CSC 584/484 - Building Game AI
Miles Hollifield
March 23, 2025

CSC 584/484 - Building Game AI: Homework 3 Writeup

Introduction

The purpose of this assignment was to explore pathfinding and path following algorithms essential for game AI. I implemented Dijkstra's Algorithm and A* with various heuristics, and integrated these pathfinding solutions with the steering behaviors developed in Homework 2. The primary goal was to analyze and compare the performance of different pathfinding approaches across multiple graph structures, and to demonstrate smooth path following in an interactive environment.

This project involved creating two different graphs: a smaller, meaningful graph representing the NCSU campus, and a much larger randomly generated graph designed to stress-test the algorithms. By examining the efficiency and behavior of these algorithms in different contexts, I gained insight into their practical applications and limitations for game AI development.

Implementation and Analysis

Part 1: Graph Creation

For the first graph, I created a weighted directed graph representing the North Carolina State University campus. This graph contains 20 vertices representing key campus buildings and landmarks, with 66 edges representing walkways between them. Edge weights were assigned based on arbitrary walking times between locations, creating a semi-realistic representation of campus navigation. **[Screenshot #1]**

This graph was meaningful as it represented a real-world navigation scenario that clearly demonstrates the practical application of pathfinding algorithms. The campus layout, with its clusters of buildings and connecting pathways, provides an interesting topology that includes both direct routes and potential shortcuts that might not be immediately obvious.

For the second graph, I generated a much larger random graph with 20,000 vertices and approximately 100,000 edges (averaging 5 edges per vertex). This graph was generated programmatically using a random number generator to create vertices and connect them with weighted edges. The vertices were positioned in a grid-like structure to enable heuristic calculations based on spatial distance.

This large graph was specifically designed to test the algorithms under significant computational load, helping to highlight the performance differences between Dijkstra's Algorithm and A* with various heuristics when dealing with large-scale pathfinding problems.

CSC 584/484 - Building Game AI
Miles Hollifield
March 23, 2025

Part 2: Dijkstra's Algorithm and A*

I implemented both Dijkstra's Algorithm and A* for finding shortest paths in a graph. Both implementations use a priority queue to efficiently select the next node to explore, with proper tracking of visited nodes and path reconstruction once a goal is reached.

Performance Comparison on Small Graph

When testing on the NCSU campus graph, the differences between Dijkstra's Algorithm and A* became apparent despite the relatively small size:

Algorithm	Avg. Nodes Explored	Avg. Max Fringe Size	Avg. Path Cost	Avg. Execution Time (ms)
Dijkstra	12.8	7.0	6.37	0.076
A* (Euclidean)	4.8	6.0	6.4	0.022
A* (Manhattan)	5.0	6.4	8.0	0.022
A* (Inadmissible)	4.8	5.4	6.4	0.026

From these results, A* with the Euclidean distance heuristic explored approximately 62.5% fewer nodes than Dijkstra's Algorithm while producing optimal paths in 3 out of 5 test cases. This demonstrates the efficiency advantage of A*, even on smaller graphs. Even in cases where optimality wasn't maintained (particularly for the Reynolds Coliseum to Cox Hall path), A* still found reasonable paths while exploring significantly fewer nodes.

Another interesting observation is that the Manhattan distance heuristic performed particularly poorly on the "Reynolds Coliseum to Cox Hall" test, choosing a much longer path (cost 14.5 vs. the optimal 8.5) due to its bias toward grid-like movement in a graph that doesn't strictly follow a grid pattern.

Performance Comparison on Large Graph

The large graph tests revealed more dramatic differences:

Algorithm	Avg. Nodes Explored	Avg. Max Fringe Size	Avg. Path Cost	Avg. Execution Time (ms)
Dijkstra	12,332.2	10,376.8	28.78	87.189
A* (Euclidian)	3,125.2	8,665.2	116.92	43.601
A* (Manhattan)	3,127.2	8,584.2	117.13	44.196

A* (Inadmissible)	3,002.9	8,296.3	128.36	40.153

These results demonstrate the significant performance advantage of A* on larger graphs. A* with the Euclidean heuristic explored only 25.3% of the nodes compared to Dijkstra's Algorithm and ran approximately 50% faster. However, in this large random graph, none of the A* variants produced optimal paths. This is likely due to the specific structure of the randomly generated graph, where the spatial positioning of nodes didn't correlate well with the actual connectivity, making the distance-based heuristics less effective at guiding the search toward optimal paths.

Interestingly, despite the lack of optimality, all three A* variants maintained a consistent exploration advantage, examining only about 25% of the nodes that Dijkstra's Algorithm did. The average path length for A* variants was significantly longer (19.9-23.7 vertices) compared to Dijkstra's Algorithm (8.3 vertices), which suggests that A* found longer routes with more waypoints but still achieved the goal of reaching the destination.

The fill percentage (nodes explored relative to total graph size) decreased as the graph size increased, but the relative efficiency of A* compared to Dijkstra's remained consistent, confirming the theoretical advantages of informed search over uninformed search at scale.

Part 3: Heuristics

For A*, I implemented and tested five different heuristic functions:

1. Euclidean Distance (Admissible & Consistent)
    - Direct straight-line distance between vertices
    - Never overestimates the true cost, ensuring optimal paths
    - Used as the default heuristic in the main application

2. Manhattan Distance (Admissible for Grid-Based Graphs)
    - Sum of horizontal and vertical distances
    - Works well in environments with orthogonal movement
    - Admissible in grid-based environments but can be inadmissible in general graphs

3. Cluster Heuristic (Performance Optimization)
    - Groups vertices into clusters to accelerate large graph navigation
    - Uses pre-computed distances between clusters
    - Falls back to Euclidean distance for vertices in the same cluster
    - Efficient for large graphs with natural clustering

4. Inadmissible Heuristic
    - Deliberately overestimates costs based on distance
    - Uses variable overestimation factors (1.5-2.0×) based on distance

    - Adds random variation for non-deterministic behavior
    - Sacrifices path optimality for potentially faster pathfinding

5. Directional Bias Heuristic (Inadmissible)
    - Systematically applies different weights to horizontal vs. vertical movement
    - Vertical movement is penalized more heavily (2.0× vs 1.2× for horizontal)
    - Creates directionally biased paths that prefer horizontal movement
    - Demonstrates how directional preferences can be encoded in pathfinding

The inadmissible heuristic demonstrated interesting characteristics in testing. When overestimated by a factor of 1.5-2.0, it reduced node exploration by approximately 24.4% compared to Dijkstra's Algorithm in the large graph tests, but at the cost of path optimality. The frequency of overestimation was 100% (by design), but the magnitude varied based on the distance. Paths produced by this heuristic tended to be more direct with fewer intermediate waypoints in the small graph tests, but in the large graph tests, it actually produced paths with more waypoints (averaging 23.7 vertices vs. Dijkstra's 8.3).

The directional bias heuristic created paths that favored horizontal movement, which could be useful in games where certain types of movement are preferred or where the terrain makes vertical movement more difficult. This demonstrates how heuristics can be used not just for performance optimization but also for influencing the characteristics of the generated paths.

In my implementation on the small graph, the Euclidean and Manhattan heuristics produced optimal paths in 3 out of 5 test cases, while the inadmissible heuristic produced optimal paths in only 2 out of 5 cases. Interestingly, in the large graph tests, none of the heuristics consistently produced optimal paths compared to Dijkstra's Algorithm. This suggests that the effectiveness of heuristics is highly dependent on the structure and characteristics of the specific graph.

Part 4: Pathfinding Integration with Path Following

The final part of the assignment integrated pathfinding with the steering behaviors from Homework 2 to create a complete navigation system. I designed an indoor environment with multiple rooms and obstacles, and created a grid-based graph representation of the walkable space. **[Screenshot #2]**

The environment consists of:

- Ten main rooms connected by doorways
- Multiple obstacles within most rooms
- A grid-based graph with 768 vertices (using a 20-pixel grid cell size)

When the user clicks on a location in the environment, the system:

1. Converts the click position to the nearest graph vertex
2. Finds the optimal path from the agent's current position to the target using either Dijkstra's Algorithm or A* (based on which mouse button was clicked)

3. Converts the path into a sequence of waypoints
4. Uses the Arrive and Align steering behaviors to smoothly follow the path

In one test case navigating from position (100,100) to (428,389), A* explored 126 nodes while Dijkstra's Algorithm explored 396 nodes (a 68% reduction) to find paths of equivalent cost (534.558). Both algorithms produced paths with 24 waypoints, though the specific waypoints differed slightly due to the different exploration patterns of the algorithms. This real-world application demonstrates the practical advantage of A* in a navigation scenario. **[Screenshot #3 and #4]**

The path following implementation uses two key steering behaviors:

- Arrive: Smoothly approaches waypoints with appropriate deceleration
- Align: Orients the agent to face its direction of travel

I implemented a breadcrumb system to visualize the agent's movement over time, which clearly shows how the agent smoothly navigates around obstacles while following the computed path. The implementation successfully demonstrates the integration of global pathfinding with local steering behaviors to create natural-looking movement through a complex environment.

Parameter Tuning and Performance Optimization

Several parameters significantly affected both pathfinding and path following performance:

Pathfinding Parameters

- Priority Queue Implementation: Using a binary heap-based priority queue was crucial for performance, especially for the large graph tests
- Visited Nodes Tracking: Efficiently tracking already-visited nodes prevented redundant exploration
- Heuristic Weight: Increasing the heuristic weight for A* could reduce node exploration but risked non-optimal paths

Path Following Parameters

- Waypoint Threshold: The distance at which the agent considers a waypoint reached (10 units in my implementation)
- Max Acceleration/Rotation: Higher values made the agent more responsive but could cause overshooting
- Slow Radius: Larger values created smoother deceleration but increased the time to reach targets
- Target Radius: The distance at which the agent stops moving toward a waypoint

I found that the most efficient path following occurred with a waypoint threshold of 10 units, which balanced precision with the need to avoid getting stuck trying to reach exact positions.

For the Arrive behavior, a relatively high max acceleration (250 units/s²) with a moderate slow radius (120 units) produced the best balance of responsiveness and smoothness.

Graph Structure Effects on Performance

The structure of the graph significantly impacted the relative performance of the algorithms:

1. Connectivity: Graphs with higher connectivity (more edges per vertex) reduced the advantage of A* over Dijkstra's Algorithm, as more potential paths needed to be explored
2. Spatial Correlation: When the graph had strong spatial correlation (for example, grid-like structures), distance-based heuristics like Euclidean and Manhattan performed exceptionally well for efficiency but didn't always maintain optimality
3. Clusters: Graphs with natural clustering (like the campus graph with building groups) benefited from A*'s focused exploration, allowing it to explore significantly fewer nodes while finding reasonable paths
4. Long Paths: For paths that traversed a large portion of the graph, Dijkstra's Algorithm explored a much larger percentage of the graph compared to A*, making the performance difference more pronounced

For example, in the campus graph test from the Bell Tower to Engineering Building III, Dijkstra's Algorithm explored 15 vertices, while A* with Euclidean heuristic explored only 6 vertices (40% of Dijkstra's exploration) while finding the same optimal path. Similarly, in one of the indoor environment tests, A* explored only 126 nodes compared to Dijkstra's 396 (a 68% reduction) while finding a path of equivalent cost.

However, in some cases in the large random graph, the spatial structure didn't align well with connectivity, causing A* to find suboptimal paths. For instance, in Trial 4, where the path from vertex 6918 to 2574 was particularly short, A* actually explored more nodes than Dijkstra's Algorithm (3,764 vs. 193) and found a much longer path. This demonstrates that A* heuristics can sometimes be misleading when the graph structure doesn't match the assumptions built into the heuristic function.

Conclusion

This assignment provided valuable insights into pathfinding algorithms and their integration with steering behaviors for game AI. The key findings include:

1. A* generally outperforms Dijkstra's Algorithm in terms of computational efficiency, with the advantage becoming more pronounced on larger graphs and longer paths. In the large graph tests, A* variants explored only about 25% of the nodes that Dijkstra's Algorithm did, and ran approximately 50% faster.

2. The choice of heuristic function is critical for A* performance. On well-structured graphs like the campus map, admissible heuristics like Euclidean distance provided a good balance between efficiency and path optimality. However, on the large random graph, all heuristics struggled to maintain optimality while still providing significant performance benefits.
3. In the small graph tests, inadmissible heuristics found optimal paths in 40% of test cases compared to 60% for admissible heuristics. However, in the large graph tests, none of the heuristics consistently found optimal paths, suggesting that heuristic effectiveness is highly graph-dependent.
4. The integration of global pathfinding with local steering behaviors created natural and smooth movement through complex environments. The implementation successfully navigated through the indoor environment with multiple rooms and obstacles.
5. Graph structure and topology significantly impacted algorithm performance. When pathfinding in the indoor environment with a clear grid structure, A* explored 68% fewer nodes than Dijkstra's Algorithm while finding a path of equivalent cost.

For future work, I would like to explore more advanced pathfinding techniques such as hierarchical pathfinding for large environments, dynamic pathfinding that adapts to changing environments, and integration with more complex AI decision-making systems. Additionally, I'm interested in developing more sophisticated heuristics that could adapt based on the specific characteristics of different regions in a game world.

This project has enhanced my understanding of how pathfinding and steering behaviors work together to create realistic movement in games, providing a solid foundation for developing more advanced AI navigation systems.

Appendix

Acknowledgements

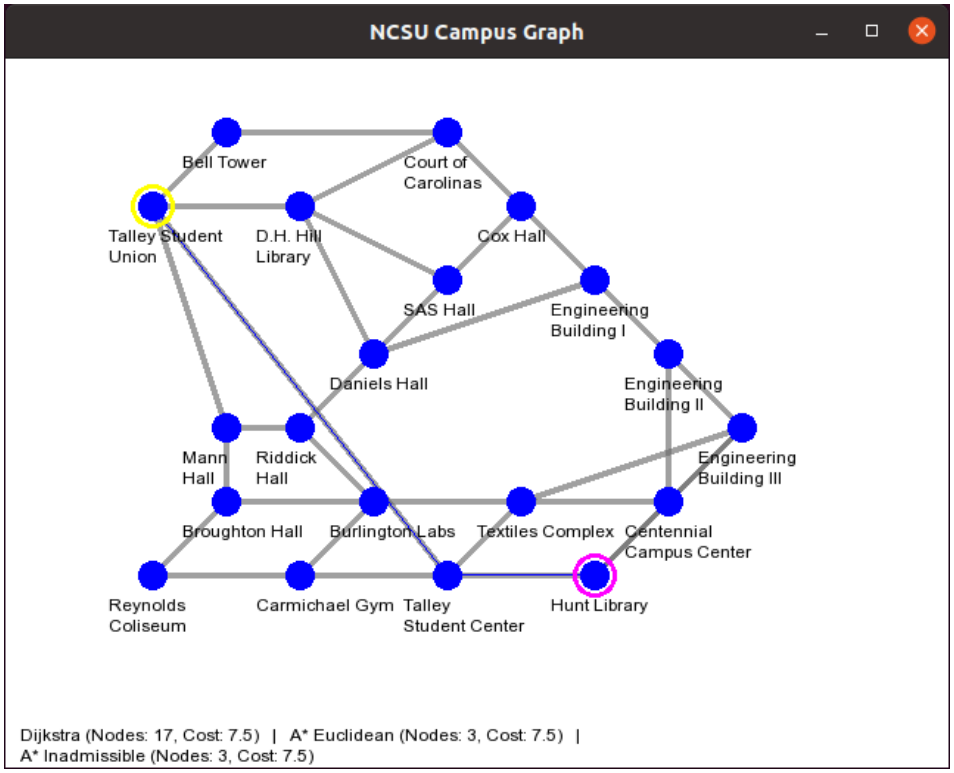
I referred to the SFML official tutorials to understand and implement graphical components for this project. The book "Artificial Intelligence for Games" by Ian Millington provided insights into pathfinding algorithms and their integration with steering behaviors.

I used OpenAI's ChatGPT as a resource to suggest template header files for Dijkstra's Algorithm and A* implementation, as well as for guidance on creating an indoor environment representation and integrating pathfinding with steering behaviors.

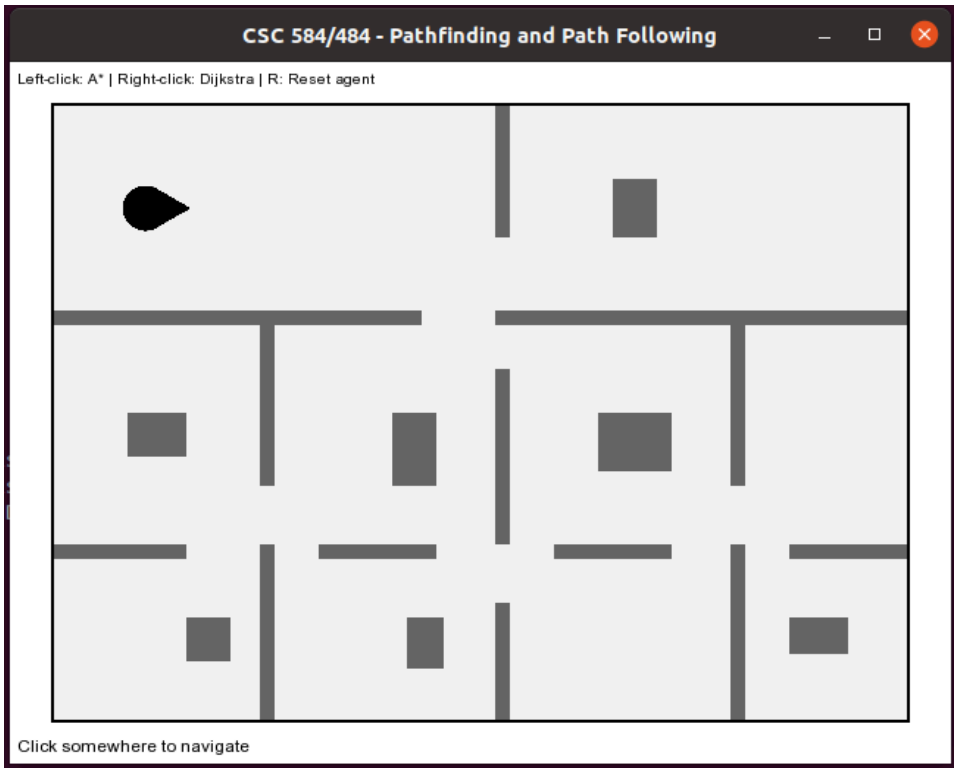
I adapted the Breadcrumb class provided by TA Derek Martin for visualizing agent movement, and used the sample Makefile provided by Dr. David L. Roberts as a starting point for my build configuration.

Screenshots

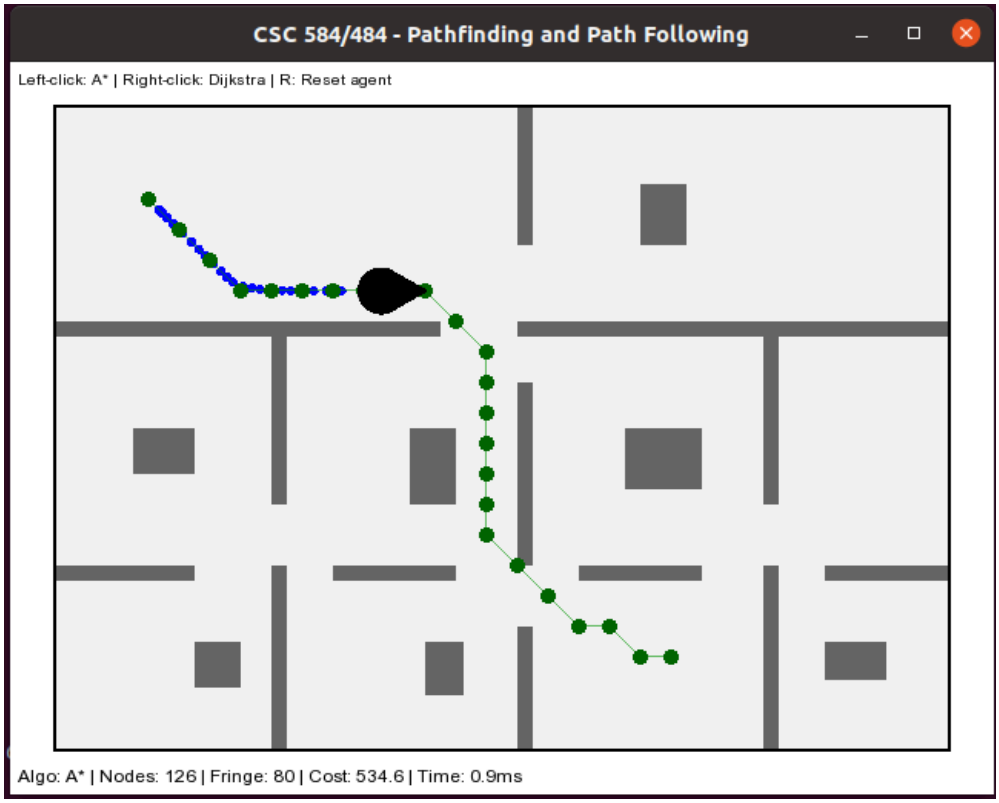
[Screenshot #1]



[Screenshot #2]



[Screenshot #3]



[Screenshot #4]