# A Comparative Study of A* and Hill Climbing Algorithms for Pathfinding Applications

Xinyi yu
*University of Goldsmiths*
*Email: yuxiny1@me.com*

*Date : 1 April , 2023*

# Table of Contents:

## Introduction:

The A* search algorithm is a widely used heuristic pathfinding algorithm in computer science, designed to efficiently find the shortest path between two points in a graph or grid. By employing heuristics to guide its search towards the goal, A* search achieves better efficiency compared to other algorithms, such as Dijkstra's algorithm. It successfully combines the advantageous features of both Dijkstra's algorithm and the greedy Best-First-Search algorithm. However, A* search comes with the drawback of high memory consumption, depending on the problem and its implementation.

It is essential to understand that A* search is designed to find the shortest path from a specified source to a particular goal, whereas Dijkstra's algorithm generates a shortest path tree by considering all possible routes. Consequently, A* search may not discover the shortest path tree from a specified source to all potential goals. While Dijkstra's algorithm might provide more optimal solutions, it is less efficient and more computationally expensive than A* search. Furthermore, this report will compare A* search with hill climbing, as both algorithms utilize heuristics.

In summary, the A* search algorithm offers a balance between route quality and algorithm execution efficiency. It can be viewed as an enhanced version of Dijkstra's algorithm, which takes this trade-off into account to efficiently determine the shortest path between two points in a graph or grid. Due to its efficiency and accuracy, A* search is commonly employed in various fields such as robotics, gaming, and logistics

## Description of A* Search Algorithm:

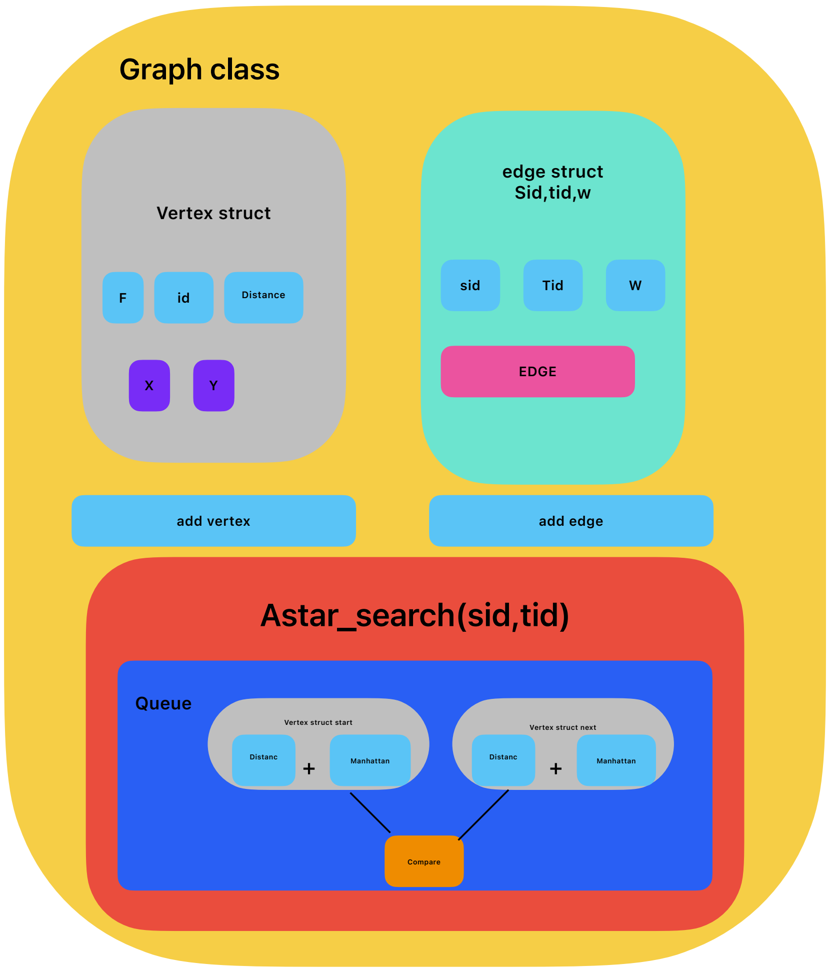
This report has already discussed that the A* search algorithm is an informed search algorithm applied to weighted graphs, enabling users to find the lowest-cost path. Starting from a specific source node, the A* search algorithm constructs a tree of paths, incrementally extending these paths by adding one edge at a time until the algorithm terminates, ultimately identifying the lowest-cost route from the source node to the destination.

The A* search algorithm selects a path according to the formula **$f(n) = g(n) + h(n)$**, where n represents the next node to be added to the path. Each path corresponds to one iteration of the main loop, with the algorithm relying on path costs and a heuristic function to estimate the total cost of extending the path through other unknown nodes to the destination.

The algorithm assesses nodes in the graph based on the cost of reaching that node from the starting node (known as the "**g(n)**" value) and an estimate of the cost of traveling from that node to the destination node (known as the **"h(n)"** value). The sum of these two values is the **"f(n)"** value, and the algorithm chooses the node with the lowest f value as the next node to expand. A* terminates when no eligible paths can be extended further or when it has already reached the destination. Furthermore, the heuristic function is admissible, meaning it will not overestimate the overall route cost.

**Implementation of A*:**

(The graph shows the structure of A* search function from the code example)



### 3.1. Open List and Node Properties

At the beginning of the A* algorithm, a priority queue called the "open list" is initialized. A* employs a priority queue (min heap) to choose the node with the lowest estimated total cost **(f(x))** to expand during each iteration, updating costs and re-estimating neighboring nodes.

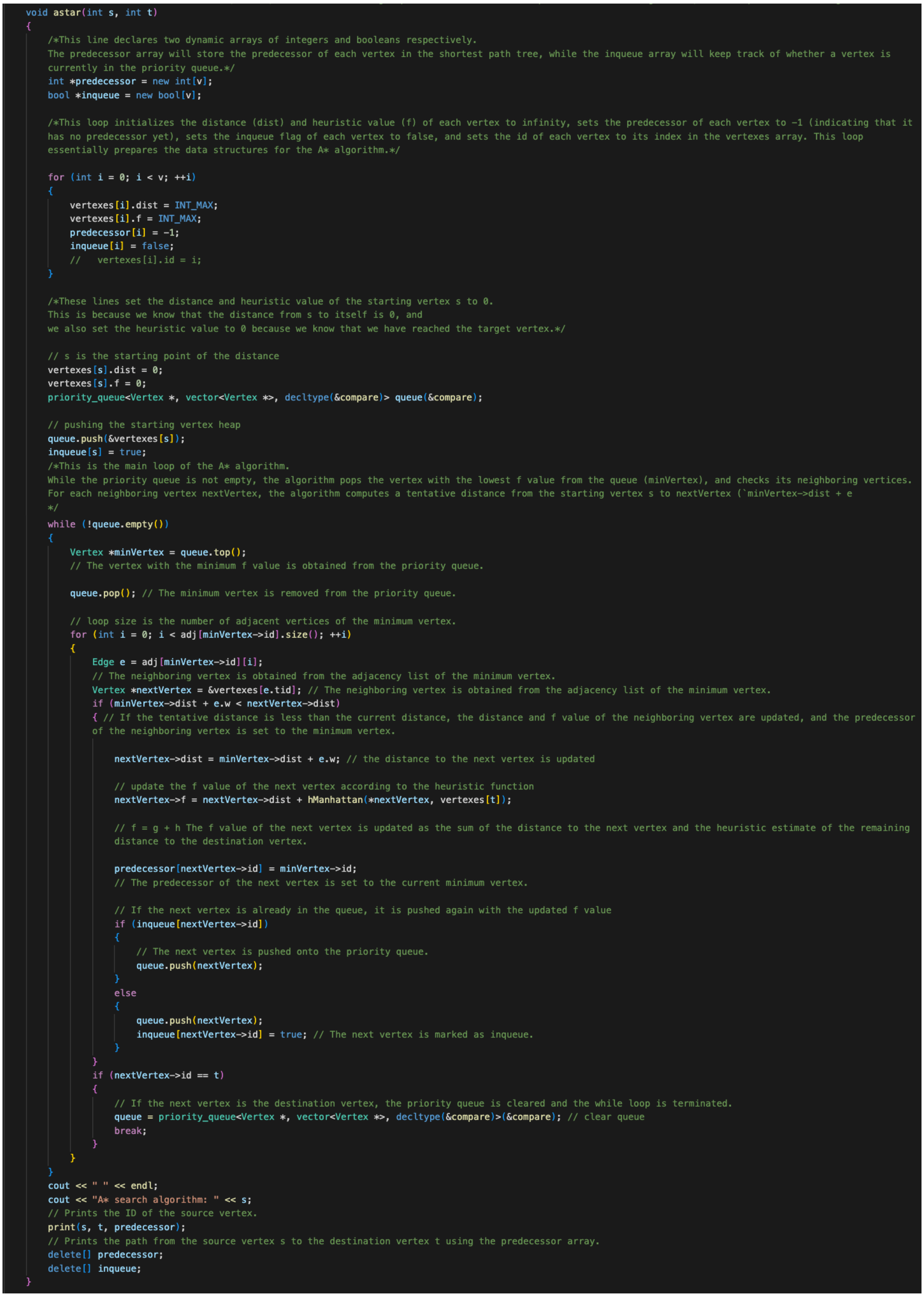
Every node in the heap is assigned **three** values: g, h, and f. The g value represents the actual cost of reaching a node from the starting node, the h value estimates the cost to reach the goal node from that node, and the f value is the sum of g and h.

In each iteration, the selected node is called the "current node," and its neighbors are examined. The priority queue keeps track of visited nodes that have not yet been expanded until the last node is reached. This indicates that no path exists from the starting node to the destination. Otherwise, the shortest path is found when the destination node is in the heap.

### 3.2. Main Loop and Path Expansion

To optimize the algorithm, A* is admissible by ensuring the estimate from a node to a successor node is always less than or equal to the estimate from the starting node to that successor node, plus the cost of moving from the node to its successor.

The a_star function takes the start and goal nodes, as well as three functions: neighbors to return a node's neighbors, distance to calculate the distance between two nodes, and heuristic to estimate the distance from a node to the goal. It then initializes the open set with the start node and the **g-score** and **f-score** maps with the start node's values. The algorithm loops through the open set, choosing the node with the lowest f-score and updating its neighbors' g-scores and f-scores if a better path is found. Once the goal is reached, the function returns the reconstructed path, or None if the goal was not reached.



```cpp
void astar(int s, int t)
{
    /*This line declares two dynamic arrays of integers and booleans respectively.
    The predecessor array will store the predecessor of each vertex in the shortest path tree, while the inqueue array will keep track of whether a vertex is
    currently in the priority queue.*/
    int *predecessor = new int[v];
    bool *inqueue = new bool[v];

    /*This loop initializes the distance (dist) and heuristic value (f) of each vertex to infinity, sets the predecessor of each vertex to -1 (indicating that it
    has no predecessor yet), sets the inqueue flag of each vertex to false, and sets the id of each vertex to its index in the vertexes array. This loop
    essentially prepares the data structures for the A* algorithm.*/

    for (int i = 0; i < v; ++i)
    {
        vertexes[i].dist = INT_MAX;
        vertexes[i].f = INT_MAX;
        predecessor[i] = -1;
        inqueue[i] = false;
        //   vertexes[i].id = i;
    }

    /*These lines set the distance and heuristic value of the starting vertex s to 0.
    This is because we know that the distance from s to itself is 0, and
    we also set the heuristic value to 0 because we know that we have reached the target vertex.*/

    // s is the starting point of the distance
    vertexes[s].dist = 0;
    vertexes[s].f = 0;
    priority_queue<Vertex *, vector<Vertex *>, decltype(&compare)> queue(&compare);

    // pushing the starting vertex heap
    queue.push(&vertexes[s]);
    inqueue[s] = true;
    /*This is the main loop of the A* algorithm.
    While the priority queue is not empty, the algorithm pops the vertex with the lowest f value from the queue (minVertex), and checks its neighboring vertices.
    For each neighboring vertex nextVertex, the algorithm computes a tentative distance from the starting vertex s to nextVertex (`minVertex->dist + e
    */
    while (!queue.empty())
    {
        Vertex *minVertex = queue.top();
        // The vertex with the minimum f value is obtained from the priority queue.

        queue.pop(); // The minimum vertex is removed from the priority queue.

        // loop size is the number of adjacent vertices of the minimum vertex.
        for (int i = 0; i < adj[minVertex->id].size(); ++i)
        {
            Edge e = adj[minVertex->id][i];
            // The neighboring vertex is obtained from the adjacency list of the minimum vertex.
            Vertex *nextVertex = &vertexes[e.tid]; // The neighboring vertex is obtained from the adjacency list of the minimum vertex.
            if (minVertex->dist + e.w < nextVertex->dist)
            { // If the tentative distance is less than the current distance, the distance and f value of the neighboring vertex are updated, and the predecessor
            of the neighboring vertex is set to the minimum vertex.

                nextVertex->dist = minVertex->dist + e.w; // the distance to the next vertex is updated

                // update the f value of the next vertex according to the heuristic function
                nextVertex->f = nextVertex->dist + hManhattan(*nextVertex, vertexes[t]);

                // f = g + h The f value of the next vertex is updated as the sum of the distance to the next vertex and the heuristic estimate of the remaining
                distance to the destination vertex.

                predecessor[nextVertex->id] = minVertex->id;
                // The predecessor of the next vertex is set to the current minimum vertex.

                // If the next vertex is already in the queue, it is pushed again with the updated f value
                if (inqueue[nextVertex->id])
                {
                    // The next vertex is pushed onto the priority queue.
                    queue.push(nextVertex);
                }
                else
                {
                    queue.push(nextVertex);
                    inqueue[nextVertex->id] = true; // The next vertex is marked as inqueue.
                }
            }
            if (nextVertex->id == t)
            {
                // If the next vertex is the destination vertex, the priority queue is cleared and the while loop is terminated.
                queue = priority_queue<Vertex *, vector<Vertex *>, decltype(&compare)>(&compare); // clear queue
                break;
            }
        }
    }
    cout << " " << endl;
    cout << "A* search algorithm: " << s;
    // Prints the ID of the source vertex.
    print(s, t, predecessor);
    // Prints the path from the source vertex s to the destination vertex t using the predecessor array.
    delete[] predecessor;
    delete[] inqueue;
}
```

## 4.Time complexity of A* search algorithm:

The time complexity of the A* search algorithm depends on the graph structure and the heuristic function. As per the *astar.cpp* file, in the worst case, the A* algorithm is a heuristic search algorithm that finds the shortest path between two points using a heuristic function to estimate the remaining Manhattan distance to the goal.

The vertex object has multiple properties, and the algorithm updates the '**f**' property according to the starting and ending points.

In the core function of the A* search algorithm, it maintains a priority queue of vertices to visit next, starting with the source vertex '**s**'. The algorithm repeatedly selects the vertex with the lowest '**f**' value, which is the sum of the distance from the starting vertex plus the heuristic estimate of the remaining distance to the ending vertex. Every time it checks the number of neighboring vertices and updates the distance and '**f**' value according to the heuristic function if a shorter path is found.

Thus, the time complexity of the A* algorithm is ***O((E+V)logV)***, where **E** is the number of edges in the graph, and **V** is the number of vertices. This is because each edge is visited once, and each vertex is added to and removed from the heap at most once (a boolean check is used to determine whether it is in the heap).

In C++, the time complexity of a priority queue is ***O(n*log(n))*** because the priority queue structure is a binary heap. Elements are stored in a binary tree, and adding and removing items individually requires ***O(n*log(n))*** because both deletion and insertion requires ***O(log n)***. Furthermore, when a new element is added to the priority queue, it is placed at the bottom of the binary tree and then "bubbled up" to the correct position by comparing its priority with its parent node's priority and swapping them if necessary. The height of a binary tree is logarithmic, so typically it is ***n*log(n).***

As a result, maintaining the priority queue requires the number of edges and vertices multiplied by ***log(v)*** due to the graph structure.

## 5. Cost Functions and comparison with Dijkstra

To gain a deeper understanding of the A* function, the report will discuss the cost functions of Dijkstra and A*.

The cost function is **g(x);**

The cost function of A*, **f(x) = g(x) + h(x);**

In simple terms, Dijkstra is a special case of A* when the heuristics are zero, which means that Dijkstra and A* are the same if the **h(x)** in the cost function of A* is zero.

Dijkstra has one cost function, which is the real cost value from the source to each node: **f(x) = g(x)**. It finds all possible routes from the source point by considering only the real cost.

## 5. The advantage of A* search:

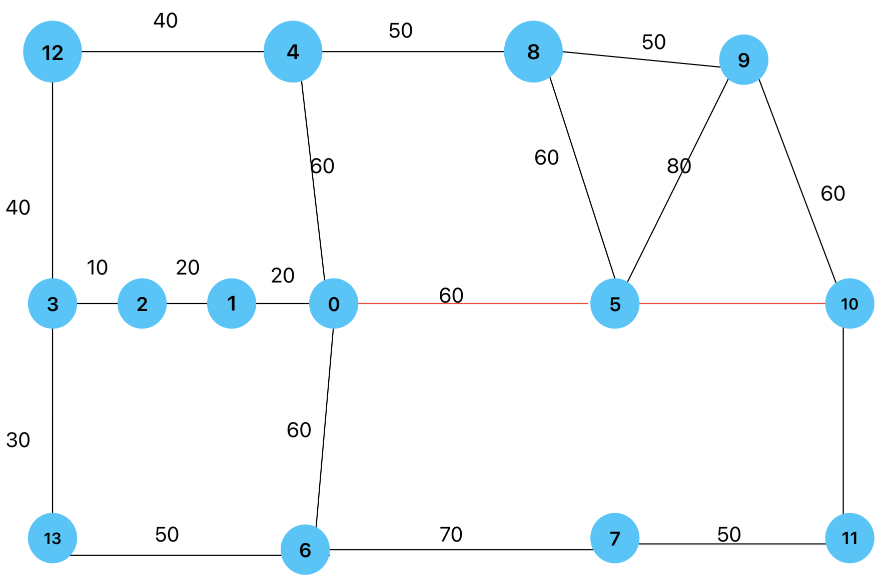
1 **g(x):** is as same as Dijkstra, the real cost from source point to the destination .

2 **h(x):** this is the heuristic function that approximates the cost from node x to the destination node. This heuristic function should not overestimate the cost, which means the real cost for the selected route from node x to the destination will always greater or equal to h(x). In another way, it is called admissible heuristic.

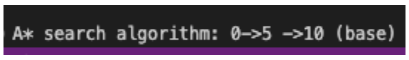
Admissible heuristic is a type of heuristic that estimates distance between current node and goal node. In A* search algorithm, it simplify the whole map into a grid and calculate the manhanthen distance between two nodes.

The A* algorithm prioritises vertices using a combination of the actual distance from the starting point and an estimated distance to the endpoint. This differs from Dijkstra's algorithm, which only considers the actual distance. A* may not always find the shortest path but is generally faster.

To apply A* in games, we can represent the map as a graph with squares as vertices and adjacent squares connected by edges. This allows A* to find a path between two points in a directed weighted graph.



| ID | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| X | 320 | 300 | 280 | 270 | 320 | 360 | 320 | 370 | 350 | 390 | 400 | 420 | 270 | 270 |
| Y | 630 | 630 | 625 | 630 | 700 | 620 | 590 | 580 | 730 | 690 | 620 | 580 | 700 | 590 |



```
A* search algorithm: 0->5 ->10 (base)
```

(A* search found the optimal route from 0 to 10)

## 8. Hill Climbing Algorithm

The Hill Climbing algorithm is a local search optimisation method used to find approximate solutions for optimisation problems. It iteratively refines an initial solution by incrementally modifying it, moving towards an increasing objective function value (or decreasing for minimisation problems).

### 8.1 Overview and Steps

The algorithm involves the following steps:

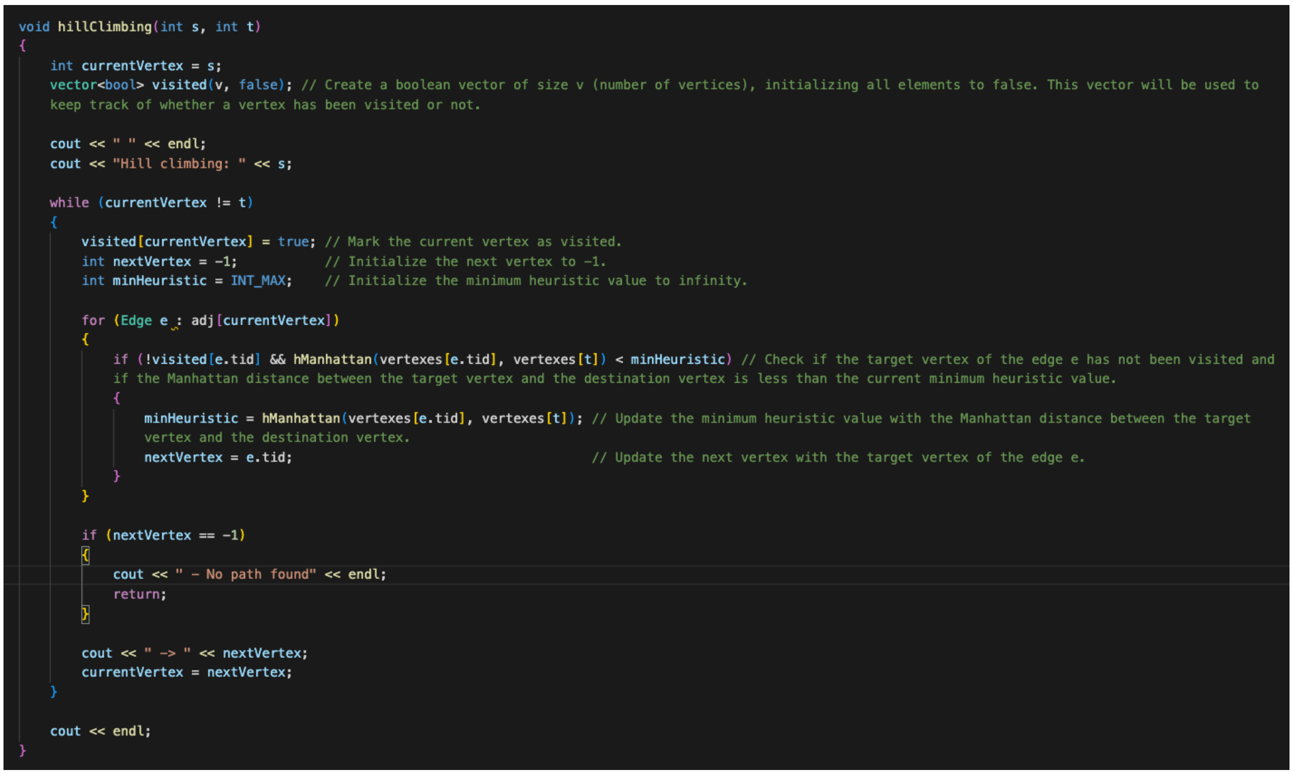
1. Generate a starting solution, either randomly or based on heuristics.
2. Assess the objective function for the current solution.
3. Create a neighbouring solution with minor changes to the current one.
4. Assess the objective function for the neighbouring solution.
5. Compare the neighbouring solution's objective function value with the current solution's value.
    - Update the current solution with the neighbouring solution if it's better.
    - Keep the current solution if the neighbouring solution is worse.
6. Repeat steps **3-5** until reaching a stopping criterion, such as a specific number of iterations, a time limit, or minimal solution improvement. (In the code, iteration stops if found the target vertex)

### 8.2 Key features :

1. Greedy nature: The algorithm makes locally optimal decisions at each step, aiming for a globally optimal solution. However, this can result in being trapped in local optima.
2. Local search: The algorithm examines the current solution's neighbours instead of the entire solution space.
3. Stochastic elements: Variants like Stochastic Hill Climbing or Random-Restart Hill Climbing use randomness to select neighbours or restart the search, helping to escape local optima and improve global optimum chances.

### 8.3 Use cases and Limitations :

Hill Climbing is useful when the solution space is too vast for exhaustive searches and when an approximate solution is acceptable. However, it doesn't guarantee finding the global optimum, and its performance may depend on the initial solution and neighborhood structure choices.



```cpp
void hillClimbing(int s, int t)
{
    int currentVertex = s;
    vector<bool> visited(v, false); // Create a boolean vector of size v (number of vertices), initializing all elements to false. This vector will be used to
    keep track of whether a vertex has been visited or not.

    cout << " " << endl;
    cout << "Hill climbing: " << s;

    while (currentVertex != t)
    {
        visited[currentVertex] = true; // Mark the current vertex as visited.
        int nextVertex = -1;           // Initialize the next vertex to -1.
        int minHeuristic = INT_MAX;    // Initialize the minimum heuristic value to infinity.

        for (Edge e : adj[currentVertex])
        {
            if (!visited[e.tid] && hManhattan(vertexes[e.tid], vertexes[t]) < minHeuristic) // Check if the target vertex of the edge e has not been visited and
            if the Manhattan distance between the target vertex and the destination vertex is less than the current minimum heuristic value.
            {
                minHeuristic = hManhattan(vertexes[e.tid], vertexes[t]); // Update the minimum heuristic value with the Manhattan distance between the target
                vertex and the destination vertex.
                nextVertex = e.tid;                                      // Update the next vertex with the target vertex of the edge e.
            }
        }

        if (nextVertex == -1)
        {
            cout << " - No path found" << endl;
            return;
        }

        cout << " -> " << nextVertex;
        currentVertex = nextVertex;
    }

    cout << endl;
}
```

## 9. Comparison between A*search algorithm and Hill climbing algorithm:

### 9.1 Time and Space complexities of Hill Climbing

Hill Climbing Algorithm: Time Complexity: In the given code, the main loop of the Hill Climbing algorithm runs until the current vertex is the target vertex. In the worst case, the algorithm will visit all vertices, resulting in a time complexity of ***O(V)*,** where n is the number of vertices. Inside the loop, there is another loop that iterates over the adjacent vertices. Since the total number of edges in a graph is ***O(E)***, the overall time complexity is ***O(E + V)***.

Space Complexity: The space complexity of the Hill Climbing algorithm in the given code consists of the visited vector, which has ***O(V)*** elements. Therefore, the space complexity is ***O(V)***.

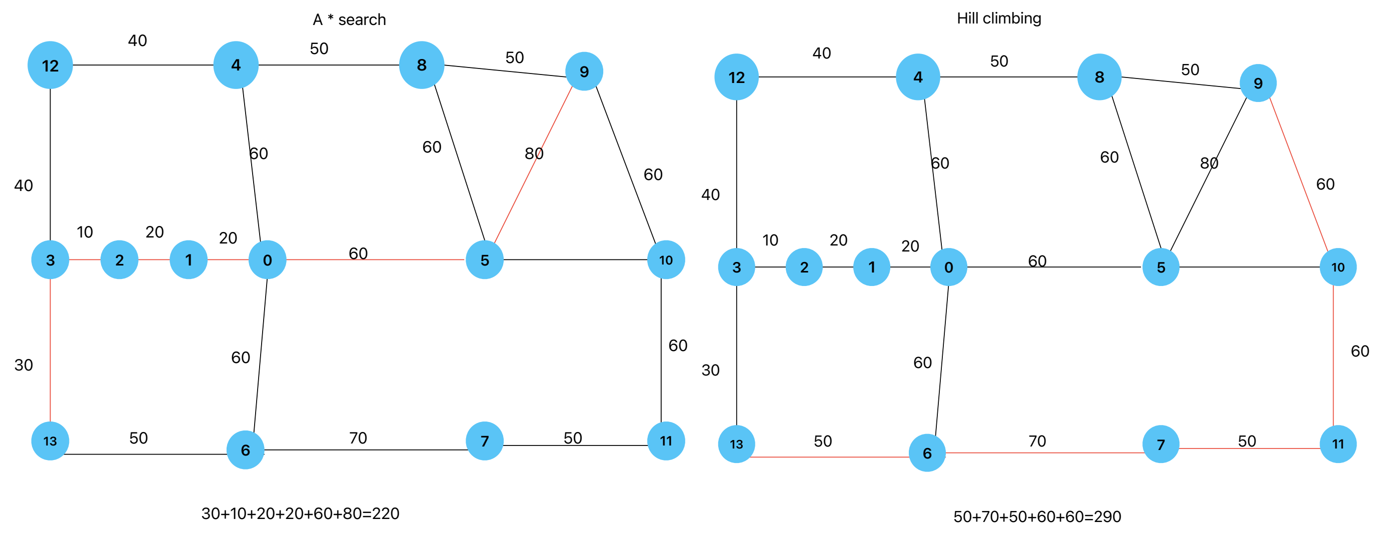
Comparison: In the given code, the A* algorithm typically has a higher time complexity ***O((E+V)logV)***, and space complexity ***O(V+E)*** compared to the Hill Climbing algorithm (time complexity: ***O(E+V)***, space complexity: ***O(V)***). However, the A* algorithm is guaranteed to find the optimal solution, assuming an admissible and consistent heuristic is used. The Hill Climbing algorithm is faster and has a lower space complexity, but it may not always find the optimal solution since it is a greedy local search algorithm and can get stuck in local optima.

In conclusion, if finding the optimal solution is crucial, the A* algorithm in the given code is the better choice, but if the focus is on speed and lower space complexity, the Hill Climbing algorithm in the given code might be preferred, with the caveat that it may not always find the optimal solution.

## 9.2 Performance in Complex Graphs

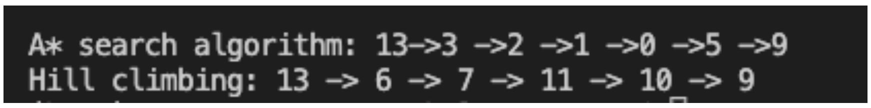
Comparing the two algorithms, the A* search algorithm is generally more efficient in finding the optimal path. It guarantees finding the shortest path (if it exists), while Hill Climbing can sometimes get stuck in local minima and fail to find the optimal path. However, the A* search algorithm has a higher time complexity due to its use of a priority queue.

For more complex or partially seen unseen graphs, the A* algorithm is likely to perform better when using a suitable heuristic function. The Hill Climbing algorithm may fail to find the optimal path in these situations, as it doesn't consider the global structure of the graph and is more prone to getting stuck in local minima.



*(Left graph shows the solution of A* search, right graph shows the solution of hill climbing search. A* solution distance 220, hill climbing is 290)*

| ID | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| X | 320 | 300 | 280 | 270 | 320 | 360 | 320 | 370 | 350 | 390 | 400 | 420 | 270 | 270 |
| Y | 630 | 630 | 625 | 630 | 700 | 620 | 590 | 580 | 730 | 690 | 620 | 580 | 700 | 590 |



```
A* search algorithm: 13->3 ->2 ->1 ->0 ->5 ->9
Hill climbing: 13 -> 6 -> 7 -> 11 -> 10 -> 9
```

In summary, the A* search algorithm has a higher time complexity but guarantees finding the optimal path, whereas the Hill Climbing algorithm has a lower time complexity but may not always find the optimal path. For complex graphs, the A* algorithm is generally a better choice, provided an appropriate heuristic function is used.

## 9.3 Example Comparison

In the given example, the goal is to find the shortest path from node **13** to node **9** in a known environment graph. The A* algorithm finds a path with a total distance of **220**, while the Hill Climbing algorithm finds a path with a total distance of **290**.

In this test, the A* search algorithm proves to be more efficient than Hill Climbing. This is because the A* search's heuristic function calculates not only the Manhattan distance between the next node and the target node but also the accumulated distance between the start node and the target node. On the other hand, Hill Climbing only considers the distance between the next node and the target node, which is insufficient for obtaining the best result. As a result, the A* algorithm finds a better route in this test, and similar outcomes are observed in other tests as well. However, the evaluation is limited by the size and structure of the graph. Partially observable or unobservable environment is required for further experimentation.

## Conclusion:

In conclusion, the A* search algorithm and the Hill Climbing algorithm each have their unique advantages and disadvantages when it comes to finding the shortest path in graphs. The A* search algorithm is more efficient in finding the optimal path, even in complex graphs, thanks to its heuristic function, which takes into account both the Manhattan distance and the accumulated distance from the start node. However, it comes with a higher time complexity compared to the Hill Climbing algorithm.

On the other hand, the Hill Climbing algorithm has a lower time complexity, but its greedy nature and reliance on local search might lead it to get stuck in local minima and fail to find the optimal path. It is essential to consider the specific problem context and the requirements of the application when deciding which algorithm to use. In general, the A* search algorithm is preferable for complex graphs and when an optimal path is desired, as long as an appropriate heuristic function is employed.

## Appendix:

*Hill climbing https://en.wikipedia.org/wiki/Hill_climbing*
*A* search algorithm https://en.wikipedia.org/wiki/A*_search_algorithm*