# hjweide / a-star

Branch: master ▾ **a-star** / **astar.cpp** Find file Copy path

**hjweide** Fix spacing. 805df86 21 days ago

**2** contributors

117 lines (95 sloc) 3.61 KB

```cpp
#include <queue>
#include <limits>
#include <cmath>

// represents a single pixel
class Node {
  public:
    int idx;     // index in the flattened grid
    float cost;  // cost of traversing this pixel

    Node(int i, float c) : idx(i),cost(c) {}
};

// the top of the priority queue is the greatest element by default,
// but we want the smallest, so flip the sign
bool operator<(const Node &n1, const Node &n2) {
  return n1.cost > n2.cost;
}

bool operator==(const Node &n1, const Node &n2) {
  return n1.idx == n2.idx;
}

// See for various grid heuristics:
```

```cpp
// http://theory.stanford.edu/~amitp/GameProgramming/Heuristics.html#S7
// L_\inf norm (diagonal distance)
float linf_norm(int i0, int j0, int i1, int j1) {
  return std::max(std::abs(i0 - i1), std::abs(j0 - j1));
}

// L_1 norm (manhattan distance)
float l1_norm(int i0, int j0, int i1, int j1) {
  return std::abs(i0 - i1) + std::abs(j0 - j1);
}

// weights:        flattened h x w grid of costs
// h, w:           height and width of grid
// start, goal:    index of start/goal in flattened grid
// diag_ok:        if true, allows diagonal moves (8-conn.)
// paths (output): for each node, stores previous node in path
extern "C" bool astar(
      const float* weights, const int h, const int w,
      const int start, const int goal, bool diag_ok,
      int* paths) {

  const float INF = std::numeric_limits<float>::infinity();

  Node start_node(start, 0.);
  Node goal_node(goal, 0.);

  float* costs = new float[h * w];
  for (int i = 0; i < h * w; ++i)
    costs[i] = INF;
  costs[start] = 0.;

  std::priority_queue<Node> nodes_to_visit;
  nodes_to_visit.push(start_node);

  int* nbrs = new int[8];

  bool solution_found = false;
  while (!nodes_to_visit.empty()) {
    // .top() doesn't actually remove the node
    Node cur = nodes_to_visit.top();

    if (cur == goal_node) {
      solution_found = true;
      break;
    }

    nodes_to_visit.pop();

    int row = cur.idx / w;
    int col = cur.idx % w;
    // check bounds and find up to eight neighbors: top to bottom, left to right
    nbrs[0] = (diag_ok && row > 0 && col > 0)          ? cur.idx - w - 1   : -1;
    nbrs[1] = (row > 0)                                ? cur.idx - w       : -1;
    nbrs[2] = (diag_ok && row > 0 && col + 1 < w)      ? cur.idx - w + 1   : -1;
    nbrs[3] = (col > 0)                                ? cur.idx - 1       : -1;
    nbrs[4] = (col + 1 < w)                            ? cur.idx + 1       : -1;
    nbrs[5] = (diag_ok && row + 1 < h && col > 0)      ? cur.idx + w - 1   : -1;
    nbrs[6] = (row + 1 < h)                            ? cur.idx + w       : -1;
    nbrs[7] = (diag_ok && row + 1 < h && col + 1 < w ) ? cur.idx + w + 1   : -1;

    float heuristic_cost;
    for (int i = 0; i < 8; ++i) {
      if (nbrs[i] >= 0) {
        // the sum of the cost so far and the cost of this move
        float new_cost = costs[cur.idx] + weights[nbrs[i]];
        if (new_cost < costs[nbrs[i]]) {
          // estimate the cost to the goal based on legal moves
          if (diag_ok) {
            heuristic_cost = linf_norm(nbrs[i] / w, nbrs[i] % w,
                                       goal    / w, goal    % w);
          }
          else {
            heuristic_cost = l1_norm(nbrs[i] / w, nbrs[i] % w,
                                     goal    / w, goal    % w);
          }

          // paths with lower expected cost are explored first
          float priority = new_cost + heuristic_cost;
          nodes_to_visit.push(Node(nbrs[i], priority));

          costs[nbrs[i]] = new_cost;
          paths[nbrs[i]] = cur.idx;
        }
      }
    }
  }

  delete[] costs;
  delete[] nbrs;

  return solution_found;
}
```