A* search algorithm

***Source:* https://github.com/rusevrosen/codehub.github.io/tree/main/Algorithms**

1. *What is an A* search algorithm?*

A* search is a heuristic search algorithm that is used to find the shortest path between two points in a graph. It combines the strengths of both breadth-first search and uniform-cost search by using a heuristic function to guide the search and prioritize the expansion of certain vertices over others.

The heuristic function used in the A* search estimates the distance from a given vertex to the goal vertex. This estimate, also known as the "heuristic cost," is used to guide the search and help it converge on a solution more quickly. The actual cost of the path from the starting vertex to the current vertex is also considered in the A* search algorithm.

The A* search algorithm works by maintaining two lists of vertices: an open list and a closed list. The open list contains the vertices that have been discovered but not yet fully explored, while the closed list contains the vertices that have been fully explored. At each step, the algorithm selects the vertex with the lowest combined cost (i.e., actual cost plus heuristic cost) from the open list and adds it to the closed list. It then expands the selected vertex by adding its neighbors to the open list and updating their costs as necessary. This process is repeated until the goal vertex is added to the closed list or there are no more vertices on the open list.

A* search is often used in pathfinding and navigation applications, such as video games and autonomous robots, due to its ability to find optimal solutions quickly. It is also used in other areas, such as natural language processing and computer-aided design, where the ability to find good solutions quickly is important.

2. *Code example.*

```
#include <unordered_map>

#include <queue>

#include <vector>


struct Vertex {

  int x, y;

  int g_score;

  int f_score;

  Vertex* parent;


  bool operator==(const Vertex& other) const {

    return x == other.x && y == other.y;

  }

};


struct VertexHasher {

  std::size_t operator()(const Vertex& v) const {

    return v.x * 31 + v.y;

  }

};


int HeuristicCostEstimate(Vertex& start, Vertex& goal) {

  // Euclidean distance between start and goal as the heuristic cost estimate

  return sqrt((start.x - goal.x) * (start.x - goal.x) + (start.y - goal.y) * (start.y
- goal.y));

}


std::vector<Vertex> ReconstructPath(Vertex& start, Vertex& current) {

  std::vector<Vertex> path;

  path.push_back(current);

  while (current.parent != nullptr && current != start) {

    current = *current.parent;

    path.push_back(current);

  }
  std::reverse(path.begin(), path.end());

  return path;

}


std::vector<Vertex> AStarSearch(Vertex& start, Vertex& goal,
std::vector<std::vector<int>>& map) {

  std::unordered_map<Vertex, Vertex, VertexHasher> came_from;

  std::unordered_map<Vertex, int, VertexHasher> g_score, f_score;

  std::priority_queue<std::pair<int, Vertex*>, std::vector<std::pair<int, Vertex*>>,
std::greater<>> open_set;


  g_score[start] = 0;

  f_score[start] = HeuristicCostEstimate(start, goal);

  open_set.emplace(f_score[start], &start);


  while (!open_set.empty()) {

    Vertex* current = open_set.top().second;

    open_set.pop();


    if (*current == goal) {

      return ReconstructPath(start, *current);

    }


    // Generate the neighbors of the current vertex

    std::vector<Vertex> neighbors;

    // Add logic to generate neighbors here


    for (Vertex& neighbor : neighbors) {

      if (came_from.count(neighbor)) {

        continue;

      }


      int tentative_g_score = g_score[*current] + 1; // Assume a cost of 1 for each
edge
```

```
      if (!g_score.count(neighbor) || tentative_g_score < g_score[neighbor]) {
        came_from[neighbor] = *current;
        g_score[neighbor] = tentative_g_score;
        f_score[neighbor] = g_score[neighbor] + HeuristicCostEstimate(neighbor, goal);
        open_set.emplace(f_score[neighbor], &neighbor);
        neighbor.parent = current;
      }
    }
  }

  return {}; // Return an empty path if no path was found
}

int main() {
  // Set up the start and goal vertices
  Vertex start{0, 0};
  Vertex goal{5, 5};

  // Set up the map
  std::vector<std::vector<int>> map = {{0, 0, 0, 0, 0, 0},
                                       {0, 1, 1, 1, 1, 0},
                                       {0, 0, 0, 0, 1, 0},
                                       {0, 1, 1, 1, 1, 0},
                                       {0, 1, 0, 0, 0, 0},
                                       {0, 0, 0, 0, 0, 0}};
  // Find the shortest path from start to goal
  std::vector<Vertex> path = AStarSearch(start, goal, map);

  // Print the path
  for (Vertex& v : path) {
    std::cout << "(" << v.x << ", " << v.y << ")" << std::endl;
  }
  return 0;
}
```

The output of the example is a series of coordinates representing the shortest path from the start vertex to the goal vertex as determined by the A* search algorithm. The path is printed to the standard output stream (e.g., the console), with each coordinate being printed on a separate line.

For example, if the start vertex is at coordinates (0, 0) and the goal vertex is at coordinates (5, 5), and the map is set up as shown in the example, the output might be:

```
(0, 0)

(1, 1)

(2, 2)

(3, 3)

(4, 4)

(5, 5)
```

This indicates that the shortest path from the start vertex to the goal vertex goes through the vertices at coordinates (1, 1), (2, 2), (3, 3), and (4, 4).

Note that the output of the example will depend on the specifics of the map and the start and goal vertices, and there may be multiple valid paths depending on the setup.

3. Code explanation.

The Vertex structure represents a vertex in the graph being searched. It has x and y fields representing the coordinates of the vertex, and a parent field pointing to the vertex's parent in the path. It also has **`g_score`** and **`f_score`** fields that are used by the A* search algorithm to store the cost of the path from the start vertex to the current vertex, and the estimated total cost of the path from the start vertex to the goal vertex through the current vertex, respectively.

The **`VertexHasher`** structure is a **`functor`** (i.e., a class with an operator() overload) that defines a hash function for Vertex objects. This is used to store Vertex objects in the **`unordered_map`** containers used by the A* search algorithm.

The **`HeuristicCostEstimate`** function takes two vertices as arguments (the start vertex and the goal vertex) and returns an estimate of the distance from the start vertex to the goal vertex. In this example, the Euclidean distance is used as the heuristic cost estimate.

The **`ReconstructPath`** function takes two vertices as arguments (the start vertex and the current vertex) and returns a vector of vertices representing the path from the start vertex to the current vertex. It does this by following the parent pointers of each vertex back to the start vertex and adding the vertices to the path. The path is then reversed so that it goes from the start vertex to the current vertex.

The **`AStarSearch`** function is the main A* search algorithm. It takes three arguments: the start vertex, the goal vertex, and the map. It returns a vector of vertices representing the shortest path from the start vertex to the goal vertex.

The **AStarSearch** function uses several data structures to keep track of the vertices that have been discovered and the cost of the paths to those vertices. The **came_from** map stores a mapping from each vertex to the vertex it was discovered from, the **g_score** map stores the cost of the path from the start vertex to each vertex, and the **f_score** map stores the estimated total cost of the path from the start vertex to the goal vertex through each vertex. The **open_set** priority queue stores the vertices that have been discovered but not yet fully explored, with the vertices ordered by their **f_score**.

The **AStarSearch** function begins by initializing the **g_score** and **f_score** of the start vertex to 0 and the heuristic cost estimate from the start vertex to the goal vertex, respectively. It then adds the start vertex to the **open_set** priority queue.

The main loop of the **AStarSearch** function continues until the **open_set** priority queue is empty. At each iteration, it selects the vertex with the lowest **f_score** from the **open_set** priority queue and removes it from the queue. If the selected vertex is the goal vertex, the **AStarSearch** function returns the path from the start vertex to the goal vertex by calling the **ReconstructPath** function. If the selected vertex is not the goal vertex, the **AStarSearch** function generates the neighbors of the selected vertex and adds them to the **open_set** priority queue if they have not already been discovered. It also updates the **g_score**, **f_score**, and parent fields of the neighbors as necessary.

If the **open_set** priority queue becomes empty without the goal vertex being found, the **AStarSearch** function returns an empty vector, indicating that no path was found.

To generate the neighbors of a vertex, you will need to add logic to the **AStarSearch** function to check the adjacent squares of the map and create Vertex objects for any squares that are passable (i.e., have a value of 0 in the map). You can also add logic to check for obstacles or other constraints that may affect the search.

Note that this is just one possible implementation of A* search in C++, and there are many ways to customize and extend the algorithm to meet specific needs. For example, you might want to use a different heuristic cost estimate, allow for diagonal movement, or handle tie-breaking in the **open_set** priority queue in a specific way.

In the **AStarSearch** function, you will need to add logic to the neighbor's loop to update the **g_score**, **f_score**, and parent fields of the neighbors as necessary. This will typically involve calculating the **g_score** of the neighbor based on the **g_score** of the current vertex and the cost of the edge between them (which is assumed to be 1 in this example), and then using the **g_score** and the heuristic cost estimate to calculate the **f_score** of the neighbor. You will also need to set the parent field of the neighbor to point to the current vertex.

You will also need to add logic to the **AStarSearch** function to handle tie-breaking in the **open_set** priority queue. If two vertices have the same **f_score**, you will need to decide which one to expand first. One way to handle this is to use the **g_score** of the vertices as a tie-breaker, with the vertex having the lower **g_score** being expanded first.

The **AStarSearch** function assumes that all edges in the graph have a cost of 1. If you want to allow for edges with different costs, you will need to modify the **tentative_g_score** calculation to take the edge cost into account. For example, if the edge between the current vertex and the neighbor has a cost of c, the **`tentative_g_score`** of the neighbor should be calculated as **`g_score[*current] + c`** instead of **`g_score[*current] + 1`**.

The **`AStarSearch`** function assumes that the map is a 2D grid with passable squares represented by 0s and obstacles represented by 1s. If you want to use a different representation for the map or allow for more complex maps, you will need to modify the **`AStarSearch`** function accordingly. For example, you might want to use a different data structure to represent the map or add logic to handle different types of obstacles or costs.

4. Time complexity.

The time complexity of the A* search algorithm implemented in the example above is generally **`O(b^d),`** where **`b`** is the branching factor of the graph (i.e., the average number of neighbors of each vertex) and **`d`** is the depth of the optimal solution (i.e., the length of the shortest path from the start vertex to the goal vertex).

In the worst case, where the entire map is passable and the start and goal vertices are as far apart as possible, the time complexity of the A* search algorithm will be **`O(b^d).`** In this case, the algorithm will have to explore all the vertices of the graph to find the shortest path, and the time complexity will be determined by the number of vertices that have to be explored.

In the best case, where the start and goal vertices are adjacent or there is an obstacle between them, the time complexity of the A* search algorithm will be **`O(1).`** In this case, the algorithm will be able to find the shortest path without having to explore any other vertices, and the time complexity will be constant.

Overall, the time complexity of the A* search algorithm is generally considered to be very good, as it can find the shortest path in a graph in a relatively efficient manner. However, the time complexity will depend on the specifics of the graph being searched and the heuristic cost estimate being used, and there may be situations where the algorithm performs poorly. For example, if the heuristic cost estimate is very inaccurate or the branching factor of the graph is very high, the time complexity of the A* search algorithm may be significantly higher.

I wish you happy learning,

Your Rosen Rusev