Verifying Correctness of the A* Algorithm in C Implementation for Shortest Path Problem

Qingyang Sun, Victoria Duan
GitHub Repository

Introduction

The shortest-path problem is a classic problem in the field of computer science and mathematics, which focuses on finding the shortest and most efficient path between two points in a graph. It's interesting and useful because of its wide range of real-world applications. For instance, navigation systems like Google Maps or Apple Maps use it to guide millions of users to find the most efficient route for them to commute from one place to another; food delivery apps such as DoorDash or Uber Eats rely on it to optimize delivery routes for delivery drivers. Moreover, researchers and scientists who worked on solving these problems have made major breakthroughs in algorithm design and research, pushing computer science and mathematics forward in research.

In essence, the shortest path problem involves traversing a weighted (or uniformly weighted) graph and identifying a path with the minimum distance and cost from the starting point (or node) to the end (or the target node). There are numerous factors to consider when developing the most suitable solution for this problem, including the graph's structure, the nature of the weights (i.e., distance, time, or cost), and the computational constraints (i.e., time/space complexity, hardware limitations, etc).

Algorithms like Dijkstra's[4] and Bellman-Ford's[2][5][8] have been extensively studied and are considered the foundational benchmarks for solving the shortest-path problem. However, due to its high efficiency, the A* algorithm is more applicable in scenarios that require dynamic or resource-constrained path-finding (i.e., usage in robotics, game development, and real-time navigation systems). Unlike Dijkstra's algorithm, which explores all possible paths within a graph, A* leverages heuristic guidance to prioritize paths that are more likely to reach the target efficiently. While this heuristic-driven approach enhances efficiency, it also introduces complexity that can lead to suboptimal or incorrect results, especially in critical systems where reliability and safety are prioritized.

This project aims to verify the correctness of the A* algorithm's implementation in C for solving the shortest-path problem. By utilizing formal verification tools such as C Bounded Model Checker (CBMC)[7] and Seahorn[1], we ensure the algorithm's logical correctness, computational soundness, memory safety, and overall robustness. C was chosen for this implementation due to its precise control over system-level resources and strong compatibility with advanced verification tools, making it well-suited for performance-critical and reliability-focused applications.

Method

Verification Details

We used the C Bounded Model Checker (CBMC) in version 5.10 on the Windows Subsystem for Linux (WSL) to verify the correctness and robustness of its implementation in C. Below are specifications on how we verified our implementation of A* in C is correct in general (i.e., software-wise) and logic-wise.

Verification of the correctness of general properties

We applied the following CBMC commands with assertion specified to check the general properties of the entire program **astar.c** and the core function implementing the A* search algorithm **aStar()**:

```
cbmc filename.c   --bounds-check --pointer-check --memory-leak-check
--div-by-zero-check --signed-overflow-check --unsigned-overflow-check
--pointer-overflow-check --conversion-check --undefined-shift-check
--float-overflow-check --nan-check
```

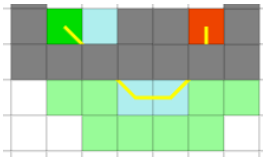
```
cbmc filename.c --function aStar    --bounds-check --pointer-check
--memory-leak-check     --div-by-zero-check --signed-overflow-check
--unsigned-overflow-check     --pointer-overflow-check --conversion-check
--undefined-shift-check     --float-overflow-check --nan-check
```

More specifically, the assertions above check the following properties:

- **Buffer overflows:** Check whether the upper and lower bounds are violated for each array access.
- **Pointer safety:** Ensure no NULL or invalid pointer dereferences.
- **Memory leaks:** Check whether the program constructs dynamically allocated data structures that are subsequently inaccessible.
- **Division by zero:** Check whether the program has a division by zero.
- **Not-a-Number:** Check whether floating-point computation may result in NaNs.
- **Arithmetic overflow:** Check whether a numerical overflow occurs during an arithmetic operation.
- **Undefined shifts:** Check for shifts with excessive distance.

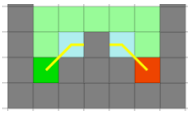
Verification of the correctness of returning a valid shortest path

Our definition of a valid path is a path that 1) is in the range of the map (i.e., within the graph), 2) is a walkable path with no obstacle, and 3) is a continuous path that is reachable from start to end. Therefore, we defined the following properties for us to verify if the algorithm returns a valid path:



- **Within Bounds:** Path stays within grid boundaries.

- **Walkable:** Path avoids obstacles.
- **Step Rule (+1 step):** Path moves one step at a time (left, right, up, down).
- **Continuity:** Path forms a connected sequence of steps

 or

To verify those properties, the following assertion was added to the code:

```
C/C++
__CPROVER_assert(//(row, col) used in the algorithm falls within
the valid grid boundaries);
__CPROVER_assert(//walkable path, not a obstacle);
__CPROVER_assert(//next_step is valid following rule);
__CPROVER_assert(//path is continuous);
```

We first run the A* algorithm with a selected starting node and target node and compute the shortest path (say pathA). Next, using the Breadth-First Search (BFS) algorithm that we wrote, which is a more straightforward path-finding algorithm with the same starting node and target node, compute the shortest path using BFS (say pathB). Finally, we compare the two paths (i.e., pathA and pathB), and check if they have the same length by using the following assertion:

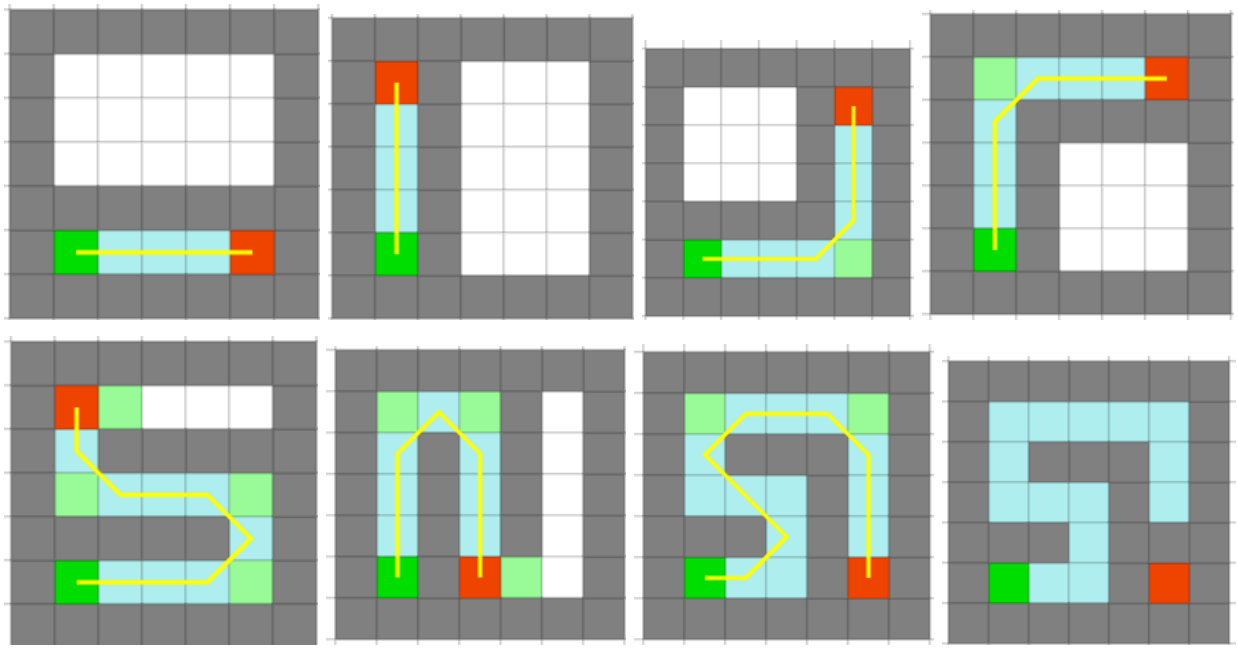
```
C/C++

assert(//aStarLength == bfsLength);
```

This confirms that the returned path is valid and the shortest among all possible paths.

Scalability Analysis

After verifying that the algorithm works logically and syntactically, we need to verify its scalability (i.e., how large of a matrix can we verify our algorithm on).

We chose a 5x5 matrix (with a total of 2^{25} possible grid combinations) with a starting point of (0, 0) and an endpoint of any point within the matrix. The 5 by 5 grid is sufficient enough to represent a wide variety of paths and scenes, including but not limited to straight paths, turns (i.e., left, right, U-turn), obstacles blocking the path to go around it, snake-like paths, and dead ends. This diversity makes the 5x5 grid an ideal test case for evaluating the algorithm's ability to handle different pathfinding challenges.



The total test cases from

0	0	0	0	0
0	0	0	0	0
0	0	0	0	0
0	0	0	0	0
0	0	0	0	0

to

1	1	1	1	1
1	1	1	1	1
1	1	1	1	1
1	1	1	1	1
1	1	1	1	1

are $25 * 2^{25} = 838,860,800$, since there are 2^{25} possible grid combinations where each cell could be walkable or blocked and 25 choices of endpoints. By iterating through all $838,860,800$ cases with assertions, checking the correctness of general properties, and returning a valid shortest path, we can conclude that the A* algorithm correctly computes the shortest path in all cases for a 5x5 grid with the start end of (0,0).

Result and Analysis

The following verifications are run on a 5x5 grid, and running all verifications on both the correctness of general properties and that it returns a valid shortest path proves the correctness on the 5x5 grid.

Correctness of general properties

CBMC verified the overall correctness of **astar.c** and the **aStar()** function:

- No buffer overflows, pointer issues, or arithmetic errors.
- Memory safety is confirmed.

```
[aStar.overflow.11] arithmetic overflow on signed + in rear + 1: SUCCESS
[aStar.overflow.12] arithmetic overflow on signed + in i + 1: SUCCESS
[__CPROVER__start.memory-leak.1] dynamically allocated memory never freed in __CPROVER_memory_leak =
NULL: SUCCESS

** 0 of 75 failed (1 iteration)
VERIFICATION SUCCESSFUL
```

```
[aStar.overflow.9] arithmetic overflow on signed + in rear + 1: SUCCESS
[aStar.overflow.10] arithmetic overflow on signed + in i + 1: SUCCESS
[__CPROVER__start.memory-leak.1] dynamically allocated memory never freed in __CPROVER_memory_leak ==
NULL: SUCCESS

** 0 of 110 failed (1 iteration)
VERIFICATION SUCCESSFUL
```

Correctness of returning a valid shortest path

The algorithm consistently returned valid paths adhering to all defined properties:

- Within bounds.
- Avoiding obstacles.
- Step-by-step movement.
- Continuous connectivity.

```
[aStar.overflow.12] arithmetic overflow on signed + in rear + 1: SUCCESS
[aStar.overflow.13] arithmetic overflow on signed + in i + 1: SUCCESS
[aStar.array_bounds.38] array 'cost' lower bound in cost[(signed long int)goalRow]: SUCCESS
[aStar.array_bounds.39] array 'cost' upper bound in cost[(signed long int)goalRow]: SUCCESS
[aStar.array_bounds.40] array 'cost'[] lower bound in cost[(signed long int)goalRow][(signed long int)
goalCol]: SUCCESS
[aStar.array_bounds.41] array 'cost'[] upper bound in cost[(signed long int)goalRow][(signed long int)
goalCol]: SUCCESS
[aStar.assertion.9] If no path found, cost at goal remains infinite: SUCCESS
[__CPROVER__start.memory-leak.1] dynamically allocated memory never freed in __CPROVER_memory_leak ==
NULL: SUCCESS

** 0 of 158 failed (1 iteration)
VERIFICATION SUCCESSFUL
```

Additionally, the path lengths computed by A* matched the BFS results for all tested cases, confirming correctness.

```
Tested 833000000 matrices so far...
Tested 834000000 matrices so far...
Tested 835000000 matrices so far...
Tested 836000000 matrices so far...
Tested 837000000 matrices so far...
Tested 838000000 matrices so far...
Total Matrices Tested without any violation: 838860800
```

Discussion

We also experimented with Frama-C but encountered challenges due to incompatible software and hardware versions, as well as outdated technology files from the software.
Given the current CPU capabilities of our setup, a full verification for a 5x5 matrix (iterating through all possible combinations with any starting and ending point) takes approximately 43 minutes. Based on this, we estimate that verifying all cases for a 6x6 matrix with a fixed starting point at (0,0) and any ending point would take about 85 hours. Extending this to all possible starting and ending points on a 6x6 matrix would require approximately 3,037.6 hours (126 days)

due to the increased complexity (2^36 * 36 * 36 cases). Because of this, we decided not to use Frama-C for our formal verification process, as the computational demands and compatibility issues outweighed its potential benefits.

Conclusion

The A* algorithm implementation in C was successfully verified using CBMC, which verifies the correctness of general properties of the algorithm, which ensured the code works syntactically, and the correctness of finding a valid shortest path, which ensured the code works logically. We also verified that the algorithm can handle diverse scenarios (e.g., obstacles and complex paths). Lastly, we verified that the algorithm works for a 5x5 grid; hence, it could potentially be scaled to larger grids given sufficient computational resources.

Future Work

While this project successfully verified the correctness of the A* algorithm implementation using formal verification tools, there are still many possible directions for improvements or solution exploration for pathfinding problems.

We could start by experimenting with different heuristic functions. Our current implementation uses Manhattan distance as the heuristic function, but we could examine the algorithm using other heuristic functions like Euclidean distance or domain-specific heuristics and evaluate their impact on the algorithm's correctness and performance. This could reveal whether different heuristics improve efficiency or accuracy under varying conditions and which heuristics work best.

Moreover, we could use the Java Bounded Model Checker (JBMC), developed by Daniel Kroening, the same creator of CBMC, to verify the A* algorithm in Java implementation. This could allow us to do a comparative analysis of implementation-specific behaviors and potential discrepancies and help us gain a deeper understanding of the algorithm across different programming languages.

Furthermore, we could also further test the algorithm's ability to handle dynamic environments, such as real-time updates to obstacles or edge weights. Dynamic adaptation is especially important for applications like robotics and autonomous navigation, where environments are constantly changing. Incremental search techniques or hybrid approaches could be explored to adapt the A* algorithm efficiently in such scenarios.

References

[1] A.Gurfinkel, T.Kahsai, A. Komuravelli, J.A.Navas. The SeaHorn Verification Framework. At CAV 2015. LNCS 9206, pp. 343-361. 2015 https://seahorn.github.io/papers/cav15.pdf

[2] Bellman, Richard (1958). "On a routing problem". Quarterly of Applied Mathematics. 16: 87–90. doi:10.1090/qam/102435. MR 0102435

[3] C. Urban, A. Gurfinkel, and T. Kahsai, “Synthesizing Ranking Functions from Bits and Pieces.” Accessed: Dec. 07, 2024. [Online]. Available: https://seahorn.github.io/papers/termination_tacas16.pdf

[4] Dijkstra, E. W. (1959). A note on two problems in connexion with graphs. Numerische Mathematik, 1(1), 269–271. https://www.cs.yale.edu/homes/lans/readings/routing/dijkstra-routing-1959.pdf

[5] Ford, Lester R. Jr. (August 14, 1956). Network Flow Theory. Paper P-923. Santa Monica, California: RAND Corporation

[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): 100–7. doi:10.1109/TSSC.1968.300136.

[7] Kroening, D., Schrammel, P., & Tautschnig, M. (2023, February 5). CBMC: The C bounded model Checker. arXiv.org. https://arxiv.org/abs/2302.02384

[8] Shimbel, A. (1955). Structure in communication nets. Proceedings of the Symposium on Information Networks. New York, New York: Polytechnic Press of the Polytechnic Institute of Brooklyn. pp. 199–203

[9] T. Kahsai, J. Navas, D. Jovanović, and M. Schäf, “Finding Inconsistencies in Programs with Loops.” Accessed: Dec. 07, 2024. [Online]. Available: https://seahorn.github.io/papers/lpar2015.pdf