A* search algorithm

The A* search algorithm is an extension of Dijkstra's algorithm (/wiki/Dijkstra%27s_algorithm) useful for finding the lowest cost path between two nodes (aka vertices) of a graph. The path may traverse any number of nodes connected by edges (aka arcs) with each edge having an associated cost. The algorithm uses a heuristic which associates an estimate of the lowest cost path from this node to the goal node, such that this estimate is never greater than the actual cost.

A* search algorithm is a **draft** programming task. It is not yet considered ready to be promoted as a complete task, for reasons that should be found in its talk page (/wiki/Talk:A*_search_algorithm).

The algorithm should not assume that all edge costs are the same. It should be possible to start and finish on any node, including ones identified as a barrier in the task.

Task

Consider the problem of finding a route across the diagonal of a chess board-like 8x8 grid. The rows are numbered from 0 to 7. The columns are also numbered 0 to 7. The start position is (0, 0) and the end position is (7, 7). Movement is allow by one square in any direction including diagonals, similar to a king in chess. The standard movement cost is 1. To make things slightly harder, there is a barrier that occupy certain positions of the grid. Moving into any of the barrier positions has a cost of 100.

The barrier occupies the positions (2,4), (2,5), (2,6), (3,6), (4,6), (5,6), (5,5), (5,4), (5,3), (5,2), (4,2) and (3,2).

A route with the lowest cost should be found using the A* search algorithm (there are multiple optimal solutions with the same total cost).

Print the optimal route in text format, as well as the total cost of the route.

Optionally, draw the optimal route and the barrier positions.

Note: using a heuristic score of zero is equivalent to Dijkstra's algorithm and that's kind of cheating/not really A*!

Extra Credit

Use this algorithm to solve an 8 puzzle. Each node of the input graph will represent an arrangement of the tiles. The nodes will be connected by 4 edges representing swapping the blank tile up, down, left, or right. The cost of each edge is 1. The heuristic will be the sum of the manhatten distance of each numbered tile from its goal position. An 8 puzzle graph will have 9!/2 (181,440) nodes. The 15 puzzle has over 10 trillion nodes. This algorithm may solve simple 15 puzzles (but there are not many of those).

See also

- Wikipedia webpage: A* search algorithm (https://en.wikipedia.org/wiki/A*_search_algorithm).
- An introduction to: Breadth First Search |> Dijkstra's Algorithm |> *A** (https://www.redblobgames.com/pathfinding/a-star/introduction.html)

Related tasks

- 15 puzzle solver (/wiki/15_puzzle_solver)
- Dijkstra's algorithm (/wiki/Dijkstra%27s_algorithm)

Contents

C (/wiki/Category:C)

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <float.h>
/* and not not_eq */
#include <iso646.h>
/* add -lm to command line to compile with this header */
#include <math.h>

#define map_size_rows 10
#define map_size_cols 10

char map[map_size_rows][map_size_cols] = {
    {1, 1, 1, 1, 1, 1, 1, 1, 1, 1},
    {1, 0, 0, 0, 0, 0, 0, 0, 0, 1},
    {1, 0, 0, 0, 0, 0, 0, 0, 0, 1},
    {1, 0, 0, 0, 0, 1, 1, 1, 0, 1},
    {1, 0, 0, 1, 0, 0, 0, 1, 0, 1},
    {1, 0, 0, 1, 0, 0, 0, 1, 0, 1},
    {1, 0, 0, 1, 1, 1, 1, 1, 0, 1},
    {1, 0, 0, 0, 0, 0, 0, 0, 0, 1},
    {1, 0, 0, 0, 0, 0, 0, 0, 0, 1},
    {1, 1, 1, 1, 1, 1, 1, 1, 1, 1}
};

/* description of graph node */
struct stop {
    double col, row;
    /* array of indexes of routes from this stop to neighbours in array of all rout
    int * n;
    int n_len;
    double f, g, h;
    int from;
};

int ind[map_size_rows][map_size_cols] = {
    {-1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
    {-1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
    {-1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
    {-1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
    {-1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
    {-1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
```

```
    {-1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
    {-1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
    {-1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
    {-1, -1, -1, -1, -1, -1, -1, -1, -1, -1}
};

/* description of route between two nodes */
struct route {
    /* route has only one direction! */
    int x; /* index of stop in array of all stops of src of this route */
    int y; /* intex of stop in array of all stops od dst of this route */
    double d;
};

int main() {
    int i, j, k, l, b, found;
    int p_len = 0;
    int * path = NULL;
    int c_len = 0;
    int * closed = NULL;
    int o_len = 1;
    int * open = (int*)calloc (http://www.opengroup.org/onlinepubs/009695399/functi
    double min, tempg;
    int s;
    int e;
    int current;
    int s_len = 0;
    struct stop * stops = NULL;
    int r_len = 0;
    struct route * routes = NULL;

    for (i = 1; i < map_size_rows - 1; i++) {
        for (j = 1; j < map_size_cols - 1; j++) {
            if (!map[i][j]) {
                ++s_len;
                stops = (struct stop *)realloc (http://www.opengroup.org/onlinepubs
                int t = s_len - 1;
                stops[t].col = j;
                stops[t].row = i;
                stops[t].from = -1;
                stops[t].g = DBL_MAX;
                stops[t].n_len = 0;
                stops[t].n = NULL;
                ind[i][j] = t;
```

```
            }
        }
    }

    /* index of start stop */
    s = 0;
    /* index of finish stop */
    e = s_len - 1;

    for (i = 0; i < s_len; i++) {
        stops[i].h = sqrt (http://www.opengroup.org/onlinepubs/009695399/functions/
    }

    for (i = 1; i < map_size_rows - 1; i++) {
        for (j = 1; j < map_size_cols - 1; j++) {
            if (ind[i][j] >= 0) {
                for (k = i - 1; k <= i + 1; k++) {
                    for (l = j - 1; l <= j + 1; l++) {
                        if ((k == i) and (l == j)) {
                            continue;
                        }
                        if (ind[k][l] >= 0) {
                            ++r_len;
                            routes = (struct route *)realloc (http://www.opengroup.
                            int t = r_len - 1;
                            routes[t].x = ind[i][j];
                            routes[t].y = ind[k][l];
                            routes[t].d = sqrt (http://www.opengroup.org/onlinepubs
                            ++stops[routes[t].x].n_len;
                            stops[routes[t].x].n = (int*)realloc (http://www.opengr
                            stops[routes[t].x].n[stops[routes[t].x].n_len - 1] = t;
                        }
                    }
                }
            }
        }
    }

    open[0] = s;
    stops[s].g = 0;
    stops[s].f = stops[s].g + stops[s].h;
    found = 0;

    while (o_len and not found) {
```

```
min = DBL_MAX;

for (i = 0; i < o_len; i++) {
    if (stops[open[i]].f < min) {
        current = open[i];
        min = stops[open[i]].f;
    }
}

if (current == e) {
    found = 1;

    ++p_len;
    path = (int*)realloc (http://www.opengroup.org/onlinepubs/009695399/fun
    path[p_len - 1] = current;
    while (stops[current].from >= 0) {
        current = stops[current].from;
        ++p_len;
        path = (int*)realloc (http://www.opengroup.org/onlinepubs/009695399
        path[p_len - 1] = current;
    }
}

for (i = 0; i < o_len; i++) {
    if (open[i] == current) {
        if (i not_eq (o_len - 1)) {
            for (j = i; j < (o_len - 1); j++) {
                open[j] = open[j + 1];
            }
        }
        --o_len;
        open = (int*)realloc (http://www.opengroup.org/onlinepubs/009695399
        break;
    }
}

++c_len;
closed = (int*)realloc (http://www.opengroup.org/onlinepubs/009695399/funct
closed[c_len - 1] = current;

for (i = 0; i < stops[current].n_len; i++) {
    b = 0;

    for (j = 0; j < c_len; j++) {
```

```
                if (routes[stops[current].n[i]].y == closed[j]) {
                    b = 1;
                }
            }

            if (b) {
                continue;
            }

            tempg = stops[current].g + routes[stops[current].n[i]].d;

            b = 1;

            if (o_len > 0) {
                for (j = 0; j < o_len; j++) {
                    if (routes[stops[current].n[i]].y == open[j]) {
                        b = 0;
                    }
                }
            }

            if (b or (tempg < stops[routes[stops[current].n[i]].y].g)) {
                stops[routes[stops[current].n[i]].y].from = current;
                stops[routes[stops[current].n[i]].y].g = tempg;
                stops[routes[stops[current].n[i]].y].f = stops[routes[stops[current

                if (b) {
                    ++o_len;
                    open = (int*)realloc (http://www.opengroup.org/onlinepubs/00969
                    open[o_len - 1] = routes[stops[current].n[i]].y;
                }
            }
        }
    }

    for (i = 0; i < map_size_rows; i++) {
        for (j = 0; j < map_size_cols; j++) {
            if (map[i][j]) {
                putchar (http://www.opengroup.org/onlinepubs/009695399/functions/pu
            } else {
                b = 0;
                for (k = 0; k < p_len; k++) {
                    if (ind[i][j] == path[k]) {
                        ++b;
```

```
                    }
                }
                if (b) {
                    putchar (http://www.opengroup.org/onlinepubs/009695399/function
                } else {
                    putchar (http://www.opengroup.org/onlinepubs/009695399/function
                }
            }
        }
        putchar (http://www.opengroup.org/onlinepubs/009695399/functions/putchar.ht
    }

    if (not found) {
        puts (http://www.opengroup.org/onlinepubs/009695399/functions/puts.html)("I
    } else {
        printf (http://www.opengroup.org/onlinepubs/009695399/functions/printf.html
        for (i = p_len - 1; i >= 0; i--) {
            printf (http://www.opengroup.org/onlinepubs/009695399/functions/printf.
        }
    }

    for (i = 0; i < s_len; ++i) {
        free (http://www.opengroup.org/onlinepubs/009695399/functions/free.html)(st
    }
    free (http://www.opengroup.org/onlinepubs/009695399/functions/free.html)(stops)
    free (http://www.opengroup.org/onlinepubs/009695399/functions/free.html)(routes
    free (http://www.opengroup.org/onlinepubs/009695399/functions/free.html)(path);
    free (http://www.opengroup.org/onlinepubs/009695399/functions/free.html)(open);
    free (http://www.opengroup.org/onlinepubs/009695399/functions/free.html)(closed

    return 0;
}
```

Output:

```
▒▒▒▒▒▒▒▒▒▒
▒x.......▒
▒.x......▒
▒.x..▒▒▒.▒
▒.x▒...▒.▒
▒.x▒...▒.▒
▒.x▒▒▒▒▒.▒
▒..xxxxx.▒
▒.......x▒
▒▒▒▒▒▒▒▒▒▒
path cost is 12:
(1, 1)
(2, 2)
(2, 3)
(2, 4)
(2, 5)
(2, 6)
(3, 7)
(4, 7)
(5, 7)
(6, 7)
(7, 7)
(8, 8)
```

C++ (/wiki/Category:C%2B%2B)

```
#include <list>
#include <algorithm>
#include <iostream>

class point {
public:
    point( int a = 0, int b = 0 ) { x = a; y = b; }
    bool operator ==( const point& o ) { return o.x == x && o.y == y; }
    point operator +( const point& o ) { return point( o.x + x, o.y + y ); }
    int x, y;
};

class map {
public:
    map() {
        char t[8][8] = {
            {0, 0, 0, 0, 0, 0, 0, 0}, {0, 0, 0, 0, 0, 0, 0, 0},
            {0, 0, 0, 0, 1, 1, 1, 0}, {0, 0, 1, 0, 0, 0, 1, 0},
            {0, 0, 1, 0, 0, 0, 1, 0}, {0, 0, 1, 1, 1, 1, 1, 0},
            {0, 0, 0, 0, 0, 0, 0, 0}, {0, 0, 0, 0, 0, 0, 0, 0}
        };
        w = h = 8;
        for( int r = 0; r < h; r++ )
            for( int s = 0; s < w; s++ )
                m[s][r] = t[r][s];
    }
    int operator() ( int x, int y ) { return m[x][y]; }
    char m[8][8];
    int w, h;
};

class node {
public:
    bool operator == (const node& o ) { return pos == o.pos; }
    bool operator == (const point& o ) { return pos == o; }
    bool operator < (const node& o ) { return dist + cost < o.dist + o.cost; }
    point pos, parent;
    int dist, cost;
};

class aStar {
public:
```

```
    aStar() {
        neighbours[0] = point( -1, -1 ); neighbours[1] = point(  1, -1 );
        neighbours[2] = point( -1,  1 ); neighbours[3] = point(  1,  1 );
        neighbours[4] = point(  0, -1 ); neighbours[5] = point( -1,  0 );
        neighbours[6] = point(  0,  1 ); neighbours[7] = point(  1,  0 );
    }

    int calcDist( point& p ){
        // need a better heuristic
        int x = end.x - p.x, y = end.y - p.y;
        return( x * x + y * y );
    }

    bool isValid( point& p ) {
        return ( p.x >-1 && p.y > -1 && p.x < m.w && p.y < m.h );
    }

    bool existPoint( point& p, int cost ) {
        std::list<node>::iterator i;
        i = std::find( closed.begin(), closed.end(), p );
        if( i != closed.end() ) {
            if( ( *i ).cost + ( *i ).dist < cost ) return true;
            else { closed.erase( i ); return false; }
        }
        i = std::find( open.begin(), open.end(), p );
        if( i != open.end() ) {
            if( ( *i ).cost + ( *i ).dist < cost ) return true;
            else { open.erase( i ); return false; }
        }
        return false;
    }

    bool fillOpen( node& n ) {
        int stepCost, nc, dist;
        point neighbour;

        for( int x = 0; x < 8; x++ ) {
            // one can make diagonals have different cost
            stepCost = x < 4 ? 1 : 1;
            neighbour = n.pos + neighbours[x];
            if( neighbour == end ) return true;

            if( isValid( neighbour ) && m( neighbour.x, neighbour.y ) != 1 ) {
                nc = stepCost + n.cost;
```

```
                dist = calcDist( neighbour );
                if( !existPoint( neighbour, nc + dist ) ) {
                    node m;
                    m.cost = nc; m.dist = dist;
                    m.pos = neighbour;
                    m.parent = n.pos;
                    open.push_back( m );
                }
            }
        }
        return false;
    }

    bool search( point& s, point& e, map& mp ) {
        node n; end = e; start = s; m = mp;
        n.cost = 0; n.pos = s; n.parent = 0; n.dist = calcDist( s );
        open.push_back( n );
        while( !open.empty() ) {
            //open.sort();
            node n = open.front();
            open.pop_front();
            closed.push_back( n );
            if( fillOpen( n ) ) return true;
        }
        return false;
    }

    int path( std::list<point>& path ) {
        path.push_front( end );
        int cost = 1 + closed.back().cost;
        path.push_front( closed.back().pos );
        point parent = closed.back().parent;

        for( std::list<node>::reverse_iterator i = closed.rbegin(); i != closed.ren
            if( ( *i ).pos == parent && !( ( *i ).pos == start ) ) {
                path.push_front( ( *i ).pos );
                parent = ( *i ).parent;
            }
        }
        path.push_front( start );
        return cost;
    }

    map m; point end, start;
```

```
    point neighbours[8];
    std::list<node> open;
    std::list<node> closed;
};

int main( int argc, char* argv[] ) {
    map m;
    point s, e( 7, 7 );
    aStar as;

    if( as.search( s, e, m ) ) {
        std::list<point> path;
        int c = as.path( path );
        for( int y = -1; y < 9; y++ ) {
            for( int x = -1; x < 9; x++ ) {
                if( x < 0 || y < 0 || x > 7 || y > 7 || m( x, y ) == 1 )
                    std::cout << char(0xdb);
                else {
                    if( std::find( path.begin(), path.end(), point( x, y ) )!= path
                        std::cout << "x";
                    else std::cout << ".";
                }
            }
            std::cout << "\n";
        }

        std::cout << "\nPath cost " << c << ": ";
        for( std::list<point>::iterator i = path.begin(); i != path.end(); i++ ) {
            std::cout<< "(" << ( *i ).x << ", " << ( *i ).y << ") ";
        }
    }
    std::cout << "\n\n";
    return 0;
}
```

Output:

```
██████████
█x.......█
█x.......█
█x...███.█
█x.█...█.█
█x.█...█.█
█.x█████.█
█..xxxx..█
█......xx█
██████████

Path cost 11: (0, 0) (0, 1) (0, 2) (0, 3) (0, 4) (1, 5) (2, 6) (3, 6) (4, 6) (5, 6)
```

Go (/wiki/Category:Go)

```go
// Package astar implements the A* search algorithm with minimal constraints
// on the graph representation.
package astar

import "container/heap"

// Exported node type.
type Node interface {
    To() []Arc               // return list of arcs from this node to another
    Heuristic(from Node) int // heuristic cost from another node to this one
}

// An Arc, actually a "half arc", leads to another node with integer cost.
type Arc struct {
    To   Node
    Cost int
}

// rNode holds data for a "reached" node
type rNode struct {
    n    Node
    from Node
    l    int // route len
    g    int // route cost
    f    int // "g+h", route cost + heuristic estimate
    fx   int // heap.Fix index
}

type openHeap []*rNode // priority queue

// Route computes a route from start to end nodes using the A* algorithm.
//
// The algorithm is general A*, where the heuristic is not required to be
// monotonic.  If a route exists, the function will find a route regardless
// of the quality of the Heuristic.  For an admissiable heuristic, the route
// will be optimal.
func Route(start, end Node) (route []Node, cost int) {
    // start node initialized with heuristic
    cr := &rNode{n: start, l: 1, f: end.Heuristic(start)}
    // maintain a set of reached nodes.  start is reached initially
    r := map[Node]*rNode{start: cr}
    // oh is a heap of nodes "open" for exploration.  nodes go on the heap
    // when they get an initial or new "g" route distance, and therefore a
```

```
    // new "f" which serves as priority for exploration.
    oh := openHeap{cr}
    for len(oh) > 0 {
        bestRoute := heap.Pop(&oh).(*rNode)
        bestNode := bestRoute.n
        if bestNode == end {
            // done.  prepare return values
            cost = bestRoute.g
            route = make([]Node, bestRoute.l)
            for i := len(route) - 1; i >= 0; i-- {
                route[i] = bestRoute.n
                bestRoute = r[bestRoute.from]
            }
            return
        }
        l := bestRoute.l + 1
        for _, to := range bestNode.To() {
            // "g" route distance from start
            g := bestRoute.g + to.Cost
            if alt, ok := r[to.To]; !ok {
                // alt being reached for the first time
                alt = &rNode{n: to.To, from: bestNode, l: l,
                    g: g, f: g + end.Heuristic(to.To)}
                r[to.To] = alt
                heap.Push(&oh, alt)
            } else {
                if g >= alt.g {
                    continue // candidate route no better than existing route
                }
                // it's a better route
                // update data and make sure it's on the heap
                alt.from = bestNode
                alt.l = l
                alt.g = g
                alt.f = end.Heuristic(alt.n)
                if alt.fx < 0 {
                    heap.Push(&oh, alt)
                } else {
                    heap.Fix(&oh, alt.fx)
                }
            }
        }
    }
    return nil, 0
```

```
}

// implement container/heap
func (h openHeap) Len() int           { return len(h) }
func (h openHeap) Less(i, j int) bool { return h[i].f < h[j].f }
func (h openHeap) Swap(i, j int) {
    h[i], h[j] = h[j], h[i]
    h[i].fx = i
    h[j].fx = j
}

func (p *openHeap) Push(x interface{}) {
    h := *p
    fx := len(h)
    h = append(h, x.(*rNode))
    h[fx].fx = fx
    *p = h
}

func (p *openHeap) Pop() interface{} {
    h := *p
    last := len(h) - 1
    *p = h[:last]
    h[last].fx = -1
    return h[last]
}
```

```go
package main

import (
    "fmt"

    "astar"
)

// rcNode implements the astar.Node interface
type rcNode struct{ r, c int }

var barrier = map[rcNode]bool{{2, 4}: true, {2, 5}: true,
    {2, 6}: true, {3, 6}: true, {4, 6}: true, {5, 6}: true, {5, 5}: true,
    {5, 4}: true, {5, 3}: true, {5, 2}: true, {4, 2}: true, {3, 2}: true}

// graph representation is virtual.  Arcs from a node are generated when
// requested, but there is no static graph representation.
func (fr rcNode) To() (a []astar.Arc) {
    for r := fr.r - 1; r <= fr.r+1; r++ {
        for c := fr.c - 1; c <= fr.c+1; c++ {
            if (r == fr.r && c == fr.c) || r < 0 || r > 7 || c < 0 || c > 7 {
                continue
            }
            n := rcNode{r, c}
            cost := 1
            if barrier[n] {
                cost = 100
            }
            a = append(a, astar.Arc{n, cost})
        }
    }
    return a
}

// The heuristic computed is max of row distance and column distance.
// This is effectively the cost if there were no barriers.
func (n rcNode) Heuristic(fr astar.Node) int {
    dr := n.r - fr.(rcNode).r
    if dr < 0 {
        dr = -dr
    }
    dc := n.c - fr.(rcNode).c
    if dc < 0 {
```

```
        dc = -dc
    }
    if dr > dc {
        return dr
    }
    return dc
}

func main() {
    route, cost := astar.Route(rcNode{0, 0}, rcNode{7, 7})
    fmt.Println("Route:", route)
    fmt.Println("Cost:", cost)
}
```

Output:

```
Route: [{0 0} {1 1} {2 2} {3 1} {4 1} {5 1} {6 2} {6 3} {6 4} {6 5} {6 6} {7 7}]
Cost: 11
```

JavaScript (/wiki/Category:JavaScript)

Animated.
To see how it works on a random map go here (http://paulo-jorente.de/tests/astar/)

```
var ctx, map, opn = [], clsd = [], start = {x:1, y:1, f:0, g:0},
goal = {x:8, y:8, f:0, g:0}, mw = 10, mh = 10, neighbours, path;

function findNeighbour( arr, n ) {
    var a;
    for( var i = 0; i < arr.length; i++ ) {
        a = arr[i];
        if( n.x === a.x && n.y === a.y ) return i;
    }
    return -1;
}
function addNeighbours( cur ) {
    var p;
    for( var i = 0; i < neighbours.length; i++ ) {
        var n = {x: cur.x + neighbours[i].x, y: cur.y + neighbours[i].y, g: 0, h: 0
        if( map[n.x][n.y] == 1 || findNeighbour( clsd, n ) > -1 ) continue;
        n.g = cur.g + neighbours[i].c; n.h = Math.abs( goal.x - n.x ) + Math.abs( g
        p = findNeighbour( opn, n );
        if( p > -1 && opn[p].g + opn[p].h <= n.g + n.h ) continue;
        opn.push( n );
    }
    opn.sort( function( a, b ) {
        return ( a.g + a.h ) - ( b.g + b.h ); } );
}
function createPath() {
    path = [];
    var a, b;
    a = clsd.pop();
    path.push( a );
    while( clsd.length ) {
        b = clsd.pop();
        if( b.x != a.prt.x || b.y != a.prt.y ) continue;
        a = b; path.push( a );
    }
 }
function solveMap() {
    drawMap();
    if( opn.length < 1 ) {
        document.body.appendChild( document.createElement( "p" ) ).innerHTML = "Imp
        return;
    }
    var cur = opn.splice( 0, 1 )[0];
```

```
    clsd.push( cur );
    if( cur.x == goal.x && cur.y == goal.y ) {
        createPath(); drawMap();
        return;
    }
    addNeighbours( cur );
    requestAnimationFrame( solveMap );
}
function drawMap() {
    ctx.fillStyle = "#ee6"; ctx.fillRect( 0, 0, 200, 200 );
    for( var j = 0; j < mh; j++ ) {
        for( var i = 0; i < mw; i++ ) {
            switch( map[i][j] ) {
                case 0: continue;
                case 1: ctx.fillStyle = "#990"; break;
                case 2: ctx.fillStyle = "#090"; break;
                case 3: ctx.fillStyle = "#900"; break;
            }
            ctx.fillRect( i, j, 1, 1 );
        }
    }
    var a;
    if( path.length ) {
        var txt = "Path: " + ( path.length - 1 ) + "<br />[";
        for( var i = path.length - 1; i > -1; i-- ) {
            a = path[i];
            ctx.fillStyle = "#999";
            ctx.fillRect( a.x, a.y, 1, 1 );
            txt += "(" + a.x + ", " + a.y + ") ";
        }
        document.body.appendChild( document.createElement( "p" ) ).innerHTML = txt
        return;
    }
    for( var i = 0; i < opn.length; i++ ) {
        a = opn[i];
        ctx.fillStyle = "#909";
        ctx.fillRect( a.x, a.y, 1, 1 );
    }
    for( var i = 0; i < clsd.length; i++ ) {
        a = clsd[i];
        ctx.fillStyle = "#009";
        ctx.fillRect( a.x, a.y, 1, 1 );
    }
}
```

```
function createMap() {
    map = new Array( mw );
    for( var i = 0; i < mw; i++ ) {
        map[i] = new Array( mh );
        for( var j = 0; j < mh; j++ ) {
            if( !i || !j || i == mw - 1 || j == mh - 1 ) map[i][j] = 1;
            else map[i][j] = 0;
        }
    }
    map[5][3] = map[6][3] = map[7][3] = map[3][4] = map[7][4] = map[3][5] =
    map[7][5] = map[3][6] = map[4][6] = map[5][6] = map[6][6] = map[7][6] = 1;
    //map[start.x][start.y] = 2; map[goal.x][goal.y] = 3;
}
function init() {
    var canvas = document.createElement( "canvas" );
    canvas.width = canvas.height = 200;
    ctx = canvas.getContext( "2d" );
    ctx.scale( 20, 20 );
    document.body.appendChild( canvas );
    neighbours = [
        {x:1, y:0, c:1}, {x:-1, y:0, c:1}, {x:0, y:1, c:1}, {x:0, y:-1, c:1},
        {x:1, y:1, c:1.4}, {x:1, y:-1, c:1.4}, {x:-1, y:1, c:1.4}, {x:-1, y:-1, c:1
    ];
    path = []; createMap(); opn.push( start ); solveMap();
}
```

Output:

```
Path: 11
[(1, 1) (2, 2) (2, 3) (2, 4) (2, 5) (2, 6) (3, 7) (4, 8) (5, 8) (6, 8) (7, 8) (8, 8
```

Kotlin (/wiki/Category:Kotlin)

```
import java.lang.Math.abs

typealias GridPosition = Pair<Int, Int>
typealias Barrier = Set<GridPosition>

const val MAX_SCORE = 99999999

abstract class Grid(private val barriers: List<Barrier>) {

    open fun heuristicDistance(start: GridPosition, finish: GridPosition): Int {
        val dx = abs(start.first - finish.first)
        val dy = abs(start.second - finish.second)
        return (dx + dy) + (-2) * minOf(dx, dy)
    }

    fun inBarrier(position: GridPosition) = barriers.any { it.contains(position) }

    abstract fun getNeighbours(position: GridPosition): List<GridPosition>

    open fun moveCost(from: GridPosition, to: GridPosition) = if (inBarrier(to)) MA
}

class SquareGrid(width: Int, height: Int, barriers: List<Barrier>) : Grid(barriers)

    private val heightRange: IntRange = (0 until height)
    private val widthRange: IntRange = (0 until width)

    private val validMoves = listOf(Pair(1, 0), Pair(-1, 0), Pair(0, 1), Pair(0, -1

    override fun getNeighbours(position: GridPosition): List<GridPosition> = validM
            .map { GridPosition(position.first + it.first, position.second + it.sec
            .filter { inGrid(it) }

    private fun inGrid(it: GridPosition) = (it.first in widthRange) && (it.second i
}


/**
 * Implementation of the A* Search Algorithm to find the optimum path between 2 poi
 *
 * The Grid contains the details of the barriers and methods which supply the neigh
 * cost of movement between 2 cells.  Examples use a standard Grid which allows mov
```

```
 * (i.e. includes diagonals) but alternative implementation of Grid can be supplied
 *
 */
fun aStarSearch(start: GridPosition, finish: GridPosition, grid: Grid): Pair<List<G

    /**
     * Use the cameFrom values to Backtrack to the start position to generate the p
     */
    fun generatePath(currentPos: GridPosition, cameFrom: Map<GridPosition, GridPosi
        val path = mutableListOf(currentPos)
        var current = currentPos
        while (cameFrom.containsKey(current)) {
            current = cameFrom.getValue(current)
            path.add(0, current)
        }
        return path.toList()
    }

    val openVertices = mutableSetOf(start)
    val closedVertices = mutableSetOf<GridPosition>()
    val costFromStart = mutableMapOf(start to 0)
    val estimatedTotalCost = mutableMapOf(start to grid.heuristicDistance(start, fi

    val cameFrom = mutableMapOf<GridPosition, GridPosition>()  // Used to generate

    while (openVertices.size > 0) {

        val currentPos = openVertices.minBy { estimatedTotalCost.getValue(it) }!!

        // Check if we have reached the finish
        if (currentPos == finish) {
            // Backtrack to generate the most efficient path
            val path = generatePath(currentPos, cameFrom)
            return Pair(path, estimatedTotalCost.getValue(finish)) // First Route t
        }

        // Mark the current vertex as closed
        openVertices.remove(currentPos)
        closedVertices.add(currentPos)

        grid.getNeighbours(currentPos)
                .filterNot { closedVertices.contains(it) }  // Exclude previous vis
                .forEach { neighbour ->
                    val score = costFromStart.getValue(currentPos) + grid.moveCost(
```

```
                    if (score < costFromStart.getOrDefault(neighbour, MAX_SCORE)) {
                        if (!openVertices.contains(neighbour)) {
                            openVertices.add(neighbour)
                        }
                        cameFrom.put(neighbour, currentPos)
                        costFromStart.put(neighbour, score)
                        estimatedTotalCost.put(neighbour, score + grid.heuristicDis
                    }
                }

    }

    throw IllegalArgumentException("No Path from Start $start to Finish $finish")
}

fun main(args: Array<String>) {

    val barriers = listOf(setOf( Pair(2,4), Pair(2,5), Pair(2,6), Pair(3,6), Pair(4
                 Pair(5,4), Pair(5,3), Pair(5,2), Pair(4,2), Pair(3,2)))

    val (path, cost) = aStarSearch(GridPosition(0,0), GridPosition(7,7), SquareGrid

    println("Cost: $cost  Path: $path")
}
```

Output:

```
Cost: 11
Path: [(0, 0), (1, 1), (2, 2), (3, 1), (4, 1), (5, 1), (6, 2), (6, 3), (6, 4), (6,
```

Lua (/wiki/Category:Lua)

```lua
-- QUEUE -----------------------------------------------------------------------
Queue = {}
function Queue:new()
    local q = {}
    self.__index = self
    return setmetatable( q, self )
end
function Queue:push( v )
    table.insert( self, v )
end
function Queue:pop()
    return table.remove( self, 1 )
end
function Queue:getSmallestF()
    local s, i = nil, 2
    while( self[i] ~= nil and self[1] ~= nil ) do
        if self[i]:F() < self[1]:F() then
            s = self[1]
            self[1] = self[i]
            self[i] = s
        end
        i = i + 1
    end
    return self:pop()
end

-- LIST ------------------------------------------------------------------------
List = {}
function List:new()
    local l = {}
    self.__index = self
    return setmetatable( l, self )
end
function List:push( v )
  table.insert( self, v )
end
function List:pop()
    return table.remove( self )
end

-- POINT -----------------------------------------------------------------------
Point = {}
```

```lua
function Point:new()
    local p = { y = 0, x = 0 }
    self.__index = self
    return setmetatable( p, self )
end
function Point:set( x, y )
    self.x, self.y = x, y
end
function Point:equals( o )
    return (o.x == self.x and o.y == self.y)
end
function Point:print()
    print( self.x, self.y )
end

-- NODE -------------------------------------------------------------------------------
Node = {}
function Node:new()
    local n = { pos = Point:new(), parent = Point:new(), dist = 0, cost = 0 }
    self.__index = self
    return setmetatable( n, self )
end
function Node:set( pt, parent, dist, cost )
    self.pos = pt
    self.parent = parent
    self.dist = dist
    self.cost = cost
end
function Node:F()
    return ( self.dist + self.cost )
end

-- A-STAR -----------------------------------------------------------------------------
local nbours = {
    {  1,  0, 1 }, {  0,  1, 1 }, {  1,  1, 1.4 }, {  1, -1, 1.4 },
    { -1, -1, 1.4 }, { -1,  1, 1.4 }, {  0, -1, 1 }, { -1,  0, 1 }
}
local map = {
        1,1,1,1,1,1,1,1,1,1,
        1,0,0,0,0,0,0,0,0,1,
        1,0,0,0,0,0,0,0,0,1,
        1,0,0,0,0,1,1,1,0,1,
        1,0,0,1,0,0,0,1,0,1,
        1,0,0,1,0,0,0,1,0,1,
```

```
        1,0,0,1,1,1,1,1,0,1,
        1,0,0,0,0,0,0,0,0,1,
        1,0,0,0,0,0,0,0,0,1,
        1,1,1,1,1,1,1,1,1,1
}
local open, closed, start, goal,
      mapW, mapH = Queue:new(), List:new(), Point:new(), Point:new(), 10, 10
start:set( 2, 2 ); goal:set( 9, 9 )

function hasNode( arr, pos )
    for nx, val in ipairs( arr ) do
        if val.pos:equals( pos ) then
            return nx
        end
    end
    return -1
end
function isValid( pos )
    return pos.x > 0 and pos.x <= mapW
           and pos.y > 0 and pos.y <= mapH
           and map[pos.x + mapW * pos.y - mapW] == 0
end
function calcDist( p1 )
    local x, y = goal.x - p1.x, goal.y - p1.y
    return math.abs( x ) + math.abs( y )
end
function addToOpen( node )
    local nx
    for n = 1, 8 do
        nNode = Node:new()
        nNode.parent:set( node.pos.x, node.pos.y )
        nNode.pos:set( node.pos.x + nbours[n][1], node.pos.y + nbours[n][2] )
        nNode.cost = node.cost + nbours[n][3]
        nNode.dist = calcDist( nNode.pos )

        if isValid( nNode.pos ) then
            if nNode.pos:equals( goal ) then
                closed:push( nNode )
                return true
            end
            nx = hasNode( closed, nNode.pos )
            if nx < 0 then
                nx = hasNode( open, nNode.pos )
                if( nx < 0 ) or ( nx > 0 and nNode:F() < open[nx]:F() ) then
```

```
                    if( nx > 0 ) then
                        table.remove( open, nx )
                    end
                    open:push( nNode )
                else
                    nNode = nil
                end
            end
        end
    end
    return false
end
function makePath()
    local i, l = #closed, List:new()
    local node, parent = closed[i], nil

    l:push( node.pos )
    parent = node.parent
    while( i > 0 ) do
        i = i - 1
        node = closed[i]
        if node ~= nil and node.pos:equals( parent ) then
            l:push( node.pos )
            parent = node.parent
        end
    end
    print( string.format( "Cost: %d", #l - 1 ) )
    io.write( "Path: " )
    for i = #l, 1, -1 do
        map[l[i].x + mapW * l[i].y - mapW] = 2
        io.write( string.format( "(%d, %d) ", l[i].x, l[i].y ) )
    end
    print( "" )
end
function aStar()
    local n = Node:new()
    n.dist = calcDist( start )
    n.pos:set( start.x, start.y )
    open:push( n )
    while( true ) do
        local node = open:getSmallestF()
        if node == nil then break end
        closed:push( node )
        if addToOpen( node ) == true then
```

```
            makePath()
            return true
        end
    end
    return false
end
-- ENTRY POINT ------------------------------------------------------------------
if true == aStar() then
    local m
    for j = 1, mapH do
        for i = 1, mapW do
            m = map[i + mapW * j - mapW]
            if m == 0 then
                io.write( "." )
            elseif m == 1 then
                io.write( string.char(0xdb) )
            else
                io.write( "x" )
            end
        end
        io.write( "\n" )
    end
else
    print( "can not find a path!" )
end
```

Output:

```
Cost: 11
Path: (2, 2) (3, 3) (3, 4) (3, 5) (3, 6) (3, 7) (4, 8) (5, 9) (6, 9) (7, 9) (8, 9)
██████████
█x.......█
█.x......█
█.x..███.█
█.x█...█.█
█.x█...█.█
█.x█████.█
█..x.....█
█...xxxxx█
██████████
```

Phix (/wiki/Category:Phix)

rows and columns are numbered 1 to 8. start position is {1,1} and end position is {8,8}. barriers are simply avoided, rather than costed at 100. Note that the 23 visited nodes does not count walls, but with them this algorithm exactly matches the 35 of Racket.

```
sequence grid = split("""
x:::::::
::::::::
::::###:
::#:::#:
::#:::#:
::#####:
::::::::
::::::::
""",'\n')

constant permitted = {{-1,-1},{0,-1},{1,-1},
                      {-1, 0},       {1, 0},
                      {-1, 1},{0,+1},{1,+1}}

sequence key = {7,0},   -- chebyshev, cost
         moves = {{1,1}},
         data = {moves}
setd(key,data)
bool found = false
integer count = 0
while not found do
    if dict_size()=0 then ?"impossible" exit end if
    key = getd_partial_key(0)
    data = getd(key)
    moves = data[$]
    if length(data)=1 then
        deld(key)
    else
        data = data[1..$-1]
        putd(key,data)
    end if
    count += 1
    for i=1 to length(permitted) do
        sequence newpos = sq_add(moves[$],permitted[i])
        integer {nx,ny} = newpos
        if nx>=1 and nx<=8
        and ny>=1 and ny<=8
        and grid[nx,ny] = ':' then -- (unvisited)
            grid[nx,ny] = '.'
            sequence newkey = {max(8-nx,8-ny),key[2]+1},
                     newmoves = append(moves,newpos)
            if newpos = {8,8} then
```

```
                moves = newmoves
                found = true
                exit
            end if
            integer k = getd_index(newkey)
            if k=0 then
                data = {newmoves}
            else
                data = append(getd_by_index(k),newmoves)
            end if
            putd(newkey,data)
        end if
    end for
end while
if found then
    printf(1,"visited %d nodes\ncost:%d\npath:",{count,length(moves)-1})
    ?moves
    for i=1 to length(moves) do
        integer {x,y} = moves[i]
        grid[x,y] = 'x'
    end for
    puts(1,join(grid,'\n'))
end if
```

Output:

```
visited 23 nodes
cost:11
path:{{1,1},{2,2},{3,3},{4,2},{5,2},{6,2},{7,3},{8,4},{8,5},{8,6},{8,7},{8,8}}
x......:
.x.....:
..x.###:
.x#...#:
.x#...#:
.x#####:
..x.....
:..xxxxx
```

Python (/wiki/Category:Python)

```
from __future__ import print_function
import matplotlib.pyplot as plt

class AStarGraph(object):
	#Define a class board like grid with two barriers

	def __init__(self):
		self.barriers = []
		self.barriers.append([(2,4),(2,5),(2,6),(3,6),(4,6),(5,6),(5,5),(5,

	def heuristic(self, start, goal):
		#Use Chebyshev distance heuristic if we can move one square either
		#adjacent or diagonal
		D = 1
		D2 = 1
		dx = abs(start[0] - goal[0])
		dy = abs(start[1] - goal[1])
		return D * (dx + dy) + (D2 - 2 * D) * min(dx, dy)

	def get_vertex_neighbours(self, pos):
		n = []
		#Moves allow link a chess king
		for dx, dy in [(1,0),(-1,0),(0,1),(0,-1),(1,1),(-1,1),(1,-1),(-1,-1
			x2 = pos[0] + dx
			y2 = pos[1] + dy
			if x2 < 0 or x2 > 7 or y2 < 0 or y2 > 7:
				continue
			n.append((x2, y2))
		return n

	def move_cost(self, a, b):
		for barrier in self.barriers:
			if b in barrier:
				return 100 #Extremely high cost to enter barrier sq
		return 1 #Normal movement cost

def AStarSearch(start, end, graph):

	G = {} #Actual movement cost to each position from the start position
	F = {} #Estimated movement cost of start to end going via this position

	#Initialize starting values
	G[start] = 0
```

```
	F[start] = graph.heuristic(start, end)

	closedVertices = set()
	openVertices = set([start])
	cameFrom = {}

	while len(openVertices) > 0:
		#Get the vertex in the open list with the lowest F score
		current = None
		currentFscore = None
		for pos in openVertices:
			if current is None or F[pos] < currentFscore:
				currentFscore = F[pos]
				current = pos

		#Check if we have reached the goal
		if current == end:
			#Retrace our route backward
			path = [current]
			while current in cameFrom:
				current = cameFrom[current]
				path.append(current)
			path.reverse()
			return path, F[end] #Done!

		#Mark the current vertex as closed
		openVertices.remove(current)
		closedVertices.add(current)

		#Update scores for vertices near the current position
		for neighbour in graph.get_vertex_neighbours(current):
			if neighbour in closedVertices:
				continue #We have already processed this node exhau
			candidateG = G[current] + graph.move_cost(current, neighbou

			if neighbour not in openVertices:
				openVertices.add(neighbour) #Discovered a new verte
			elif candidateG >= G[neighbour]:
				continue #This G score is worse than previously fou

			#Adopt this G score
			cameFrom[neighbour] = current
			G[neighbour] = candidateG
			H = graph.heuristic(neighbour, end)
```

```
                        F[neighbour] = G[neighbour] + H

        raise RuntimeError("A* failed to find a solution")

if __name__=="__main__":
        graph = AStarGraph()
        result, cost = AStarSearch((0,0), (7,7), graph)
        print ("route", result)
        print ("cost", cost)
        plt.plot([v[0] for v in result], [v[1] for v in result])
        for barrier in graph.barriers:
                plt.plot([v[0] for v in barrier], [v[1] for v in barrier])
        plt.xlim(-1,8)
        plt.ylim(-1,8)
        plt.show()
```

Output:

```
route [(0, 0), (1, 1), (2, 2), (3, 1), (4, 1), (5, 1), (6, 2), (7, 3), (6, 4), (7,
cost 11
```

Racket (/wiki/Category:Racket)

This code is lifted from: this blog post (https://jeapostrophe.github.io/2013-04-15-astar-post.html). Read it, it's very good.

```
#lang scribble/lp
@(chunk
  <graph-sig>
  (define-signature graph^
    (node? edge? node-edges edge-src edge-cost edge-dest)))

@(chunk
  <map-generation>
  (define (make-map N)
    ;; Jay's random algorithm
    ;; (build-matrix N N (λ (x y) (random 3)))
    ;; RC version
    (matrix [[0 0 0 0 0 0 0 0]
             [0 0 0 0 0 0 0 0]
             [0 0 0 0 1 1 1 0]
             [0 0 1 0 0 0 1 0]
             [0 0 1 0 0 0 1 0]
             [0 0 1 1 1 1 1 0]
             [0 0 0 0 0 0 0 0]
             [0 0 0 0 0 0 0 0]])))

@(chunk
  <map-graph-rep>
  (struct map-node (M x y) #:transparent)
  (struct map-edge (src dx dy dest)))

@(chunk
  <map-graph-cost>
  (define (edge-cost e)
    (match-define (map-edge _ _ _ (map-node M x y)) e)
    (match (matrix-ref M x y)
      [0  1]
      [1  100]
      [2 1000])))

@(chunk
  <map-graph-edges>
  (define (node-edges n)
    (match-define (map-node M x y) n)
    (append*
     (for*/list ([dx (in-list '(1 0 -1))]
                 [dy (in-list '(1 0 -1))]
                 #:when
```

```
                 (and (not (and (zero? dx) (zero? dy)))
                      ;; RC -- allowed to move diagonally, so not this clause
                      ;;(or (zero? dx) (zero? dy))
                      ))
       (cond
         [(and (<= 0 (+ dx x) (sub1 (matrix-num-cols M)))
               (<= 0 (+ dy y) (sub1 (matrix-num-rows M))))
          (define dest (map-node M (+ dx x) (+ dy y)))
          (list (map-edge n dx dy dest))]
         [else
          empty])))))

@(chunk
  <a-star>
  (define (A* graph@ initial node-cost)
    (define-values/invoke-unit graph@ (import) (export graph^))
    (define count 0)
    <a-star-setup>

    (begin0
      (let/ec esc
        <a-star-loop>
        #f)

      (printf "visited ~a nodes\n" count))))

@(chunk
  <a-star-setup>
  <a-star-setup-closed>
  <a-star-setup-open>)

@(chunk
  <a-star-setup-closed>
  (define node->best-path (make-hash))
  (define node->best-path-cost (make-hash))
  (hash-set! node->best-path      initial empty)
  (hash-set! node->best-path-cost initial 0))

@(chunk
  <a-star-setup-open>
  (define (node-total-estimate-cost n)
    (+ (node-cost n) (hash-ref node->best-path-cost n)))
  (define (node-cmp x y)
    (<= (node-total-estimate-cost x)
```

```
        (node-total-estimate-cost y)))
  (define open-set (make-heap node-cmp))
  (heap-add! open-set initial))

@(chunk
  <a-star-loop>
  (for ([x (in-heap/consume! open-set)])
    (set! count (add1 count))
    <a-star-loop-body>))

@(chunk
  <a-star-loop-stop?>
  (define h-x (node-cost x))
  (define path-x (hash-ref node->best-path x))

  (when (zero? h-x)
    (esc (reverse path-x))))

@(chunk
  <a-star-loop-body>
  <a-star-loop-stop?>

  (define g-x (hash-ref node->best-path-cost x))
  (for ([x->y (in-list (node-edges x))])
    (define y (edge-dest x->y))
    <a-star-loop-per-neighbor>))

@(chunk
  <a-star-loop-per-neighbor>
  (define new-g-y (+ g-x (edge-cost x->y)))
  (define old-g-y
    (hash-ref node->best-path-cost y +inf.0))
  (when (< new-g-y old-g-y)
    (hash-set! node->best-path-cost y new-g-y)
    (hash-set! node->best-path y (cons x->y path-x))
    (heap-add! open-set y)))

@(chunk
  <map-display>
  (define map-scale 15)
  (define (type-color ty)
    (match ty
      [0 "yellow"]
      [1 "green"]
```

```
      [2 "red"]))
  (define (cell-square ty)
    (square map-scale "solid" (type-color ty)))
  (define (row-image M row)
    (apply beside
           (for/list ([col (in-range (matrix-num-cols M))])
             (cell-square (matrix-ref M row col)))))
  (define (map-image M)
    (apply above
           (for/list ([row (in-range (matrix-num-rows M))])
             (row-image M row)))))

@(chunk
  <path-display-line>
  (define (edge-image-on e i)
    (match-define (map-edge (map-node _ sx sy) _ _ (map-node _ dx dy)) e)
    (add-line i
              (* (+ sy 0.5) map-scale) (* (+ sx 0.5) map-scale)
              (* (+ dy 0.5) map-scale) (* (+ dx 0.5) map-scale)
              "black")))

@(chunk
  <path-display>
  (define (path-image M path)
    (foldr edge-image-on (map-image M) path)))

@(chunk
  <map-graph>
  (define-unit map@
    (import) (export graph^)

    (define node? map-node?)
    (define edge? map-edge?)
    (define edge-src map-edge-src)
    (define edge-dest map-edge-dest)

    <map-graph-cost>
    <map-graph-edges>))

@(chunk
  <map-node-cost>
  (define ((make-node-cost GX GY) n)
    (match-define (map-node M x y) n)
    ;; Jay's
```

```
    #;(+ (abs (- x GX))
         (abs (- y GY)))
    ;; RC -- diagonal movement
    (max (abs (- x GX))
         (abs (- y GY)))))

@(chunk
  <map-example>
  (define N 8)
  (define random-M
    (make-map N))
  (define random-path
    (time
     (A* map@
         (map-node random-M 0 0)
         (make-node-cost (sub1 N) (sub1 N))))))

@(chunk
  <*>
  (require rackunit
           math/matrix
           racket/unit
           racket/match
           racket/list
           data/heap
           2htdp/image
           racket/runtime-path)

  <graph-sig>

  <map-generation>
  <map-graph-rep>
  <map-graph>

  <a-star>

  <map-node-cost>
  <map-example>
  (printf "path is ~a long\n" (length random-path))
  (printf "path is: ~a\n" (map (match-lambda
                                 [(map-edge src dx dy dest)
                                  (cons dx dy)])
                               random-path))
```

```
  <map-display>
  <path-display-line>
  <path-display>

  (path-image random-M random-path))
```

Output:

```
visited 35 nodes
cpu time: 94 real time: 97 gc time: 15
path is 11 long
path is: ((1 . 1) (1 . 1) (1 . -1) (1 . 0) (1 . 0) (1 . 1) (1 . 1) (0 . 1) (-1 . 1)
.
```

A diagram is also output, but you'll need to run this in DrRacket to see it.

REXX (/wiki/Category:REXX)

```
/*REXX program solves the    A*   search problem   for a  (general)   NxN   grid.
parse arg  N  sCol sRow .                          /*obtain optional arguments from t
if    N=='' |    N==","  then    N=8               /*No grid size specified?  Use def
if sCol=='' | sCol==","  then sCol=1               /*No starting column given?  "
if sRow=='' | sRow==","  then sRow=1               /* "     "     row     "     "
beg= '─0─'                                         /*mark the start of the journey in
o.=.;         p.=0                                 /*list of optimum start journey st
times=0                                            /*cntr/pos for number of optimizat
              Pc = ' 1  1  0  0   1 -1 -1 -1 '   /*the possible column moves for a
              Pr = ' 1  0  1 -1  -1  0  1 -1 '   /* "      "     row     "    "  "
Pcm=words(Pc)                                      /* [↑]  optimized for moving right
$.=1e6;  OK=0;     min$=$.                         /*# possible directions; cost; sol
@Aa= " A*  search algorithm on"                    /*a handy─dandy literal for the  S
flasher= '@. $. min$ N o. p. Pc. Pcm Pr. sCol sRow times'   /*a literal list for EX
call path 0                                        /*find a possible solution for the
@NxN= 'a '      N"x"N      ' grid'                 /*a literal used for a  SAY  state
if OK  then say 'A solution for the'    @Aa     @NxN       "with a score of "     @
       else say 'No'   @Aa   "solution for"     @NxN'.'
call show 1                                        /*invoke subroutine to display the
exit                                               /*stick a fork in it,  we're all d
/*───────────────────────────────────────────────────────────────────────────────────
@:    parse arg x,y,aChar;   if arg()==3  then @.x.y=aChar;                   retur
@p:   parse arg x,y;         if datatype(@.x.y, 'W')  then return @.x.y<m-1;  retur
/*───────────────────────────────────────────────────────────────────────────────────
barr: $=2.4 2.5 2.6 3.6 4.6 5.6 5.5 5.4 5.3 5.2 4.2 3.2  /*locations of barriers on
         do b=1  for words($);    _=word($, b);   parse var _ c '.' r;  call @ c+1,
         end   /*b*/;             return
/*───────────────────────────────────────────────────────────────────────────────────
move: procedure expose (flasher);          parse arg m,col,row   /*obtain  move,col
         do t=1  for Pcm;         nc=col + Pc.t;   nr=row + Pr.t /*a new path posit
         if @.nc.nr==.  then do;  if opti()  then iterate        /*Costlier path?
                                  @.nc.nr=m;       p.1.m=nc nr   /*Empty?  A legal
                                  p.pcm.m=nr nc-1                /*used for a fast
                                  if nc==N  then if nr==N  then return 1   /*last m
                                  if move(m + 1,  nc, nr)  then return 1   /*  "
                                  @.nc.nr=.                      /*undo the above m
                             end                                 /*try a different
         end   /*t*/                                             /* [↑]  all moves
      return 0                                                   /*path isn't possi
/*───────────────────────────────────────────────────────────────────────────────────
opti: ncm=nc-1;   nrm=nr-1;       if @p(ncm, nrm)  then return 1
                                  if @p(ncm, nr )  then return 1
                                  if @p(nc,  nrm)  then return 1
```

```
      ncp=nc+1;   nrp=nr+1;       if @p(ncp, nr )  then return 1
                                  if @p(ncp, nrm)  then return 1
                                  if @p(nc,  nrp)  then return 1
                                  if @p(ncm, nrp)  then return 1
                                  if @p(ncp, nrp)  then return 1;        return 0
/*──────────────────────────────────────────────────────────────────────────────
path: parse arg z;                t=times       /*initial move can only be one of
        do #=1  for Pcm;          @.=           /*optimize for each degree of move
        if z\==0  then  if #\==z  then iterate   /*This a particular low─cost reque
             do c=1  for  N;    do r=1  for N;   @.c.r=.;   end  /*r*/
             end   /*c*/
        iCol=sCol;  iRow=sRow;  @.sCol.sRow= beg /*all path's initial starting  pos
        call barr                               /*place the barriers on the grid.
        Pco=subword(Pc Pc, #, Pcm);  Pro=subword(Pr Pr, #, Pcm)
        parse var  Pco   Pc.1 Pc.2 Pc.3 Pc.4 Pc.5 Pc.6 Pc.7 Pc.8  /*possible direct
        parse var  Pro   Pr.1 Pr.2 Pr.3 Pr.4 Pr.5 Pr.6 Pr.7 Pr.8  /*    "         "
             do o=1  for times;  parse var o.o c r;    @.c.r=o;     iRow=r;     iCo
             end   /*o*/
        fp=move(1+times, iCol, iRow);      sol=@N.N\==. & fp
        if sol  then do;    $.#=@.N.N           /*Found a solution?  Remember the
                     OK=1;  min$=min(min$, $.#)
                     end
        end   /*#*/
      wp=1e7; wg=0;  do g=1  for Pcm; if $.g<wp & $.g>0 & t\=2  then do; wg=g; wp=$
                     end   /*g*/                /* [↑]  find minimum non-zero path
      if wg==0  then wg=8                       /*Not found?  Then use last cost f
      times=times + 1                           /*bump # times a marker has been p
      o.times= p.wg.times                       /*remember this move location for
      if times<4  then call path 0              /*only do memoization for first 3
      return
/*──────────────────────────────────────────────────────────────────────────────
show: ind=left('', 9 * (n<18) );      say       /*the indentation of the displayed
      _=substr(copies("┼───", N),2);  say ind translate('┌'_"┐", '┬', "┼")   /*grid
                                                /* [↓]  build a display for the gr
       do   c=1  for N;          if c\==1 & arg(1)  then say  ind  '├'_"┤";     L=@
         do r=1  for N; ?=@.c.r; if c ==N & r==N & ?\==.  then ?='end'; L=L"│"cente
         end   /*r*/                            /*done with   rank   of the grid.
       say ind translate(L'│', , .)             /*display a     "     "  "    "
       end     /*c*/                            /*a 19x19 grid can be shown 80 col
     say ind translate('└'_"┘",'┴',"┼");  return /*display the very bottom of the g
```

output when using the default input:

```
A solution for the  A*  search algorithm on a  8x8  grid with a score of  11:

┌───┬───┬───┬───┬───┬───┬───┬───┐
│─0─│   │   │   │   │   │   │   │
├───┼───┼───┼───┼───┼───┼───┼───┤
│   │ 1 │   │   │   │   │   │   │
├───┼───┼───┼───┼───┼───┼───┼───┤
│   │   │ 2 │   │ █ │ █ │ █ │   │
├───┼───┼───┼───┼───┼───┼───┼───┤
│   │ 3 │ █ │   │   │   │ █ │   │
├───┼───┼───┼───┼───┼───┼───┼───┤
│   │ 4 │ █ │   │   │   │ █ │   │
├───┼───┼───┼───┼───┼───┼───┼───┤
│   │ 5 │ █ │ █ │ █ │ █ │ █ │   │
├───┼───┼───┼───┼───┼───┼───┼───┤
│   │   │ 6 │   │   │   │   │   │
├───┼───┼───┼───┼───┼───┼───┼───┤
│   │   │   │ 7 │ 8 │ 9 │10 │end│
└───┴───┴───┴───┴───┴───┴───┴───┘
```

SequenceL (/wiki/Category:SequenceL)

```
import <Utilities/Set.sl>;
import <Utilities/Math.sl>;
import <Utilities/Sequence.sl>;

Point ::= (x : int, y : int);

State ::= (open : Point(1), closed : Point(1), cameFrom : Point(2), estimate : int(

allNeighbors := [(x : -1, y : -1), (x : 1, y : -1), (x : -1, y : 1), (x : 1, y : 1)
                                 (x : 0, y : -1), (x : -1, y : 0), (x : 0, y : 1),

defaultBarriers := [(x : 3, y : 5),(x : 3, y : 6),(x : 3, y : 7),(x : 4, y : 7),
        (x : 5, y : 7),(x : 6, y : 7),(x : 6, y : 6),(x : 6, y : 5),(x : 6, y : 4),
        (x : 6, y : 3),(x : 5, y : 3),(x : 4, y : 3)];

defaultWidth := 8;
defaultHeight := 8;

main(args(2)) := aStar(defaultWidth, defaultHeight, defaultBarriers, (x : 1, y : 1)

aStar(width, height, barriers(1), start, end) :=
        let
                newEstimate[i,j] := heuristic(start, end) when i = start.x and j =
                                                                foreach i within 1.
                newActual[i,j] := 0 foreach i within 1...width, j within 1...height
                newCameFrom[i,j] := (x : 0, y : 0) foreach i within 1...width, j wi

                searchResults := search((open : [start], closed : [], estimate : ne
                shortestPath := path(searchResults.cameFrom, start, end) ++ [end];
        in
                "No Path Found" when size(searchResults.open) = 0 else
                "Path: " ++ toString(shortestPath) ++ "\nCost:" ++
                toString(searchResults.actual[end.x, end.y]) ++ "\nMap:\n" ++ join(

path(cameFrom(2), start, current) :=
        let
                next := cameFrom[current.x, current.y];
        in
        [] when current = start else
        path(cameFrom, start, next) ++ [next];

drawMap(barriers(1), path(1), width, height)[i,j] :=
```

```
        '#' when elementOf((x:i, y:j), barriers) else
        'X' when elementOf((x:i, y:j), path) else
        '.' foreach i within 1 ... width, j within 1 ... height;

search(state, barriers(1), end) :=
        let
                nLocation := smallestEstimate(state.open, state.estimate, 2, 1, sta
                n := state.open[nLocation];
                neighbors := createNeighbors(n, allNeighbors, size(state.actual), s
                startState := (open : state.open[1...nLocation-1] ++ state.open[nLo
                                           estimate : state.estimate, actual : stat
                newState := findOpenNeighbors(n, startState, barriers, end, neighbo
        in
        state when size(state.open) = 0  else
        state when n = end else
        search(newState, barriers, end);

smallestEstimate(open(1), estimate(2), index, minIndex, minEstimate) :=
        let newEstimate := estimate[open[index].x, open[index].y]; in
        minIndex when index > size(open) else
        smallestEstimate(open, estimate, index + 1, minIndex, minEstimate) when new
        smallestEstimate(open, estimate, index + 1, index, newEstimate);

findOpenNeighbors(n, state, barriers(1), end, neighbors(1)) :=
        let
                neighbor := head(neighbors);
                cost := 1 + n.cost;
                candidate := state.actual[n.x, n.y] + calculateCost(barriers, n, ne
        in
                state when size(neighbors) = 0 else
                findOpenNeighbors(n, state, barriers, end, tail(neighbors)) when el
                findOpenNeighbors(n, state, barriers, end, tail(neighbors)) when el
                findOpenNeighbors(n, (open : state.open ++ [neighbor], closed : sta
                        cameFrom : setMap(state.cameFrom, neighbor, n),
                        estimate : setMap(state.estimate, neighbor, candidate + heu
                        actual : setMap(state.actual, neighbor, candidate)),
                        barriers, end, tail(neighbors));

createNeighbors(n, p, w, h) :=
        let
                x := n.x + p.x;
                y := n.y + p.y;
        in
                (x : x, y : y) when x >= 1 and x <= w and y >= 1 and y <= h;
```

```
calculateCost(barriers(1), start, end) := 100 when elementOf(end, barriers) else 1;

heuristic(start, end) :=
        let
                dx := abs(start.x - end.x);
                dy := abs(start.y - end.y);
        in
                (dx + dy) - min(dx, dy);

setMap(map(2), point, value)[i,j] :=
        value when point.x = i and point.y = j else
        map[i,j] foreach i within 1 ... size(map), j within 1 ... size(map[1]);
```

Output

```
Path: [(x:1,y:1),(x:2,y:2),(x:3,y:3),(x:4,y:2),(x:5,y:2),(x:6,y:2),(x:7,y:3),(x:7,y
Cost:11
Map:
X.......
.X......
..X.###.
.X#...#.
.X#...#.
.X#####.
..XXXXX.
.......X
```

Sidef (/wiki/Category:Sidef)

Translation of: Python

```
class AStarGraph {

    has barriers = [
        [2,4],[2,5],[2,6],[3,6],[4,6],[5,6],[5,5],[5,4],[5,3],[5,2],[4,2],[3,2]
    ]

    method heuristic(start, goal) {
        var (D1 = 1, D2 = 1)
        var dx = abs(start[0] - goal[0])
        var dy = abs(start[1] - goal[1])
        (D1 * (dx + dy)) + ((D2 - 2*D1) * Math.min(dx, dy))
    }

    method get_vertex_neighbours(pos) {
        gather {
            for dx, dy in [[1,0],[-1,0],[0,1],[0,-1],[1,1],[-1,1],[1,-1],[-1,-1]] {
                var x2 = (pos[0] + dx)
                var y2 = (pos[1] + dy)
                (x2<0 || x2>7 || y2<0 || y2>7) && next
                take([x2, y2])
            }
        }
    }

    method move_cost(_a, b) {
        barriers.contains(b) ? 100 : 1
    }
}

func AStarSearch(start, end, graph) {

    var G = Hash()
    var F = Hash()

    G{start} = 0
    F{start} = graph.heuristic(start, end)

    var closedVertices = []
    var openVertices = [start]
    var cameFrom = Hash()

    while (openVertices) {
```

```
    var current = nil
    var currentFscore = Inf

    for pos in openVertices {
        if (F{pos} < currentFscore) {
            currentFscore = F{pos}
            current = pos
        }
    }

    if (current == end) {
        var path = [current]
        while (cameFrom.contains(current)) {
            current = cameFrom{current}
            path << current
        }
        path.flip!
        return (path, F{end})
    }

    openVertices.remove(current)
    closedVertices.append(current)

    for neighbour in (graph.get_vertex_neighbours(current)) {
        if (closedVertices.contains(neighbour)) {
            next
        }
        var candidateG = (G{current} + graph.move_cost(current, neighbour))

        if (!openVertices.contains(neighbour)) {
            openVertices.append(neighbour)
        }
        elsif (candidateG >= G{neighbour}) {
            next
        }

        cameFrom{neighbour} = current
        G{neighbour} = candidateG
        var H = graph.heuristic(neighbour, end)
        F{neighbour} = (G{neighbour} + H)
    }
}

die "A* failed to find a solution"
```

```
}

var graph = AStarGraph()
var (route, cost) = AStarSearch([0,0], [7,7], graph)

var w = 10
var h = 10

var grid = h.of { w.of { "." } }
for y in (^h) { grid[y][0] = "█"; grid[y][-1] = "█" }
for x in (^w) { grid[0][x] = "█"; grid[-1][x] = "█" }

for x,y in (graph.barriers) { grid[x+1][y+1] = "█" }
for x,y in (route)          { grid[x+1][y+1] = "x" }

grid.each { .join.say }

say "Path cost #{cost}: #{route}"
```

Output:

```
██████████
█x.......█
█.x......█
█..x.███.█
█.x█...█.█
█.x█...█.█
█.x█████.█
█..xxxxx.█
█.......x█
██████████
Path cost 11: [[0, 0], [1, 1], [2, 2], [3, 1], [4, 1], [5, 1], [6, 2], [6, 3], [6,
```

zkl (/wiki/Category:Zkl)

Translation of: Python

```
   // we use strings as hash keys: (x,y)-->"x,y", keys are a single pair
fcn toKey(xy){ xy.concat(",") }

fcn AStarSearch(start,end,graph){
   G:=Dictionary(); # Actual movement cost to each position from the start position
   F:=Dictionary(); # Estimated movement cost of start to end going via this positi
      #Initialize starting values
   kstart:=toKey(start);
   G[kstart]=0;
   F[kstart]=graph.heuristic(start,end);
   closedVertices,openVertices,cameFrom := List(),List(start),Dictionary();

   while(openVertices){
      # Get the vertex in the open list with the lowest F score
      current,currentFscore := Void, Void;
      foreach pos in (openVertices){
         kpos:=toKey(pos);
         if(current==Void or F[kpos]<currentFscore)
            currentFscore,current = F[kpos],pos;

         # Check if we have reached the goal
         if(current==end){   # Yes! Retrace our route backward
            path,kcurrent := List(current),toKey(current);
            while(current = cameFrom.find(kcurrent)){
               path.append(current);
               kcurrent=toKey(current);
            }
            return(path.reverse(),F[toKey(end)])   # Done!
         }

         # Mark the current vertex as closed
         openVertices.remove(current);
         if(not closedVertices.holds(current)) closedVertices.append(current);

         # Update scores for vertices near the current position
         foreach neighbor in (graph.get_vertex_neighbors(current)){
            if(closedVertices.holds(neighbor))
               continue; # We have already processed this node exhaustively
            kneighbor:=toKey(neighbor);
            candidateG:=G[toKey(current)] + graph.move_cost(current, neighbor);

            if(not openVertices.holds(neighbor))
               openVertices.append(neighbor); # Discovered a new vertex
```

```
            else if(candidateG>=G[kneighbor])
               continue; # This G score is worse than previously found

            # Adopt this G score
            cameFrom[kneighbor]=current;
            G[kneighbor]=candidateG;
            F[kneighbor]=G[kneighbor] + graph.heuristic(neighbor,end);
         }
      }
   } // while
   throw(Exception.AssertionError("A* failed to find a solution"));
}
```

```
class [static] AStarGraph{   # Define a class board like grid with barriers
   var [const] barriers =
      T(       T(3,2),T(4,2),T(5,2),   // T is RO List
                            T(5,3),
         T(2,4),             T(5,4),
         T(2,5),             T(5,5),
         T(2,6),T(3,6),T(4,6),T(5,6) );
   fcn heuristic(start,goal){  // (x,y),(x,y)
   # Use Chebyshev distance heuristic if we can move one square either
   # adjacent or diagonal
      D,D2,dx,dy := 1,1, (start[0] - goal[0]).abs(), (start[1] - goal[1]).abs();
      D*(dx + dy) + (D2 - 2*D)*dx.min(dy);
   }
   fcn get_vertex_neighbors([(x,y)]){      # Move like a chess king
      var moves=Walker.cproduct([-1..1],[-1..1]).walk();  // 8 moves + (0,0)
      moves.pump(List,'wrap([(dx,dy)]){
         x2,y2 := x + dx, y + dy;
         if((dx==dy==0) or x2 < 0 or x2 > 7 or y2 < 0 or y2 > 7) Void.Skip;
         else T(x2,y2);
      })
   }
   fcn move_cost(a,b){  // ( (x,y),(x,y) )
      if(barriers.holds(b))
         return(100); # Extremely high cost to enter barrier squares
      1 # Normal movement cost
   }
}
```