

Figure 1:

^Bare se bort fra denne illustration :-). Det løser noget formattering jeg ikke lige har fået styr på endnu.

## Introduction

Optimizing a single entity's cost is the most common approach in path finding problems. However, the board game *Ricochet Robots* present a rather unusual case where multiple entities cooperates for a single entity to reach the goal at minimal cost. Usually the map is considered static in path finding problems, however, in *Ricochet Robots* the map is altered for each entity move, resulting in an somewhat new problem. With multiple possible moves for each map, the search space increases rapidly.

The goal of this project is to survey and design state-of-the-art algorithms within graph search and dynamic programming. I will survey existing solutions for *Ricochet Robots*, and based on the experiences, design and implement new state-of-the-art algorithms. I will be implementing prototypes of each algorithm and analyse and evaluate each algorithm from both a theoretical and practical perspective.

## Ricochet Robots

Ricochet Robots is a played on a game board with  $16 \times 16$  tiles with obstacles scattered on the board. The board contains 17 goals. Four robots in different colors are placed on the board at random at the beginning of the game. The robots cannot be positioned at the same field nor on a goal field. Each goal is colored in relation to one of the four robots except for one of the goals which is reachable by all colors. The game is played by solving the individual goal sequentially. The objective of the game is to find the minimum number of moves such that the corresponding robot lands on the given goal. All robot can be moved and all robots can be used as obstacles for the other robots.

A round starts with picking a goal to reach. All players can declare that they are able to reach the goal in a specific number of moves. When the first player declares a number, all the other players have one minute to find a lower (and better) number of moves. If possible, the player calling a lower number of moves demonstrates his solution and wins the round. If not, the first player to declare demonstrates his solution and wins the round. The game continues until all 17 goals have been played. The robots are then placed at random again and the game repeats.

Moving the robots are one of the characteristic parts of the game. A robot moves like a rook in chess, but it moves in a direction until it hits an obstacle or another robot. This counts as one move. Thus, a robot cannot stop on any of the intermediate fields. All robots can be moved and used as obstacles for the other robots. An example is given in figure 2. The robots can move in four directions. This gives that for each move, 16 possible moves can be made, also referred to as the branching factor. However, as the robots moves they are adjacent to at least one obstacle, and the branching factor is  $\sim 12$ . The number of possible

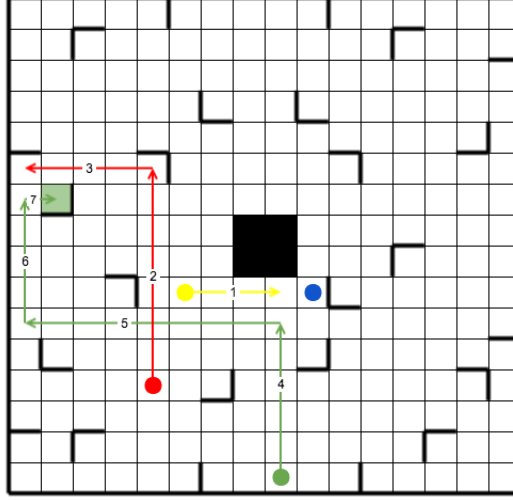


Figure 2: An example of Ricochet Robots where the green robot has to reach the goal at (7,2). The minimum number of moves is 7, and multiple solutions exists. The number on each move indicates the order of moves.

combinations for a given game is given by  $12^k$  where  $k$  is the required number of moves to reach goal. Given the exponential growth in possible combinations, the game presents several challenges for both players and computers:

- Finding a valid solution
- Guaranteeing an optimal solution
- Solving the game in reasonable time

The game has been proven NP-Hard **REFERENCE** which underlines that the complexity of the game is an exponential function of the required number of moves. Precomputing each and every robot combination for each board is infeasible given the high number of robot combinations and the complexity of each game. Thus, tradeoffs between solving the game optimally and solving the game in reasonable time are at the core of developing algorithms for this game.

## 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  $OR$  be the set of Obstacle Robots  $\in R - \{GR\}$ .

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

Let  $b$  be the branching factor describing the number of possible moves for each game state and let  $k$  be the minimum number of moves required for solving a given game.

The set of directional indicators  $D$  refers to the possible directions on the game-board. 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 feasible 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.

Solving a giving configuration uses  $O(b^k \cdot b \cdot n) = O(b^{k+1} \cdot n)$  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  $O\left(\sum_{i=1}^k b^i\right) \sim O(b^k)$  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. In the latter,  $min[i,j]$  refers to the minimum number of moves to goal from position  $(i,j)$ . It is assumed that the robots can change direction without bouncing off an obstacle as seen in figure 3. If  $min[GR] > h$  the search is pruned. If  $h = min[GR]$  only the  $GR$  is processed further in the search tree.

3	3	3	2	2	2	2	2	2	2	2	2	2	3	3	3	3
3	3	0	1	1	1	1	1	1	1	1	1	1	3	3	3	4
2	2	1	2	2	2	2	2	2	2	2	2	2	2	2	2	2
2	2	1	2	2	2	2	2	2	2	2	2	2	3	3	3	3
2	2	1	2	2	2	2	2	2	2	2	2	2	2	2	2	3
2	2	1	2	2	2	2	3	2	2	3	3	2	3	3	3	3
3	3	1	2	2	2	2	2	2	2	2	2	2	2	2	2	2
2	2	1	2	2	2	2				3	3	2	3	3	3	3
2	2	1	2	2	2	2				3	3	2	3	3	3	3
2	2	1	2		3	2	3	3	3	3	3	2	3	3	3	3
2	2	1	2	2	2	2	2	2	2	2	2	2	2	2	2	2
3	2	1	2	2	2	2	2	2	2	2	3	2	3	3	3	3
2	2	1	2	2	2	2	2	3	3	3	3	2	3	3	3	3
2	2	1	2	2	2	2	2	2	2	2	2	2	2	2	2	2
4	3	3	3	3	2	3	3	3	3	3	2	3	4	3	3	3
3	3	3	3	3	2	3	3	3	3	3	2	3	3	3	3	3

Figure 3: The minimum number of moves for the goal at (2,3) colored in grey.

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 are 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. The algorithm does not guarantee an optimal solution. 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*. The algorithm assumes that every field can only have one final state for each robot and only one intermediate state for each combination of direction and color.

Final 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 and new states are created for each visited field if no redundant states already exists for the given field. The algorithm terminates when there are no more robot states to process and the best result is returned.

### Algorithm

Let  $S[i,j]$  refer to the element storing states for  $F[i,j]$ .  $S[i,j]$  is instantiated at the first visit to  $F[i,j]$ . Each  $s \in S$  stores final states in a hash table using the color of the robot as key. In addition,  $s$  stores all intermediate states in a hash table using the direction of the robot move as the key. Finally,  $s$  stores if a robot starts on the field.

### Moving and States

Moving a robot determines the state for each field processed. The state depends upon the next field to process if it is possible to move further. Four cases is

taken into consideration when moving robot  $r$ , and all four cases are evaluated for each step until an obstacle is met. In the following  $a$  will refer to the adjacent field in direction  $d$ :

1.  $F[i,j]$  cannot move in direction  $d$ . A final state,  $f$  is added to  $S[i,j]$ . If a final state did not already exist,  $f$  is added to the queue for future processing.
2.  $F[i,j]$  can move in direction  $d$ :
  - a. No final robot states exist on  $a$ . Thus, an intermediate state is added to  $S[i,j]$  and  $a$  is processed next.
  - b. A final robot state exist on  $a$  and its color does not match robot  $r$ 's. The final state  $f$  is set to depend upon the lowest final state in  $S[a]$ , the move counts are aggregated and  $f$  is added to  $S[i,j]$ . If a final state did not already exist,  $f$  is added to the queue while  $a$  is processed next.
  - c. A robot starts on  $a$ . The same as case 1.b is applied, with the addition that all the further processing of fields in direction  $d$  for the current  $r$  will be added 1 to the number of required moves from  $a$  and forward. This refers to the required movement of the starting robot on  $a$ .

An initial move is illustrated in figure 4 displaying several of the cases.

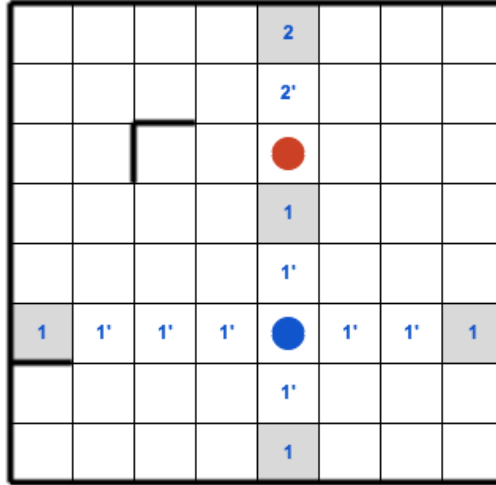


Figure 4: The initial move for the blue robot. All grey fields are final states while all fields denoted with an apostrophe are intermediate states. This includes the field of the red robot with a value of 2.

Whenever a new final state  $f$  is added to the queue, adjacent fields are searched for intermediate states on collision with  $f$ . Let  $a$  be the adjacent field in direction

$d$  and let  $d'$  be the opposite direction of  $d$ . All intermediate states  $i \in I$  where  $i.color \neq f.color \wedge i.direction = d'$  are removed from  $S[a]$  and converted to final states that depend upon  $f$ . If no final states already exists for the given color in  $S[a]$ , the new final states are added to the queue.

### Analysis

The algorithm uses a min-heap queue and insertion and removal uses  $O(\log(m))$  time where  $m$  refers to the number of elements in the queue. Since each field is only processed once for every robot,  $m = O(n^2 \cdot |R|)$ . Thus, insertion and removal in the queue uses  $O(\log(|R| \cdot n^2))$  time. Moving a robot uses  $O(2n)$  time like the naive algorithm. Inserting, lookup and removal from the hash tables in  $S$  uses  $O(1)$  time. Processing each robot state uses  $O(2n \cdot 2\log(|R| \cdot n^2))$  time and the total worst case uses  $O(n^2 \cdot 2n \cdot 2\log(|R| \cdot n^2)) = O(n^3 \cdot \log(n^2))$  time.

In the worst case, each  $s \in S$  stores  $O(|R|)$  final states and  $O(|D| \cdot |R|)$  intermediate states. The queue contains  $O(|R| \cdot n^2)$  elements. Therefore, the worst case space impact is  $O(n^2 \cdot (1 + |R| + |R| \cdot |D|)) = O(n^2)$ .

### 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  $GR$ . An example of the implications of this assumption is shown in figure 5.

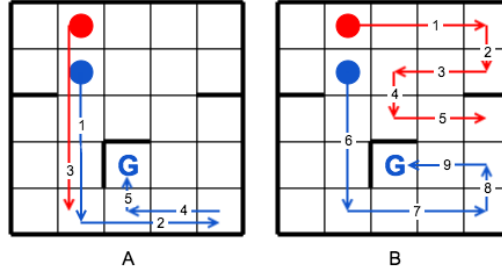


Figure 5: Case A displays the optimal solution for the given game board while but case B is the optimal solution given by the graph algorithm hence case A requires the red robot to move after the blue robot's initial move.

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 referred 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  $d \in \{East, South\}$  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 a 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 referred to as  $d'$ .

Let  $v'$  be the vertex immediately 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 all 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 figure 6 and listed below.

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  $d$ .

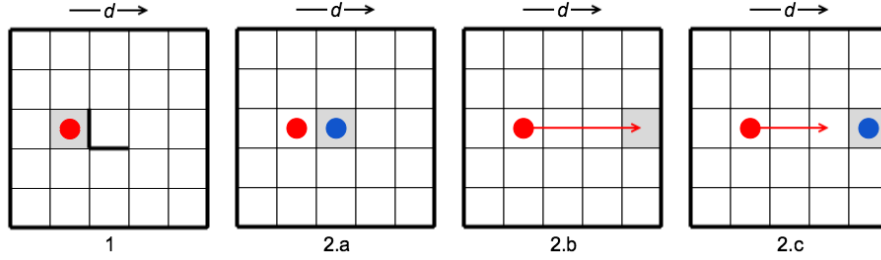


Figure 6: Each case when updating the graph in direction  $d$ . The grey field is the most distant vertex in each case.

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 current investigated game state. Since the number of moves is investigated in an 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 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 enqueueing 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  $O(n^2 \cdot |D| + 2b^k + n^2) = O(b^k + n^2 \cdot (1 + |D|)) = O(b^k + n^2)$ .

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 described 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 current 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  $(\min[c] + c.d) \geq \text{depth}$  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 remains the same. The space impact of the algorithm is  $O(b^k + n^2 \cdot (1 + |D|) + n^2) = O(b^k + n^2 \cdot (2 + |D|)) = O(b^k + n^2)$ . The intermediate result shows a slight increase in the space impact while the worst case remains the same as version 1.

## Summary

In table 1 all described algorithms are summarized with branching factor and worst case space and time analysis. As shown, only the stateless algorithm does not include the branching factor in the running time and space analysis.

Table 1: A summary of the presented algorithms.

Algorithm	Branching factor	Worst Case Time	Worst Case Space
Naive	$\sim 12$	$O(b^{k+1} \cdot n)$	$O(b^k + n^2)$
IDDFS	$\sim 12$	$O(b^k \cdot n + n^3)$	$O(b^k + n^2)$
Stateless	-	$O(n^3 \cdot \log(n^2))$	$O(n^2)$
Graph v1	$\sim 9$	$O(b^k \cdot n^2)$	$O(b^k + n^2)$
Graph v2	$\sim 9$	$O(b^k \cdot n^2 + n^3)$	$O(b^k + n^2)$

The naive algorithm has the worst worst case running time while the stateless is the fastest in the worstcase and most optimal in the space analysis. The graph algorithms are both much alike and only the minimum moves addition to version 2 affects the worst case time. The IDDFS algorithm has the best worst case running time of the four algorithms depending on the branching factor.

# Experimental Results

## Setup

I have compared the following algorithms.

- **Naive** The basic algorithm for the problem in my own implementation.
- **IDDFS** The IDDFS algorithm claimed to be the best solver in my own implementation.
- **Stateless** My own proposed stateless algorithm independent of the branching factor.
- **Graph v1** My own graph and dynamic programming based algorithm.
- **Graph v2** The optimized version of the one above. It applies many early termination techniques aggressively to shorten the average running time.

All performance tests have been conducted on a Intel i7 2.3 GHz with 4 GB heap allocated.

All of the described algorithms were implemented in Java. All performance tests have been on the same game board but with different initial game states. The gameboard is available on *appendix*. Given a game board size of  $16^2$  and 20 illegal starting fields for robots, there exist  $\sim 3$  billion possible combinations.

In the following, the combination of a initial game state and a given goal is referred to as a game configuration. For each initial game state, 16 goals exist. Each game configuration is processed by all algorithms in a sequential manner. An upper time limit at 40 seconds is introduced to terminate long running processes; at this point heap size becomes an issue and slows down the processing even further making the result ambiguous. The result of the IDDFS is used as optimal baseline for each game configuration.

I have conducted performance tests on 160,000 game configurations. In the following, *required number of moves* will refer to the minimal number of moves required for reaching goal for the given robot while the *hitrate* will refer to the percentage of game configurations where a given algorithm solves the game configuration in the required number of moves. The hitrate and running time will be the primary focus of the tests.

## Results

Using the result of the IDDFS a distribution of game configurations per required number of moves is shown in figure 7. In figure 7 it shows that the current board averages around 6-7 required moves per game configuration. It is also shown that few solutions exist which require 17 moves. Solutions having 17 required moves to goal are therefore pruned from the data sets given the insignificance and inaccuracy of these.

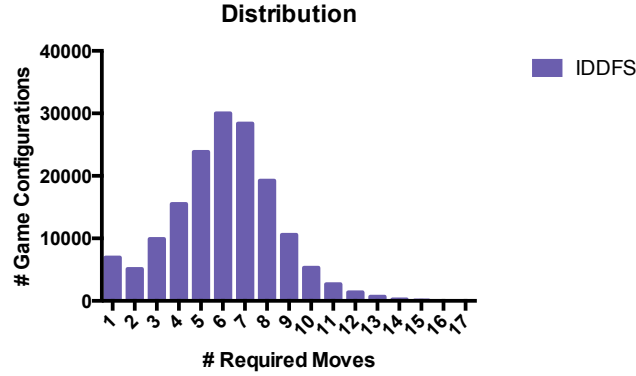


Figure 7: The distribution of game configurations.

As shown in figure 8, the naive algorithm starts hitting the 40 seconds limit around 13 moves while the IDDFS solves every configuration. Also, the stateless and graph solutions follows each other closely with a 5-10% better hitrate for the latter. However, as the game configurations become more complex, the hitrate drops to ~50% for both solutions. Even though it seems that the hitrate is low, the graph algorithms has an total accuracy close to 90% while the stateless algorithm solves the game in ~80% of the game configurations as shown in figure 9.

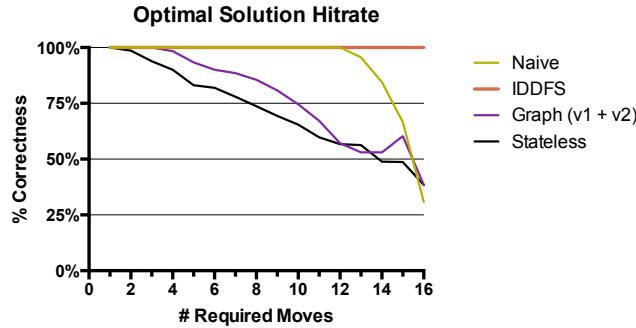


Figure 8: The hitrate as a function of required number of moves.

In figure 10 the average running time is plotted as a function of the required number of moves for the given game configuration. The naive algorithm is the slowest algorithm by several degrees and has been been ignored. The stateless algorithm uses 1-2 ms pr. solution no matter the number of required moves while the graph v2 algorithm is considerable faster than the IDDFS at 9 moves required and higher.

Several results for the graph algorithms are not optimal and therefore the av-

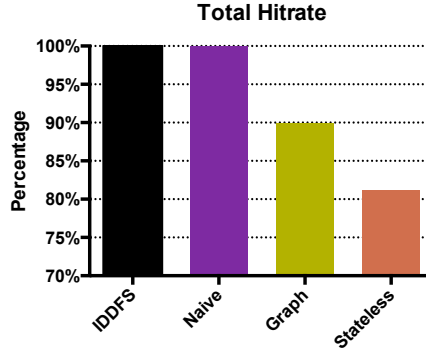


Figure 9: The total hitrate in percent.

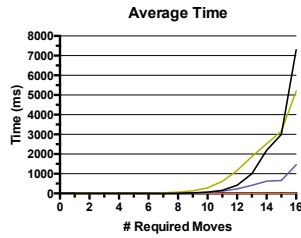


Figure 10: The average running time as a function of required number of moves.

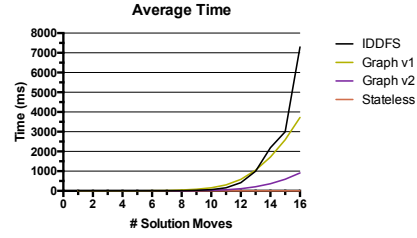


Figure 11: The average running time as a function of number of moves given by the result.

average running time given in figure 10 includes results where the algorithms investigated more moves than actually required. In figure 11 only the running time of results where the graph algorithms found the optimal solutions are compared. The stateless algorithm is omitted given the running time is not affected by the required number of moves.

All data is available in *appendix*.

## Analysis

The running time of each algorithm corresponds well to the theoretical analysis. All algorithms depending on the branching factor uses a considerable amount of time as the complexity of the game configurations increases. For the stateless algorithm, the worst case running time is constant given the fixed size of the board, which also shows in the results.

An interesting result is that for the given game board, the result of 80% of the game configurations can be solved in constant worst case running time.



It is clear that the optimizations applied to the graph algorithm have affected the average running time considerably. The extra cost of precomputing the minimum moves is not a considerable factor. Comparing the two graph algorithms, the increased performance of the graph v2 is an exponential function of the number of moves. This corresponds well to the actual optimizations applied.

An interesting fact is that the graph v2 algorithm outperforms the IDDFS on solutions with 9 or higher required number of moves. For solutions where the graph v2 algorithm found the optimal solution, it outperforms IDDFS on game configurations with 8 or higher required number of moves. On solutions which required 16 moves, the graph v2 is faster with about a factor 7.5 if considering only the cases where the graph v2 found the optimal result.

However, the results also show that as the graph v2 and the stateless algorithms outperform the IDDFS algorithm on solutions with 9 or higher required number of moves, the hitrate for each algorithm decreases fast. Thus, in the search space where the proposed algorithms outperform the known solutions, their reliability drops.

## Future Work

Given the results of my work, I would like to propose several improvements and optimizations.

### Performance tests

The performance tests are conducted on the same game board. However, it would be interesting to investigate other game boards with a different distribution of game configurations per required number of moves and how it affects the running time of each proposed algorithm.

As the prototypes are implemented in Java, the algorithms could be implemented in C/C++ for increased performance while several machine-aware techniques could be used to optimize all algorithms in general.

### Stateless

In few cases, the stateless algorithm returns a lower solution than actually possible. Thus, there exists cases that are not accounted for, and these would have to be included in the assumptions. If these cases can be included without affecting the running time, the stateless algorithm will always guarantee an upper limit for the required number of moves. If this is the case, it can be included in other algorithms and used to prune the search field.

## Graph

During my tests I found that the BFS visits ~60 nodes for graph v1, while for graph v2 it only visits ~15 on average. This number directly depends on the number of reachable fields on the board which again depends upon the number of obstacles. A more precise worst case running time analysis could be proposed. This would depend on a function of the number of fields and the number of obstacles on the board instead of the number of nodes.

Another interesting improvement would be to move all robots including the *GR*. This would increase the branching factor, but the graph algorithm would guarantee an optimal solution. Given the results of the performance tests when comparing the average running time when the graph algorithms found the optimal solution, there is still a considerable gap between the IDDFS and the graph v2. Thus, a tradeoff between increasing the running time while guaranteeing the optimal solution would be of interest.

The pruning techniques introduced in the graph v2 could be improved. They are not truly applied before the first time the BFS hits goal. Thus, the graph v2 relies on finding an upper limit before the pruning techniques works optimally. If the stateless algorithm can be improved to guarantee an upper limit for the required number of moves, this might be implemented into the graph algorithm when solving complex game configurations, setting an upper bound for the search field and initiate the pruning earlier.

## IDDFS

The IDDFS could introduce the stateless algorithm on the more complex game configurations to improve overall performance and skipping some intermediate levels of deepening. However, given the incrementing nature of the algorithm, it does not benefit the same way as the graph v2 does. Depending on the inaccuracy of the stateless algorithm, this improvement could improve the running time on the more complex algorithms.

## Appendix ...

Table 2: Average running times as a function of required moves.  
All numbers are in *ms*.

Algorithm	1	2	3	4	5	6	7	8	9	10	11	12	13	14
IDDFS	0	0	0	0	0	1	4	10	26	58	149	390	989	2714
Graph v1	0	0	0	1	4	12	29	69	140	312	606	1362	2096	2521
Graph v2	0	0	0	0	1	3	5	13	24	53	104	263	444	720
Stateless	1	1	2	1	1	1	1	1	1	1	1	1	1	1

Table 3: Average running times as a function of required moves  
where the optimal solution was found. All numbers are in *ms*.

Algorithm	1	2	3	4	5	6	7	8	9	10	11	12	13	14
IDDFS	0	0	0	0	0	1	4	10	26	58	149	390	989	2714
Graph v1	0	0	0	0	2	7	20	44	89	175	335	608	1200	1804
Graph v2	0	0	0	0	0	1	3	7	15	28	57	112	227	375