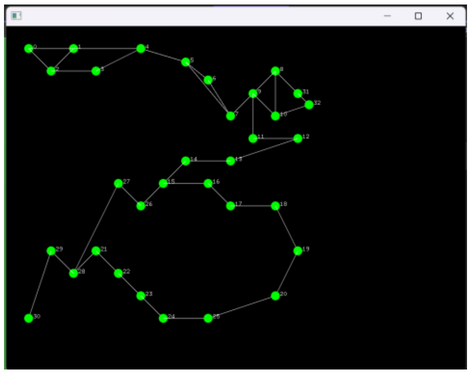
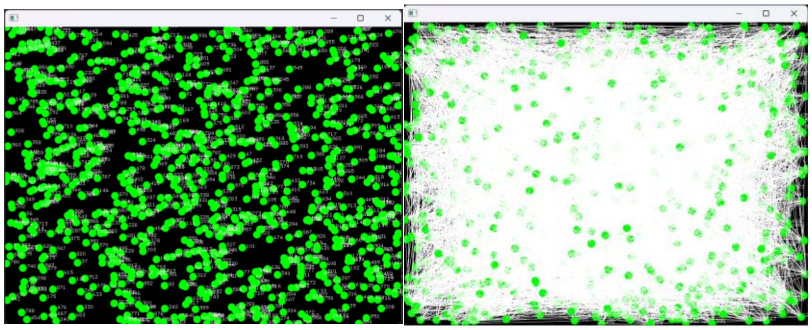
Process Letter

First Steps

The first graph represents a map of some Chinese cities and their connections. Each city is represented as a node with specific coordinates. The adjacency list defines which cities are connected to each other. I manually defined the positions and the connections between cities to form a geographical layout.

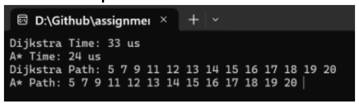


The second graph is a randomly generated graph that uses a probabilistic model to connect nodes. I created 1000 nodes, each randomly placed on the canvas, and connected them based on a probability of edge creation. This approach results in a more abstract, randomized structure, which can be useful for testing algorithm.

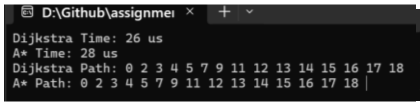


Dijkstra’s Algorithm, A*

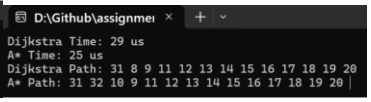
Dijkstra’s Algorithm: This algorithm explores nodes based on the current shortest distance (g value) from the start node. It uses a priority queue (min-heap) to ensure that the node with the smallest distance is processed next. For each node, it updates the distance of its neighboring nodes and adds them to the queue for further processing.
A Algorithm*: Similar to Dijkstra, A* also explores nodes based on distance, but it adds a heuristic (h value) to guide the search. The heuristic estimates the distance from a node to the goal, and A* prioritizes nodes with the lowest total cost, f(n) = g(n) + h(n). This often makes A* more efficient, especially in large graphs, as it avoids exploring unnecessary paths.
Using Small Graph : A* was slightly faster than Dijkstra, but the difference was minimal. Both algorithms visited roughly the same number of nodes. With a smaller graph, the heuristic has less impact, and both algorithms can find the path fairly quickly. As a result, there was not much difference in the number of nodes explored.



```
D:\Github\assignme
Dijkstra Time: 33 us
A* Time: 24 us
Dijkstra Path: 5 7 9 11 12 13 14 15 16 17 18 19 20
A* Path: 5 7 9 11 12 13 14 15 16 17 18 19 20
```

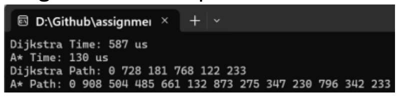


```
D:\Github\assignme
Dijkstra Time: 26 us
A* Time: 28 us
Dijkstra Path: 0 2 3 4 5 7 9 11 12 13 14 15 16 17 18
A* Path: 0 2 3 4 5 7 9 11 12 13 14 15 16 17 18
```

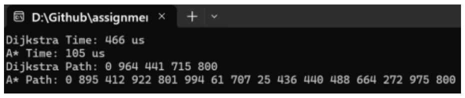


```
D:\Github\assignme
Dijkstra Time: 29 us
A* Time: 25 us
Dijkstra Path: 31 8 9 11 12 13 14 15 16 17 18 19 20
A* Path: 31 32 10 9 11 12 13 14 15 16 17 18 19 20
```

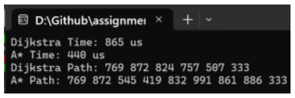
Using Large Graph : A* was 2-4 times faster than Dijkstra, and Dijkstra visited significantly fewer nodes than A*. A* was able to efficiently guide its search using the heuristic, reducing the number of nodes it explored. On the other hand, Dijkstra, without any heuristic to guide it, explored more nodes and took longer to find the path.



```
Dijkstra Time: 587 us
A* Time: 130 us
Dijkstra Path: 0 728 181 768 122 233
A* Path: 0 908 504 485 661 132 873 275 347 230 796 342 233
```



```
Dijkstra Time: 466 us
A* Time: 105 us
Dijkstra Path: 0 964 441 715 800
A* Path: 0 895 412 922 801 994 61 707 25 436 440 488 664 272 975 800
```



```
Dijkstra Time: 865 us
A* Time: 440 us
Dijkstra Path: 769 872 824 757 507 333
A* Path: 769 872 545 419 832 991 861 886 333
```

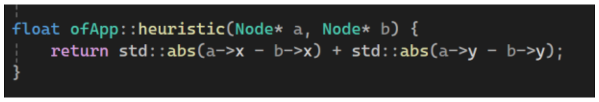
One interesting observation is that the effectiveness of A* heavily depends on the quality of the heuristic. If the heuristic is well-designed (e.g., admissible and consistent), A* can drastically outperform Dijkstra. However, if the heuristic is poorly chosen (e.g., overestimates the cost or is unrelated to the actual distances), A* may degrade to a performance similar to—or worse than—Dijkstra's.

Heuristics

1. Manhattan Distance Heuristic

Manhattan Distance is the sum of the absolute differences between the x-coordinates and y-coordinates of two points. It is typically used in grid-based environments where movement is restricted to horizontal and vertical directions.

$$h(n) = |x_n - x_{\text{goal}}| + |y_n - y_{\text{goal}}|$$

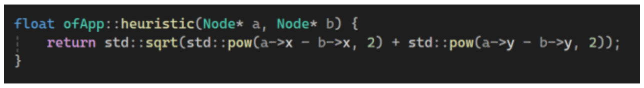


```cpp
float ofApp::heuristic(Node* a, Node* b) {
    return std::abs(a->x - b->x) + std::abs(a->y - b->y);
}
```

2. Euclidean Distance Heuristic

Euclidean Distance measures the straight-line (as-the-crow-flies) distance between two nodes.

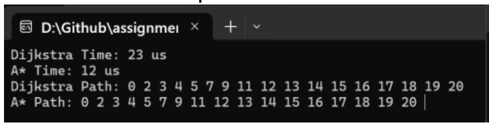
$$h(n) = \sqrt{(x_n - x_{\text{goal}})^2 + (y_n - y_{\text{goal}})^2}$$



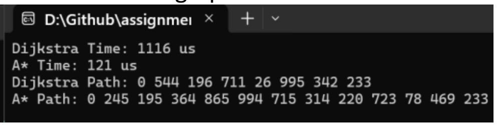
```cpp
float ofApp::heuristic(Node* a, Node* b) {
    return std::sqrt(std::pow(a->x - b->x, 2) + std::pow(a->y - b->y, 2));
}
```

Both heuristics are admissible and consistent. Manhattan Distance never overestimates and considers only horizontal and vertical movement, ensuring optimal paths. Euclidean Distance, representing straight-line distance, respects the triangle inequality and provides a more accurate estimate, improving performance when diagonal movement is allowed.

Using the Manhattan Distance Heuristic causes Dijkstra's pathfinding time to be longer in large-scale datasets. Below are the experimental results for the first and second graphs.



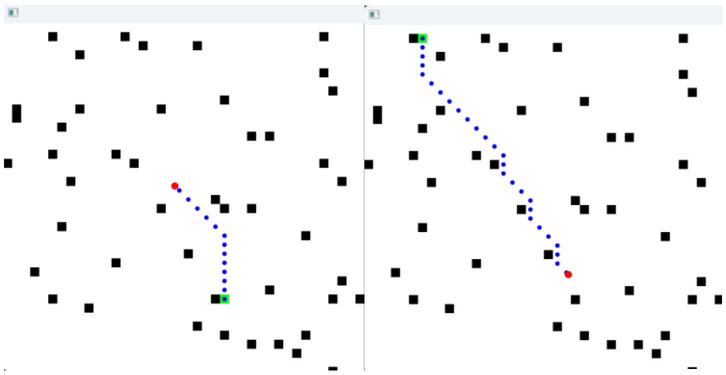
```
Dijkstra Time: 23 us
A* Time: 12 us
Dijkstra Path: 0 2 3 4 5 7 9 11 12 13 14 15 16 17 18 19 20
A* Path: 0 2 3 4 5 7 9 11 12 13 14 15 16 17 18 19 20
```



```
Dijkstra Time: 1116 us
A* Time: 121 us
Dijkstra Path: 0 544 196 711 26 995 342 233
A* Path: 0 245 195 364 865 994 715 314 220 723 78 469 233
```

Putting It All Together

In the diagram below, the white squares represent walkable paths, the black squares represent obstacles, and the blue dots indicate the computed walking path.



To avoid obstacles, I allow diagonal movement in areas without obstacles, while applying special handling for right-angle movement in areas with obstacles.