Peter the Great St. Petersburg **Polytechnic University**

Institute of Materials, Mechanical Engineering and Transport

Subject: Object oriented programming

Course work on the topic:

«Best cost path algorithm - A*»

Student group 3331506/80401 H.T. Tsvetkova

Teacher E.M. Kuznetsova

« » 2021 г.

St. Petersburg

2021 year

Table of contents

1. Introducing

Our motivation of uses path-finding algorithms is to approximate the **shortest path** in real-life situations, like- in maps, games where there can be many hindrances.

With, for example, **Breadth First Search** or **Dijkstra's Algorithm**, the search area frontier expands in all directions during algorithm. This is a reasonable choice in situations when you are trying to find an efficient path to all locations or to many locations. However, a common case is to find a path to only one location. That is why we use **A* algorithm**.

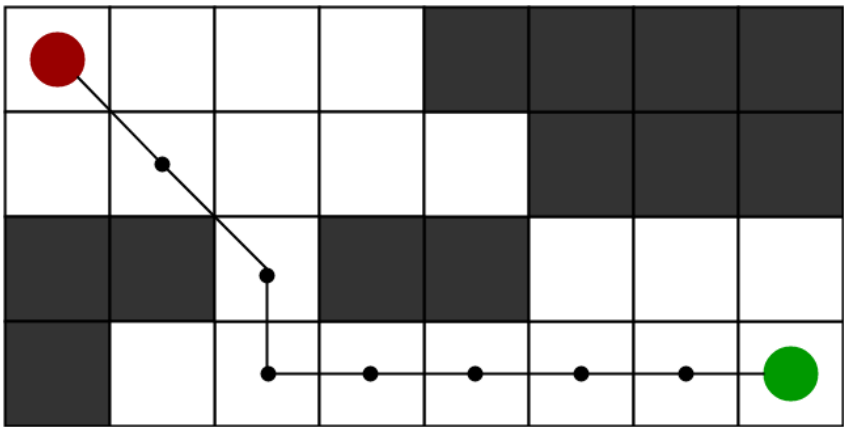
Peter Hart, Nils Nilsson and Bertram Raphael of Stanford Research Institute (now SRI International) first published the algorithm in 1968. It can be seen as an extension of Dijkstra's algorithm. A* achieves better performance by using heuristics to guide its search.

A* Search algorithm is one of the best and popular technique used in pathfinding and graph traversals.

2. The idea of algorithm

A* is a modification of Dijkstra's Algorithm that is optimized for a single destination. Dijkstra's Algorithm can find paths to all locations; A* finds paths to **one location**, or the closest of several locations. It prioritizes paths that seem to be leading closer to a goal.

Let us consider a square grid having many obstacles and we are given a starting cell and a target cell. We want to reach the target cell from the starting cell as quickly as possible. At the picture 1 we can see an easy example of such grid, source-starting cell and the destination cell.



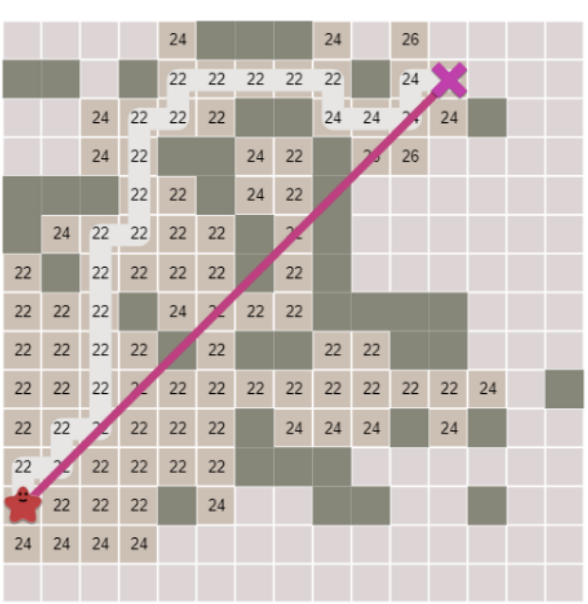
Picture 1 – Given grid, source cell, target cell.

The A* algorithm uses a **heuristic function** to help decide which path to follow next. The heuristic function provides an estimate of the minimum cost between a given node and the target node. At each step we will pick the node according to a value - F which is a parameter equal to the sum of two other parameters – G and H. Algorithm picks the node/cell having the lowest F, and process that node/cell.

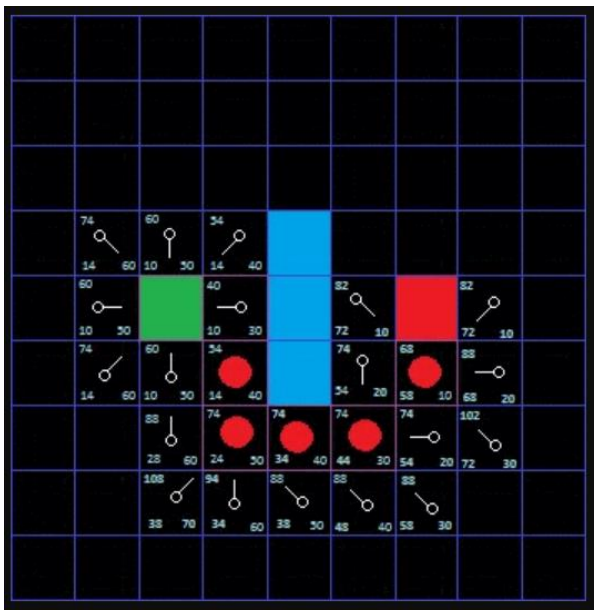
G - the movement cost to move from the starting point to a given square on the grid.

H - the estimated movement cost to move from that given square on the grid to the destination. This is often referred to as the heuristic.

At the picture 2 we can see an example of process of pathfinding with choosing the nodes according to the F value. The F value for each cell/node is represented in the squares. At the picture 3 we can see the same process, but at the squares we can see also information about G and H values.



Picture 2 – pathfinding using heuristic function (value F)

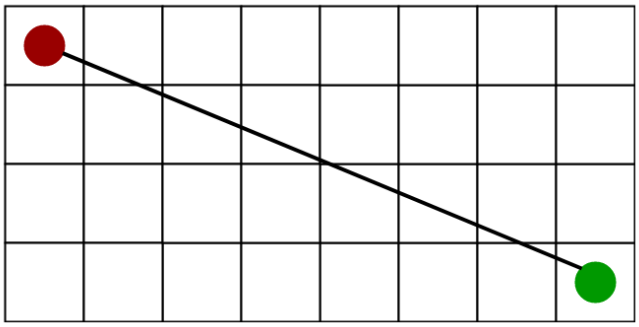


Picture 3 – pathfinding using heuristic function (F = G+H)

We can calculate heuristic value as:

$$H = \sqrt{(curX - goalX)^2 + (curY - goalY)^2}$$

The Euclidean Distance Heuristics is shown at the picture 4 below (assume red spot as source cell and green spot as target cell).



Picture 4 - Euclidean Distance Heuristics

3. The effectiveness of algorithm

The time complexity of A* depends on the heuristic. The time complexity is polynomial when the search space is a tree, there is a single goal state, and the heuristic function h meets the following condition:

$$|h(x) - h^*(x)| = O(\log h^*(x))$$

where h* is the optimal heuristic, the exact cost to get from x to the goal. In other words, the error of h will not grow faster than the logarithm of the "perfect heuristic" h* that returns the true distance from x to the goal.

The space complexity of A* is roughly the same as that of all other graph search algorithms, as it keeps all generated nodes in memory. In the worst case of an unbounded search space, the number of nodes expanded is exponential in the depth of the solution (the shortest path) d: O(b^d), where b is the branching factor (the average number of successors per state). This assumes that a goal state exists at all, and is reachable from the start state; if it is not, and the state space is infinite, the algorithm will not terminate.

4. Results

We can use any data structure to implement open list and closed list but for best performance we use a set data structure of C++ STL (implemented as Red-Black Tree) and a bool hash table for a closed list. At the picture 5 we can see the solution of optimal pathfinding using different algorithms.

Picture 5 – the solution by different methods

When Greedy Best-First Search finds the right answer, A* finds it too, exploring the same area. When Greedy Best-First Search finds the wrong answer (longer path), A* finds the right answer, like Dijkstra's Algorithm does, but still explores less than Dijkstra's Algorithm does.

A* is the best of both worlds. As long as the heuristic does not overestimate distances, A* finds an optimal path, like Dijkstra's Algorithm does. A* uses the heuristic to reorder the nodes so that it's more likely that the goal node will be encountered sooner.

5. List of references

1. https://www.geeksforgeeks.org/a-search-algorithm/
2. https://www.redblobgames.com/pathfinding/a-star/introduction.html
3. https://en.wikipedia.org/wiki/A*_search_algorithm
4. Седжвик, Р. Фундаментальные алгоритмы на C++. Алгоритмы на графах: пер. с англ./Роберт Седжвик. - СПб: 2002. – 496 с.

6. Software implementation

```
void aStarSearch(int grid[][COL], coordinates src, coordinates dest) {
  if (!is_valid(src.first, src.second)) {
    cout << "Source is invalid\n";
    return;
  }
  if (!is_valid(dest.first, dest.second)) {
    cout << "Destination is invalid\n";
    return;
  }
  if (!is_unblocked(grid, src.first, src.second)
    || !is_unblocked(grid, dest.first, dest.second)) {
    cout << "Source or the destination is blocked\n";
    return;
  }
  if (is_destination(src.first, src.second, dest)) {
    cout << "We are already at the destination\n";
    return;
  }

  bool closed_list[ROW][COL] = {false};
  set <pPair> open_list;
  cell cells_grid[ROW][COL];

  int x, y;
  x = src.first, y = src.second;
  cells_grid[x][y].set_fgh(0.0, 0.0, 0.0);
  cells_grid[x][y].set_parents(x, y);
  open_list.insert(make_pair(0.0, make_pair(x, y)));

  while (!open_list.empty()) {

    pPair cur_point = *open_list.begin();
    open_list.erase(open_list.begin());
    x = cur_point.second.first;
    y = cur_point.second.second;
    closed_list[x][y] = true;

    double gNew = 0.0;
    double hNew = 0.0;
    double fNew = 0.0;
    double step = 0.0;

    for(int x_step = -1; x_step <= 1; x_step++) {
      for(int y_step = -1; y_step <= 1; y_step++) {

        if(x_step == 0 && y_step == 0) {
          continue; // the current position
        }
        if(closed_list[x + x_step][y + y_step]) {
          continue; // already in the closed_list
        }
        if(!is_valid(x + x_step, y + y_step) || !is_unblocked(grid, x +
x_step, y + y_step)) {
          continue; // not valid or blocked position
        }
        if (is_destination(x + x_step, y + y_step, dest)) {
          cells_grid[x + x_step][y + y_step].set_parents(x, y);
          cout << "The destination cell is found\n";
          trace_path(cells_grid, dest);
          return;
        }
```

```
        if(abs(x_step) == abs(y_step)) {
          step = 1.414; // diagonal
        } else step = 1.0; // straight

        gNew = cells_grid[x][y].get_g() + step;
        hNew = calculate_H(x + x_step, y + y_step, dest);
        fNew = gNew + hNew;

        if (cells_grid[x + x_step][y + y_step].get_f() == FLT_MAX
        || cells_grid[x + x_step][y + y_step].get_f() > fNew) {
          open_list.insert(make_pair(fNew, make_pair(x + x_step, y +
y_step)));
          cells_grid[x + x_step][y + y_step].set_fgh(fNew, gNew, hNew);
          cells_grid[x + x_step][y + y_step].set_parents(x, y);
        }
      }
    }
  }
  cout << "Failed to find the Destination Cell\n";
}
```