A Star Search Algorithm

The A* search algorithm is a widely popular algorithm used in pathfinding and graph traversal. It is a combination of a shortest path algorithm and a heuristic for evaluating nodes. A* uses a best-first search approach and is guaranteed to find the shortest possible path in any weighted graph, given certain conditions.

21k-3211 Shayan Haider

21k-3297 Hasan Iqbal

21k-3385 Rafed Naeem

Introduction to A* Algorithm

The A* algorithm is essentially a smarter version of Dijkstra's shortest path algorithm. It was developed in 1968 by Peter Hart, Nils Nilsson, and Bertram Raphael of the Stanford Research Institute. A* uses a heuristic function to guide the search towards the goal node. Unlike Dijkstra's algorithm, A* considers both the actual cost from the start node and the estimated cost to the goal node when evaluating paths.

Heuristics in A* Algorithm

Admissible Heuristics

Admissible heuristics are those that never overestimate the cost to reach the goal node. They are essential to ensure the optimality of the A* algorithm.

> Example: The Manhattan distance heuristic is admissible in a grid-like environment.

Consistent Heuristics

Consistent heuristics are those that satisfy the triangle inequality. They ensure that once a node is expanded, the cost of reaching it will not decrease later on.

> Example: The Euclidean distance heuristic is consistent in a two-dimensional space.

Cost Function in A* Algorithm

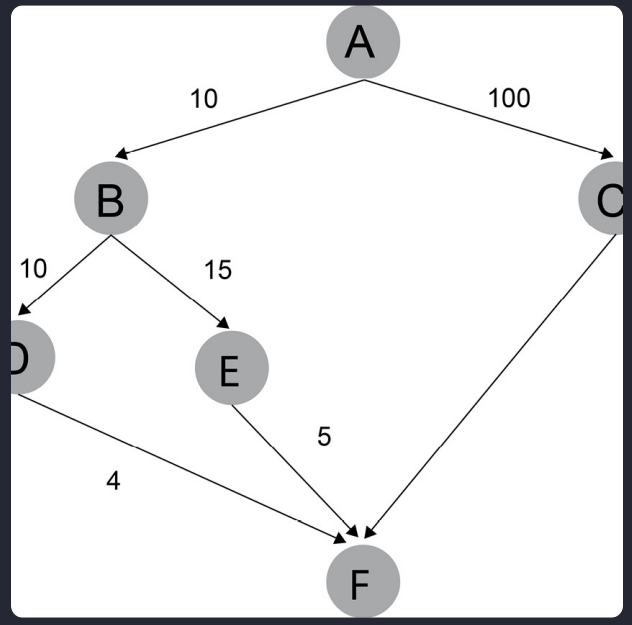
The cost function in the A* algorithm is simply the sum of the actual cost from the start node to that node and the estimated cost to reach the goal node. This cost determines the priority of the next node to expand. A* chooses the next node with the lowest total cost among all nodes that have been evaluated and are yet to be expanded.

Pseudocode for A* Algorithm

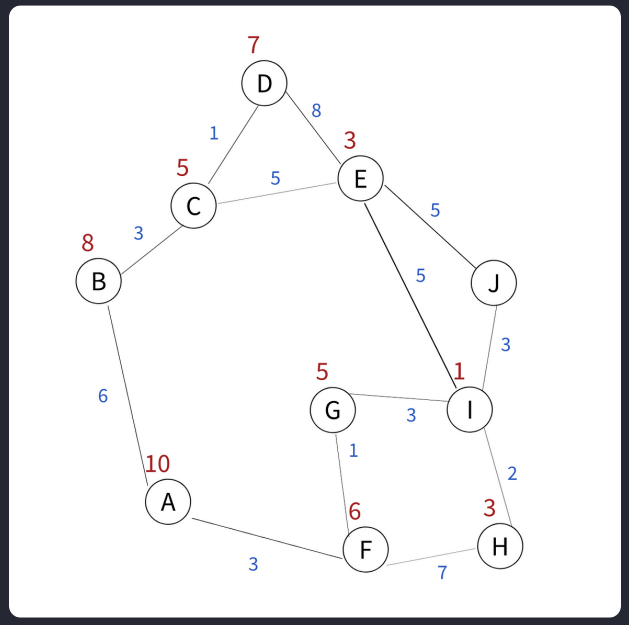
```
Procedure A*(start,goal)
 open = priority queue containing start
 closed = empty set
 while open is not empty
  current = node in open with the lowest cost
  if current = goal
   return path
  for each neighbor of current
   if neighbor in closed
    continue
   if neighbor not in open or new cost to neighbor
    less than old cost then
    set f_cost = g_cost + h_cost
    set parent = current
    if neighbor not in open
     add neighbor to open
  add current to closed
 return None
```

Working of A* Algorithm

The A* algorithm maintains two lists of nodes: open and closed. The open list contains nodes that are not yet evaluated and the closed list contains nodes that have already been evaluated. A* expands the node with the lowest total cost (f-cost), where f-cost = g-cost + h-cost, where g-cost is the actual cost to reach the node from the start node, and h-cost is the estimated cost to reach the goal node from that node.



Example of A* algorithm graph traversal.



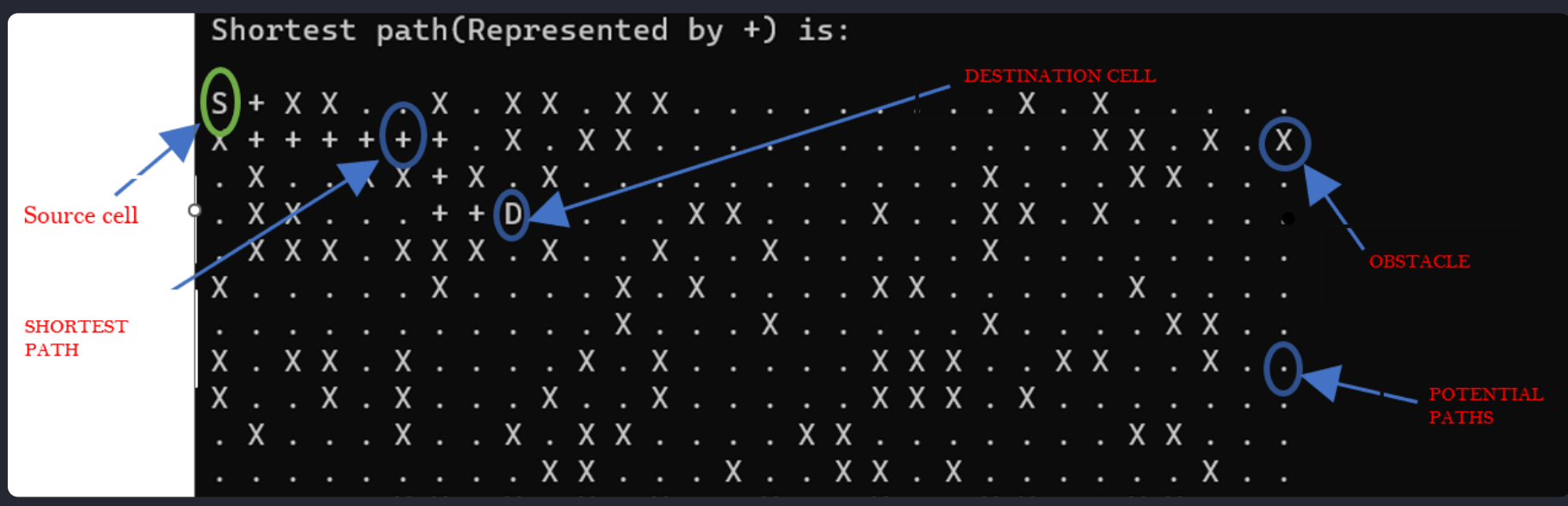
Example of A* algorithm pathfinding.

CODE IMPLEMENTATION:

The given code is an implementation of the A* algorithm for finding the shortest path between two points on a 2D grid. The grid is represented as a 2D array of integers where 0 represents an empty cell and any other integer represents an obstacle. The algorithm finds a path that connects a source cell with a destination cell while avoiding obstacles on the grid.

```
Shortest path(Represented by +) is:

S + X X . . X . X X . X X . . . . . . . . . X . X . . . . .
X + + + + + + . X . X X . . . . . . . . . . . . X X . X . X
. X . . X X + X . X . . . . . . . . . . . . X . . . X X . . .
. X X . . . + + D . . . . X X . . . X . . X X . X . . . . . .
. X X X . X X X . X . . X . . X . . . . . X . . . . . . . . .
X . . . . . X . . . . X . X . . . . X X . . . . . X . . . . .
. . . . . . . . . . . . X . . . X . . . . . X . . . . X X . .
X . X X . X . . . . X . X . . . . . X X X . . X X . . X . . .
X . . X . X . . . X . . X . . . . . X X X . X . . . . . . . .
. X . . . X . . X . X X . . . . X X . . . . . . . . X X . . .
. . . . . . . . . X X . . . X . . X X . X . . . . . . . X . .
. . . . . X X . X . X . X . . X . . X . . X X . . X X . . . .
. X . X . . . . X X . . . X . . . X . . . . . X . . . X X .
. . X . . X . X . . . . . . . X . . X . X . X X . X X X . .
. . X . . . . . X . X . . . X . . X X . . . . . . . X . . X
X . . X X . . X . . X . . . X . . . . X . . . X X . . . . . .
. X . . X . . . . . . . . X . . . X . . X X . . . . . . . . .
. . X . . . X . . X . . . X X . . X . . . X . X . . . X . . .
. . X . . . . X . X . . . X . . X X . . . . . . . X . . X . .
X . . . X X . X X . X X . . X . . . X . . . X X X . . X . . .
X X X . . . . X X . X X . . . . X X . . . . . X X . . X . . .
```



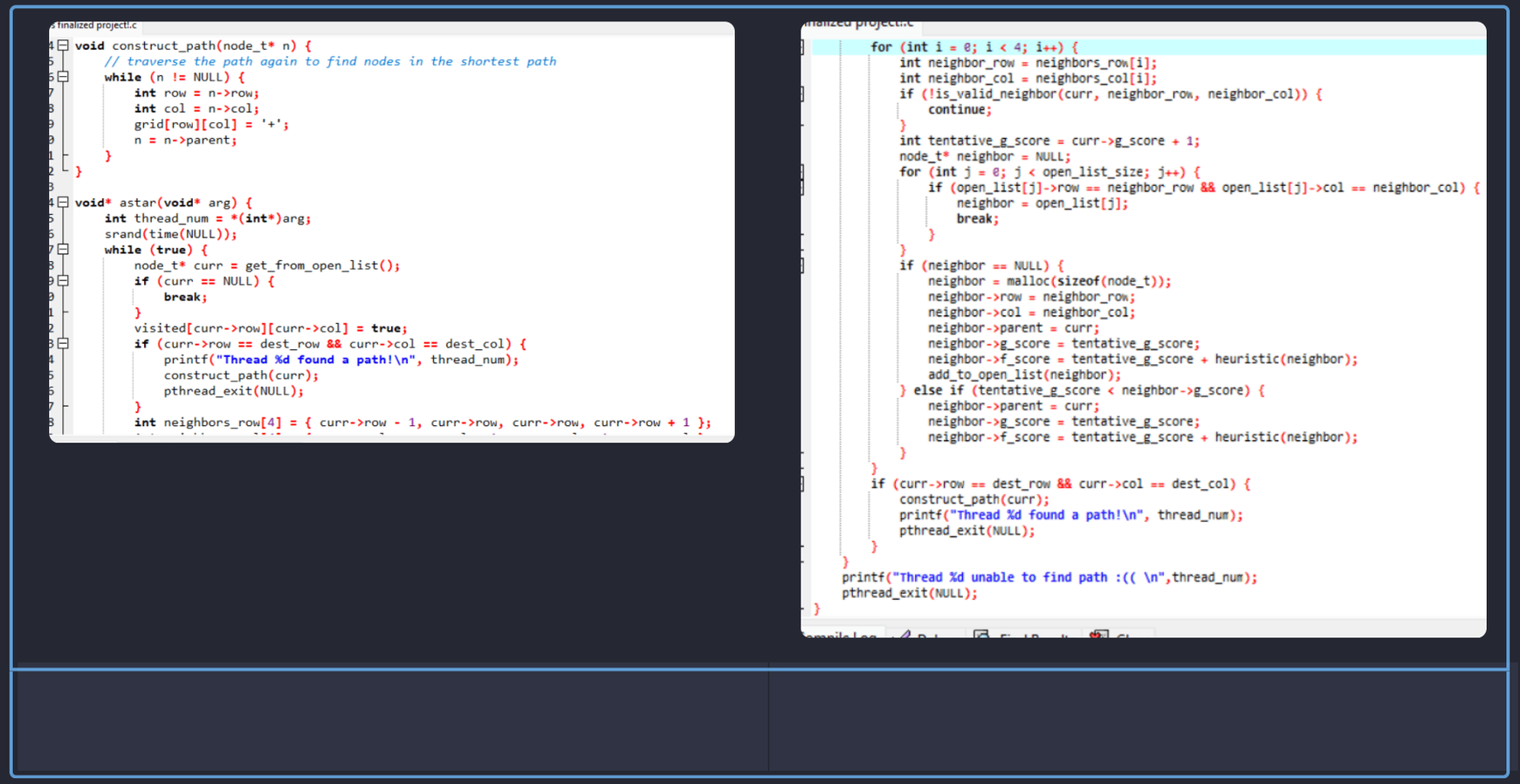
The implementation uses a struct node_t to represent a cell on the grid. Each node has a row and column number, a parent pointer, and two scores - g_score and f_score. The g_score represents the cost of reaching a node from the source node, while f_score is the sum of g_score and the estimated cost of reaching the destination node from the current node. The estimated cost is calculated using the Manhattan distance heuristic.

```
typedef struct node {
    int row;
    int col;
    int f_score;
    int g_score;
    struct node* parent;
} node_t;
```

The implementation uses a priority queue to store the nodes that are yet to be explored. The priority queue is implemented using an array open_list and a variable open_list_size. The array stores the pointers to the nodes, and the open_list_size keeps track of the number of nodes in the queue.

```
node_t* open_list[MAX_NODES];
int open_list_size = 0;
```

The algorithm runs in multiple threads. The astar function is the entry point for each thread. Each thread repeatedly removes the node with the lowest f_score value from the priority queue, expands it, and adds its neighbors to the priority queue. If the destination node is found, the function construct_path is called to mark the shortest path on the grid. If no path is found, the thread exits.



```c
void construct_path(node_t* n) {
    // traverse the path again to find nodes in the shortest path
    while (n != NULL) {
        int row = n->row;
        int col = n->col;
        grid[row][col] = '+';
        n = n->parent;
    }
}

void* astar(void* arg) {
    int thread_num = *(int*)arg;
    srand(time(NULL));
    while (true) {
        node_t* curr = get_from_open_list();
        if (curr == NULL) {
            break;
        }
        visited[curr->row][curr->col] = true;
        if (curr->row == dest_row && curr->col == dest_col) {
            printf("Thread %d found a path!\n", thread_num);
            construct_path(curr);
            pthread_exit(NULL);
        }
        int neighbors_row[4] = { curr->row - 1, curr->row, curr->row, curr->row + 1 };
```

```c
        for (int i = 0; i < 4; i++) {
            int neighbor_row = neighbors_row[i];
            int neighbor_col = neighbors_col[i];
            if (!is_valid_neighbor(curr, neighbor_row, neighbor_col)) {
                continue;
            }
            int tentative_g_score = curr->g_score + 1;
            node_t* neighbor = NULL;
            for (int j = 0; j < open_list_size; j++) {
                if (open_list[j]->row == neighbor_row && open_list[j]->col == neighbor_col) {
                    neighbor = open_list[j];
                    break;
                }
            }
            if (neighbor == NULL) {
                neighbor = malloc(sizeof(node_t));
                neighbor->row = neighbor_row;
                neighbor->col = neighbor_col;
                neighbor->parent = curr;
                neighbor->g_score = tentative_g_score;
                neighbor->f_score = tentative_g_score + heuristic(neighbor);
                add_to_open_list(neighbor);
            } else if (tentative_g_score < neighbor->g_score) {
                neighbor->parent = curr;
                neighbor->g_score = tentative_g_score;
                neighbor->f_score = tentative_g_score + heuristic(neighbor);
            }
        }
        if (curr->row == dest_row && curr->col == dest_col) {
            construct_path(curr);
            printf("Thread %d found a path!\n", thread_num);
            pthread_exit(NULL);
        }
    }
    printf("Thread %d unable to find path :(( \n",thread_num);
    pthread_exit(NULL);
}
```

The implementation uses a mutex lock to protect the priority queue from race conditions caused by multiple threads accessing it concurrently.

```
    pthread_mutex_unlock(&lock);
}

node_t* get_from_open_list() {
    pthread_mutex_lock(&lock);

    if (open_list_size <= 0) {
        pthread_mutex_unlock(&lock);
        return NULL;
    }

    int min_f_score = open_list[0]->f_score;
    int min_index = 0;

    for (int i = 1; i < open_list_size; i++) {
        if (open_list[i]->f_score < min_f_score) {
            min_f_score = open_list[i]->f_score;
            min_index = i;
        }
    }

    node_t* result = open_list[min_index];

    for (int i = min_index + 1; i < open_list_size; i++) {
        open_list[i - 1] = open_list[i];
    }

    open_list_size--;

    pthread_mutex_unlock(&lock);
```

1. `int heuristic(node_t* n)`: This function calculates the heuristic value of the given node `n` using the Manhattan distance. The Manhattan distance is the sum of the absolute differences between the row and column coordinates of the current node and the destination node. The function returns the heuristic value.
2. `bool is_valid_node(int row, int col)`: This function checks if the given coordinates `(row, col)` represent a valid node on the grid. The function returns `true` if the coordinates are within the grid boundaries, the node is either empty or it's the source or destination node, and the node hasn't been visited before.
3. `bool is_valid_neighbor(node_t* curr, int row, int col)`: This function checks if the given coordinates `(row, col)` represent a valid neighbor of the current node `curr`. The function returns `true` if the neighbor is a valid node and is not the same as the current node `curr`.

```
int heuristic(node_t* n) {
    return abs(n->row - dest_row) + abs(n->col - dest_col);
}

bool is_valid_node(int row, int col) {
    return row >= 0 && row < ROWS && col >= 0 && col < COLS &&
           (grid[row][col] == 0 || (row == source_row && col == source_col) || (row == dest_row && col == dest_col)) &&
           !visited[row][col];
}


bool is_valid_neighbor(node_t* curr, int row, int col) {
    if (row == curr->row && col == curr->col) {
        return false;
    }

    if (!is_valid_node(row, col)) {
        return false;
    }

    if (abs(curr->row - row) + abs(curr->col - col) > 1) {
        return false;
    }
    //visited[row][col] = 1; // Mark the visited cell with 1
    return true;
}
```

Applications of A* Algorithm

Video Games

A* is frequently used in game development for pathfinding and is particularly popular in real-time strategy games and first-person shooters.

1

Robotics

A* is commonly used in robotics for obstacle avoidance, path planning, and localization.

2

3

Geographical Information Systems

A* is used in GIS for route planning, location-based services, and spatial analysis.

Conclusion

The A* algorithm has proven to be an efficient and effective algorithm for pathfinding and graph traversal. Its success lies in the use of heuristics to guide the search towards the goal node, making it faster and more efficient than other traditional search algorithms.