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.

To move a robot, all fields in a given direction is processed until an obstacle is met. It uses $O(n)$ time. This is done for each robot for each direction for each game state. Processing each game state takes $O(b \cdot n)$ time where b is the branching factor.

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

Each robot is adjacent to at least one obstacle after the initial move. Therefore the branching factor is given by $b = |R| \cdot 3 = 12$.

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 is the incrementing search limit while $depth$ is the distance to the root in the search tree. The IDDFS includes two additional pruning techniques:

- A hash table stores a combination of game state s and h. If s has been reached before by the same height or heigher, it is pruned from the search.
- A two-dimensional array stores the minimum number of moves from each field to goal. This is refered to as *min*. It is assumed that 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 in the search tree.

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 is guaranteed to be more optimal. 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 and uses $O(n^2)$ space.

Finding the optimal solution uses $O(b^k \cdot n + n^3)$ time where b is the branching factor and k is the number of moves. However, the space impact of the search itself is only $O(k)$ since the algorithm is recursive. The space analysis of the hash table is the same as in the naive algorithm. The worst case space impact is $O(b^k + k + n^2) = O(b^k + n^2)$.

Solutions

Stateless

This algorithm is an attempt to solve the game without keeping track of each game state. The obvious advantage of this is to eliminate the branching factor's impact on the running time. Several tradeoffs are used, and the algorithm does not guarantee an optimal solution. However, the algorithm guarantees that the optimal number of moves is either the same as the result of the algorithm or less when a solution is found.

Algorithm

The algorithm stores information about every time a robot interacts with a field. The interaction can be split in two - when the robot moves over the field or when it lands on the field. In the latter, landing on a field is denoted a *final state* while moving over a field is denoted an *intermediate state*. Finale states are queued in a min-heap queue, ordered by the number of moves. These states represent only the single robot's state, thus for each element in the queue, only one robot is processed. The robot is moved in all directions followed by a validation if the goal has been reached. A heuristic is introduced as the termination state. Let r be the current best result and k be the current searched depth, then the search is terminated if $r \leq (k + 2)$.

The algorithm stores information about every time a robot interacts with a field. The interaction can be split in two - when the robot moves over the field or when it lands on the field. Each of these are described below:

- **Intermediate states**: An intermediate robot state is when a robot moves past a field and no obstacle nor robot is met. Thus, the robot can reach on the given field, but it cannot land on it yet. Therefore, a intermediate robot state is stored on the field for the given robot, it's direction and the current amount of moves required for the robot to reach this field.
- **Final states**: A final robot state is given when a robot bounces off an obstacle or another robot. The final robot state is stored on the field for the given robot and the number of required moves to reach it. The robot state is also queued for further movement.

Instead of treating each move with a robot as individual game states, all moves are stored for the given fields

Hvad gør den stateless? Den rykker sig i alle retninger samtidig! Den bruger data på brættet i stedet for at huske hele game states assumptions - mange tradeoffs

Hvad er tilføjet for hvert eneste felt? Prioritized queue

Moving

Update adjacent fields

Hvert move koster |n| -> hvert niveau koster b * |n| At fine resultatet koster k * b * |n| * prioritized queue insert/remove Pris? Additional memory for each field...

Vi ha

The stateless algorithm is an attempt to avoid the exponential running times of most solutions at the cost of increased space usage. Stateless refers to that each robot move only depends on other moves that affects the robot itself. Thus,

Algorithm

Analysis

Graph v1

In the following the graph based algorithm will be presented. The algorithm assumes that all moves by the *OR* can be applied before the *GR* moves towards goal. The *GR* can depend on *OR* states but *OR* can only depend on the starting state of *GR*.

The algorithm will update the graph with *OR* states and perform a BFS with the *GR* as the source vertex and stores the result. Like the naive algorithm, each *OR* are now moved to new positions and new game states are queued in a first-in, first-out queue. As each level in the search tree is discovered in an incremental order, the algorithm guaranties an optimal solution under the given assumption above. As with the naive algorithm, a hash table is used to prune the search tree.

Algorithm

Let G be the graph representing 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 let *G.Adj[v]* refer to all edges where the source is the vertex v. G will be represented in a two 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, later refered to as the *child* of e. Each e is also denoted a directional indicator $d \in D$. While v has at most one edge for each direction, then $|E| \leq |D|$ is given.

Construct Graph

The graph is constructed in two passes, where the first pass is conducted in a topdown left to right approach. The first pass processes edges in the directions $d \in \{West, North\}$. Each *G.V[i,j]* is added an edge in direction *d* if *F[i,j]* has no obstacle in direction *d*. In the case of $d \in \{West, North\}$, the respective vertices *G.V[i,j-1]* and *G.V[i-1,j]* will be investigated. As an example, in case *F[i,j]* can move in direction *d = North* and *G.V[i-1,j]* has an edge in direction *d*, the child of that edge will also be the child of the edge in direction *d* for vertex *G.V[i,j]*. If *G.V[i-1,j]* has no edge in direction *d*, *G.V[i-1,j]* will be the child of the edge at direction *d* for vertex *G.V[i,j]*. The second pass processes edges where *d* ∈ *{ East, South }* in a bottom-up right to left approach.

Adapting the Graph

The graph has to be adapted to the current *OR*'s positions before the BFS can be applied for the *GR*. Let *(i,j)* be the position where an robot has to be placed. Let *d* be a direction from *(i,j)* and let *V* be the set of vertices in direction *d* which need to be adjusted to the new board configuration. All vertices in *V* need to update the edges in the opposite direction of *d*, in the latter refered to as *d'*.

Let *v'* be the vertex immidiately adjacent to *G.V[i,j]* in direction *d*. Let *E* be the set of $e \in G.Adj[V - \{v'\}]$ where $e.d = d'$. The child property of alle edges in *E* has to be set to *v'* while the edge of *v'* in direction *d'* has to be removed.

To find the set *V*, the most distant vertex is found and then all vertices in direction *d'* is processed until *G.V[i,j]* is met. Several cases exist when finding the distant vertex which is illustrated in ***FIGUR SOMETHING*** 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*:
   a. *G.V[i,j]* has no edge in direction *d*. Thus, another robot stands on the adjacent field in direction *d*.
   b. *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 most distant vertex from *G.V[i,j]* in direction *d*.
   c. *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*.

Given the position *(i,j)*, the fields *F* and the graph *G*, all directions are processed and the graph is updated with the new robot position.

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, this functionality will not be described deeper in this section.

Solver

In the graph-based solver the search tree is expanded one level at a time. A first-in, first-out queue is used to store awaiting game states. In this case, a game state refers to a list of *OR* states and a property indicating the total number of moves for the given game state. Like the naive algorithm, a hash table is introduced to track already processed game states and prunes redundant states.

The solver keeps track of the best known solution so far in B. $B.s$ is the game state, $B.bfs$ is the result of the BFS and $B.best$ is the combined number of moves for the given solution. A heuristic is introduced as the termination state:

> 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.

The solver instantiates the hash table and constructs the graph. Then the initial game state is queued and a loop is performed until a solution is found. For each game state, the heuristic is applied. If the termination state is not fulfilled, the graph is updated with the the positions of the *OR* for the given game state. The BFS is applied and the result of the goal field is found. If it was reached by the *GR*, the combined result is compared to B and updated if better. Then, the graph is adapted to the position of the *GR*, and all the *OR* are moved in all directions. The new game states are validated against the hash table and added to the queue if they are not redundant. Finally, the robots are all removed from the graph and the next game state is processed.

Analysis

Processing a single field in the construction of the graph uses $O(1)$ time, and each pass uses $O(n^2)$ time. The space impact is given by $O(n^2 \cdot |D|) = O(n^2)$ since there is at most one edge per direction per vertex.

Placing and removing a robot in the graph affects at most $2 \cdot n$ vertices. This 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 uses *O(1)* time. The total time for placing or removing a robot from the graph uses $O(2n) = O(n)$ 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 given by the number of vertices and edges, resulting in $O(n^2 \cdot |D|)$. The BFS requires a two dimensional array to keep track of the search progress using $O(n^2)$ space. However, this is discarded when the BFS is completed and the result is returned.

Validating, inserting and enqueuing a new game state uses $O(1)$ time. Moving all robot uses $O(b)$ time, where b is the branching factor. Therefore, the time used for each processed game state is $O(8 \cdot n + n^2 \cdot |D| + b) = O(n^2)$. In total, the worst case time usage for the algorithm is $O(b^k \cdot n^2)$ where k is the number of moves.

The space impact of the queue and the hashtable is $O(b^k)$. The worst case space impact of the algorithm is $O(n^2 \cdot |D| + b^k + n^2) = O(b^k + n^2 \cdot (1 + |D|))$.

After the initial move, each OR is adjacent to at least one obstacle. The branching factor is given by $b = |OR| \cdot 3 = 9$.

Graph v2

The second version of the graph solver has a few improvements inspired by the other solvers. In the following, only the additions will be described.

Improvements

The graph construction, inserting and removal of robots in the graph is the same as bescribed for version 1 while the same basic assumptions as stated in version 1 applies. The additions is focused on the heuristic function and the BFS while the solver itself has no further additions.

Heuristic function

The heuristic function has been improved to facilitate an earlier termination of the solver. Using the minimum moves precomputation given in the IDDFS solver, the heuristic function has been updated to the following:

> B is the optimal solution if $B.best < (s.moves + min[GR])$, where s is the curent investigated game state and *min[GR]* is the minimum number of moves from the GR to goal. If a solution satisfies this condition, no possible other solution can reach goal in fewer moves than the given best hence the incremental order of the search.

Early termination of BFS

When a given solution is found, the BFS is still applied until the heuristic function is satisfied. However, when a solution is found, an upper limit for the possible number of moves is set, drastically reducing the search tree. The following optimizations are an attempt to do so:

1. The depth of the BFS can be limited by the best solution so far and the minimum required moves to goal. Let $depth = B.best - s.moves$, then it's given that $depth - 1$ is the maximum searchable depth in the BFS that can improve the current result. The BFS is terminated if this condition is fulfilled.
2. Let c be the current node processed in the BFS and let $c.d$ denote the distance to the root. If $(min[c] + c.d) \geq depth$ is true, the current searched branch in the BFS tree is obsolete and the rest of this branch is skipped. However, the BFS is not terminated.

Analysis

The minimum moves computation is the only addition to the worst case analysis. As already described in the IDDFS solver, it uses $O(n^3)$ time and $O(n^2)$ space. Therefore, the total worst case time usage is $O(n^3 + b^k \cdot n^2) = O(b^k \cdot n^2)$. Since the branching factor continues to be the dominant factor, the worst case time stays the same. The space impact of the algorithm is $O(b^k + n^2 \cdot (1 + |D|) + n^2) = O(b^k + n^2 \cdot (2 + |D|))$ and shows a slightly increase in the space impact.

Notes

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

Graph v2: 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.

In general for graphs: A function for obstacle count vs fields -> how much of the graph is actually searched. - Find the maximum number of obstacles for different sizes

Experimental results

Setup

Results