

## Rush Hour Puzzle

### Abstract

Rush hour is a sliding block puzzle where game pieces representing vehicles stuck in traffic on a 6x6 grid. The objective of the puzzle is to move a special vehicle, the red car, out of a gridlocked “traffic jam”. In this report, I search for features that are characteristic for configuration within 6x6 Rush Hour, by applying Dijkstra’s algorithm and breadth-first search (BFS) algorithm.


### Introduction

Rush Hour was invented by Nob Yoshigahara in the 1970s, and then first sold in the United States in 1996. It is now being manufactured by ThinkFun. There are eleven regular cars, four trucks, and one special red car. A car or truck occupies two or three successive grid spaces, respectively. Moreover, the vehicles must be placed vertically or horizontally, not diagonally, and they are restricted in their movement as they can only move forwards or backwards, but not through other vehicles or over the edge of the grid. The purpose is to manipulate the vehicles in order that the red car will escape from the “traffic jam”.

The game consists of a deck of 40 cards as different standards, which are rated “beginner” (cards 1 through 10), “intermediate” (numbers 11 through 20), “advanced” (21 through 30), and “expert” (cards 31 through 40). Each card indicates an initial placement of some subset of the vehicles. Also, the red car must be contained in every initial configuration, and positioned somewhere in the second row from the top, because the only exit is located at the right-hand side of this row. The solution of every standard appears on the back of cards.

In this report, I firstly compute Dijkstra’s algorithm to discover a solution, which minimize the number of moves. To speed up this algorithm, breadth-first search algorithm will be derived, because Dijkstra’s algorithm requires full generality. Then I solve the related problem of finding a solution with a minimum number of slides. Finally, I apply both Dijkstra’s algorithm and breadth-first search algorithm to each of the 40 original game cards, and conclude with some observations on the Rush Hour puzzle.

### A Simplified Example

First of all, Let’s start with a simplified but full-sized version of Rush Hour using a  $4 \times 4$  grid. As the example illustrated in *Figure 1*, the goal is to manage the vehicles so that the red car, symbolized as “”, can escape from the grid.

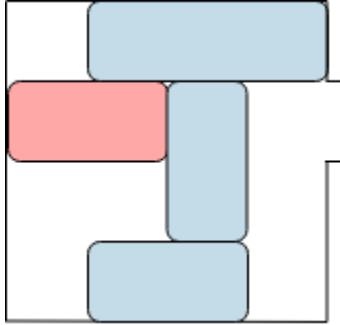


Figure 1: Simplified Rush Hour

A “move” is defined as one car or truck sliding forward or backward any number of grid spaces within the restriction that it cannot take an occupied place. For instance, in *Figure 1*, 3 possible movements can be demonstrated from the original statement – the bottom car can move left or right one grid space, or the top truck can move left one grid space. Moreover, we consider a “slide” to be a move of exactly one grid space. With respect to *Figure 1*, we claim that three moves are demanded to have the red car exited the grid, while a minimal number of slides is five. However, in the situation of a more complicated initial configuration, we would like to develop an algorithmic approach for determining minimal-move and minimal-slide solutions. One of the most useful methods is to build a graph, where every vertex represents a valid board configuration, and edges represent the possible moves. Since we can figure out the minimal-move solution from the graph; after we weigh the edges with the length of the corresponding slides, a minimal-slide solution should be yielded.

In the example of *Figure 1*, the graph is created by having every placement of each car or truck within its respective row or column, and then we reduce the graph to the one that no overlapping vertices consist of all resulting configurations. For the initial configuration in *Figure 1*, 81 placements of the vehicles are investigated, because three placements for each car and two for the truck lead to a total of  $2 \cdot 3^3 = 54$ , where we list the 12 non-overlapping configurations in *Figure 2*. The initial configuration is configuration 0. The configuration 6 to configuration 11 are marked with an asterisk (\*) are winning configurations, because they are just one move away from a win. Based on the index in *Figure 2*, an example of a minimal winning series of moves is provided by vertices (0, 2, 6), and the unique minimal-slide solution is (0, 2, 6, 8, 10).

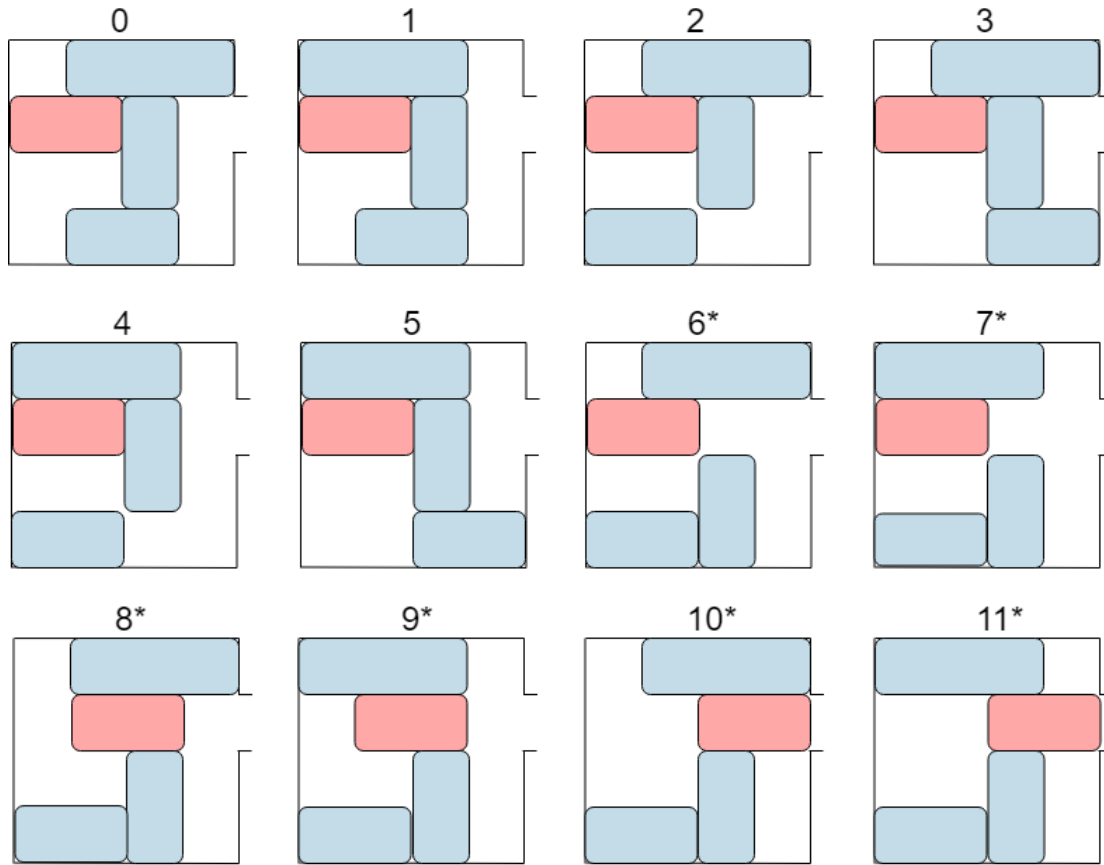


Figure 2: Board configurations for simplified Rush Hour example

## Dijkstra's algorithm

Dijkstra's algorithm is used for finding the lowest-cost path between two specified vertices of an edge weighted graph. In our particular case, the weights of the edge are all non-negative. The algorithm is widely used in real life notably network routing protocols. For instance, if the vertices of the graph denote cities and edge path costs denote driving distances between pairs of cities connected by a direct road, Dijkstra's algorithm discovers the shortest route between one city and all other cities. Consequently, computer networks are using so-called link state routing, such as IS-IS (Intermediate System to Intermediate System) and Open Shortest Path First (OSPF).

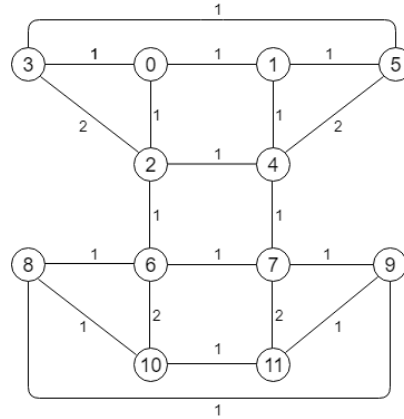


Figure 3: All-move graph for simplified Rush Hour example

Suppose  $G$  is an edge-weighted graph. In order to discover a minimal path with weights of edge from an initial vertex  $v_i$  to the final vertex  $v_f$ , we firstly set all vertices of  $G$  to white, and then the initial vertex  $v_i$  is changed to blue and all of its neighboring vertices are changed to yellow. Determine the distance of the next vertex  $v_j$  to be the minimum weight of a path from  $v_i$  to  $v_j$ . The following is the procedure of the algorithm.

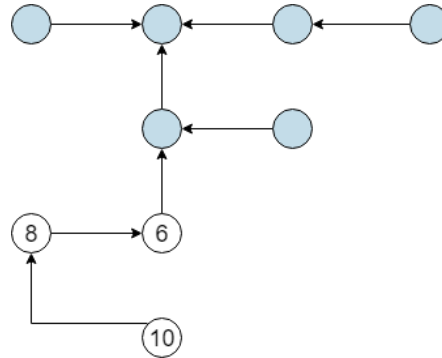
1. If there are no more yellow vertices, a solution does not exist. Otherwise, a yellow vertex  $v_y$  of minimum distance from  $v_i$  is chosen.
2. If  $v_y = v_f$ , a solution is found.
3. If  $v_y \neq v_f$ , set  $v_y$  to blue and record its weight from  $v_i$  (the starting vertex), and change all of  $v_j$ 's white adjacent vertices to yellow.
4. Repeat Step 1.

However, the algorithm can only distinguish whether the final vertex is found or not, instead of recording the path from  $v_i$  to  $v_f$ . As a result, some additional bookkeeping will be required to discover a path. Meanwhile, the initial or final vertex can be simply replaced by a set of vertices.

Now we apply Dijkstra's algorithm to the graph in *Figure 3*, where our purpose is to discover a solution with minimal number of moves. In terms of the graph, we are looking for a shortest path from vertex zero to any of the winning vertices, particularly, vertices one, two, ..., and eleven. We assign all weights of edge equivalently to 1, because we want a minimal-move solution, not a minimal-slide solution. To begin Dijkstra's algorithm, we set vertex zero to blue, and its neighboring vertices, one, two, and three, are set to yellow. Since all three vertices are the same distance, the choice of next vertex is arbitrary. If we choose vertex one as next, we will set it to blue, and also set its white neighbor, vertex four and five, to yellow.

According to the edge weights of one, the yellow vertices are two, three, four, and five, which separately with distance 1, 1, 2, 2 from vertex zero. Therefore, we can select vertex two with minimum distance among all yellow vertices. We change the color of vertex two to blue, and its white neighbors' vertex four and six are changed to yellow. As we proceed with searching, a termination occurs once the first winning vertex is reached. *Figure 4* below allows us to restructure the winning series of moves from various directed edges, like (0, 2, 6). For the

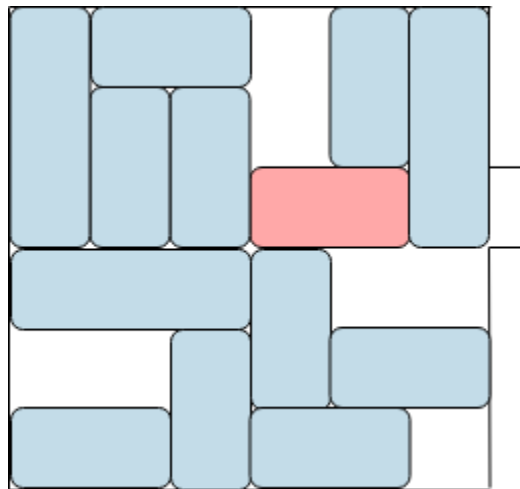
minimal-slide solution, we apply Dijkstra's algorithm on the graph in *Figure 3* with the given edge-weights, the process is very similar by having vertex 8 and 10.



*Figure 4: Dijkstra's algorithm for the simplified example*

### Example of Rush Hour Puzzle (Breadth-first Search Algorithm)

In this section, we analyze card number 40 from real Rush Hour game, and conclude by breadth-first search algorithm from accelerating Dijkstra's algorithm. *Figure 5* is the initial configuration of card number 40. It has six rows and columns with a 2-length red car, three 3-length trucks, and nine 2-length ordinary cars. In order to discover the vertices of the all-move graph, we would like to know all possible placements of the vehicles, and keep track of the result in valid non-overlapping configurations. After examining card number 40, there are  $5^6 * 4^3 * 6^2 = 36000000 = 3.6 * 10^7$  placements. Correspondingly, the all-move graph has 4805 vertices with 18729 edges where 4780 of the vertices are achievable from the initial configuration. So, if a solution exists, it must appear in one of these 4780 vertices, because the solution must start at initial vertex.



*Figure 5: Rush Hour—card number 40*

To discover the minimal move solution, both approaches – trying all  $3.6 * 10^7$  possible placements of the vehicles to create an all-move graph and implementing Dijkstra's algorithm

to find a shortest path – require significantly more expense than what we necessarily need to solve the problem.

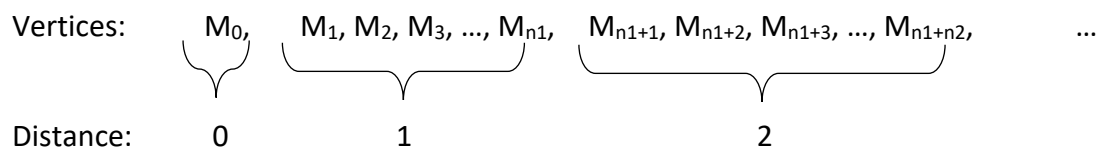
Evidently, the most difficult part for card number 40 is to efficiently discover the all-move graph, and we found that not all of the graph is required by Dijkstra’s algorithm. Respect with the observations from *Figure 3* and *Figure 4*, there exists obvious computational strategies to considerably decrease the required amount of work and computer memory for a minimal-move solution in any Rush Hour puzzle. Hence, we transform Dijkstra’s algorithm to the breadth-first search algorithm.

Let us assume that we consecutively found all vertices that are distance one, distance two, ..., and distance  $n$  from the initial vertex, and we are not getting to any of the winning configurations. Let us also assume that vertex  $m$  is distance  $n$  from the initial vertex. Any vertex adjacently connects to  $m$  would allow us to move the game pieces on the configuration corresponding to  $m$ . For any vertex  $w$  neighbors vertex  $m$ , one of the following four situations must be true.

1. The vertex  $w$  appears in the list of vertices at distance  $n - 1$  from the initial vertex.
2. The vertex  $w$  appears in the list of vertices at distance  $n$  from the initial vertex.
3. The vertex  $w$  appears in the (partial) list of distance  $n + 1$  vertices.
4. The vertex  $w$  does not appear in the list of distance  $n - 1$ , distance  $n$ , or distance  $n + 1$  vertices.

The first three cases define that vertex  $w$  is established, and we continue to the next vertex adjacent to  $w$ . However, if we have case 4, it means that  $w$  is a new vertex at distance  $n + 1$  from the initial vertex. We should primarily check whether  $w$  corresponds to a winning configuration. If so, we are completed. If not, we append  $w$  to the distance  $n + 1$  list, then continue to consider the next vertex adjacent to  $v$ . We keep repeating the process for each distance  $n$  vertex.

For far, we know that the vertices at every contiguous distance from the initial vertex will keep searching until the first winning configuration is found. We can build a graph with its vertices ordered by distance from the initial vertex, and then develop a list using queue as showed below, which is known as a breadth-first search.



From book *Introduction to Algorithms* by Corman, Leiserson, and Rivest, section 22.2 indicates that the time complexity of the breadth-first search is  $O(|E| + |V|)$ , where  $V$  is the number of vertices, and  $E$  is the number of Edges, and section 24.3 indicates that the time complexity of Dijkstra’s algorithm usually is  $O(|E| + |V|\log|V|)$ . Even though both algorithms have the same impact on edges,  $|V|$  and  $|V|\log|V|$  are greatly varied. For example, if we have a fully connected graph with 6000 vertices,  $|V|$  is 6000, but  $|V|\log|V|$  is over 75000.

The breadth-first search is more effective than Dijkstra’s algorithm to find a minimal-move solution, because we can avoid the cost to build the entire all-move graph. When finding the

minimal-slide solution we must use Dijkstra's algorithm, but we can still build the graph as needed by avoiding the effort of pre-computing the all-move graph.

Both the breadth-first search and Dijkstra's algorithm generate a recovered solution only involves one link from each vertex when finding the series of moves. Specifically, if vertex  $w$  at distance  $n + 1$  is discovered that are considered as neighboring vertices to  $m$ , we can rebuild the move from  $w$  back to  $m$  by having the directed edge between  $m$  and  $w$ . After we find the first winning vertex, a minimal sequence of moves can be traced back to the initial vertex. The optimal solution can be recorded by revering the list (queue). Besides, no pre-computing of all-move graph is asked, and the graph we build will gradually have fewer vertices than that in the all-move graph, which means that fewer edges will be constructed.

Nonetheless, there is still a disadvantage in our method, because most likely we build a particular vertex numerous times. More precisely, in an identical all-move graph, the number of times that we discover a vertex will approximately equal the number of edges, which is the degree of the vertex. Undoubtedly more workload is added to rebuild adjacent vertices by moving pieces on the grid than selecting known edges, but this extra work can be balanced out by no pre-computing of the all-move graph.

## Observation

Breadth-first search algorithm of Rush Hour Puzzle is implemented by Python programming language, we have two files `sol.py` and `parse.py`, respectively. File `parse.py` generates a temporary document to save all the reachable paths, and check whether the path is previously visited. File `sol.py` prints out the number of unique states with all possible movements on sequential the 6x6 boards.

We input every initial configuration of the 40 cards into our breadth-first search implementation to find a minimal-move solution. *Table 1* shows the minimal-move performance and Table 2 shows minimal-slide performance. Both tables have 12 representative cards that we randomly chosen from beginner, intermediate, advanced and expert.

card number	graph vertices	winning edges	minimum moves
2	20691	1304	8
6	2912	194	9
8	949	3	12
14	45591	29838	17
17	2919	127	24
19	474	16	22
21	254	5	21
25	9010	277	27

29	4323	30	31
32	651	84	37
36	3489	236	44
40	3432	203	51

*Table 1: Table 1: Minimal-move performance for 12 cards*

card number	vertices	moves	minimum slides
2	14	8	3913
6	18	9	2363
8	22	15	951
14	34	18	17203
17	47	28	2202
19	44	22	527
21	49	23	267
25	52	32	8937
29	54	36	4342
32	62	41	625
36	63	46	2983
40	81	57	3163

*Table 2: Table 1: Minimal-slide performance for 12 cards*

From Table 1, we observe that the graphs involve all vertices that could be arrived from the initial vertex, but not including the winning vertices. The “unique states” column lists all non-overlapping configurations, the “winning edges” column shows the number of paths that winning vertices can be achieved from one of the “graph vertices”, and the “minimum moves” gives the minimal number of edges from non-winning vertices to winning vertices. In real Rush Hour Puzzle, the “winning edges” column displays the number of moves that put the red car into a winning position. Also, we realize that some Rush Hour configurations are easier to solve while others are much harder.

Based on the data of Table 1, we perceive that a bigger diameter and more vertices would make the problem harder, yet more winning edges might tend to make the puzzle easier. However, the minimum number of moves seems like to be the only reliable reference of difficulty. It is almost impossible to find "smart" combination of the graph parameters that have good measure of difficulty.



## Conclusion

According to Dijkstra algorithm and breadth-first search, we discuss two computational methods - minimal-move and minimal-slide solution- of Rush Hour configuration. One challenging problem is to discover the most difficult possible initial configuration, as measured by the minimum number of moves—or slide—required to win. Computational methods that do not assure a minimal solution, but require less work, would also be worth pursuing.

## Reference

T. Corman, C. Leiserson and R. Rivest, Introduction to Algorithms, MIT Press, 1990.

R. Perlman, Interconnections: Bridges, Routers, Switches and Internetworking Protocols, Second Edition, Addison–Wesley, 2000.

Assema, J. V. (2014). Figure 2f from: Irimia R, Gottschling M (2016) Taxonomic revision of *Rocheftia* Sw. (Ehretiaceae, Boraginales). Biodiversity Data Journal 4: E7720. <https://doi.org/10.3897/BDJ.4.e7720>. *On the Hardness of 6x6 Rush Hour*,4-11. doi:10.3897/bdj.4.e7720.figure2f

Rush Hour®. (n.d.). Retrieved December 2, 2018, from <https://www.thinkfun.com/products/rush-hour/>