A* Algorithm Implementation Cheatsheet

Overview

A* algorithm is an informed search algorithm that finds the shortest path between two given nodes on a graph. It uses a heuristic function to estimate the cost of reaching the goal from the current node. A* algorithm is guaranteed to find the shortest path if the heuristic is admissible and consistent.

Implementation

1. Define the heuristic function that estimates the cost of reaching the goal from the current node.
2. Define the data structures to be used: `open_list`, `closed_list`, `g_score`, `h_score`, and `f_score`.
3. Add the starting node to the `open_list` with an initial `g_score` of 0 and `h_score` calculated using the heuristic function.
4. While the `open_list` is not empty, select the node with the lowest `f_score` and remove it from `open_list`.
5. If the selected node is the goal, return the path.
6. Add the selected node to the `closed_list`.
7. For each neighboring node of the selected node, calculate its tentative `g_score` and `h_score` using the heuristic function.
8. If the neighboring node is already in the `closed_list`, skip it.
9. If the neighboring node is not in the `open_list`, add it to the `open_list`.
10. If the neighboring node is already in the `open_list` and the tentative `g_score` is greater than or equal to its current `g_score`, skip it. Otherwise, update the `g_score`, `h_score`, and `f_score` of the neighboring node and add it to the `open_list`.
11. Repeat steps 4-10 until the `open_list` is empty.

Python Example

```
import heapq

def heuristic(a, b):
    return abs(b[0] - a[0]) + abs(b[1] - a[1])

def astar(array, start, goal):
    neighbors = [(0,1),(0,-1),(1,0),(-1,0),(1,1),(1,-1),(-1,1),(-1,-1)]

    close_set = set()
    came_from = {}
    gscore = {start:0}
    fscore = {start:heuristic(start, goal)}
    oheap = []

    heapq.heappush(oheap, (fscore[start], start))

    while oheap:
        current = heapq.heappop(oheap)[1]
        if current == goal:
```

```
            data = []
            while current in came_from:
                data.append(current)
                current = came_from[current]
            return data
        close_set.add(current)
        for i, j in neighbors:
            neighbor = current[0] + i, current[1] + j
            tentative_g_score = gscore[current] + heuristic(current, neighbor)
            if 0 <= neighbor[0] < array.shape[0]:
                if 0 <= neighbor[1] < array.shape[1]:
                    if array[neighbor[0]][neighbor[1]] == 1:
                        continue
                else:
                    continue
            else:
                continue

            if neighbor in close_set and tentative_g_score >= gscore.get(neighbor,
0):
                continue

            if  tentative_g_score < gscore.get(neighbor, 0) or neighbor not in
[i[1]for i in oheap]:
                came_from[neighbor] = current
                gscore[neighbor] = tentative_g_score
                fscore[neighbor] = tentative_g_score + heuristic(neighbor, goal)
                heapq.heappush(oheap, (fscore[neighbor], neighbor))

    return None
```

C++ Example

```
#include <iostream>
#include <queue>
#include <vector>
#include <functional>
#include <utility>
#include <cmath>
#include <unordered_map>
#include <set>

using namespace std;

typedef pair<int, int> pii;

const int INF = 1e9;

int heuristic(pii a, pii b) {
    return abs(b.first - a.first) + abs(b.second - a.second);
```

```
}

vector<pii> astar(vector<vector<int>>& grid, pii start, pii goal) {
    int n = grid.size(), m = grid[0].size();
    vector<pii> dirs = {{0,1}, {0,-1}, {1,0}, {-1,0}, {1,1}, {1,-1}, {-1,1},
{-1,-1}};
    priority_queue<pii, vector<pii>, greater<pii>> pq;
    unordered_map<int, unordered_map<int, int>> gscore;
    unordered_map<int, unordered_map<int, int>> fscore;
    unordered_map<int, unordered_map<int, pii>> came_from;
    set<pii> closed_set;

    gscore[start.first][start.second] = 0;
    fscore[start.first][start.second] = heuristic(start, goal);
    pq.push({fscore[start.first][start.second], (start.first << 16) |
start.second});

    while (!pq.empty()) {
        auto curr = pq.top(); pq.pop();
        int x = curr.second >> 16, y = curr.second & 0xFFFF;
        if (make_pair(x, y) == goal) {
            vector<pii> path;
            while (x != start.first || y != start.second) {
                path.push_back({x, y});
                int tmp_x = x, tmp_y = y;
                x = came_from[tmp_x][tmp_y].first;
                y = came_from[tmp_x][tmp_y].second;
            }
            path.push_back({x, y});
            reverse(path.begin(), path.end());
            return path;
        }
        closed_set.insert({x, y});
        for (auto dir : dirs) {
            int nx = x + dir.first, ny = y + dir.second;
            if (nx < 0 || nx >= n || ny < 0 || ny >= m || grid[nx][ny] == 1)
continue;
            if (closed_set.count({nx, ny})) continue;
            int tentative_gscore = gscore[x][y] + heuristic({x, y}, {nx, ny});
            if (!gscore.count(nx) || !gscore[nx].count(ny) || tentative_gscore <
gscore[nx][ny]) {
                came_from[nx][ny] = {x, y};
                gscore[nx][ny] = tentative_gscore;
                fscore[nx][ny] = tentative_gscore + heuristic({nx, ny}, goal);
                pq.push({fscore[nx][ny], (nx << 16) | ny});
            }
        }
    }
    return {};
}

int main() {
```

```
    vector<vector<int>> grid = {
        {0, 0, 0, 0, 0},
        {0, 1, 1, 1, 0},
        {0, 1, 0, 0, 0},
        {0, 1, 0, 1, 0},
        {0, 0, 0, 1, 0}
    };
    pii start = {0, 0}, goal = {4, 4};
    auto path = astar(grid, start, goal);
    for (auto p : path) {
        cout << "(" << p.first << ", " << p.second << ") ";
    }
    cout << endl;
    return 0;
}
```

Resources

- A* Search Algorithm: Wikipedia page on A* algorithm
- Introduction to A* Pathfinding: Red Blob Games tutorial on A* algorithm
- A* Search Algorithm: GeeksforGeeks tutorial on A* algorithm
- Pathfinding with A*: Amit's A* Pages with detailed explanations and interactive examples
- A* Pathfinding for Beginners: AI Junkie tutorial on A* algorithm with code samples
- A* Pathfinding Visualization Tool: Pathfinding.js Visualizer for A* algorithm
- A* Pathfinding for Beginners - Introduction to Pathfinding: YouTube video tutorial on A* algorithm by Sebastian Lague