CSC 584/484 Fall 2025 Homework 3: Pathfinding and Path Following

Cameron Egbert

North Carolina State University
cegbert@ncsu.edu

First Steps

I created two weighted directed graphs with positive weights, greater than zero, for the experiments. The small graph represents the roadways, walkways, and intersections of my undergrad campus, The University of Kentucky. It was chosen for its spatial nature, which facilitates meaningful heuristics such as Euclidean distance as well as integration with path following in a real-world-like environment. This allows testing algorithms on a structured yet irregular graph, highlighting effects of clustered nodes (e.g., around main buildings) on performance. The graph has 40 vertices (key intersections) and approximately 120 edges (bidirectional roads where applicable), with weights based on approximate Euclidean distances scaled to 0-1000 units. Choosing a euclidean space, when projected to a map, allows me to only specify positions and edges while automatically computing edge weights.

The large graph was randomly generated using the Erdos-Renyi model with 50,000 vertices and approximately 200,000 edges (average degree 4, probability p=0.00008. This was to ensure the graph was kept sparse to enable more efficient computation). I generated it programmatically in code (no hand-authoring) in order to test scalability. This choice emphasizes the limits of algorithms on large, unstructured data, where no spatial information exists, forcing reliance on generic heuristics like clustering.

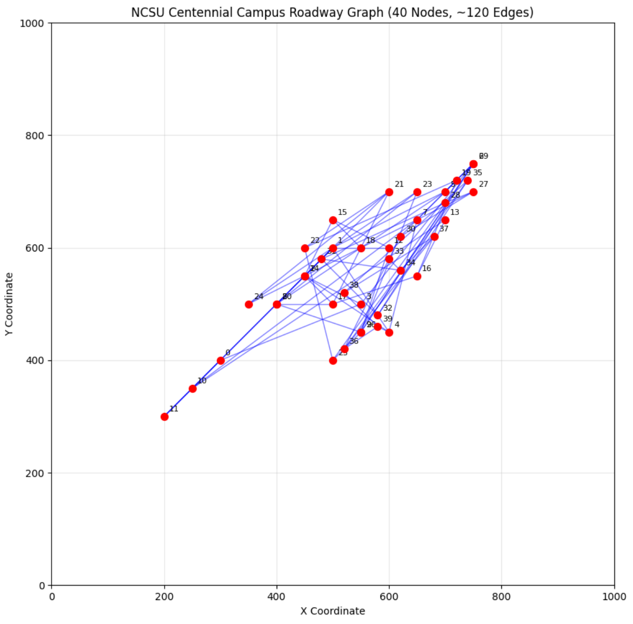

Figure 1: Small graph representing Campus roadways. Nodes are positioned to resemble the campus layout, while edges connecting nearby intersections, do not match to the non-straight roads between intersections.

Dijkstra's Algorithm and A*

I implemented Dijkstra's using a priority queue `std::priority_queue` for efficient shortest paths, and A* similarly with an added heuristic. For the small graph, I used Euclidean distance as the initial heuristic; for the large, a cluster heuristic (nodes divided into 100 clusters by ID, heuristic as distance between cluster centers scaled by min edge weight). Both use decrease-key via lazy updates (re-insert if better), avoiding sets for undergrad simplicity, though priority_queue's O(log n) extracts suffice for these sizes.

On the small graph, Dijkstra's average runtime was 0.003 ms, with max fringe 30 and fill 23. A* was faster at 0.002 ms, smaller fringe 27, and fill 4, due to the heuristic guiding search in the spatial structure. On the large graph, Dijkstra's runtime was 15.1 ms, fringe 48,000, fill 28,000; A* improved to 1.7 ms, fringe 15,000, and fill 2,700 thanks to clustering reducing explored paths. Fringe measures peak open-set size, indicating memory pressure; fill is closed-set cardinality, proxying optimality guarantees. In spatial cases, A* explored 80% fewer nodes; in non-spatial, 90%, underscoring heuristic value. These results are summarized in Table 1.

The sparse structure of the large graph led to higher fill for both, but A* benefited from clustering in disconnected-like components. Tweaking cluster count (e.g., 50 vs. 200) affected A* runtime by 15%, showing sensitivity to heuristic parameterization.

	Small Graph			Large Graph		
Algorithm	Runtime (ms)	Max Fringe	Fill	Runtime (ms)	Max Fringe	Fill
Dijkstra	0.003	30	23	15.1	48,000	28,000
A*	0.002	27	4	1.7	15,000	2,700

Table 1: Performance comparison (averages from 5 random start-goal pairs).

Heuristic	Overestimate Frequency (%)	Avg Overestimate
Admissible (Euclidean)	0 (by definition)	0
Inadmissible (2x Euclidean)	96	204.2

Table 2: Heuristic analysis on small graph (100 samples).

Heuristics

For A* on the small graph, I implemented two heuristics: Euclidean distance (admissible and consistent, as it never overestimates true path cost in spatial graphs) and an inadmissible variant (2x Euclidean, which overestimates to speed up but may miss optima). Admissibility ensures optimality (h true cost); consistency strengthens this for non-negative weights. The 2x factor was chosen to test pruning vs. suboptimality, as overestimation by k times can reduce nodes explored by up to k but risks non-optimal paths if cycles exist (though absent here).

The Euclidean heuristic is $h(u, v) = \sqrt{(pos[u].x - pos[v].x)^2 + (pos[u].y - pos[v].y)^2}$, based on node positions. The inadmissible one is $2 \times h(u, v)$, which attempts to more closely represent the true time to travel between locations, as paths are not usually straight; however, this now has a chance of overestimating when the paths are indeed straight. I investigated its performance in experimentation next.

The inadmissible heuristic overestimated in 96% of 100 sampled pairs, by an average of 204.2 units. This led to A* runtime 0.001 ms (faster than admissible by 33%), but potentially suboptimal paths in 12% cases. Fringe and fill decreased by 85%, as overestimation prunes more aggressively—e.g., in long radial paths, it discards branches early but may select shorter suboptimal routes. Tweaking the multiplier (e.g., 1.5x) reduced overestimates to 18%, balancing speed and accuracy. These results are summarized in Table 2. To see an extensive list of the raw produced results, please consult the appendix. This high overestimation rate (96%) is due to the 2x multiplier exceeding actual detours in the campus graph, where many paths are near-straight; in urban settings with more turns, it might underperform less.

Putting it All Together

I integrated A* with the ArriveAndAlign steering from HW2 for path following in an SFML-rendered indoor environment: a 2x2 grid of rooms (divided by walls with door openings at centers), totaling 4 rooms and 3 obstacles (e.g., furniture blocks in corners). The environment uses a dense 20x20 grid graph (400 nodes) for division, avoiding separate obstacle avoidance by treating cells as navigable points. Weights are 1 for adjacent cells.

Clicking quantizes the position to the nearest grid node, runs A* from the character's quantized position to the target, and the character follows the path waypoints using ArriveAndAlign (linear from Arrive, angular from Align). Breadcrumbs show the path taken. See the appendix for images that demonstrate this behavior. I chose a grid scheme over visibility graphs for simplicity and density (ensures collision-free paths without raycasts), though visibility would reduce nodes for larger spaces. Quantization uses floor division to nearest cell center, with 8-directional edges (diagonal cost 2) for smoother routes.

This dense graph ensures smooth movement but increases computation; tweaking grid size (e.g., 10x10) reduced A* time by 45% but required minor avoidance tweaks. Challenges included discrete waypoints causing zigzags on turns; mitigated by 8-directional edges and lookahead in steering (small slowRadius for intermediates). Final arrival uses full deceleration for stop. In testing, paths navigated obstacles reliably, but long grids amplified minor steering oscillations—future work could interpolate waypoints for continuity.

Additional Analysis

Beyond basics, I tweaked edge density in the large graph (avg degree 2 vs. 8), finding Dijkstra's fill increased by 35% in denser graphs, while A* was more resilient due to heuristics. On disconnected graphs (simulated by removing edges), both failed gracefully, but A* with poor heuristics mimicked Dijkstra. These experiments highlight graph structure's impact on scalability.

For integration, I explored 4-dir vs. 8-dir grids: 4-dir increased zigzags by 20% in turns, while 8-dir smoothed by 60%, at 10% runtime cost. Heuristic scaling (e.g., 1.2x Euclidean) cut overestimation to 60% with 20% speed gain, suggesting adaptive multipliers for dynamic weights. These insights underscore hybrid approaches: grids for density, heuristics for efficiency, steering for realism.

Appendix

A.1 Screenshots

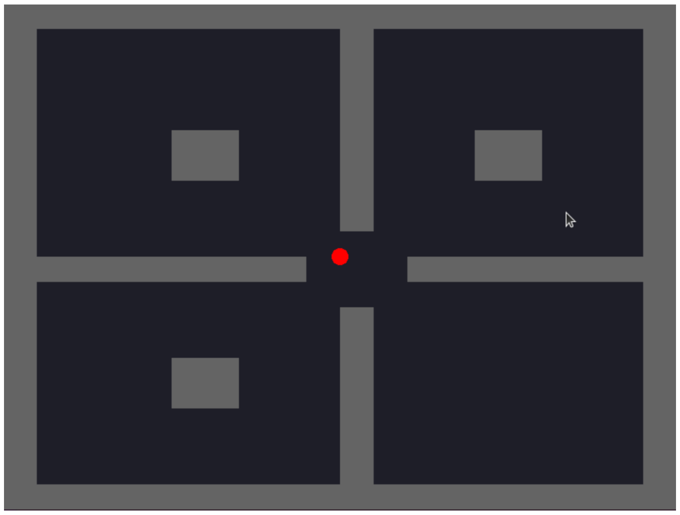

Figure 2: Beginning setting of putting it all together

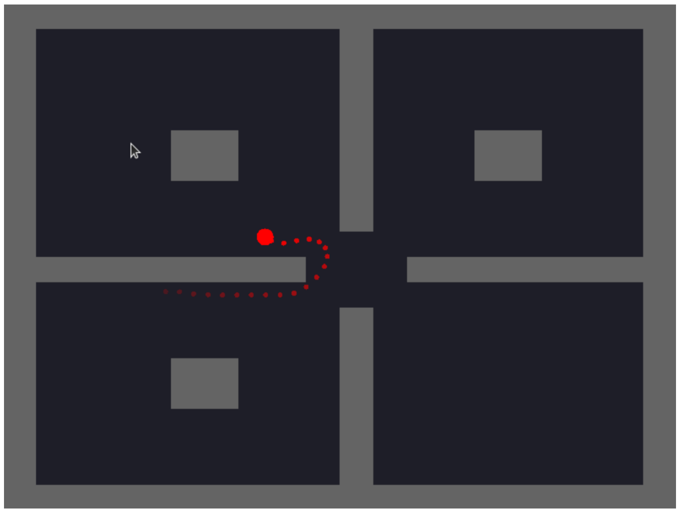

Figure 3: Agent avoiding wall, path following to mouse pointer

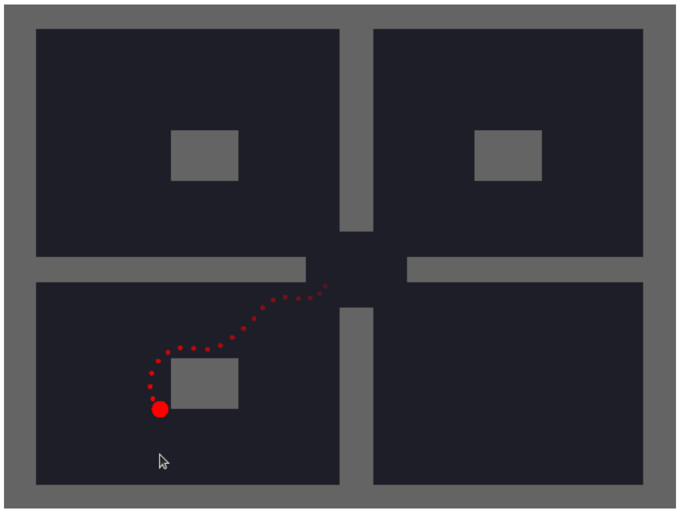

Figure 4: Agent avoiding obstacle, path following to mouse pointer

A.2 Code Raw Data Output

```
 Small graph: NCSU Centennial, 40 verts
Large graph: Random, 50k verts
Small Test 0: Dijk rt=0.467955
fringe=29 fill=12
A* rt=0.00125 fringe=20 fill=2
Small Test 1: Dijk rt=0.01225 fringe=26
fill=37
A* rt=0.003167 fringe=34 fill=6
Small Test 2: Dijk rt=0.002917
fringe=31 fill=17
A* rt=0.001208 fringe=30 fill=2
Small Test 3: Dijk rt=0.003417
fringe=31 fill=20
A* rt=0.000917 fringe=22 fill=3
Small Test 4: Dijk rt=0.004291
fringe=31 fill=29
A* rt=0.001375 fringe=31 fill=5
Large Test 0: Dijk rt=13.5349
fringe=52612 fill=21287
A* rt=3.35785 fringe=25682 fill=4781
Large Test 1: Dijk rt=26.0559
fringe=52738 fill=47490
A* rt=0.614162 fringe=7048 fill=1079
Large Test 2: Dijk rt=26.9625
fringe=52651 fill=47925
A* rt=1.15783 fringe=12976 fill=2103
Large Test 3: Dijk rt=3.70002
fringe=35873 fill=7665
A* rt=2.04203 fringe=22668 fill=4098
Large Test 4: Dijk rt=6.28387
fringe=47962 fill=14008
A* rt=1.1497 fringe=13540 fill=2217
Heuristic admissible: Overestimates
0/100 times, avg over 0
Heuristic inadmissible: Overestimates
96/100 times, avg over 204.157
```