach a goal location, assuming no obstacles. While it does not account for box interactions or deadlocks, it provides a computationally efficient lower bound that guides the search effectively in practice. [4]

Algorithms Implemented. We implement five classical search algorithms within a unified framework to ensure consistency in evaluation:

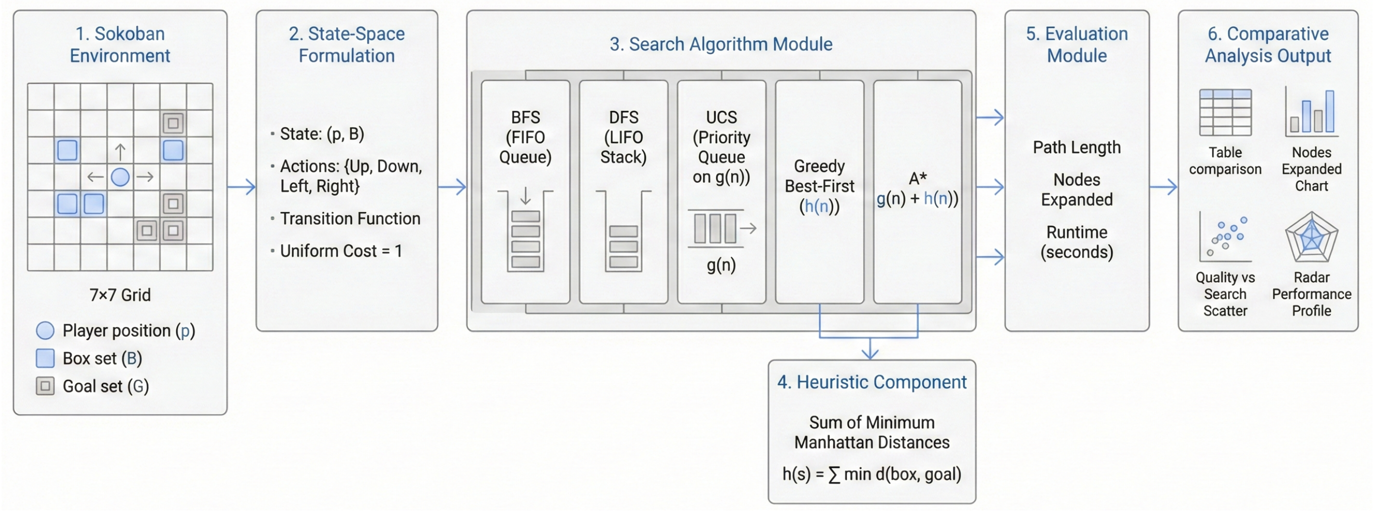

Fig. 1: System architecture of the Sokoban search algorithm evaluation pipeline.

- **Breadth-First Search (BFS)**: Uses a FIFO queue and guarantees the shortest solution path under uniform step costs.
- **Depth-First Search (DFS)**: Uses a LIFO stack with a configurable depth limit to prevent unbounded exploration.
- **Uniform Cost Search (UCS)**: Expands nodes in order of cumulative path cost $g(n)$ using a priority queue.
- **Greedy Best-First Search**: Selects nodes based solely on the heuristic value $h(n)$.
- **A* Search**: Expands nodes according to $f(n) = g(n) + h(n)$, combining path cost and heuristic guidance. [3]

Benchmark and Evaluation Setup. All algorithms are evaluated on the same 7×7 Sokoban level containing two boxes and two goal locations. To ensure a fair comparison, experiments are conducted under identical computational conditions within a controlled environment. Performance metrics recorded include solution path length, number of nodes expanded, and wall-clock runtime. This controlled setup allows us to directly compare empirical behavior with theoretical expectations regarding optimality, completeness, and computational efficiency.

III. Results and Analysis

This section presents a comparative evaluation of the five implemented search algorithms on the Sokoban benchmark level. The algorithms are analyzed along three primary dimensions: solution quality (path length), search effort (number of nodes expanded), and computational cost (wall-clock runtime). Since Sokoban is modeled with uniform step costs, the optimal solution for the tested configuration consists of 15 moves.

A. Overall Performance Comparison

Table I summarizes the experimental outcomes. The results clearly reflect the theoretical properties of the algorithms.

BFS, UCS, and A* successfully find the optimal solution of length 15, consistent with their theoretical guarantees under uniform costs. DFS, while relatively fast, produces a significantly longer path (199 moves), highlighting its tendency to explore deeply without regard to solution quality. Greedy Best-First Search also finds a 15-step solution in this instance; however, this outcome is not guaranteed in general since it does not consider accumulated path cost.

TABLE I: Experimental Results of Classical Search Algorithms on Sokoban

Algorithm	Path Length	Nodes Expanded	Time (s)
UCS	15	5,223	0.0208
DFS	199	4,927	0.0126
BFS	15	4,309	0.0947
A*	15	2,465	0.0140
Greedy BFS	15	36	0.0002

*DFS completeness depends on the imposed depth limit (set to 200 in this experiment).

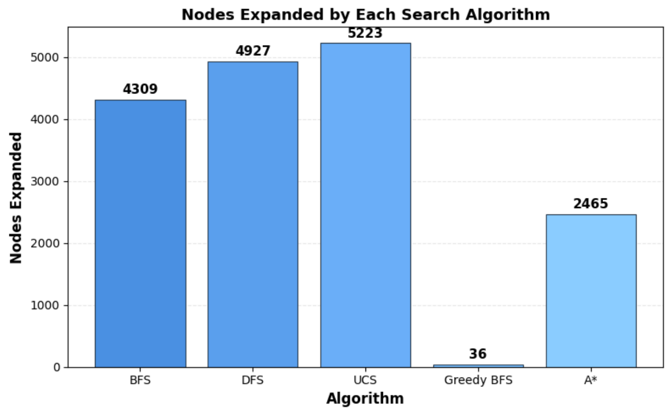

Fig. 2: Comparison of nodes expanded across algorithms.

B. Search Efficiency and Heuristic Influence

A key indicator of search efficiency is the number of nodes expanded before reaching the goal. Figure 2 compares node expansions across all algorithms.

A clear distinction emerges between uninformed and heuristic-guided approaches. BFS and UCS expand over

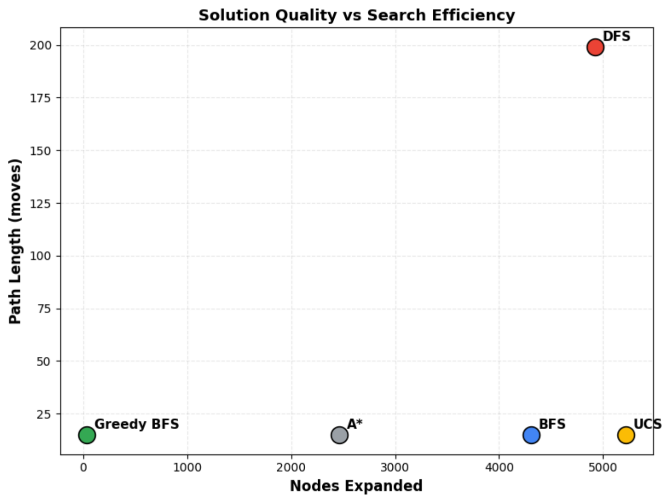

Fig. 3: Trade-off between solution quality (path length) and search effort (nodes expanded).

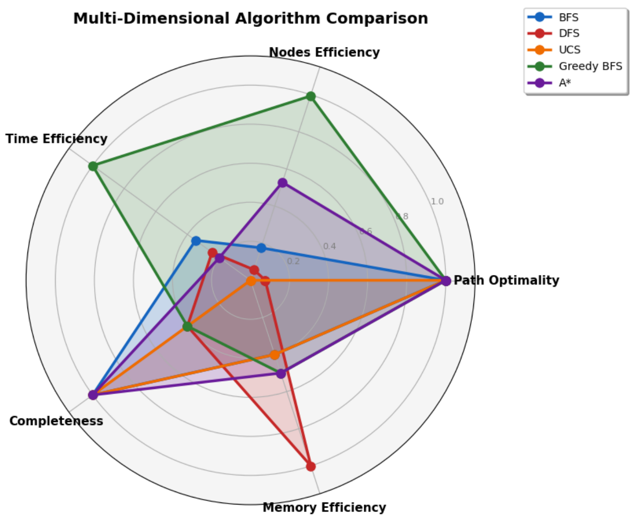

Fig. 4: Multi-dimensional performance comparison of search algorithms.

4,000 nodes, reflecting their exhaustive exploration of the search space. In contrast, A* expands roughly half as many nodes, demonstrating the practical value of incorporating heuristic guidance. Greedy Best-First Search expands only 36 nodes, indicating extremely aggressive pruning; however, this efficiency comes at the cost of reliability in more complex scenarios.

C. Solution Quality versus Search Effort

To better visualize the trade-off between solution quality and computational effort, Figure 3 presents a scatter plot relating path length to nodes expanded.

DFS clearly illustrates the downside of uninformed depth exploration, producing a substantially longer solution despite exploring a comparable number of states to BFS. A* occupies a balanced region in the plot: it preserves optimality while reducing search effort relative to BFS and UCS. This balance highlights the advantage of combining path cost and heuristic estimation.

D. Multi-Dimensional Performance Profile

For a more comprehensive view, Figure 4 presents a radar chart comparing algorithms across five dimensions: path optimality, node efficiency, time efficiency, completeness, and memory efficiency.

The radar visualization shows that DFS is relatively memory-efficient due to its stack-based implementation but performs poorly in terms of solution quality. BFS and UCS guarantee correctness but incur high memory and expansion costs. Greedy Best-First Search excels in speed and low node expansion but lacks theoretical guarantees. A* demonstrates the most balanced performance profile, achieving optimal solutions with substantially reduced search effort.

Overall, the empirical findings align with theoretical expectations: heuristic-informed search, particularly A*, offers the most effective trade-off between optimality and computational efficiency for this Sokoban benchmark.

IV. Discussion

The experimental findings are consistent with established results in the search literature [1]. In particular, A* Search demonstrates the most favorable balance between optimality and computational efficiency. By combining cumulative path cost with heuristic guidance, A* avoids the exhaustive exploration characteristic of uninformed methods while still guaranteeing optimal solutions under an admissible heuristic. The Manhattan-distance heuristic used in this study provides a simple and computationally efficient lower bound on the remaining cost. Although effective, it does not account for box interactions, narrow corridors, or deadlock configurations that are common in Sokoban. Performance could potentially be improved by incorporating domain-specific enhancements such as deadlock detection mechanisms or solving the box-to-goal assignment optimally using techniques like the [8]. These refinements would likely produce a tighter cost estimate and further reduce unnecessary exploration.

Overall, the results reinforce a well-known principle in search theory: incorporating informed guidance significantly improves practical performance without sacrificing correctness, provided that the heuristic satisfies admissibility and consistency conditions.

V. Limitations

This study is limited by the use of a single benchmark level of modest size. While sufficient to illustrate algorithmic behavior, larger and more complex Sokoban instances would provide a more rigorous evaluation. In particular, instances with more boxes would increase branching complexity and better expose scalability differences among algorithms.

Additionally, memory consumption was not measured explicitly. Algorithms such as BFS and A* are known to require substantial memory due to frontier storage, whereas DFS operates with linear memory relative to search depth. Evaluating memory usage on larger instances would provide a more comprehensive comparison.

Future work could extend this analysis to multiple benchmark levels, incorporate advanced heuristics, and perform deeper scalability studies to better understand algorithmic behavior under increased problem complexity.

VI. Conclusion

In this work, we implemented and systematically compared five classical search algorithms on the Sokoban puzzle within a unified experimental framework. By modeling Sokoban as a formal state-space search problem and evaluating each method under identical computational conditions, we were able to examine both their theoretical guarantees and their practical behavior.

The results confirm several well-established insights from search theory. A* Search offers the most effective balance between optimality and efficiency when combined with an admissible heuristic, significantly reducing node expansions while preserving solution quality. BFS and UCS reliably produce optimal solutions under uniform costs, but they incur substantially higher computational overhead due to exhaustive exploration. DFS, although memory-efficient and often faster in runtime, sacrifices both optimality and completeness when depth bounds are imposed. Greedy Best-First Search demonstrates impressive speed and minimal exploration in this instance, yet it lacks guarantees and may fail on more complex configurations.

Overall, the study highlights the practical importance of heuristic guidance in solving combinatorial planning problems such as Sokoban. Future work may investigate more sophisticated heuristics, including pattern databases and deadlock detection techniques, as well as alternative strategies such as Iterative Deepening A* (IDA*) [6]. Expanding the evaluation to larger and more diverse benchmark levels would provide deeper insight into scalability and memory-performance trade-offs.

Acknowledgment

We used generative AI to enhance the content and produce figures; however, all content was verified by us.

References

[1] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 4th ed. Pearson, 2021.

[2] J. C. Culberson, "Sokoban is PSPACE-complete," Technical Report TR-97-02, Department of Computing Science, University of Alberta, 1997.

[3] P. E. Hart, N. J. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," *IEEE Transactions on Systems Science and Cybernetics*, vol. 4, no. 2, pp. 100–107, 1968.

[4] J. Pearl, *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Addison-Wesley, 1984.

[5] A. Junghanns and J. Schaeffer, "Sokoban: Evaluating standard single-agent search techniques in the presence of deadlock," in *Advances in Artificial Intelligence*, Springer, 2001, pp. 1–15.

[6] R. E. Korf, "Depth-first iterative-deepening: An optimal admissible tree search," *Artificial Intelligence*, vol. 27, no. 1, pp. 97–109, 1985.

[7] R. E. Korf, "Finding optimal solutions to Rubik's Cube using pattern databases," in *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, 1997, pp. 700–705.

[8] H. W. Kuhn, "The Hungarian method for the assignment problem," *Naval Research Logistics Quarterly*, vol. 2, no. 1-2, pp. 83–97, 1955.

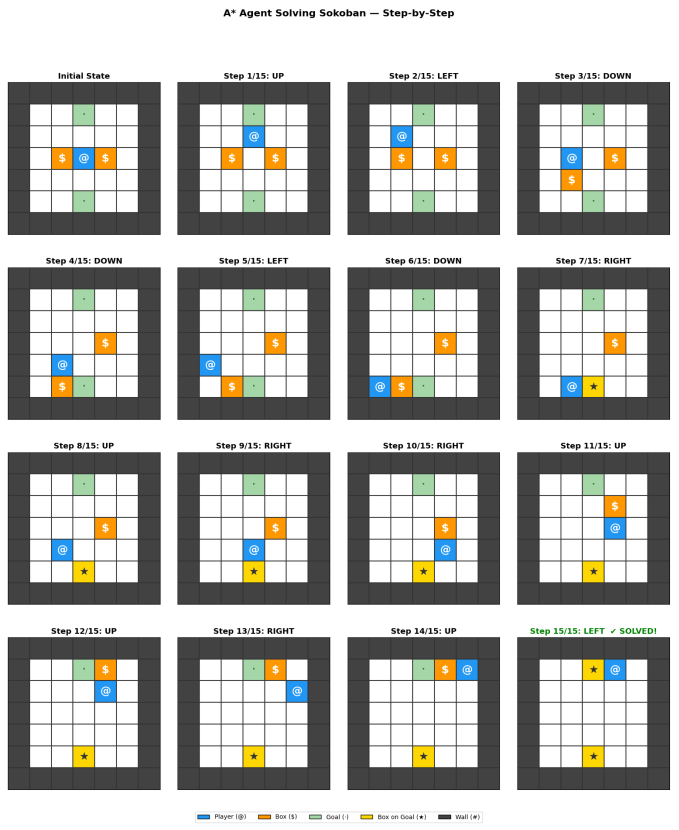

Fig. 5: A* agent execution on the benchmark Sokoban level.

Appendix A
Appendix

A. *Agent Execution (Figure 5)*

Figure 5 illustrates the step-by-step execution of the A* agent solving the benchmark Sokoban level. The figure visualizes the complete solution trajectory consisting of 15 moves, starting from the initial configuration and ending in the solved state where all boxes are placed on their designated goal tiles.

Each subfigure corresponds to a single state transition generated by the A* search algorithm. The player position, box locations, goal tiles, and walls are color-coded for clarity. The sequence highlights how the agent incrementally pushes boxes while avoiding invalid moves and deadlock configurations. The final frame confirms successful completion of the task with both boxes correctly aligned with the goal positions.

This visualization serves two purposes: (1) it provides qualitative insight into how heuristic guidance shapes the search trajectory, and (2) it validates that the returned solution path is executable and consistent with Sokoban rules.

B. *Computation and Implementation Details*

All experiments were conducted using Python in a Kaggle notebook environment. The implementation was written from scratch and includes:

- A formal state representation (p, B) with immutable structures for efficient hashing.
- Explicit transition logic enforcing wall constraints and box-push mechanics.
- A priority-queue-based implementation of UCS, Greedy Best-First Search, and A*.
- Instrumentation to record nodes expanded, execution time, and solution length.

The Manhattan-distance heuristic was used for informed search. No additional pruning techniques such as deadlock detection or pattern databases were applied, ensuring that performance differences reflect only algorithmic behavior rather than domain-specific optimizations.

C. *Experimental Environment*

All comparisons were executed within the same controlled computational environment to ensure fairness. The experiments were run on Kaggle's standard CPU notebook runtime with identical system conditions for each algorithm. No parallelization or hardware acceleration was used. Execution time was measured using Python's built-in timing utilities, and node expansions were counted explicitly within the search loop.

D. *Reproducibility*

To facilitate reproducibility, the complete implementation, experimental setup, and visualization scripts are provided in the accompanying Kaggle notebook. The notebook contains:

- Source code for all five search algorithms.
- Benchmark level definition.
- Automated metric collection routines.
- Plotting scripts used to generate the performance figures.

Because all experiments were performed in a fixed and documented environment, the reported results can be reproduced by executing the notebook without modification. Minor runtime variations may occur due to shared cloud resources, but solution lengths and node expansion counts remain deterministic for the given benchmark.

Overall, Figure 5 complements the quantitative results by providing a transparent and interpretable demonstration of the A* agent's behavior on the Sokoban instance.
[2]

[2]Resources and code implementations can be found here: https://github.com/SaurabMishra12/Search-Algorithms-in-Sokoban