Introduction

Ricochet Robots

Beskrivelse af spillet - Bevæger sig som rooks i skak

Goals

Previous Solutions

Baseline

Et navn for mål-robotten Et navn for ikke-mål-robotterne Et state: En robot position Et state i søgetræet, s S

Solutions

JPS+

Algorithm

Analysis

Graph-DP

Define basics, Graph G, Fields F, K = 16, GR, OR,

Algorithm

The base idea of the graph based algorithm is to construct a graph, G, for the entire board with directional edges for each vertex and represent this in a two dimensional array. Constructing the graph is done in two passes with dynamic programming.

A basic assumption in this solution is that all moves by the Obstacle Robots, *OR*, can be applied before the actual Goal Robot, *GR*, moves towards goal. Thus, the *GR* can depend on *OR* states while *OR* can depend upon all other *OR* states but only on the *GR*'s starting state.

Afterwards, the solution will move the *OR* in all possible directions, evaluating the search tree with a branching factor of 9^n where n is the number of moves evaluated so far. The combined *OR* state is checked against a hash set of current

states evaluated up to this level, and recurences will not be evaluated again. For each new state, G updated with the positions of OR and a BFS is run with GR's vertex as root. A heuristic function is used to evaluate the current best solution compared to number of moves evaluated so far.

G is represented as a two dimensional array of Vertices V, where each $v \quad V$ has a list of Edges E. Each edge $e \quad E$ has a pointer to the next Vertex and a directional indicator (i.e. North, South, East, West) d, while v has at most one edge for each d, thus $sum(e) <= sum(d)$. //TODO

Construct Graph

The graph is constructed in two passes, where the first pass is conducted from the upper left corner to the lower right corner of the board, where each Vertex *(i,j)* v is added edges based on the corresponding edges at the given directions d *{ West, North }* if the Field *(i,j)* has no obstacles in the given directions. Thus, the respective vertices at *(i-1,j)* and *(i,j-1)* will be investigated. As an example, in case Field *(i,j)* can move in direction d = *North* and Vertex *(i-1,j)* has no edge in direction d, Vertex *(i-1,j)* will be the child of the edge e at direction d for vertex v. The opposite is the case for the second pass starting at the bottom right corner where $d \quad$ *{ East, South }*.

```
func ADD-EDGES(i, j, D, G, F)
1   For d in D
2       if CAN-MOVE(F[i,j], d)
3           v = GET-CHILD(i, j, G, d)
4           ADD-EDGE(G[i,j], d, v)

func GET-CHILD(i, j, G, d)
1   v = NEXT-VERTEX(i,j,G,d)
2   e = GET-EDGE(v,d)
3   if e == NIL
4       return v
5   else
6       return e.child
```

The primary functions in the graph construction phase.

The function ADD-EDGES(..), given the indices (i,j) of the current vertex, the directions to investigate D, the graph G and the representation of the fields of the board F, processes each vertex once for both passes, moving from upper left to bottom right corner and back, while only the directions D in the input is changed.

PlaceRobot()

The graph has to be adapted to the current robot positions before the BFS can be applied for the *GR*. Thus, the *OR*'s has to be inserted into the graph and the graph updated to conform to these changes. All edges pointing towards or past *(i,j)* has to be updated. When placing a robot on a given field *(i,j)*, all vertices V in direcion d until a obstacle is met need to update edges in the opposite direction of d. Thus, *d' = opposite(d)*. For all edges e where *direction = d'* the child property has to be set to the vertex immidiately adjacent to *Vertex[i,j]* in d. Thus, in case d = *North*, for all e set *child = Vertex[i-1,j]*. My algorithm finds the most distant vertex in direction d in constant time and loops back visiting each vertex in V only once for each direction. However, several edge cases exist when finding all V as illustrated in figure ***FIGUR NUMMER?*** and listed below.

ILLUSTRATE THE FOUR CASES

1. *Field[i,j]* cannot move in direction d. No edges has to be updated.
2. *Field[i,j]* can move in direction d:
    1. *Vertex[i,j]* has no edge in direction d. Thus, another robot stands on the adjacent field in direction d.
    2. *Vertex[i,j]* has an edge e in direction d. The field for *e.child* cannot move in direction d. Thus, *e.child* is the vertex furthest away from *Vertex[i,j]* in direction d.
    3. *Vertex[i,j]* has an edge e in direction d. The field for *e.child*, f, can move in direction d. Thus, a robot is present on the adjacent field to f in direction d.

The algorithm for finding the most distant vertex for position *(i,j)* given the fields F, the graph G and the current direction d to process returns either NIL or the distant vertex, and it is present in ***ILLUSTRATION XX***. Please notice the notation *d'* which refers to *d' = opposite(d)*.

```
DISTANT-VERTEX(i,j,F,G,d)
1   if NOT CAN-MOVE(F[i,j])
2       return NIL
3   e = GET-EDGE(G[i,j], d)
4   if e1 == NIL
5       v = NEXT-VERTEX(i,j,G,d)
6       e2 = GET-EDGE(v, d')
7       REMOVE-EDGE(v,e2)
8       return NIL
9
10  if CAN-MOVE(F[e.child.i, e.child.j],d)
11      return NEXT-VERTEX(e.child.i, e.child.j, d)
12  else
13      return e.child
```

Algorithm for finding the most distant vertex or NIL in case Vertex[i,j] does not have an edge in direction d.

Line 1-2 validates if it's possible to move in direction *d* for *Field[i,j]* and returns NIL if it's not the case, since no edge will be added in direction *d* for *Vertex[i,j], v,* then. Line 3-8 gets the edge *e* of *v* in direction *d*, and in case none is present, a robot is standing on the next field. Therefore, the vertex found on line 5 has to remove it's edge in direction *d'* since the placed robot will be blocking that direction. In line 10-13 it's determined if the child of *e* has an adjacent robot in direction *d*. Thus, *e.child* or the adjacent vertex, where the robot is placed, is returned.

The next rutine validate all directions and update the child properties of all edges in direction *d'* for all vertices in *V*.

```
func PLACE-ROBOT(i,j,F,G)
1   for d in directions
2       v1 = DISTANT-VERTEX(i,j,F,G,d)
3       if v1 == NIL
4           NEXT d
5       v2 = NEXT-VERTEX(i,j,G,d)
6       do
7           e = GET-EDGE(v1, d')
8           if v1 == v2
9               REMOVE-EDGE(v1,e)
10          else
11              e.child = v2
12          v1 = NEXT-VERTEX(i,j,G,d')
13      while v1 != v2
```

Algorithm for validation all directions and updating the child properties for each vertex affected by the graph transformation.

In line 1 each direction in *{ North, East, South, West }* is evaluated. Line 2-4 retrieves the distant vertex and continues the loop in case none is found. Line 5 finds the new child-vertex for each vertex in *V - {v2}* ***TODO: Minus som side 595 i BFS***. Line 7 finds the related edge *e* for direction *d'*. Line 8-9 together with line 13 is the termination clause, in case *v1 == v2*, *e* is removed from *v1* (and *v2*, since they references the same vertex) and the loop terminates. Else, in line 10-12 the child property of *e* is updated with *v2* and the next vertex to process is found.

RemoveRobot

Heuristic function

Analysis

Experimental results

Setup

Results