Bachelor Informatica

University of Amsterdam

Performance optimization of Rush Hour board generation.

Jelle van Dijk

June 8, 2018

Supervisor(s): dr. ir. A.L. (Ana) Varbanescu

Signed: Signees

Abstract

Rush Hour is a puzzle game. The objective is moving cars such that the target car has a clear path to the exit. Puzzles like Rush Hour offer very little in terms of replay value. Once a puzzle is solved the challenge is gone. In this thesis we will use a naive generator to generate puzzles and two types of solvers to validate the generated boards. The first solver is based on a breadth first search algorithm and one based on depth first search. Based on the performance of the solver two optimizations are implanted for the breadth first search solver. These optimizations focus on reducing the work done by the solver. The best performing parts are used to create a hybrid solver. The hybrid solver uses both the breadth and depth first search solver. The final solver is able to generate simple boards in less then 1 second.

Contents

CHAPTER 1

Introduction

Games have been around a long time. For example, The Royal Game of Ur originated in the middle east and it is known it is at least 4,500 years old, [3]. In more recent years video games have become increasingly more popular. A type of game enjoyed by many people is a puzzle game. In puzzle games there is a set of rules and an objective. The player must reach the objective by performing actions within the rules of the game. There are many types of puzzle games - a few examples can be seen in Figure 1.1.

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

Figure 1.1: Examples of puzzles. Left Sudoku, right lunar lockout

The puzzle game that we focus on in this project is Rush Hour. Rush Hour is a game developed by Nob Yoshigahara in the 1970s. The game consists of a 6×6 board and multiple cars. The cars are either of size 1×2 or 1×3. The goal is to move the cars out of the way of the target car. If the target car can escape the board at the right side exit, the puzzle is solved. An example puzzle setup for Rush Hour is given in Figure 1.2

Puzzles like Rush Hour are accessible to players of varying skill levels. Rush Hour originally came with 40 different levels of varying difficulty. The problem with these kinds of puzzles is replayability: it is fun to solve all the levels, but once they are solved there is no challenge in replaying them. To keep the game interesting, Rush Hour released multiple expansions which offered more puzzles. Games like *Unblock Me*[1] offer a similar puzzle experience to the original Rush Hour. With the original Rush Hour buying the expansions costs money. Apps like *Unblock Me* make money through advertisements. If people stop playing, the money stops, too. But in order to keep people playing a lot of levels are needed to keep the game interesting. The goal of our work is to provide automated ways to keep puzzle games interesting.

To keep these puzzles interesting for players of different skill levels, the puzzles need to have varying difficulty. Creating these levels by hand is a time consuming task. Procedural Content Generation(PCG) is an area of computing which generates content algorithmically, and originates from the need to allow more complicated games without using too much memory to store all

[1] `https://play.google.com/store/apps/details?id=com.kiragames.unblockmefree&hl=en`

Figure 1.2: Rush Hour card presenting a puzzle setup

content. In such cases, game areas could be defined by some parameters, and generated when needed (i.e., on-demand) instead of pregenerating and storing them. With the growth of available memory, this is not so much a problem these days, but PCG is still used for generating random or personalized content for games, [4]. In this thesis we want to generate content of varying difficulty for Rush Hour.

1.1 Research question & Approach

The research question we want to answer is: *How can we efficiently generate puzzles for Rush Hour?*

The generated puzzle must contain a starting configuration, like in Figure 1.2, and the best solution to that configuration. In the original Rush Hour game, the solution is provided for reference. Thus, because we want to generate content for this game, we also need the solution for every puzzle we propose. The efficiency of the generation is measured in how much puzzles can be generated in a specific time.

In order to answer this question we use an empirical approach: we devise algorithms and build prototype implementations able to generate rush hour boards and their solutions. To improve the efficiency of these algorithms, we use a performance-engineering process: we measure performance, we use the performance data to determine performance bottlenecks, and we solve them using (algorithmic) performance optimization techniques. We report the gain after each performance optimization, and combine the best performing parts to create a final implementation. We measure the performance of this implementation and propose further optimizations for future research..

1.2 Thesis structure

This thesis starts with an introduction and presents the needed background knowledge. Chapter 2 presents the previous work done in puzzle generation and why this project is different. The same chapter provides the basic Rush Hour knowledge, i.e., the game rules and the difficulty metric.

In Chapter 3 we present the design and implementation of the generation program. For this program we created a naive generator and two types of solvers. The first solver is based on a depth first search (DFS) approach, and the second uses a breadth first search (BFS) approach. Chapter 4 shows the performance of the generator and the solvers. From these performance measurements it is clear that the generation time is insignificant compared to the execution time of the solvers. In order to increase the performance of our PCG, we focused further on improving

the performance of the solvers.

Two performance optimization strategies, aimed at improving the solvers, are presented in Chapter 5. They both focus on minimizing the amount of work that has to be done by the solver: *queue pruning* provides a mechanism to avoid duplicate work, while *early solution detection* is a method for stopping the solver sooner.

Finally, based on the achieved performance after the optimization, a final implementation is created. The final implementation is found in Chapter 6. This hybrid implementation combines the best performing parts of the two initial solvers and the performance improvements. The performance of the hybrid implementation is also measured - and reported as the best performance we achieved for our generator.

This thesis concludes with Chapter 7, where we revisit our research question, list the most interesting findings of the project, and enumerate potential future work directions.

CHAPTER 2

Background

Procedural Content Generation(PCG) was used in games to decrease memory pressure: instead of creating the whole game and storing it, the content of the game can be generated on-demand using PCG. With the growth of available storage, PCG was not necessary anymore, as the game content could be created by hand and stored. But with the growth of the gaming industry, PCG becomes popular again, for costs saving and personalized content generation. Specifically, the time required to create all the content in the large games produced these days becomes too expensive, and people want more and more the ability not to have a static play. PCG can provide solutions for both these problem [4].

2.1 Related work

Content generation for puzzles can be done in different ways. W. J. M. Meuffels et. al. [6] solved battleships using integer programming. Battleships is a grid based puzzle game. The objective is placing a set of ships on the board based on some limitations. Their research formalized the rules of the battleship game; these formal rules were used to create two different models that are able to generate content for new battleships puzzles.

D. Oranchak [7] used en evolutionary algorithm for generate entertaining Shinro puzzles. Similar to battleships, shinro is a grid based puzzle that requires one to place objects based on some restrictions. Based on these rules, Oranchak created logical conclusions for certain situations, i.e., if this situation occurs there an object must be placed on that location. These conclusion were used to create a solver for shinro. To generate the board, a simple genetic algorithm was used. Each board could was also given an entertainment value. This value was based on multiple metrics, one of them being the which and how many of the previously mentioned conclusions where needed to solve the puzzle. After generating a set of boards, the most entertaining boards where kept. To these boards random mutations where applied. These steps where continued till there was no more increase in the entertainment value of the boards.

Martin Hunt et. al. [5] generated Sudoku boards, not based on their entertainment value, but on their difficulty level. The method used by many soduku generators is to create random puzzles, and solve them to find their difficulty; they tend to discard the ones that are not difficult enough. Hunt et. al. described the difficulty of a puzzle based on the thought processes needed to solve it. They also worked on a framework which used a solver to created solved boards. Then, by removing numbers, a starting configuration was created.

Different puzzles require different generation approaches. B.Pintér at al. [8] created odd one out puzzles. In these puzzles a set of words is given where all but one have something in common. The player has to find the word that does not belong. In order to generate these puzzles sets of words are created. These sets all have a close similarity in some way. Four words from this set

are taken and one from a set which has weak similarity to the other set.

We observe that most of the related work focuses on smart generation of grid-based puzzles, based on placing the correct object or number in the correct location on the grid. Rush Hour requires the player to move the already placed pieces. The closest related work was performed by M. Broekstra [2]. Broekstra used the GPU to generated puzzles for Lunar Lockout, a puzzle game which is also based on moving pieces around in order to let a target piece reach its goal. The puzzles were generated in a similar way to the approach we used. A naive generator creates the boards which are then solved to determine their difficulty. The solver used in this research was based on a depth first search algorithm. In contrast to our research, his focus was on parallelizing this solving procedure using the GPU, rather then optimizing the algorithm.

2.2 Rush Hour

The rules of Rush Hour are straightforward. Two types of cars are placed on the board, cars of 2 $times$1 blocks and of 3×1 blocks. They can move left to right or up and down, based on their rotation. Vertical placed cars can move left to right, horizontal cars move up and down. The goal is to move the target car outside the board. For our boards, this means moving it against the rightmost wall. Cars cannot move trough each other or outside the boundaries of the grid. In each turn, a car can move any number of places as long as it does not violate the previous rule. Figure 2.1 presents an example of a board and its solution.

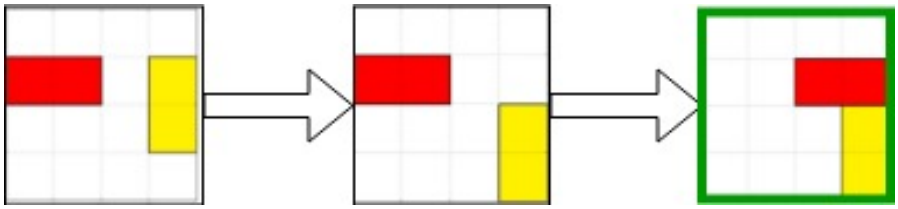


Figure 2.1: Example of solving a Rush Hour board. The red car is the target car.

The difficulty of a puzzle is based on a very simple metric: the minimum number of moves required to solve the puzzle is the difficulty level of the puzzle. Overall, solving a board using 20 moves takes more effort then solving it with 5 moves. While one could argue this is not the best metric for difficulty (for example, Hunt et. al. [5] used multiple kinds of thought processes to determine difficulty), it is a simple metric, and research on improving it is beyond the scope of this project.

CHAPTER 3

First Prototype

The board-generation framework consists of two parts, the *generator* and the *solver*. The generator creates boards while the solver solves them. The solver will find the best solution for the generated board. Based on the best solution of a board the difficulty level is determined, the metric for difficulty is the minimum number of moves required to solve the board. The structures in Figure 3.1 are used to store a board.

```
struct Board{
    int size, nCars, nMoves;
    Move *moves;
    Car *cars;
};
```

Figure 3.1: Struct used to represent a board

`Size`, `nCars` and `nMoves` are integers which indicate the size of the board, the number of cars and the number of moves which for a board. `Cars` points to an array of size `nCars`, each element represents a car. `Moves` points to an array of size `nMoves`, when the board is solved this array represents the best solution (i.e a way to reach the final configuration in the least amount of moves).

```
struct Car{
    int x, y;
    int size, direction;
};

struct Move{
    int carId, move;
};
```

Figure 3.2: Structs used to represent a Car and a Move

A `Car` element represents a car on the board, see Figure 3.2. The x and y values are the coordinates of the lowest left most point of the car, the orientation can be either horizontal, represented by a 1, or vertical, represented by a 0. A `Move` represents the movement of one of the cars on the board, see Figure 3.2. A `Move` holds the id of the moved car and the move, i.e., the number of places the specified car moves. The direction of the move is based on the orientation of the car, if car should move 1 place this means the car will move 1 position up or to the left, if the move is negative the car will either be moved down or to the left. To determine the moves a car can perform the movement range, i.e., the smallest and largest possible move,

can be determined. For each car each value $\in$ [min range, max range] where move $\neq 0$ is a new move, all these moves combined give all possible moves for a board. A visualization of a search tree see Figure 3.3.

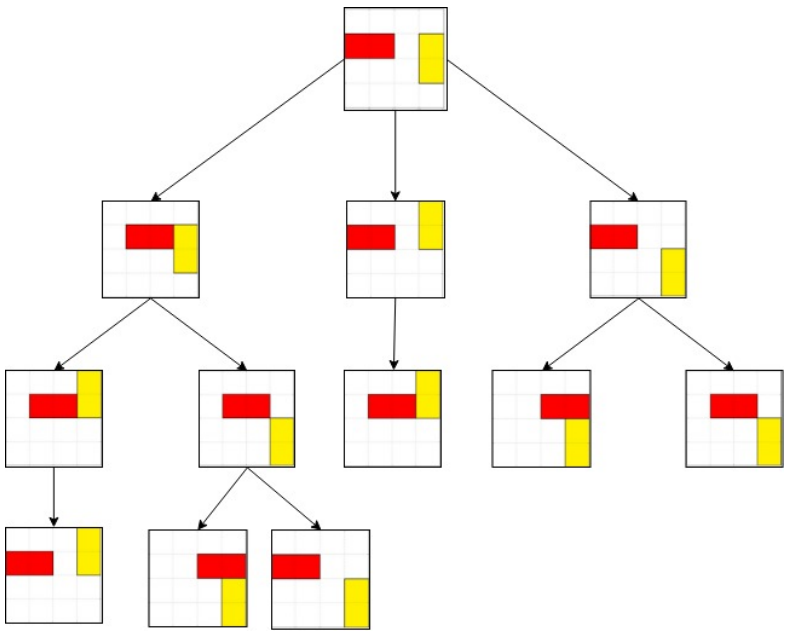


Figure 3.3: Example of an incomplete search tree. Red car is target car.

3.1 The generator

The generator is responsible, i.e., placing the cars on the board in an initial position so that the player/solver can solve it. The generator could try to build a board with a certain difficulty, (for example by reverse engineering a initial board from a solved board), or a board can be generated by placing the cars in random positions. Based on this decision the solver will have more or less work to do: validating a solution or finding one, the random approach was chosen, meaning all of the analysis will be done by the solver.
A few parameters will be given to the generator. The parameters given to the generator are, the number of cars with size two, the number of cars with size three, the size of the board and the y value of the exit. The position of the cars are generated randomly, one by one, each placement is validated. When the position is invalid (e.g., blocking, overlapping, etc) a new position is generated. If within 5 times the car position is still not valid the previously placed car is placed again. The process ends when all cars are placed in a valid starting position.

3.2 The Solver

The solver is given two arguments a starting board, (as built by the generator) and a maximum search depth. The solver finds the shortest solution for the starting board if there exists a solution which uses less moves then the maximum search depth. A board is solved if $targetCar.x >= board.size - targetCar.size$, the sequence of moves which when applied to the starting board create a solved board is a solution, when there is no solution which uses less moves the solution is considered an optimal solution. For the first prototype two solvers where implemented a breadth first search (BFS) and a depth first search (DFS) algorithm one. Both DFS and BFS are algorithms for performing a search in a tree or graph. In order to use DFS and BFS the search space is thought of as a tree. Each board configuration, i.e., a board created by performing moves on the board that is to be solved, is considered a node, the children of this node are all

configurations that can be made by performing one move on the parent node. When a node is a solved configuration the solution is the moves used to traverse from the starting board to the solved node.

3.2.1 Breadth first search

The BFS algorithm starts at the root node of the search tree and explores each node with the same depth before moving on to the nodes on the next level, i.e., nodes with a higher depth, for example first the root node is explored followed by all of its children, when none of these children are correct solutions the children of the children are explored. This continues until a solution is found or the complete tree is explored up to the maximum depth. When a solution is found,it is guaranteed to be an optimal solution because all nodes that are not explored at this point are either at the same depth or a higher one. The BFS order of traversal is visualized in Figure 3.4

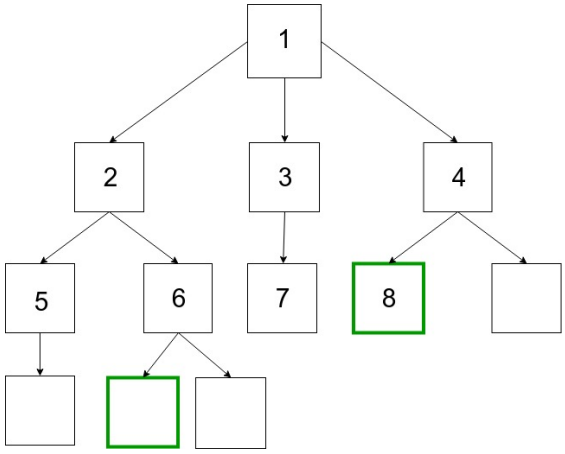


Figure 3.4: Example of the BFS solver traversal order of the tree seen in Figure 3.3. Green marks a solved configuration node.

To keep track of the nodes that need to be visited and there order a queue is used. Each element in the queue holds a list of moves, these moves are the moves needed to go from the starting board, to the board represented in that element. Each element in the queue represents a node in the search tree.

The BFS based algorithm for solving the Rush Hour board, presented in Algorithm 1, starts by adding the empty list to the queue, to represents the root node, i.e., the starting board. While the queue is not empty the next element of the list is popped from the queue, based on this move list the current board is computed. If the current board represents a solved board the algorithm is stopped and the move list is returned an optimal solutions. If the current board is not solved the algorithm checks whether the board is considered a leaf node, i.e., the number of moves is equal to the max search depth. When the board is neither solved or a leaf node all child nodes are added to the queue. For each car that was not moved during the last move the movement range, see beginning Chapter 3, is calculated, using the movement range all moves are created. To each move the current move list is prepended, this new list is pushed to the queue. The moves are added to the queue largest first, and the cars are handled in the order that they are stored in in the board. Because the target car is the first car in the list this order assures that if the target car has a free path to the exit this node is the first one added to the queue. A visualization of the BFS solver with queue can be seen in Figure 3.5

In Algorithm 1 from each element a board is computed and further used for checks and to compute the new nodes. However the list of moves is stored in the queue and not the board itself. Although storing the board would reduce the computation time when the node is traversed it would require the use of more memory. On average the search tree grows exponentially for each move that is checked, for example when three cars of size two are placed next to each other on a

Algorithm 1 BFS based algorithm for solving a Rush Hour board

Precondition: board ▷ board is the configuration that needs solving

```
1:  QUEUEPUSH([ ])                                          ▷ Empty list of moves
2:
3:  while QUEUE ! = empty do
4:      curNode ← QUEUEPOP( )
5:      curBoard ← CREATEBOARDFROMMOVES(board, curNode)
6:
7:      if ISBOARDSOLVED(curBoard) then                     ▷ Stop search if solution is found
8:          return curNode
9:
10:         if ISBOARDLEAF(curBoard) then                   ▷ Goto next node if end of tree
11:             continue with next iteration
12:
13:     for all car ∈ curBoard.cars | car ≠ last_moved_car do
14:         for all move ∈ POSIBLEMOVES(curBoard, car) do
15:             QUEUEPUSH(curNode + [move])                 ▷ Append move to curNode
```

default size board, 6×6, this board would have 12 possible first moves and 8 for the other moves. When solving this board with a maximum search depth of 10 this would generate a tree with $12 * 8^9$ or $1,610,612,736$ nodes. Storing the full board for each node would requires $4 + 8 * nCars$ bytes more then only storing the moves. For this example $45,097,156,608$ extra bytes or $\approx 45,1$ GB would be needed. Only storing the moves instead of the full boards allows for larger search trees to be handled by the algorithm before the computer runs out of memory.

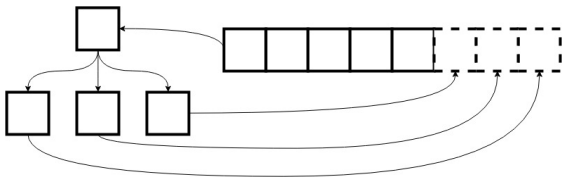


Figure 3.5: Visualization of the BFS solver and queue. An item is taken from the queue. Its children are created and appended to the queue.

3.2.2 Depth first search

DFS starts at the root node, i.e., the starting board, and explores the nodes till it finds a leaf node, when a leaf node is found the algorithm starts backtracking. Backtracking is an algorithmic technique used for going back up the search tree. When the algorithm start backtracking it will return to the closest node which has unexplored children, there the search is continued. When the backtracking returns to the starting board and there ar no children left to traverse all nodes have been traversed and the search stops. A visualization of the DFS solver traversal order can be seen in Figure 3.6.

Algorithm 2 is the recursive DFS based algorithm for solving a rush hour board. The algorithm uses the function `DFSSolver` which is recursively called for each node in the tree. For each node the `DFSsolver` function performs the same checks as the BFS algorithm, presented in Algorithm 1, firstly if a solution has been found and secondly if the node is considered a leaf node. If a solution has been found it is stored as the current optimal solution and the backtracking process is started. To check whether a node is a leaf its depth is not only checked against the maximum search depth, as done in Algorithm 1, but also against the current optimal solution. If the current node's depth is one above the number of moves in the current optimal solution the node is considered a leaf node, because all nodes that can be reached throught this node either

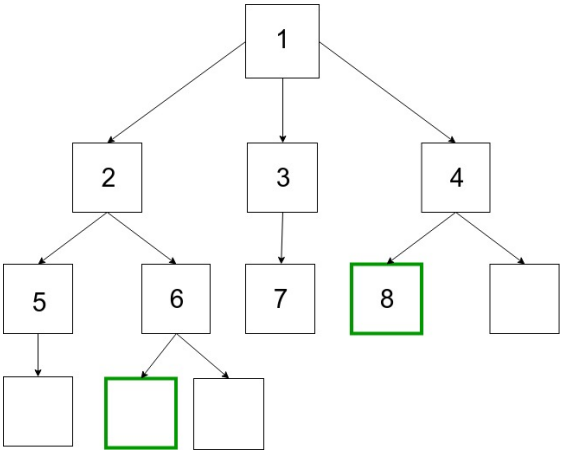


Figure 3.6: Example of the DFS solver traversal order of the tree seen in Figure 3.3. Green marks a solved configuration node.

have the same or a higher number of moves then the current optimal solution. If the board is considered a leaf node the backtracking process is started.

If the board does not start the backtracking process, i.e., no solution is found and the node is not a leaf, the `DFSSolver` function is called for each child of the current node. To creation of all moves is the same as with the BFS algorithm, for each car that was not moved in in the last move, each move traversed in descending order. Meaning the first car is on the board is moved as far forward as possible so if the board is solvable in one move this move is the first one to be checked.

Algorithm 2 DFS based algorithm for solving a Rush Hour board

Precondition: board ▷ board is the configuration that needs solving

```
1:
2:  procedure DFSSolver(board)
3:      if isBoardSolved(curBoard) then          ▷ Start backtracking if solution is found
4:          storeSolution(board)                 ▷ current best solution is stored
5:          return
6:
7:      if isBoardLeaf(curBoard) then            ▷ Start backtracking if leaf node
8:           return
9:
10:     for all car ∈ board.cars | car ≠ last_moved_car do
11:         for all move ∈ posibleMoves(board, car) do
12:             DFSSolver(board)                 ▷ Append move to curNode
```

CHAPTER 4

Performance First Prototype

To increase the performance of the program, we need to measure it and determine what can be improved. The performance we care about is the time it takes the program to generate and determine the optimal solution within the given maximum search depth, i.e, the time taken from board generation to final solution.

All measurements are performed on machines running `CentOS Linux 7.4`, with an Intel Xeon E5-2630v3 2.4 GHz with 64GB of memory. These machines are part of the DAS-5 supercomputer [1]. Because the experiments are run on the DAS-5, there is a 15 minute cap on the execution time of each solver. If the solver attempts to run beyond the 15 minutes, the DAS-5 scheduling software will interrupt the program, and no (partial) results will be produced.

4.1 End to End performance

In order to see which part of the program takes the longest to execute, the full program is run. The execution time of the solver and the generator are measured separately. For both solvers the program is run four times, and in each run a board is generated and solved. The following parameter values are used in each run:

- Board size: **6**
- Exit height: **6**
- Number of cars size 2: **5**
- Number of cars size 3: **3**
- Maximum search depth: **10**

The timings for all the runs can be found in Table 4.1 and 4.2. We make the following observations based on these results.

- *There is no significant increase in the execution time of the solver when the size of the optimal solution increases.*
  In Table 4.2, it looks like the generation time increase when the board with optimal solution size 10 is generated. This does not seem significant because in Table 4.1 the higher value occurs when generating the board of optimal solution size 2, but the time decreases again for the most difficult board created. Even if the rise in generation time is due to the generation of more difficult boards, the increase is not significant when compared with the increase of solver time.

- *Solver time increases with optimal solution size.*
  Our results show that the solver time grows more than quadratic with the size of the optimal solution. This behavior is due to the search tree growing exponentially with the number of possible moves. The larger solution, the more levels the solver has to traverse.

Size optimal solution	Generator time (ms)	Solver time (ms)
1	< 1	< 1
2	< 1	< 1
2	1	1
6	< 1	1048

Table 4.1: DFS: End to end performance

Size optimal solution	Generator time (ms)	Solver time (ms)
2	< 1	< 1
2	< 1	< 1
4	< 1	5
10	1	135271

Table 4.2: BFS: End to end performance

The results provide empirically evidence to confirm the statement made in Section 3.1: because of our simplistic generator, the bulk of the computation has to be done by the solver. Since the execution time of the solver is the most significant part of the total execution time, optimizing the performance of the solver brings much more benefits than optimizing the generator. Thus, the remainder of this chapter focuses on the performance of the solvers.

4.2 BFS solver performance

For the rest of the performance experiments, we use pregenerated boards, a choice enabling accurate comparisons between implementations. The seven boards used in this chapter are presented in Table 4.3. The visualization and complete details for all boards used in this project can be found in Appendix A. Because the boards are pregenerated, no time is spend on the generation when the program is run.

Name	Size optimal solution	Cars of size 2	Cars of size 3	Total cars
B1	4	7	3	10
B2	6	7	4	11
B3	8	4	4	8
B4	10	7	3	10
B5	12	9	3	12
B6	16	11	2	13
B7	20	8	3	11

Table 4.3: Boards used for experiments. Size: 6×6, exit height: 3

To see the performance of the BFS solver each of the pregenerated boards is solved. The following metrics are collected for each board: total solver time, number of nodes traversed, queue size upon finding the solution, and the average time per node. To calculate the time per node $\frac{\texttt{Solver time}}{\texttt{nodes traversed}}$ is used. When using the BFS solver, changing the max search depth value does not change the execution time, as long as the value is greater than the solution size. The goal of the benchmarking performed on these first prototypes is to see where most of the performance can be won.

Board	Solver Time (ms)	Total nodes	Time per node (μs)	Queue size
B1	18	6997	2.57	53051
B2	380	217398	1.74	1709014
B3	9340	6543626	1.43	53890174
B4	4035	3984420	1.01	18466224
B5	3207	2557036	1.25	15183928
B6	Did not complete	-	-	-
B7	Did not complete	-	-	-

Table 4.4: First prototype BFS solver

The measurement results can be found in Table 4.4. When solving boards B6 and B7, the solver did not return a result, i.e., it did not manage to finish normally, although the DAS-5 scheduler did not kill the program. When the program was not able to finish, "`Did not complete`" is placed in the table. Such cases appear when the system running the program runs out of memory. The following observations are made based on the results in Table 4.4.

- *Queue size is always larger than the number of traversed nodes.*
  The size of the tree grows exponentially with the average number of possible moves. This means that, on average, for each node that is taken from the queue, the average number of moves (i.e., nodes) is added back into the queue. This exponential growth is also the reason for which the program can quickly run out of memory.

- *Execution time and the number of traversed nodes are correlated.*
  When ordering the boards by execution time or by number of nodes traversed, the same order would appear, indicating that reducing the number of traversed nodes would also reduce the execution time.

- *Optimal solution size is not the only variable that determines solver time or number traversed nodes.*
  If a board would be solved twice, once normally and once with the first move of the optimal solution already applied, the solver would need to traverse less nodes in the second run. This is rather obvious, because of the way the algorithm traverses the search space. The data from Table 4.4 clearly shows that a larger optimal solution does not guarantee more traversed nodes.

Without any data to compare the BFS solver performance against, no statement can be made on how well it performs. To increase the performance of the BFS solver, reducing the number of traversed nodes is the first logical step, as it should both reduce the amount of work that needs to be done, and allows the solver to handle larger search trees.

4.3 DFS solver performance

To measure the DFS solver performance, we perform a similar set of experiments as in Section 4.2: the DFS solver has to solve the same pregenerated boards, B1-B7, and we use the same performance metrics.

Additionally, when measuring the performance of the DFS solver changing, the max search depth value also matter. A higher max search depth might result in a higher number of nodes traversed, and impact therefore the execution time. In order to see the impact of changing this value, all experiments are done twice, once with the value fixed at 20, the other one with the value set to the size of the optimal solution.

Board	Solver Time (ms)	Total nodes	Time per node (μs)
B1	380	905604	0.42
B2	3044	9059932	0.35
B3	6655	25463507	0.26
B4	7123	24900099	0.28
B5	4161	11803006	0.35
B6	Out of time	-	-
B7	Out of time	-	-

Table 4.5: First prototype DFS solver, max search depth: 20

Board	Solver Time (ms)	Total nodes	Time per node (μs)
B1	5	6997	0.71
B2	115	217398	0.53
B3	1856	6543626	0.28
B4	1305	3984420	0.34
B5	1057	2557036	0.41
B6	Out of time	-	-
B7	Out of time	-	-

Table 4.6: First prototype DFS solver, max search depth: size optimal solution

The results for the experiments are presented in Table 4.5 and Table 4.6, for both max search depth (20 and the optimal solution size, respectively). We note that the DFS solver is not able to solve B6 and B7, for which the time to solution went over the 15 min limit set by the DAS-5 scheduler. Based on the results of the DFS solver benchmarking, we make the following observations.

- *Time per node changes on different runs of the same board.*
  Comparing the time per node for each board between Table 4.5 and Table 4.6 shows that the time per node value is not stable for each node. This can have multiple causes: when the same board is solved in more time, the overhead for setting up the DFS solver could be more significant. There is some overhead in starting the DFS solver, i.e., memory space allocation for the solution and the board. The extra time created by the overhed is included in the solver time measurements. The overhead is not dependent on the board size. If the time taken to solve the board is lower the overhead has more impact on the time per node. When looking back at the data from the end to end performance measurements, presented in Table 4.1, it is clear that this cannot be the only reason: in the end to end test, the solver took < 1 ms to solve the two simplest boards. This time, including the overhead, is less than 0.1% of the solver time for B5. The rise in time per node between the two runs is too large to be only attributed to overhead. The other explanation could relate to measurement errors, as all tests are performed once. Multiple runs might eliminate potential outliers and give a better indication of the expected average time per node.
  Each test is only run once because we don't need very accurate time measurements. The total number of nodes stays the same if we run the test multiple time. From the time measuremntes we only want to know where most of the execution time is spend. This is not reliant on ms accurate measurements.

- *Reducing the max search depth increases performance.*

This observation was expected based on the way the DFS algorithm works. Reducing the max search depth does decrease the work that has to be done, as it will reduce the number of nodes that need to be traversed. Only if the first path traversed is the optimal solution will lowering the max search depth not decrease the number of traversed nodes.

- *Reducing the max search depth does not increase performance equally across all tests.* The first reason for this is that the reduction is not equally large in all tests. With B5 the max search depth was reduced from 20 to 12, for B2 it was reduced from 20 to 6. This is not the only explanation, because, with the max search depth set at 20, B3 took longer to solve than B5, with the reduced max search depth B5 was solved faster than B3. A few other factors are important in determining why reducing the max search depth affects some boards more than others: how fast a solution is found, and how fast the optimal solution is found. When a solution is found, all nodes on an equal or lower level will not be traversed, i.e., the initial max search depth will have no effect from then on. When an optimal solution is found the max search depth will be 1 lower than the level set in the second experiment.

Setting the max search depth as low as possible is important in increasing the performance of the DFS solver. This is difficult because there is no knowledge about the solution size before the board is solved. When the value is set lower than the solution size the DFS solver will not be able to solve the board. Setting the value too high will decrease the performance of the solver.

4.4 Comparing the solvers: BFS vs DFS

When looking at the two solver separately, possible performance improvements can be suggested based on the conducted measurements. In order to say something about how good a solver performs it has to be compared with something. Figure 4.1, 4.2, 4.3 compare the solver time, number of nodes traversed and the time per node of the two solvers.

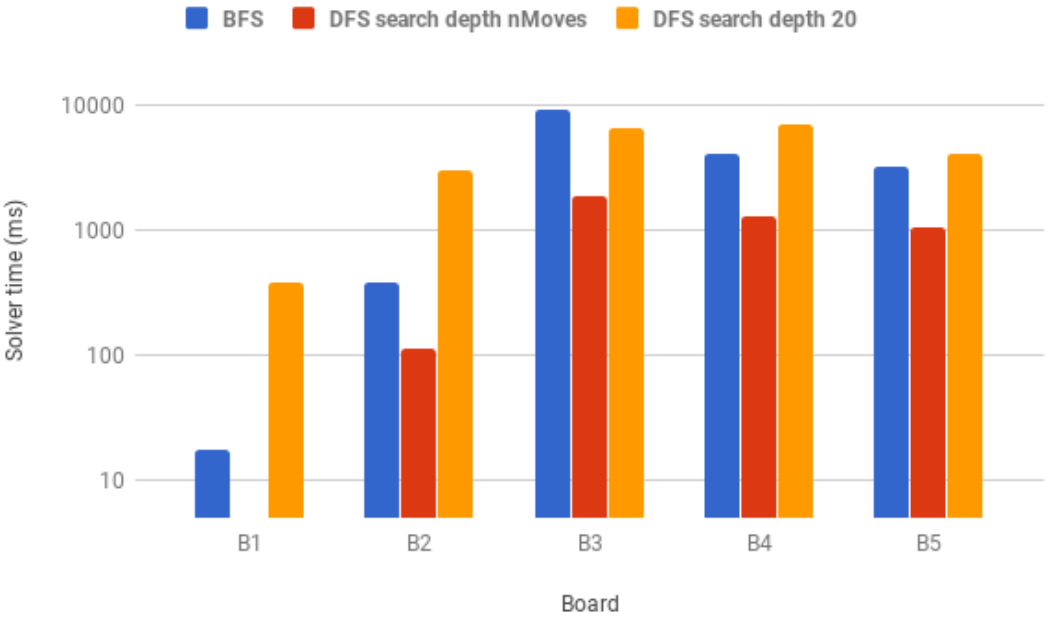


Figure 4.1: Solver time comparison BFS and DFS solver. Vertical axis on log scale.

When comparing the two solvers the following observations are made.

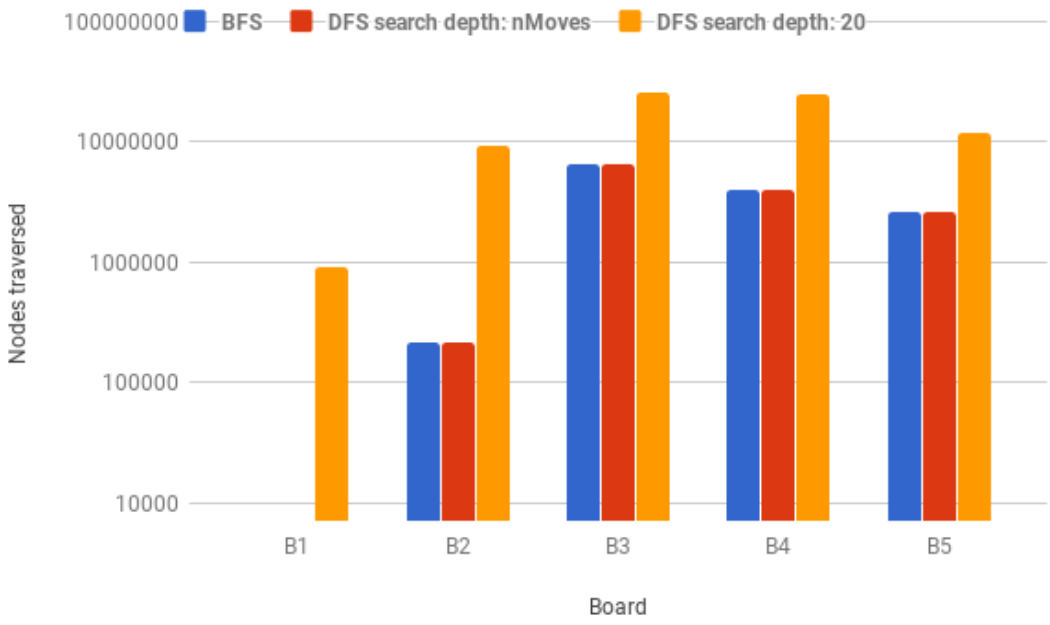


Figure 4.2: Traversed nodes comparison BFS and DFS solver. Vertical axis on log scale.

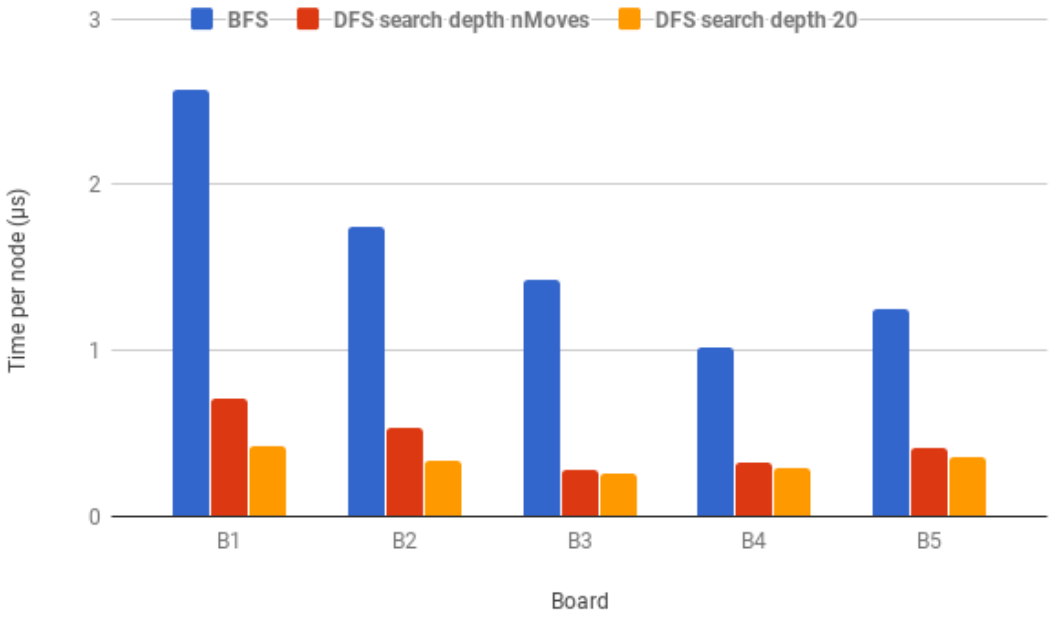


Figure 4.3: Time per node comparison BFS and DFS solver.

- *The DFS solver uses less time per node.*
  The BFS solver spends on average at least twice as much time on each node as the DFS solver. This can be explained by the fact that the BFS needs to perform a lot of memory operations, i.e., allocating and freeing of the queue items. The DFS also has to recreate each node based on the list of moves. These two operations increase the time per node of the BFS.
- *When the search depth is set to the optimal solution size the DFS and BFS traverse the*

same number of nodes.
This make sense when looking at the way the two solver traverse the tree. The BFS will traverse all nodes with a lower depth and all node with the same depth that are before the last node in the optimal solution. The DFS will traverse sub trees before the optimal solution, assuming no non optimal solution is found. When finding the optimal solution the DFS solver will continue traversing up until the depth of the solution minus one. When the max search depth is set to the optimal solution size this results in the two solver traversing the same node, thought be it in a different order.

- *When the search depth is set to 20 the BFS is faster is most of the tests.*
  Although the DFS solver takes less time per node than the BFS solver. When max search depth is larger than the optimal solution size the BFS solver needs to traverse less nodes. This difference in the number of traversed nodes make the BFS solver faster in solving all but one board.

- *The number of nodes traversed for all setups decrease after B3*
  When solving B3 the BFS solver almost traverses twice the number of nodes compared to B4. For B4 and B5 this values is much closer together. An explanation for this is the number of cars, B3 has the lowest number of cars. With less cars on the board there are more empty spaces for the cars to move to. It could be that B3 has more possible moves per configuration. This results in more possibilities that need to be traversed.

Based on the measured performance the conclusion is made that the DFS solver is faster if the max search depth is low enough. The problem is that this knowledge is not available beforehand. Setting the max search depth very low will allow the DFS to finish sooner, but it will not solve more difficult boards. Because a simplistic generator is used, there is very little knowledge about the board before solving it. The advantage of the BFS solver is that it will always traverses the same or less nodes than the DFS solver.

In order to increase the performance either the time per node has to decrease or the number of traversed nodes. Decreasing the number of traversed nodes has the added benefit of decreasing the queue size. Decreasing the queue size will allow the BFS solvers to solve more difficult boards. Because of this advantage, the performance improvements will focus on reducing the number of node traversed.

CHAPTER 5

Performance Improvements

5.1 Queue pruning

5.1.1 Implementation

The idea behind the first performance improvement is to *decrease the number of nodes that are traversed* when solving a board. We envision two options.

First, we could devise a more intelligent traversing order, replacing the order based on the cars list (like in the first prototype) with a an order where the path with the best chance to reach the solution is checked first. While this is a very appealing solution, it would require extensive analysis and prediction of path to success, which are not available for Rush Hour. While mathematical models could be devised to support this approach, we decided this would be difficult to define and prove in such a short time. Thus, we discarded this solution.

The second option is a more naive approach, where we ensure the algorithm does no redundant work, i.e., it is not traversing previously seen configurations. This approach does not require extra knowledge about the game, and can be solved computationally, at runtime. Creating intelligent strategies to solve Rush Hour is outside the scope of this project, for this reason the focus is on the second approach

Redundant work elimination is based on the following observation: If two nodes in the tree represent the same configurations, i.e., all cars are in the same position, their sub trees are also identical. Traversing *both* these sub trees introduces unnecessary computation. To detect if a configuration has already been traversed, a list of all previously seen configurations would be needed, but creating such a list would be memory- and computationally-intensive work. However, the BFS solver already has a list with configurations, which is the queue itself. The queue holds all configurations that are to be processed. Thus, by removing all duplicate configurations from the queue, we perform *queue pruning*. In turn, this process removes unnecessary nodes from the queue, resulting in fewer traversed nodes.

Two methods for queue pruning are implemented for this project: post-processing and pre-processing. Post-processing queue pruning is done upon processing a node from the queue: when a node is taken from the queue, all subsequent nodes in the queue representing the same configuration are removed. Pre-processing queue pruning happens when a new item is added to queue. Each item that is added to the queue is compared to all items already present in the queue until a match is found. If no match is found the item is added to the queue.

In order to determine if two nodes result in the same configuration, the total movement per car, i.e., the distance from the starting position, can be compared. Since each queue item only holds a list of moves, the total movement per car must be calculated. This is done by taking the sum of all moves done per car: if this sum is the same for all cars, the configurations are equal, see Algorithm 3.

Algorithm 3 Algorithm for determining end state equality of two queue items

Precondition: item1, item2 ▷ Two queue items
Precondition: cars1, cars2 ▷ List of size nCars filled with 0's

```
1: areItemsEqual ← 1
2:
3: for all move ∈ item1.moves do                    ▷ Some all movements for item1
4:     cars1[move.car] ← cars1[move.car] + move.move
5:
6: for all moves ∈ item2.moves do                   ▷ Some all movements for item2
7:     cars2[move.car] ← cars1[move.car] + move.move
8:
9: for i ∈ [0, nCars] do                            ▷ Check for equality
10:    if cars1[i] ≠ cars2[i] then
11:        areItemsEqual ← 0
```

5.1.2 Performance

To measure the queue pruning performance the BFS solver will solve all boards presented in Table 4.3 both with pre and post-processing. The results for B1 can be found in Table 5.1. For both pre and post-processing, all the other boards (except B1) did not finish within the 15 minute time limit.

A comparison between the BFS with and without queue pruning is presented in Figure 5.1 and Figure 5.2. The following observations are made based on the results.

Queue pruning method	Board	Solver Time (ms)	Total nodes	Time per node (μs)	Queue size
Pre	B1	15074	6858	2198.02	51889
Post	B1	1803	6445	279.75	48837

Table 5.1: BFS solver with both queue pruning methods. All boards that finished within 15 minutes

- *Pre-processing is slower than post-processing*
  A big factor in this is the frequency of the queue pruning. The post-processing loops through each node in the queue each time a node is traversed. Pre-processing needs to loop through the queue for each node that is added to the queue. Although on average processing one configuration with the pre-processing method take less time then with the post-processing method, pre-processing needs to process more nodes. The reason pre-processing takes less time on average is because it does not always have to loop through the entire queue, it stops when a duplicate is found.

- *Pre-processing traverses more nodes then post-processing.*
  When looking at Figure 5.1 it is clear that post-processing traverses fewer nodes then pre-processing, i.e., it is more efficient at avoiding duplicate configuration traversals. The difference between the two methods is that pre-processing looks at the nodes that are traversed between the current node and itself, post-processing looks at all nodes that are known to come after it. Although it is likely that this difference impacts the overall performance difference, no good explanation was found on why/how this difference make post-processing more efficient.

- *No solvers stopped due to memory shortage.*
  This behavior is due to the solver being too slow to fill the available memory before the

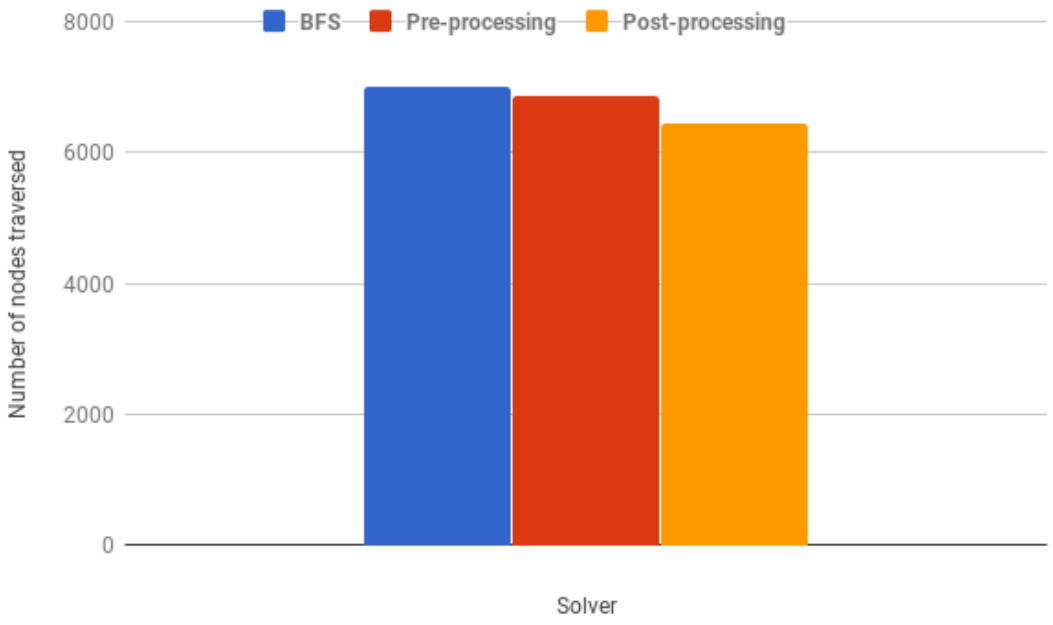


Figure 5.1: Number of nodes traversed while solving B1 for BFS with and without queue pruning methods.

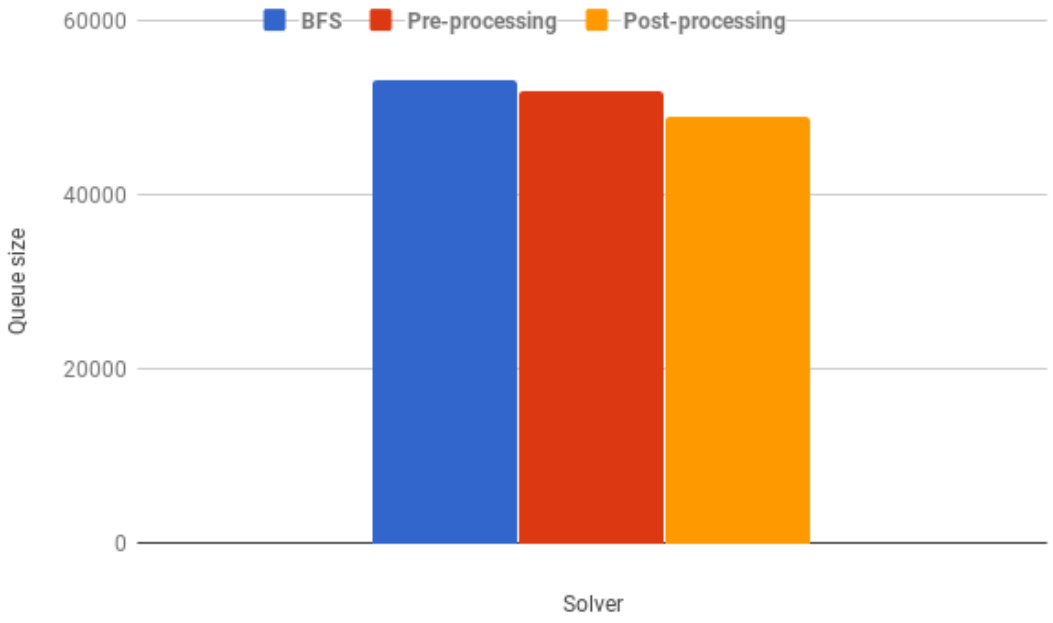


Figure 5.2: Queue size when solved for B1 for BFS with and without queue pruning methods.

15 minute mark. This conclusion is based on the large increase in solver time and the fact that the standard BFS solver was able to solve B2 - B5.

- *Both queue pruning methods are slower than the standard BFS solver*
  Although both queue pruning methods reduced the number of traversed nodes, they both also increased the processing time per node significantly. The extra execution time from pruning the queue was, in both cases, more then the time gained by traversing fewer nodes.

Although both queue pruning methods achieve the desired effect of reducing the number of traversed nodes, the reduction is not sufficient to outweigh the extra computation of pruning the queue. The reduction of traversed nodes is less than the predicted number of duplicate configurations from Section 5.1.1. This is because the queue is only a selection of the nodes that will be traversed. Nodes that have already been traversed will not be removed from the queue. The added computation required to loop through the queue as often as the queue pruning methods do is much more than the benefit gained from traversing fewer nodes. Three different approaches are possible to alleviate this problem. First, the queue could be pruned less frequently, reducing the time spend on pruning, while still reducing the number of traversed nodes. Second, a better representation of the configuration could be stored with the move list, requiring more memory, as explained in Section 3.2.1, but reducing the time taken to loop through the queue. Third, and last, we could store every configuration that has bee traversed and check new configurations against this list; this solution also requires extra memory, but allows the pruning methods to be more efficient in avoiding duplicate configurations.

5.2 Early solution detection

5.2.1 Implementation

The queue pruning process is able to reduce the number of traversed nodes, but it is too slow to improve the BFS solver performance. A faster method for reducing the number of traversed nodes is needed. The BFS solver recognizes the solved configuration when it is taken from the queue for processing. This solved configuration has been in the queue since its parent node was processed. Early solution detection will detect the solved configuration before it is added to the queue.

The BFS solver using early solution detection checks if a node is solved before it is added to the queue. When a solved configuration is found, the BFS solver needs to be sure it is an optimal solution. The solved configuration is guaranteed to be part of an optimal solution if none of the items in the queue represent solved configurations. If no queue items represent a solved configuration, the new configuration is the first solved configuration, and thus the queue item holds an optimal solution. When early solution detection is used, it is guaranteed that none of the queue items represent solved configurations, because all items where checked before being added to the queue.

In practice the solver does not have to check each created child node. For each node that creates child nodes, only one node can be a solved configuration, the largest movement of the target car, because the only way to get a solved configuration from an unsolved one is to move the target car as far right as possible. Each node taken from the queue checks if $targetCar.x + moveRange.max = board.size - targetCar.size$ is true, where `moveRange` is the movement range of the target car as described in Chapter 3. If the statement is true, the current node plus the move $(0, moveRange.max)$ is and optimal solution. The solver stops the solving processes and returns this solution.

When early solution detection is used, the number of nodes traversed + the queue size +1 is equal to the number of nodes that would be traversed by our first BFS solver. Each item in the queue when the solver using early solution detection solves the board, would have been processed by the standard BFS solver before it finding the same solution. In Chapter 4 we saw that when the board is solved, the queue size is always larger then the number of traversed nodes. This means that, in theory, using early solution detection should reduce the number of traversed nodes by more than half.

5.2.2 Performance

The performance of early detection is measured by solving each board represented in Table 4.3. The results are presented in 5.2, and a visual comparison of the early detection data against the

BFS solver can be found in Figure 5.3, Figure 5.4 and Figure 5.5.

Board	Solver Time (ms)	Total nodes	Time per node (μs)	Queue size
B1	2	809	2.47	6187
B2	58	24524	2.37	192873
B3	1092	712380	1.53	5831245
B4	844	724120	1.17	35831245
B5	612	379267	1.61	2177768
B6	Did not complete	-	-	-
B7	Did not complete	-	-	-

Table 5.2: First prototype BFS solver

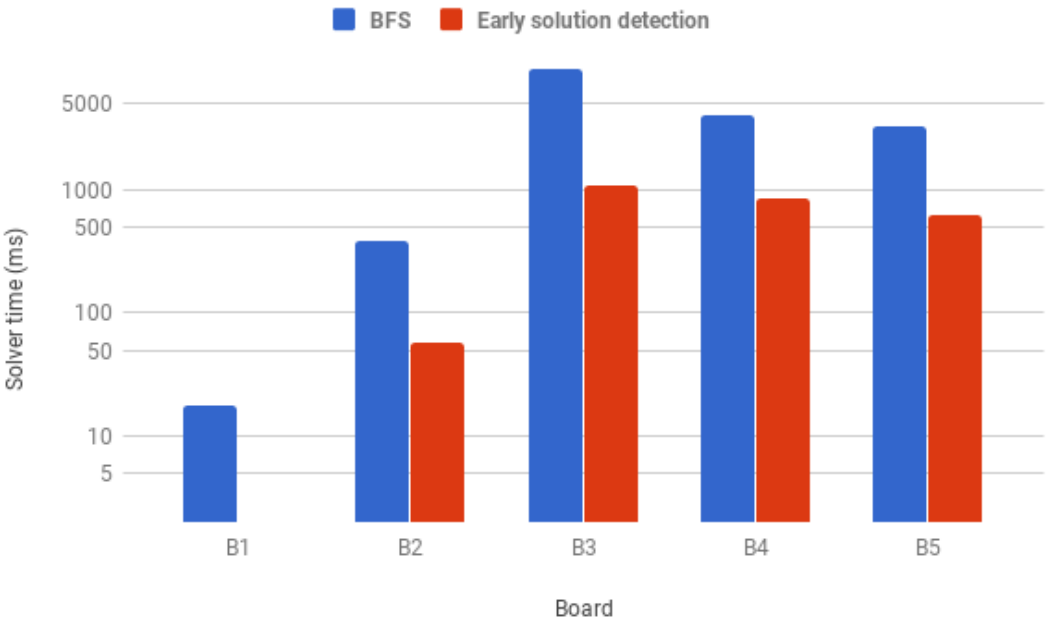


Figure 5.3: Solver time early detection and BFS solver. Vertical axis on log scale.

Based on the results, the following observations are made.

- *The BFS solver with or without early detection is not able to run for 15 min when solving B6 and B7.*
  When solving B6 and B7, the BFS solver stop because it runs out of memory. With early detection, the BFS solver is still not able to complete the solving of these boards without running out of memory. This means that at the moment the BFS solver runs out of memory, the solution is not in the queue.
- *The BFS solver performs better with early detection for all boards.*
  The extra computation required to detect if one of the child nodes is a solved board is minimal. The reduction in the number of traversed nodes is close to the size of the queue upon solving. This reduction does decrease the execution time more than the extra computation that is required.

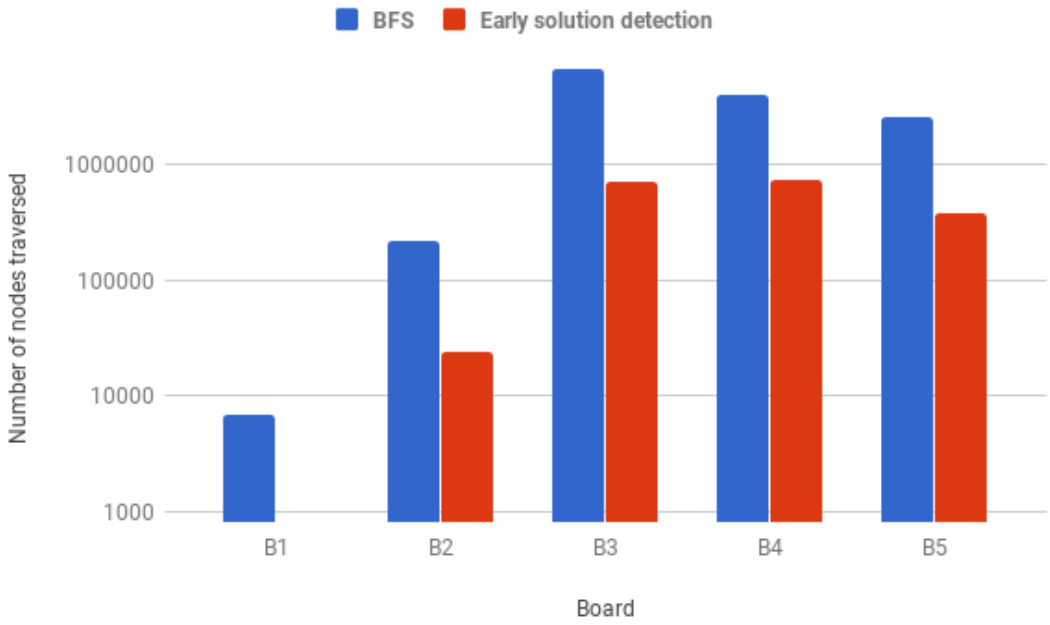


Figure 5.4: Number of nodes traversed early detection and BFS solver. Vertical axis on log scale.

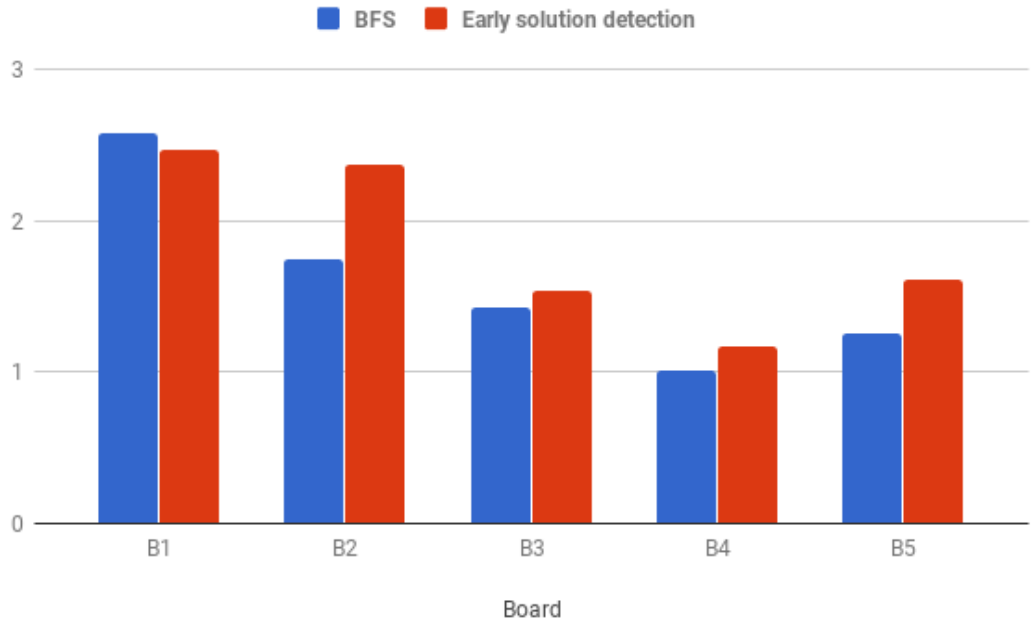


Figure 5.5: Time per node early detection and BFS solver. Vertical axis on log scale.

- *The number of nodes traversed is less than half that of the BFS solver.*
  As predicted in Section 5.2.1, using early solution detection lowers the number of traversed nodes by more than half when compared to the first prototype BFS solver.

The goal of the early detection is to reduce the execution time by reducing the number of traversed nodes. Using early detection reduces the number of nodes traversed and the execution time significantly. However, early detection does not reduce the number of nodes to be processed

before the solution, i.e., the configurations placed in the queue before the solved configuration are the same. This means that while the solution is not in the queue, the solver runs the same as the BFS solver.

In Section 4, we observed that the DFS solver traversed the same number of nodes as the BFS solver if the max search depth was set to the solution size. Because the DFS solver spends less time on each node, when the max search depth was low enough, the DFS solver performed better than the BFS solver. With early solution detection, if a board is solved, the BFS solver is now guaranteed to traverse fewer nodes then the DFS solver for every max search depth value.

CHAPTER 6

Hybrid solution

With early solution detection, the BFS solver is now faster for all the pregenerated boards. The problem is still that the BFS solver uses all the available memory if the search space is too large. Although the DFS solver is slower it does not have this problem. To be able to solve larger boards but keep the BFS speed, we propose *a hybrid solution using a search-depth cutoff point.* Specifically, the hybrid solution uses the BFS solver until a predefined cutoff point, from where the DFS solver takes over by solving each configuration in the queue.

6.1 Implementation

The hybrid implementation, combining the BFS and DFS solver, is presented in Algorithm 4. The BFS solver runs with the cutoff point as maximum search depth. The solver runs normally until the node being processed has a depth higher than the max search depth, i.e, until the cutoff point is reached. The node is then added back to the queue, and the BFS immediately stops; the current queue is the one to be used further by the DFS. Each item of the queue must now be solved by the DFS solver: as long as the queue is not empty, a new item is taken from the queue, its board configuration is reconstructed (from the list of moves), and the board is given to the DFS solver. The DFS solver will traverse the sub tree created by the configuration and return. The best found solution, if there is one, is given to the DFS solver for the next configuration. The size of the best known solution is used as the max search depth, as is done in the first prototype DFS solver.

Algorithm 4 Hybrid solver combining DFS and BFS

Precondition: board ▷ The board that needs solving
Precondition: cutoff ▷ The depth where the DFS will take over

```
1:  solution ← BFSSolver(board, cutoff)                    ▷ Empty list of moves
2:
3:  if solution ! = NULL then                              ▷ If a solution is found
4:      return solution
5:
6:  while queue ! = empty do
7:      item ← popFromQueue( )
8:      curBoard ← createBoardFromMoves(board, item.moves)
9:      solution ← DFSSolver(curBoard, solution)
10:
11: return solution
```

6.2 Results

For insightful data, solving each board once is insufficient: the boards need to be solved with different cutoff points. First, there are two cutoff points which will not result in any useful data. For example, setting the cutoff point to 0 will result in the “pure" DFS solver. When the cutoff point is higher then the solution size, only the BFS solver will be used.

Avoiding these corner cases, each board is now tested for all possible even cutoff points between 0 and the *solution size* - e.g., when the solution size is 8, the cutoff points 2, 4, 6 are tested. The BFS solver used for the hybrid solver uses early solution detection. Because the cutoff point is always at least 2 lower then the solution size, early solution detection will not find the solutions. Early solution detection is still enabled to create accurate measurements of the best performing solver.

As performance indicators, we measure the same metrics as in before. These metrics are, solver time, number of nodes traversed, average time per node and queue size. Here the queue size is the size when the BFS solver reaches the cutoff point. Additionally, the individual solver times for the BFS and DFS solvers are also reported. The conversion of the queue items to boards is included in DFS's solve time measurement. The max search depth is set to 20. The results can be found in Table 6.1. Figure 6.1 and Figure 6.2 illustrate a comparison of the performance of the hybrid solver against the DFS and BFS solvers. Figure 6.3 and Figure 6.4 show the number of traversed nodes.

Based on these results, we make the following observations.

- *The hybrid solution performs worse than the BFS solver with early solution detection.*
  This happens because the BFS solver using early solution detection will have to traverse fewer nodes then the hybrid solver does. If the cutoff point was set higher, the hybrid solver would perform identical to the BFS solver with early solution detection.
- *Changing the cutoff point has a large, and potentially unpredictable, effect on the solver time.*
  Per configuration, the generated child nodes are ordered the same in the BFS and DFS solver. This should mean that traversing the first levels using BFS and then switching to DFS should not change total number of nodes traversed when compared to the DFS solver. Changing the cutoff point should not change the total number of nodes traversed. From the experiments, this assumptions seems to be false. There is one point where an item in the queue changes position which could change the total number of traversed nodes. This point is when the cutoff point is found by the BFS solver: when the first item is taken from the queue, it is determined that the DFS solver will take over. At this point the configuration is pushed to the queue, changing its position from first to last. This changes the order in which the DFS solver traverses the nodes, and can be the cause of the different number of traversed nodes and change in execution time.
- *The hybrid solver spends most of its time in the DFS solver.*
  The DFS solver has to traverse more nodes then the BFS solver in the hybrid solution. When the cutoff point is reached, the number of nodes traversed is fewer than the size of the queue. After solving, the total number of nodes traversed is more than double the queue size in each test. This means that the nodes traversed by the DFS solver are significantly more than those solved by the BFS solver, because the DFS solver still needs to traverse till the max search depth, which is set at 20. The overhead generated by creating the boards from the queue items also adds to the longer DFS solver execution time.
- *The hybrid solver was not able to solve B6 and B7.*
  Although the hybrid solver is not able to solve B6 and B7, using the hybrid solver allows the solver to reach the 15 minute mark. Because the solver is able to run till the 15 minute mark, we are confident that providing more time for the solving would eventually enable a solution to be found.

Board	BFS search depth	Solver Time (ms)	BFS Time (ms)	DFS Time (ms)	Total nodes	Time per node (μs)	Queue size
B1	2	304	0	304	689533	0.44	98
B2	2	2473	0	2473	7087724	0.44	89
B2	4	53288	2	53286	165570841	0.35	7153
B3	2	10641	0	10641	40467749	0.26	95
B3	4	5414	2	5412	20042950	0.27	7485
B3	6	9722	128	9581	36548327	0.266	616006
B4	2	10008	0	10008	34459235	0.29	16
B4	4	11059	0	11059	38468192	0.29	379
B4	6	6408	4	6404	22028530	0.29	9797
B4	8	9890	89	9795	34354806	0.29	275736
B5	2	2087	0	2087	5560972	0.38	5
B5	4	956	0	956	2204078	0.43	27
B5	6	1108	0	1108	2660821	0.42	241
B5	8	3763	2	3761	10438198	0.36	5761
B5	10	3958	74	3879	10896821	0.36	231397
B6	2 - 10	OOT	-	-	-	-	-
B6	12 - 14	DNC	-	-	-	-	-
B7	2 - 8	OOT	-	-	-	-	-
B7	10 - 16	DNC	-	-	-	-	-

Table 6.1: Hybrid solver with different cutoff points. DNC: Did not complete. OOT: Out of time

- *The different cutoff points do not perform the same over different tests.*
  This behavior is a side effect of the fact that changing the cutoff point has an unpredictable effect on the number of nodes traversed. The only known explanation for this is provided above, i.e., moving one queue item.

Overall, the hybrid solver performance results indicate that the BFS solver with early solution detection remains the best solver in terms of execution time. Ideally, this solver would be used to solve all boards, but this is not possible due to its very large memory footprint: the system would quickly run out of memory before finding solutions for the more complex boards. However, our hybrid solution successfully deals with this by switching over to the DFS solver, which is less time-efficient, but has not memory footprint limitations.

The unpredictable change in the number of traversed nodes poses a problem for determining the best cutoff point. We simplify the problem by making the assumption that the number of traversed nodes stays constant for every cutoff point that is lower then the solution size. Then, the best cutoff point is the one that solves a board as fast as possible, which is different for each board. If we know that the BFS solver can solve the board without running in to memory issues, setting a cutoff point higher then the solution size gives the best performance. When the BFS

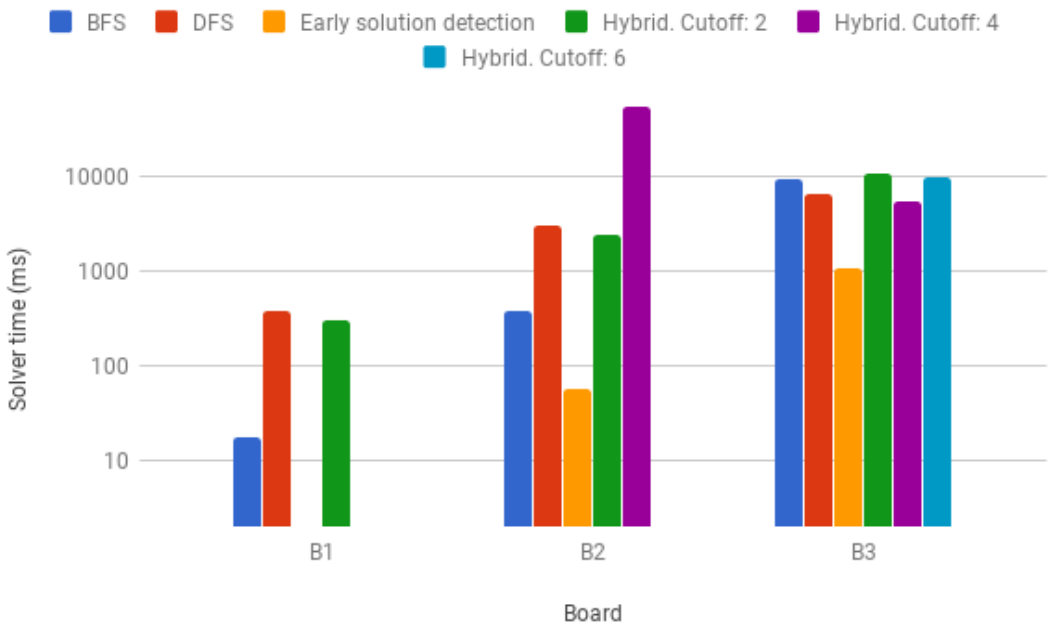


Figure 6.1: Solver time hybrid solver against BFS and DFS solver. Max search depth is 20 in all runs. Vertical axis on log scale

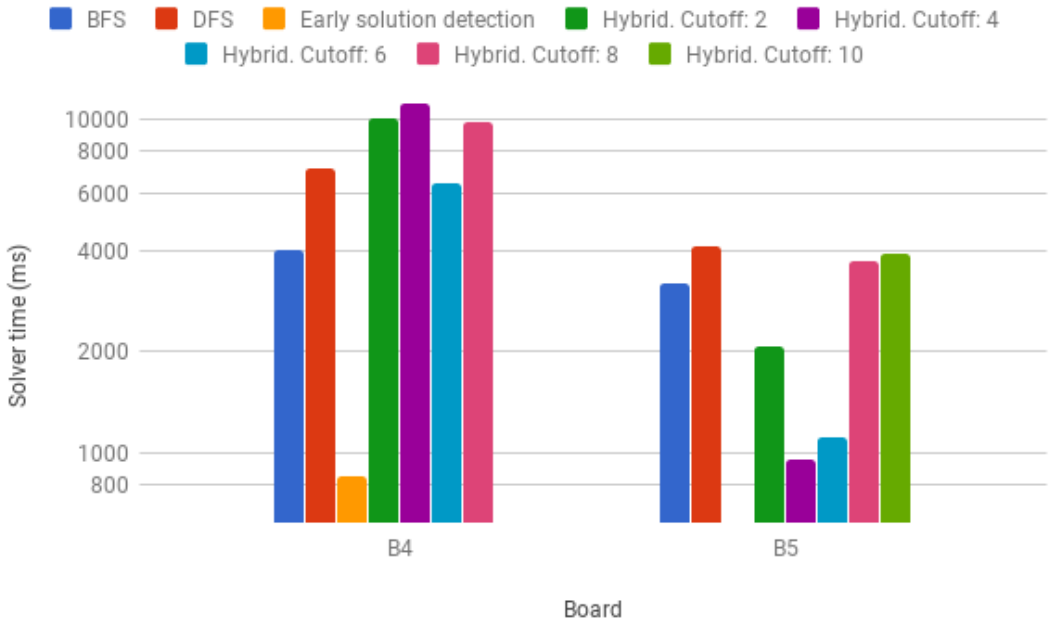


Figure 6.2: solver time hybrid solver against BFS and DFS solver. Max search depth is 20 in all runs. Vertical axis on log scale

solver is not able to solve the board without running in to memory issues the cutoff point must be lowered. It should be set lower then the point where the BFS solver runs out of memory. We know from Chapter 4 that the DFS solver processes each node faster then the BFS solver. When assumed that the number of traversed nodes stay constant. If we lower the cutoff point more of the nodes will be traversed by the DFS solver. Since the DFS solver is faster per node the

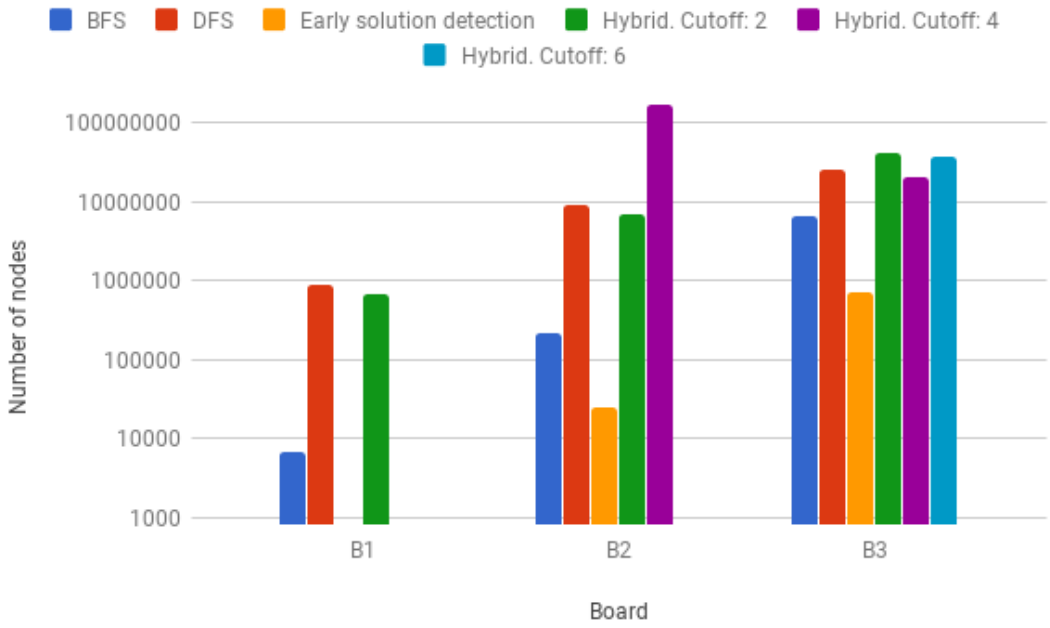


Figure 6.3: Total nodes traversed hybrid solver against BFS and DFS solver. Max search depth is 20 in all runs. Vertical axis on log scale

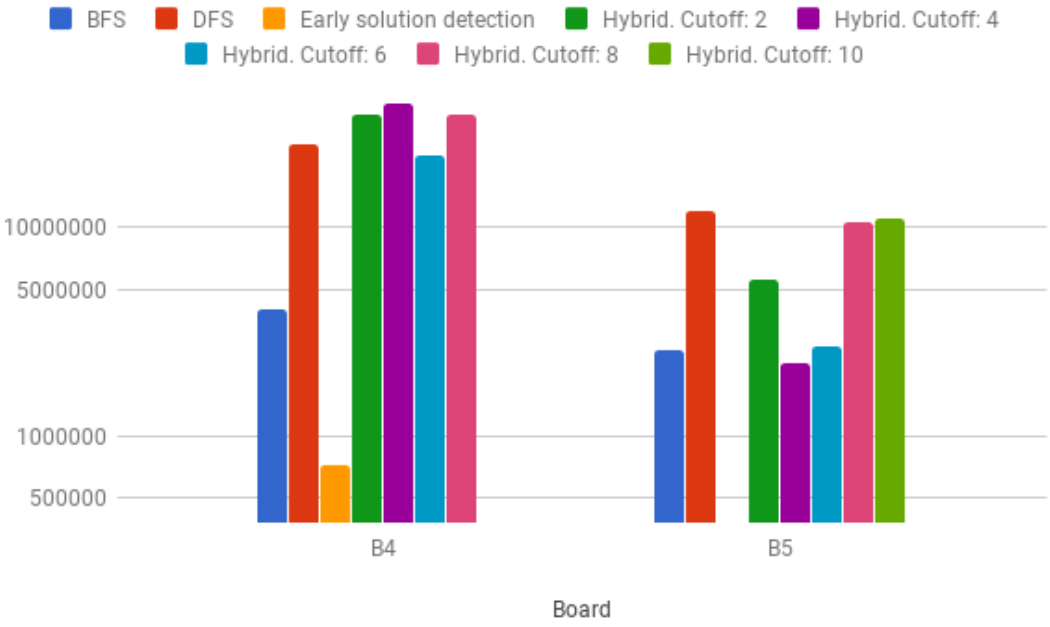


Figure 6.4: Total nodes traversed hybrid solver against BFS and DFS solver. Max search depth is 20 in all runs. Vertical axis on log scale

execution time should be lower for a lower cutoff point. Because the solution size is unknown prior to solving setting the best cutoff point is depend on how much slower the BFS solver is per node and how likely it is that the BFS solver is able to solve it alone.

Based on the data collected when focusing on throughput, the best cutoff point is as high as

possible. Setting the cutoff point as high as possible lets the BFS solver solve as many boards as possible. Since the BFS solver with early solution detection is orders of magnitude faster then the DFS solver, using it to solve as much boards as possible is beneficial for performance.

CHAPTER 7

Conclusion

Rush Hour is a popular puzzle game, which - like many other games - faces a replay-ability challenge. Once a player has solved all the provided levels, there is no reason to keep playing. In order to solve this problem, Procedural Content Generation is used to generate Rush Hour boards. This thesis investigates how we can generate such puzzles efficiently. Puzzle generation requires the creation of a puzzle board and providing a optimal solution for that board. Efficiency is measured in its throughput, i.e., how many such puzzles can be generated within a time unit.

For puzzle generation, we provide an application for Rush Hour puzzle generation. The application has two parts: the generator and the solver. A naive generator generates random boards. The solver finds the optimal solution for each of these boards.

To gain a better understanding of the efficiency of this application, we measured the performance of the puzzle generation, which is dominated by the solving time. In order to improve the performance of the whole program, the performance of the solver needed to increase. Thus, the core of our investigation is the design, implementation, analysis, and optimization of two different approaches for this solver: a BFS one and a DFS one.

We further provide extensive performance data for each version of each solvers, and analyze the differences between them in detail. We find that the biggest differences in performance between the DFS and BFS solver are the time spend per node and the number of nodes traversed.

The DFS solver spend less time on each node. When the maximum search depth, i.e., the traversing depth where the DFS solver starts backtracking, was low enough the number of traversed nodes was equal. However if the maximum search depth is higher the BFS solver traverses significantly fewer nodes. With this high max search depth the BFS approach solves the boards faster. Setting the max search depth low seems like the best solution, but this is not always possible. The solvers will not find a solution whose depth is higher than the max search depth. Since a naive generator is used no prior knowledge about the solution size is available. When using the program to create more difficult boards, the BFS solver will perform better on most boards. The problem with the BFS solver is its memory footprint. When the search space of the board is too large, the BFS solver will run out of memory and crash.

In order to increase the performance and allow the BFS solver to handle more difficult boards, we choose to decrease the number of nodes that needed to be traversed. The first method implemented for this was queue pruning. The BFS solver uses a queue to remember what nodes need to be traversed in what order. If a board configuration occurs multiple times in the queue all but the first can be removed. We implemented two methods of queue pruning. The BFS solver performed worse with both of these methods. The time it takes to compare a node against all the nodes in the queue was too high. The second method for reducing the number of traversed nodes is early solution detection. Early solution detection detects the solution the moment it is added to the queue. This lowers the number of traversed nodes by at least half, usually more. This method improves the performance if a solution is found. If no solution is found the performance is identical to the default BFS solver.

The final solver we implemented was a hybrid solver. The hybrid solver uses both the BFS solver with early solution detection and the DFS solver. The BFS solver traverses the node till a predefined depth. When this depth is met the DFS solver takes over. This allows the program to use the faster BFS solver without running into memory issues. The performance for different cutoff points is not as expected. Changing the cutoff point changed the traversed nodes unpredictably. Although the performance is unpredictable the hybrid solver was successful in avoiding memory issues.

Based on all our results, we can state that *using a BFS solver with early solution detection is an efficient method for generating simple boards.* The simple simple boards (like B1 - B5) can be generated and solved in less then 1 second. We have no data for this but we believe it is obvious that the average player is unable to solve these boards in such a short time. The program we created is thus able to produce boards faster than they will be solved. For the context set around the problem, this is efficient enough. More complex boards (like B6 and B7) take a long time to solve and thus also to generate. The execution times we measured for solving these boards where more then enough for a person to solve them by hand. Based on the research we did the key to a more efficient method is reducing the number nodes traversed during execution, as attempted by queue pruning. When this is done efficiently the hybrid solver has the potential to also solve these more complex boards efficiently.

Future work

Based on the results gathered during the project, we propose several promising research directions.

Reducing the number of nodes that are traversed during the solving process.
We have already attempted such a reduction using the queue pruning method, but it was not efficient enough. Other methods for reducing the number of traversed nodes could be explored. Reducing this number allows the BFS solver to solve larger boards and will improve the performance of both solvers. For the hybrid solution, a new queue pruning method could be explored: removing duplicate nodes in the queue before using the DFS solver could significantly reduce the workload for the DFS solver.

Setting a suitable max search depth.
In the performance measurements from the DFS solver it is clear that a low max search depth is beneficial for its performance. The problem is that setting the max search depth too low results in an unsolved board. Knowing the solution size or an estimation of the solution size can be beneficial for the solver performance. Smarter generation could help with this, by generating boards whose solution sizes are guaranteed to be within a certain range. One way to achieve such behavior is to start from a solved board first, and "reverse engineer" moves towards a starting configuration. The number of moves done to create the staring board can then be used as an upper limit for the solution size.

Parallel processing
This project focused on solving each individual board as fast as possible. Parallelization could also be used to further increase the efficiency of our puzzle generation approach. There are at least two possibilities for parallelization: solving multiple boards at the same time, or solving one board using multiple parallel threads. The generator is fast enough to provide multiple process with boards to solve. Also, for solving one board in parallel multiple possibilities exist. For example, the BFS solver can let multiple processes handle queue items in parallel. The queue pruning process could also be done by a different process, to reduce the work done by the main solver. The hybrid solution also offers simple parallelization options. When the DFS solver takes over from the BFS solver each queue item could be solved by a different process.

Bibliography

[1] H. Bal et al. "A Medium-Scale Distributed System for Computer Science Research: Infrastructure for the Long Term". In: *Computer* 49.5 (May 2016), pp. 54–63. ISSN: 0018-9162. DOI: 10.1109/MC.2016.127.

[2] Martijn Broekstra. "Generating Lunar Lockout Puzzles on the GPU". MA thesis. Vrije Universiteit Amsterdam, 2014.

[3] Irving Finkel and Tom Scott. *The Royal Game of Ur*. Youtube. 2018. URL: https://www.youtube.com/watch?v=WZskjLq040I.

[4] Bjarki Guðlaugsson. "Procedural Content Generation". In: *New Technology* (2006).

[5] Martin Hunt, Christopher Pong, and George Tucker. "Difficulty driven sudoku puzzle generation". In: *UMAPJournal* (2007), p. 343.

[6] WJM Meuffels and Dick den Hertog. "Puzzle Solving the Battleship puzzle as an integer programming problem". In: *informs Transactions on Education* 10.3 (2010), pp. 156–162. DOI: doi.org/10.1287/ited.1100.0047.

[7] David Oranchak. "Evolutionary algorithm for generation of entertaining shinro logic puzzles". In: *European Conference on the Applications of Evolutionary Computation*. Springer. 2010, pp. 181–190.

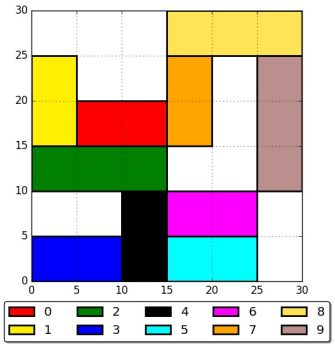
[8] Balázs Pintér et al. "Automated Word Puzzle Generation via Topic Dictionaries". In: *CoRR* abs/1206.0377 (2012). arXiv: 1206.0377. URL: http://arxiv.org/abs/1206.0377.

APPENDIX A

Boards used in experiments

Each board used for testing purposes can be found here. The boards are presented as a visualization and in a text format. The first line in the text format represents the size of the board, the y level of the exit, the number of cars and the number of moves, respectively. Each following line represents a car on the board, each car consists of the x value, the y value, the size and the orientation, 1 for horizontal and 0 for vertical.

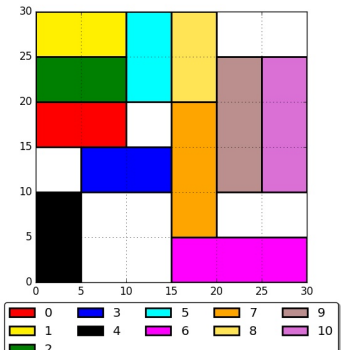
A.1 First prototype




```
6  3  10  0

1  3  2  1
0  3  2  0
0  2  3  1
0  0  2  1
2  0  2  0
3  0  2  1
3  1  2  1
3  3  2  0
3  5  3  1
5  2  3  0
4  1  2  1
```

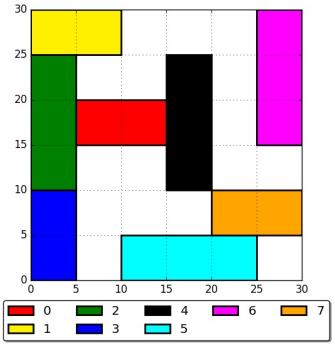
Figure A.1: B1



6 3 11 0

0 3 2 1
0 5 2 1
0 4 2 1
1 2 2 1
0 0 2 0
2 4 2 0
3 0 3 1
3 1 3 0
3 4 2 0
4 2 3 0
5 2 3 0

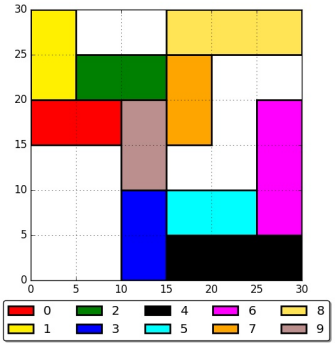
Figure A.2: B2



6 3 8 0

1 3 2 1
0 5 2 1
0 2 3 0
0 0 2 0
3 2 3 0
2 0 3 1
5 3 3 0
4 1 2 1

Figure A.3: B3

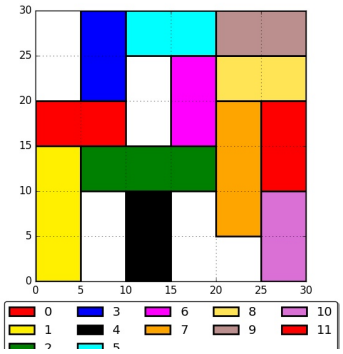


6 3 10 0

0 3 2 1
0 4 2 0
1 4 2 1
2 0 2 0
3 0 3 1
3 1 2 1
5 1 3 0
3 3 2 0
3 5 3 1
2 2 2 0

Figure A.4: B4

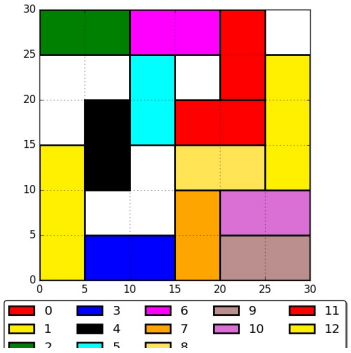
6 3 12 0



0 3 2 1
0 0 3 0
1 2 3 1
1 4 2 0
2 0 2 0
2 5 2 1
3 3 2 0
4 1 3 0
4 4 2 1
4 5 2 1
5 0 2 0
5 2 2 0

Figure A.5: B5

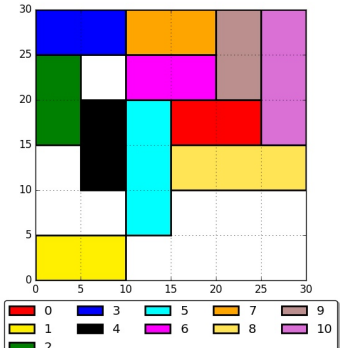
6 3 13 0



3 3 2 1
0 0 3 0
0 5 2 1
1 0 2 1
1 2 2 0
2 3 2 0
2 5 2 1
3 0 2 0
3 2 2 1
4 0 2 1
4 1 2 1
4 4 2 0
5 2 3 0

Figure A.6: B6

6 3 11 0



3 3 2 1
0 0 2 1
0 3 2 0
0 5 2 1
1 2 2 0
2 1 3 0
2 4 2 1
2 5 2 1
3 2 3 1
4 4 2 0
5 3 3 0

Figure A.7: B7