Introduction

Ricochet Robots

Beskrivelse af spillet - Bevæger sig som rooks i skak

Goals

Baseline

Let n be the dimensions of the board where $n = 16$. Let F be the set of fields where *F[i,j]* refers to the field at position *(i,j)* starting from the top-left corner. Each field contains information about obstacles in each direction.Let R be the set of robots, let GR be the Goal Robot to reach the goal and let OR (Obstacle Robots) be the set of Robots $\in R - \{GR\}$.

A robot state refers to the position of the robot and the required moves to reach the destination, while a game state refers to the complete set of robot states for a current configuration.

The set of directional indicators, D refers to the possible directions on the gameboard. Thus D = *{ North, East, South, West }*.

Each algorithm is given the board of fields F, the set of Robots R and the goal to reach G as input.

Previous Solutions

Several solutions exist for Ricochet Robots and the two most common are described below.

Naive algorithm

The naive algorithm searches the entire search tree in an incremental order and guarantees an optimal solution. The board representation is a two dimensional array with an additional attribute for each field indicating if a robot stands on the given field. A first-in first-out queue is used for keeping track of the to-be processed game states. Since this solution would never be feasable for solutions requiring more than a couple moves, a hash table is introduced to keep track of already searched game states. Therefore, duplicate game states will be pruned from the search tree.

Moving a robot in direction d processes all fields until an obstacle is met. It uses $O(|n|)$ time. This is done for each robot for each game state. Each robot

is adjacent to at least one obstacle after the initial move. Thus, a branching factor of $|R| \cdot 3 = 12$ is given, where R is the set of robots.

Processing each game state takes $O(12 \cdot |n|)$ time and finding the optimal solution takes $O(12^k \cdot 12 \cdot |n|) = O(12^{k+1} \cdot |n|)$ time where k is the number of moves. The algorithm uses $O(n^2)$ space for the additional attribute for each field, $O(12^k)$ space for the queue and $O(\sum_{i=1}^{k} 12^i) \sim O(12^k)$ for the hash table. The worst case space impact of the algorithm is $O(12^k)$.

IDDFS

The Iterative Deepening Depth First Search algorithm is the claimed to be fastest solver and guarantees an optimal solution. The algorithm uses the same board representation as the naive algorithm including the additional attribute for flagging robot locations. In the following, h will refer to the height of the remaining search for the iteration. Thus, $h = MAX - depth$, where MAX incrementing search limit while $depth$ is the distance to the root in the search tree. The IDDFS is expanded with two additional pruning techniques:

- A hash table is used much like in the naive algorithm, and game state s and h are stored. If s has been reached before by the same height or heigher, it is pruned from the search.
- A two-dimensional array, min storing the minimum number of moves to goal from each field on the board assuming the robots can change direction without bouncing off an obstacle as seen in ***FIGUR SOMETHING***. If $min[GR] > h$ the search is pruned. If $h = min[GR]$ only the GR is processed further.

The minimum moves for each field is computed with the goal as the root. The number of moves for the root is set to 0. Each direction is processed incrementing the number of moves with 1. For each direction, all fields is processed until an obstacle or a field with a lower minimum number of moves is met. Due to this incremental movement, every field is only queued once but can be visited several times during the computation. When processing a single field, only two directions is relevant since the field was discovered from one of the other two directions and therefore, these directions will have lower minimum moves. When processing a single field, only $O(|n|)$ fields can be visited. Since all fields is queued exactly once, the minimum moves is computed in $O(|n|^2 \cdot |n|) = O(|n|^3)$ time.

Moving each robot uses $O(|n|)$ time like the naive algorithm

- worst case for IDDFS

Solutions

JPS+

Algorithm

Analysis

OVERSRKFIT

Graph-DP

In the following the graph based algorithm will be presented. A basic assumption in this solution is that all moves by the *OR* can be applied before the *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.

The algorithm will update the graph with *OR* states and perform a BFS with the *GR* as the source vertex and store the result. For each investigated game state, all *OR* are moved in each possible directions and queued as a new to-be investigated game state. As each level in the search tree is discovered in an incremental order, the algorithm guaranties an optimal solution under the given assumption above. A hashtable is used for pruning the search tree for already searched game states.

Algorithm

Let G be the graph of the board where $G.V$ is the set of vertices in the graph where *G.V[i,j]* refers to the vertex at position *(i,j)* at the board starting from the top-left corner. Let $G.E$ be all directional edges in G and *G.Adj[v]* refers to all edges where the source is the vertex v. G will be represented in a dimensional array with directional edges stored in an adjacency list for each vertex. Constructing the graph is done in two passes with dynamic programming. Let $v \in G.V$ and let $E = G.Adj[v]$. Each edge $e \in E$ has a pointer to the next vertex, in the following refered to as the *child* attribute, and a directional indicator $d \in D$, while v has at most one edge for each d, then $|E| \leq |D|$ is given.

Throughout the algorithm, multiple support functions is used and excluded for ease of reading. They can be seen in the list below with a short description.

- *CAN-MOVE(Field, Direction)*: Returns if it is possible to move to the next field in the given direction.
- *NEXT-VERTEX(i,j,G,Direction)*: Returns the adjacent vertex in the given direction for the given coordinates.

- *GET-EDGE(Vertex,Direction)*: Returns the edge for the vertex with the given direction or NIL.
- *OPPOSITE(Direction)*: Returns the opposite direction of the given input. I.e., given North, South is returned.
- *ADD-EDGE(Vertex1, Direction, Vertex2)*: Adds an edge to Vertex1 in the given direction and Vertex2 as the child property.
- *REMOVE-EDGE(Vertex, Edge)*: Removes Edge from the adjacency list of Vertex.

Construct Graph

The graph is constructed in two passes, where the first pass is conducted in a topdown left to right approach. In the first pass, each *G.V[i,j] v* is added edges based on the corresponding edges at the given directions $d \in$ *{ West, North }* if the *F[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 *F[i,j]* can move in direction $d =$ *North* and *G.V[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 in a bottom up right to left approach where $d \in$ *{ East, South }*.

```
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.V[i,j], d, v)

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.

In ADD-EDGES(..) line 1, each direction d in the given set of directions, D, is iterated once. In line 2, it is validated if an obstacle exists in the given direction while line 3-4 retrieves the to-be child vertex for *G.V[i,j]* for the edge in direction d.

Adapting the Graph

The graph has to be adapted to the current *OB*'s positions before the BFS can be applied for the *GR*. Let *(i,j)* be the position where an robot has to be placed.

When placing a robot on *(i,j)*, all vertices V in direcion d until a obstacle is met need to update edges in the opposite direction of *d*. In the following, *d' = OPPOSITE(d)*. For all edges $e \in G.Adj[V]$ where $e.d == d'$ the child property has to be set to the vertex immidiately adjacent to *G.V[i,j]* in direction *d*. Thus, in case *d = North*, for all *e* set *e.child = G.V[i-1,j]*. My algorithm finds the most distant vertex in direction *d* in *O(1)* time and updates all $v \in V$ in $O(|V|)$ time. However, several edge cases exist when finding all V as illustrated in figure ***FIGUR NUMMER?*** and listed below.

ILLUSTRATE THE FOUR CASES

1. *F[i,j]* cannot move in direction *d*. No edges has to be updated.
2. *F[i,j]* can move in direction *d*:
    1. *G.V[i,j]* has no edge in direction *d*. Thus, another robot stands on the adjacent field in direction *d*.
    2. *G.V[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 *G.V[i,j]* in direction *d*.
    3. *G.V[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*.

As given in ***ILLUSTRATION XX*** the procedure for finding the most distant vertex given position *(i,j)*, the fields *F*, the graph *G* and the current direction *d* to process returns either NIL or the distant vertex.

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

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

Line 1-2 validates if it's possible to move in direction d for *F[i,j]* and returns NIL if it's not the case, since no edge will be added in direction d for *G.V[i,j], v,* then. Line 3-9 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 11-14 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.

```
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       d' = OPPOSITE(d)
7       do
8           e = GET-EDGE(v1, d')
9           if v1 == v2
10              REMOVE-EDGE(v1,e)
11          else
12              e.child = v2
13          v1 = NEXT-VERTEX(i,j,G,d')
14      while v1 != v2
```

Algorithm for validation all directions and updating the child properties for each edge 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 $V - \{v2\}$. Line 8 finds the related edge e for direction *d'*. Line 9-10 together with line 14 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 11-13 the child property of e is updated with *v2* and the next vertex to process is found.

Placing and removing a robot from the board involves many of the same features:

1. Identify relevant directions to investigate.
2. Identifying relevant vertices to update.
3. Add or update edges for all affected vertices.

Thus, the REMOVE-ROBOT(..) function will not be described deeper during this section. However, much like PLACE-ROBOT(..), updating each affected vertex takes *O(1)* and for each direction, the algorithm takes $O(|V|)$ time.

Solver

In the graph-based solver the search tree is expanded one level at a time. A first-in, first-out queue Q is used to manage the set of robot moves for each possible robot state, hence the set of robot states in a given configuration will be refered to as a game state. A game state s holds a list of OR states and an attribute, $s.moves$, representing the total number of moves required to achieve the given game state. However, since many move combinations will give the same game state with the same or higher number of moves required, a hash table, m, is introduced to ensure that each game state with the exactly same robot positions is evaluated only once. Thus, redundant states is pruned from the search tree.

While the possible combinations is ever increasing, a heuristic is introduced to ensure that when a best solution is found, the solver teminates, discarding the remaining uninvestigated game states. Each time a possible solution is found, if it is better than the known best solution, it is stored in the datastructure B in which we stores the result of the BFS in the attribute $B.bfs$, the required game state for the BFS in $B.s$ and the combined required moves from the BFS and the gamestate in $B.best$. All this gives the folowing heuristic:

> B is the optimal solution if $B.best < (s.moves + 2)$, where s is the curent investigated game state. Since the number of moves is investigated in a incremental order, and since s requires at least one additional move for the goal robot to reach goal, s will never be a better solution than B, and therefore the solver terminates when this condition is fulfilled.

Given the fields F, the goal to reach, $Goal$, the set of OR and the GR, the best solution is found under the assumption that all OR moves can be applied before the GR moves.

```
SOLVER(F, Goal, OR, GR)
1   m = ∅
2   Q = ∅
3   G = BUILD-GRAPH(F)
4   s = GAME-STATE(OR)
5   s.best = 0
6   B.best = unlimited
7
8   ADD-STATE(m,s)
9   ENQUEUE(Q,s)
10  while Q != EOF
11      s = DEQUEUE(Q)
12
13      if B.best < (s.moves + 2)
```

```
14          return B
15
16      for R in s.OR
17          PLACE-ROBOT(R, G, F)
18      r = BFS(G,GR, Goal)
19
20      if r != NIL V (r.d + s.moves) < B.best
21          B.best = r.d + s.moves
22          B.s = s
23          B.bfs = r
24
25      PLACE-ROBOT(GR, G, F)
26      for R in s.OR
27          MOVE-ROBOT(R, Q, G, F, m)
28      for R in s.OR
29          REMOVE-ROBOT(R, G, F)
30      REMOVE-ROBOT(GR, G, F)
```

FIGUR SOMETHING illustrates the progress of the Solver on a given gameboard configuration.

Line 1-2 instantiates the queue and the hashtable while line 3 builds the graph as described earlier. Line 4-6 creates the initial game state and sets the default values for the game state and the best overall result. Line 8-11 adds the game state to the hash table, enqueues the state and dequeues the first element of the queue. Notice that the while condition on line 10 is not the primary termination state, which appears on line 13-14, where the heuristic is applied to the current game state. At line 16-18 the *OR* is placed on the board and the BFS is conducted with *GR* as the source vertex. The BFS is slightly modified to return the node corresponding to the given Goal Field or NIL if not found during the search. Line 20-23 validates the BFS result and in case the sum of the game state moves and the BFS distance to goal is lower than the previous result, B is updated. The *B.bfs* attribute is stored for backtracking the actual solution. Line 25-27 updates G with the position of the *GR* before the moving phase, where all $r \in OR$ is moved once for every edge in $G.Adj[r']$, where *r'* is the corresponding vertex for each *OR*. In the function MOVE-ROBOT(..) each new game state created is validated against m in *O(1)* to ensure no redundant entries. In line 28-30 all robots are removed from G, thus bringing it back to default state before next game state is evaluated.

Analysis

Processing each field uses $O(1)$ time, and there is $n \cdot n$ fields. Thus each pass takes $O(n^2)$ time. Hence the complexity of the construction phase is $O(2 \cdot n^2) \cdot O(1) = O(n^2)$. The memory used is $O(n^2) \cdot O(|e|)$ where $|e|$ is the number of edges for each vertex. Since e $\sim$ 4, the memory used is $O(n^2)$.

Placing and removing a robot in the graph affects at most $2 \cdot n$ vertices, which is the case where the robot is placed where no obstacles nor other robots exist for the given row and column. Finding the distant vertex takes *O(1)* time, thus, the total time for placing or removing a robot from the graph uses $O(2 \cdot n) \cdot O(1) = O(n)$ time.

For the solver, each game state processed requires 4 robot placements and removals thus $O(8 \cdot n)$ time. The BFS worst case is $O(n^2) \cdot O(e)$, but given the nature of the movement limitations, the number of visited vertices is approximately $O(4 \cdot n)$. ***Can I make this assumption based on experimental results?*** During the search, a two dimensional matrix sized $n \cdot n$ is used to maintain state of the BFS, but is discarded when the result is returned. Thus, only a temporarily matrix of size $O(n^2)$ is used which also approximates $O(4 \cdot |n|)$ in memory impact due to the nature of the graph. Moving each robot creating a new game state uses $O(|OR|) \cdot O(|e'|)$ time where $/e'/$ is the number of edges for each *OR*. Given the nature of the game, after the initial move, $e' \sim 3$ since each *OR* has bounced of some obstacle. In addition, the cost of looking up each state at $O(1)$ time and adding a new state uses constant time. Thus, creating new game states uses $O(|OR| \cdot |e'|)$ time.

Thus, for each processed game state, the time impact is $O(n) + O(n) + O(|OR| \cdot |e'|) = O(n)$.

The branching factor is given by the number of new states created for each game state which given the assumptions above equals $k^{|OR| \cdot |e'|} \sim k^9$ where k is the number of total moves required for the *OR*. The memory impact of the queue is $O(k^9)$. Thus, the worst case time used by the solver is $O(n^2) + O(k^9 \cdot n) = O(k^9 \cdot n)$ and the memory impact is $O(n^2 + n + k^9)$.

Notes

It's actually not k^9 since the hashtable prunes the search tree, making it more like k^x where x decreases as k increases. How to include this?

Graph v2

Just notes for future improvements

v1 Move all robots in each direction - including the goal robot. v2 Calculate the minimum number of moves from each field to goal. Use this as a heuristic to eliminate In case the goal robot cannot move to goal in the same or less than the combined solution given by the BFS + current state. Will prune MANY states away.

Combine v1 + v2 for an always optimal solution.

Possible optimization: Prune BFS in case it's not possible to reach the goal within the number of remaining moves up to the best known optimal solution. A lot of BFS's will return prematurely then.

The tricky part: Use the robo solver for finding an estimated maximum up front. The lowest count is gurantied optimal or larger. Might be used for pruning the BFS search up front.

Experimental results

Setup

Results