Theory of Computer Games

Homework 1 (SOKOBAN)

Muhammad Inas Farras Tsamarah

611221308

1. Description of the Picked Puzzle

For this project, I selected the Sokoban puzzle. Sokoban is a logic-based game where the player, represented as a worker, pushes boxes onto goal positions on a grid-like board. The worker can only push, not pull, and boxes can be moved one at a time. The challenge of Sokoban lies in strategizing to avoid getting the boxes stuck in positions where they cannot be moved to a goal, which introduces significant complexity as levels grow in size and difficulty.

2. Methods Implemented and Used

I started with an existing Sokoban game fetched from GitHub (https://github.com/morenod/sokoban/tree/master) and extended it by adding several new features, including auto-solve functionality using different algorithms. The original implementation used a procedural approach, with all logic in a single script without leveraging Object-Oriented Programming (OOP) principles. I refactored the codebase to follow an OOP structure, which improved code readability and modularity and allowed for easier feature expansion.

Specifically, I implemented three methods to solve the Sokoban puzzle: breadth-first search (BFS), depth-first search (DFS), and A* search. Each of these methods approaches the puzzle in different ways:

- **BFS** explores all nodes level by level. It guarantees to find the shortest solution but may require large amounts of memory since all the possible board states must be stored.
- **DFS** explores each path as far as possible before backtracking. It has a lower memory requirement than BFS but does not guarantee the shortest path and may get stuck exploring deep but irrelevant paths.
- *A Search** uses a heuristic to estimate the cost to reach the solution, effectively balancing exploration depth and path optimization. In Sokoban, the heuristic is based on the minimum distance of boxes from goal positions, accounting for obstacles.

3. Improvements Made to the Original Code

The original Sokoban implementation was not structured in an Object-Oriented manner, and it was written as a single monolithic script. Here are the key improvements I made:

- **Refactoring to OOP**: I refactored the code into classes, such as Game, LevelManager, Solver, and Button, to better organize the different responsibilities. This made the codebase more modular and maintainable.
- **Auto-Solve Feature**: I added an auto-solve feature that allows the user to choose from BFS, DFS, or A* Search to solve the puzzle automatically. This was achieved by implementing separate solver classes, making adding or modifying solving algorithms easy.
- **Real-Time Visualization**: Using Pygame, I implemented real-time visualization of the solver's progress. This feature was not present in the original version and provided valuable insight into how each algorithm approached the solution, aiding in debugging and comparison.
- **Expanded User Interface**: I added buttons to the user interface to control the game, such as resetting the level, moving to the next or previous level, and initiating different solving algorithms. This improved the user experience significantly compared to the original command-based movement.
- **Bug Fixes and Stability Improvements**: The original code contained several bugs that affected the game's stability. For example, improper boundary checks sometimes cause crashes when accessing elements out of the matrix bounds. Additionally, the handling of worker and box movements had issues that occasionally led to invalid game states, such as misplaced boxes or incorrect worker positions. By refactoring the code and implementing thorough boundary and state checks, I significantly improved the stability and robustness of the game.

4. Comparison of Different Methods or Algorithms

To compare these methods, I focused on the following metrics: solvable problem size, solving time, memory usage, and time complexity.

- **Solvable Problem Size**: BFS could solve smaller puzzles effectively but struggled with larger boards due to the rapid growth of the search space. DFS could handle more giant puzzles to an extent but often needed help finding an optimal solution or becoming stuck in lengthy paths. A* performed best for complex puzzles, given its heuristic guidance, which allowed it to solve impractical puzzles for BFS or DFS efficiently.
- **Solving Time**: BFS had a predictable, albeit sometimes long, solving time due to its exhaustive search nature. DFS was faster in some instances, particularly for more straightforward puzzles, but often took longer on more complex puzzles due to backtracking. A* had the fastest solving time for most puzzles, as its heuristic effectively guided the search towards promising paths.
- **Memory Usage**: BFS requires the most memory because it stores all nodes at each level. DFS, in contrast, had lower memory requirements but required significant recursion depth, which presented a risk of stack overflow in more giant puzzles. A* required moderate memory, balancing storage needs with efficient exploration.
- **Time Complexity**: The time complexity for each algorithm varied significantly.
  - **BFS** has a time complexity of O(b^d), where b is the branching factor and d is the depth of the solution. This makes it impractical for large puzzles as the number of nodes grows exponentially.
  - **DFS** also has a time complexity of O(b^d), but its depth-first nature means it can go deep without finding an optimal solution, which affects its efficiency.
  - **A*** has a time complexity of O(b^d) in the worst case but often performs much better due to the heuristic function, which guides the search towards the goal more efficiently.
- **Space Complexity**: The space complexity also varied.
  - **BFS** requires O(b^d) space to store all nodes at each level.
  - **DFS** has a space complexity of O(d) because it only needs to store a single path from the root to a leaf, along with unexpanded sibling nodes for each node on the path.

- **A*** has a space complexity of O(b^d) in the worst case, similar to BFS, but can often be optimized depending on the heuristic.

5. Possible Extensions of the Work

A possible extension of this work is to implement more advanced heuristics for A*, such as pattern databases, which can pre-compute distances for specific puzzle patterns to improve solving time. Another extension could be to integrate machine learning techniques to predict effective moves based on the current board state, potentially improving the performance of the solver.

Another potential extension could involve adapting the solver to handle 3D Sokoban or incorporating elements of randomness in the puzzle structure to test the robustness of the algorithms.

6. Related Work and Additional Thoughts

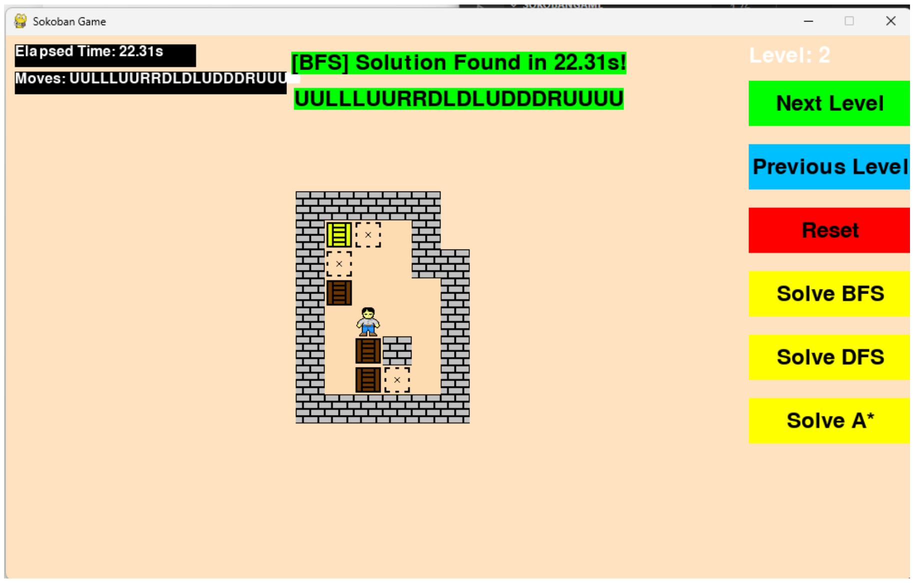
Sokoban is often used as a benchmark for evaluating search algorithms due to its complexity. Related work in this area includes various AI methods, such as Monte Carlo Tree Search (MCTS) and Reinforcement Learning, which have been used to solve similar puzzles with impressive results. Considering these approaches for future extensions could provide additional insights and improve the performance of the solver.

In my implementation, I also focused on the real-time visualization of the solver's progress using Pygame, which allowed for a better understanding of how each algorithm approached the solution. This visualization not only helped in debugging but also highlighted the differences in exploration patterns among the algorithms.

7. Benchmark Results

To evaluate the performance of BFS, DFS, and A*, I conducted benchmark tests using level 2 of the Sokoban game. Below are the results and observations for each algorithm, along with the corresponding screenshots.

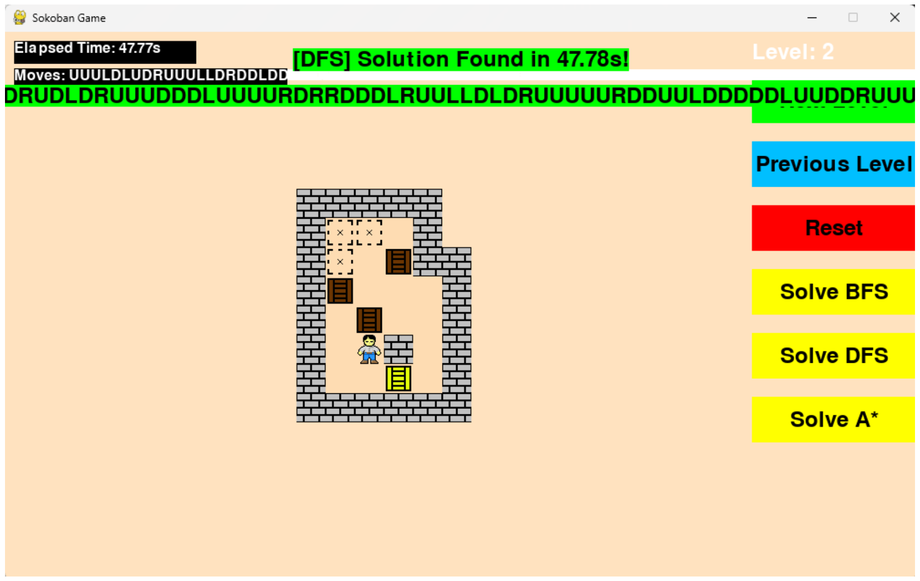
- **BFS Results**



```
Elapsed Time: 22.31s
Moves: UULLLUURRDLDLUDDDRUUU
[BFS] Solution Found in 22.31s!
UULLLUURRDLDLUDDDRUUUU
```

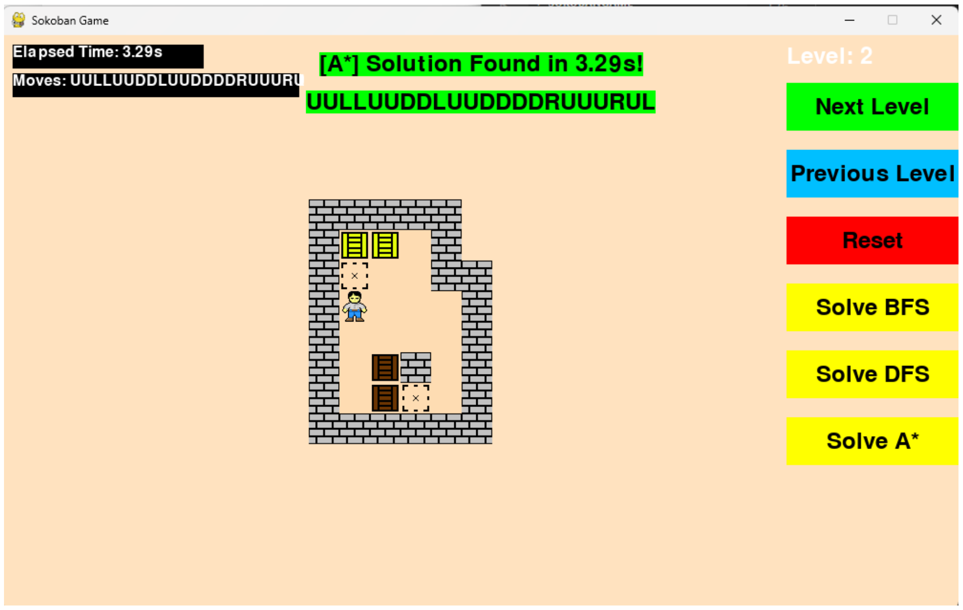
  - **Elapsed Time**: 22.31 seconds
  - BFS took 22.31 seconds to find a solution with the move sequence provided. While it guarantees the shortest path, its exhaustive exploration makes it slower in some cases.

- **DFS Results**



```
Elapsed Time: 47.77s
Moves: UUULDLUDRUUULLDRDDLDD
[DFS] Solution Found in 47.78s!
DRUDLDRUUUDDDLUUUURDRRDDDLRUULLDLDRUUUUURDDUULDDDDDLUUDDRUUU
```

    - **Elapsed Time**: 47.78 seconds
    - DFS took 47.78 seconds with a much longer sequence of moves. DFS tends to explore deeper paths, leading to inefficient exploration and longer solving time.
- ***A** Result**



Elapsed Time: 3.29s
Moves: UULLUUDDLUUDDDDRUUURU

[A*] Solution Found in 3.29s!
UULLUUDDLUUDDDDRUUURUL

    - **Elapsed Time**: 3.29 seconds
    - A* found the solution in just 3.29 seconds, indicating the efficiency of the heuristic used. The move sequence was shorter, showcasing the advantage of guided search.

8. Conclusions from Benchmarks

The benchmark results clearly show the effectiveness of A* over BFS and DFS in terms of solving time and solution efficiency. The heuristic used in A* allowed it to significantly reduce both solving time and the number of moves compared to BFS and DFS. BFS, while guaranteeing the shortest path, was slower due to its exhaustive search nature. DFS, on the other hand, struggled with the search space, leading to suboptimal results.

These benchmarks emphasize the importance of using heuristics to guide search algorithms in solving complex puzzles like Sokoban. Future work can further optimize the heuristic for A* or explore hybrid approaches to achieve even better performance.

References

1. GeeksforGeeks. (n.d.). Breadth-First Search (BFS) and Depth-First Search (DFS) are used to solve puzzles. Retrieved from https://www.geeksforgeeks.org/breadth-first-search-or-bfs-for-a-graph/
2. Inas Farras. (n.d.). Sokoban Game. GitHub repository. Retrieved from https://github.com/inasfarras/SokobanGame/tree/main
3. Moreno, D. (n.d.). Sokoban Game. GitHub repository. Retrieved from https://github.com/morenod/sokoban/tree/master
4. Junghanns, A., & Schaeffer, J. (1999). Sokoban: Enhancing general single-agent search methods using domain knowledge. *Artificial Intelligence*, 129(1-2), 219-251.
5. Culberson, J. C. (1997). Sokoban is PSPACE-complete—proceedings *of the International Conference on Fun with Algorithms*.
6. Hart, P. E., Nilsson, N. J., & Raphael, B. (1968). A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2), 100-107.
7. Russell, S., & Norvig, P. (2010). *Artificial Intelligence: A Modern Approach* (3rd ed.). Prentice Hall.
8. Browne, C., Powley, E., Whitehouse, D., Lucas, S. M., Cowling, P. I., Rohlfshagen, P.,... & Colton, S. (2012). A survey of Monte Carlo Tree Search methods. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(1), 1–43.