2001

Rush Hour® and Dijkstra's Algorithm

Mark Stamp, *MediaSnap, Inc.*
Brad Engel
McIntosh Ewell
Victor Morrow

Graph Theory Notes of New York XL, 23–30 (2001)
New York Academy of Sciences.

[GTN XL:5] RUSH HOUR® AND DIJKSTRA'S ALGORITHM†

Mark Stamp,[1] Brad Engel,[2] McIntosh Ewell,[3] and Victor Morrow[4]

[1]MediaSnap Inc.
2635 North 1st. Street
San Jose, California 95134, U.S.A.
<mstamp1@earthlink.net>

[2]2027 Jolly Road,
Baltimore, Maryland 21209, U.S.A.

[3]5811 Garden Drive
NW Clinton, Maryland 21237, U.S.A.

[4]1334 Farragut Street
Washington, DC 20011, U.S.A.

Abstract

The game of Rush Hour® includes a 6×6 grid and game pieces representing cars and trucks. The object of the puzzle is to move a special car out of a gridlocked "traffic jam". In this note we apply Dijkstra's algorithm and a breadth-first search to solve any Rush Hour configuration.

1. Introduction

The *Binary Arts*® game of *Rush Hour*® includes a 6×6 plastic grid, eleven ordinary cars, four trucks, and one special red car. A car occupies two consecutive grid spaces, and a truck three. Furthermore, a car or truck must be placed horizontally or vertically, not diagonally. A vehicle can be moved forward or backward, with the restriction that it cannot move into an occupied space. The object is to manipulate the vehicles so that the red car can exit from the grid, thus escaping from the *traffic jam*.

The game also includes a deck of 40 cards. Each card lists an initial placement of some subset of the vehicles and every initial configuration must include the red car. Furthermore, the red car must reside somewhere in the second row from the top, since the only exit out of the grid is from the right-hand side of this row. For the benefit of the easily-frustrated, a solution appears on the back of each card.

Three additional 40-card packs are currently available. These cards include a few minor variations, such as a limousine (functionally equivalent to a truck) that, in a few cases, replaces the red car. Also, some of these extra cards have two cars in the exit row, both of which must exit in order to win the game. These modifications are easily handled by the methods discussed in this paper.

In this note we show how to apply Dijkstra's algorithm to find a solution that is minimal with respect to the number of moves. Attempts to speed up the algorithm lead us to the breadth-first search algorithm. Then we solve the related problem of finding a solution with a minimum *slide*. In this latter case, we find that the full generality of Dijkstra's algorithm is required. We then apply both algorithms to each of the 40 original game cards. We conclude with some observations on the Rush Hour puzzle and we mention several related problems.

2. A Simplified Example

First we consider a simplified version of *Rush Hour* using a 4×4 grid. An example appears in Figure 1. As with the full-sized version of the game, the object is to manipulate the vehicles so that the red car, denoted by ▦ in Figure 1, can exit the grid.

† Much of this work was completed at the Center for Mathematics, Science and Technology (CMST), a summer camp for High School students sponsored by the National Security Agency.

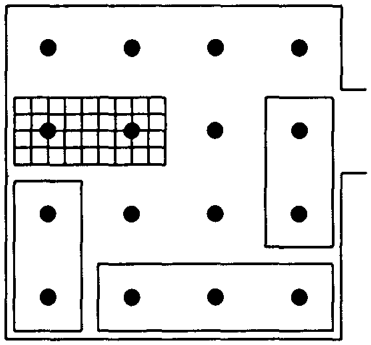

Figure 1: Simplified *Rush Hour.*

A *move* consists of one car or truck sliding forward or backward any number of grid spaces, with the restriction that it cannot move into an occupied position. For example, in Figure 1, only two initial moves are possible—either the red car can move right one space or the rightmost vertical car can move up one position. We define a *slide* to be a move of exactly one grid position. The reader might want to verify that five moves are required to get the red car out of the grid in Figure 1. A total slide of five is also minimal.

For complicated initial configurations, we would like to have an algorithmic method to find minimal-move and minimal-slide solutions. One approach is to construct a graph, where each vertex represents a valid board configuration and the edges are the possible moves (see **[1]** for graph theory background and definitions). Such a graph would allow us to find a minimal-move solution. By weighting the edges of the graph with the length of the corresponding slide, we can also find a minimal-slide solution.

Consider the example in Figure 1. To construct the required graph, we could try every placement of each car or truck within its respective row or column. Then we would let the vertices of our graph consist of all resulting configurations that contain no overlapping vehicles. For the initial configuration in Figure 1 we would examine 54 placements of the vehicles (three placements for each car and two for the truck, for a total of $2 \cdot 3^3 = 54$), from which we find the 16 non-overlapping configurations in Figure 2. Note that in Figure 2, configuration zero is the initial configuration, and those configurations marked with an asterisk are winning

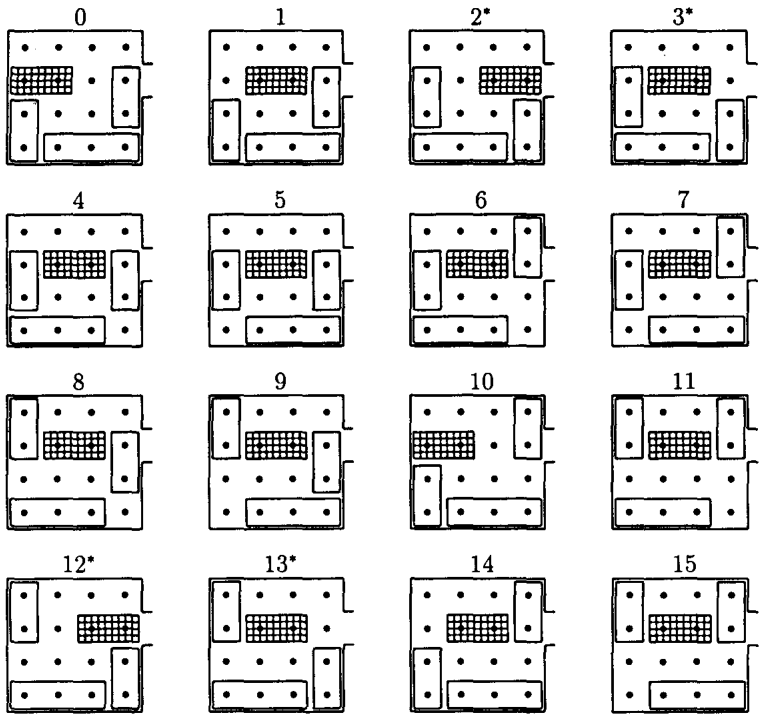

Figure 2: Board configurations for simplified *Rush Hour* example.

configurations, i.e., they are one move away from a win. Using the numbering in Figure 2, an example of a minimal winning series of moves is given by vertices (0, 1, 9, 8, 13). The unique minimal-slide solution is (0, 1, 5, 4, 3).

We refer to the graph discussed in the previous paragraph as the *all-moves graph*. The weighted all-moves graph for our simplified example appears in Figure 3.

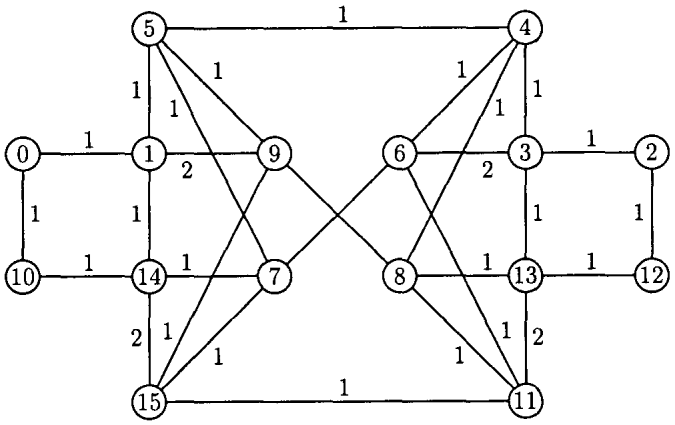

Figure 3: All-moves graph for simplified *Rush Hour*.

3. Dijkstra's Algorithm

Dijkstra's algorithm **[2]** finds the lowest-cost path between two specified vertices of an edge weighted graph, provided that the edge weights are all non-negative. The algorithm finds application in a surprising number of situations. For example, computer networks using so-called link state routing—such as the well-known *open shortest path first* (OSPF) *routing protocol*— implement Dijkstra's algorithm. For networks, there is an interesting twist since the graph itself is distributed amongst the nodes (i.e., routers). Several subtle problems arise from the distributed nature of this problem; see Chapter 12 of Reference **[3]** for an excellent discussion.

For a given edge-weighted graph G, suppose we want to find a minimal path (with respect to the edge weights) from an initial vertex v_i to a final vertex v_f. To begin, we color all vertices of G white. Then the initial vertex v_i is colored black and all of its neighboring vertices are colored gray. Define the distance to a vertex v to be the minimum weight of a path from v_i to v. The algorithm then proceeds as follows.

1. If there are no gray vertices, a solution does not exist. Otherwise, select a gray vertex v_g of minimum distance from v_i.
2. If $v_g = v_f$, then we have found a solution. If not, color v_g black and record its distance from the starting vertex. Color all of the white neighbors of v_g gray; go to 1.

As stated, the algorithm does not record the path from v_i to v_f. In order to find such a path, some additional bookkeeping is all that is required. Furthermore, note that the initial (or final) vertex could easily be replaced by a set of vertices.

Suppose we apply Dijkstra's algorithm to the graph in Figure 3, where our goal is to find a minimal-move solution. Then, in terms of the graph, we want to find a shortest path from vertex zero to any of the winning vertices, namely, vertices two, three, twelve, and thirteen. Since we are looking for a minimal-move solution, not a minimal-slide solution, we set all edge weights equal to one. To start the algorithm, we color vertex zero black and its adjacent vertices, one and ten, are colored gray. Then we pick a gray vertex of minimum distance from vertex zero. Both gray vertices are the same distance so the choice is arbitrary. Suppose we select vertex one. We then color vertex one black and its white neighbors (vertices five, nine and fourteen) are colored gray.

At this point in the algorithm, the gray vertices are 5, 9, 10, and 14, which—since we are using edge weights of one—are distance 2, 2, 1, and 2, respectively, from vertex zero. Hence, the gray vertex of minimum distance from vertex zero is vertex 10. We color vertex 10 black and, since it has no white neighbors, we proceed to once again find the gray vertex of minimum distance. Continuing, we could, depending on several arbitrary choices, arrive at the graph in Figure 4, where we have stopped as soon as the first winning vertex was

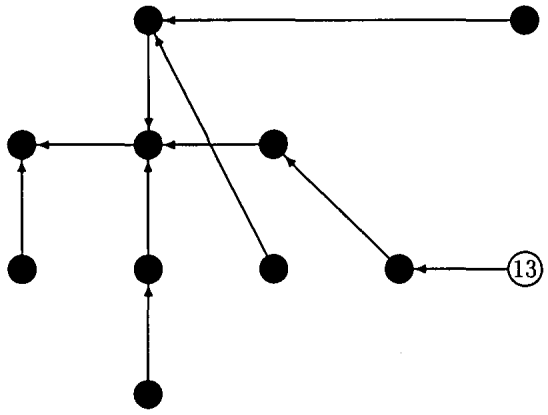

Figure 4: Dijkstra's algorithm applied to simplified example.

reached. Note that the directed edges in Figure 4 allow us to reconstruct the winning sequence of moves, which, for this example is (0, 1, 9, 8, 13).

To find the minimal-slide solution we simply run Dijkstra's algorithm on the graph in Figure 3 with the edge-weights as shown. This case is similar to the previous example and we omit the details.

In the next section we take a closer look at the minimal-move problem. For this problem, we are able to find several ways to speed up Dijkstra's algorithm. This analysis leads us to the breadth first search algorithm.

4. A Real Example

The *Rush Hour* initial configuration from card number 40 appears in Figure 5. For this particular configuration, there are six rows or columns with a single car, three with a single truck and two each with two cars. To

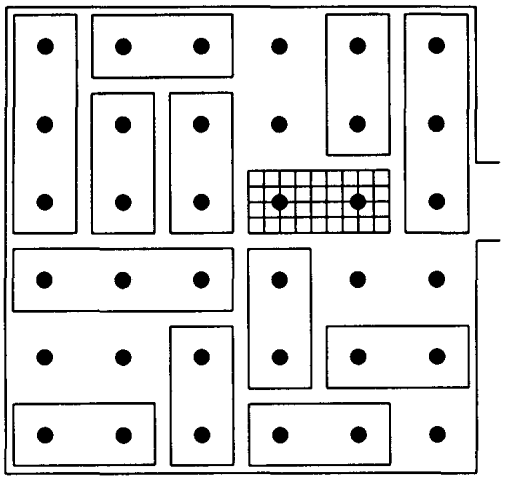

Figure 5: *Rush Hour*—the real thing.

find the vertices of the all-moves graph, we consider all possible placements of the vehicles, keeping those that result in valid non-overlapping configurations. In this case we would need to examine $5^6 \cdot 4^3 \cdot 6^2 = 36{,}000{,}000$ placements. However, we find that the all-moves graph has only 4,805 vertices (and 18,729 edges) and only 4,780 of the vertices are reachable from the given initial configuration. In graph theory terms, the component of the all-moves graph containing the initial vertex consists of 4,780 vertices. If a solution exists, it must occur at one of these 4,780 vertices.

To find a minimal-move solution for the initial configuration in Figure 5, we could first try all 3.6×10^7 possible placements of the vehicles to find the all-moves graph. Then we would need to examine each vertex of the all-moves graph to determine which of them correspond to winning configurations. Finally, we could apply Dijkstra's algorithm to find a shortest path—if such a path exists—from the initial configuration to one

of the winning configurations. This approach has been programmed and it does succeed, but it requires far more work than is necessary to solve the problem.

Evidently, the overwhelming majority of the work—at least for *Rush Hour* card 40—is in simply finding the all-moves graph. For the toy problem of the previous section we found that not all of the graph is required by Dijkstra's algorithm; see Figures 3 and 4. We make use of this simple observation, and some obvious computational tricks to greatly reduce the work and computer memory required to find a minimal-move solution to any *Rush Hour* puzzle. In the process, we convert Dijkstra's algorithm to the breadth-first search algorithm.

Suppose we have successively identified all vertices that are distance one, distance two, through distance n from the initial vertex, and we have not yet arrived at a winning configuration. Suppose vertex v is distance n from the initial vertex. Then we find all vertices adjacent to v, which is easily accomplished by simply moving the game pieces on the configuration corresponding to v. For any vertex w adjacent to v, one of the following four cases must hold 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). Vertex w does not appear in the list of distance $n-1$, distance n, or distance $n+1$ vertices.

If (1), (2), or (3) hold, then vertex w is already known and hence we proceed to the next vertex adjacent to v. However, if (4) occurs, then w is a new vertex at distance $n+1$ from the initial vertex. In this case we first check to see if w corresponds to a winning configuration. If so, we are finished. If not, we append w to the distance $n+1$ list, then proceed to consider the next vertex adjacent to v. Of course, we repeat the process for each distance n vertex.

Using the approach described in the previous paragraph, we find the vertices at each successive distance from the initial vertex, until the first winning configuration is found. As a result, we are able to construct a graph with its vertices ordered by distance from the initial vertex. We can implement such a list using a queue as illustrated below

$$\begin{array}{ll} \text{Vertices:} & \underbrace{v_0}, \quad \underbrace{v_1, v_2, \cdots, v_{n_1}}, \quad \underbrace{v_{n_1+1}, v_{n_1+2}, \cdots, v_{n_1+n_2}}, \quad \cdots \\ \text{Distance:} & \quad 0 \qquad\quad 1 \qquad\qquad\qquad 2 \end{array}$$

The algorithm described above is known as a breadth-first search **[4]**. Our implementation uses a queue to store the vertices, whereas the asymptotically optimal implementation employs a Fibonacci heap **[5]**.

The breadth-first search is more efficient than Dijkstra's algorithm, but an even bigger savings comes from the fact that we find a minimal-move solution while avoiding the expense required to construct the entire all-moves graph. When finding the minimal-slide solution we must use Dijkstra's algorithm, but we can still construct the graph as needed, again avoiding the cost of precomputing the all-moves graph.

For both the breadth-first search and Dijkstra's algorithm, finding the sequence of moves that yields a recovered solution only requires one link from each vertex. More precisely, suppose vertex w at distance $n+1$ is found when considering those vertices adjacent to v. Then by keeping a directed edge from w to v, we will be able to reconstruct the move from w back to v. When the first winning vertex is found, we will be able to recover a minimal series of moves back to the initial vertex. Simply reversing this list will give us the desired solution.

The graph we construct will generally have fewer vertices than found in the all-moves graph and, more significantly, it will have far fewer edges. Consequently, the memory required to store the graph will be much less than that required to store the all-moves graph. Furthermore, we do not need to precompute the all-moves graph, which, as we have seen, can require far more work than finding the minimal-move or minimal-slide solution.

However, our approach does have one drawback. When finding adjacent vertices, we often need to construct a particular vertex many times. In fact, the number of times that we find a vertex is approximately equal to the number of incident edges (that is, the degree of the vertex) in the corresponding all-moves graph. There is certainly more work involved in repeatedly reconstructing adjacent vertices by moving pieces on the board than

simply following known edges. However, this extra work is more than offset by the substantial savings obtained by not precomputing the all-moves graph. As discussed in the previous paragraph, the memory savings are substantial.

In the next section we discuss some results obtained when computing the minimal-move and minimal-slide solutions to the *Rush Hour* cards. These results clearly illustrate the computational advantages of our approach.

5. Results

The 40 standard *Rush Hour* cards are rated *beginner* (cards 1 through 10), *intermediate* (numbers 11 through 20), *advanced* (21 through 30) and *expert* (cards 31 through 40). The beginner cards are trivial, intermediate are somewhat challenging, advanced are more so, and expert cards are difficult. Subjectively, the ratings seem about right.

For each of the 40 cards we input the initial configuration and ran our breadth-first search implementation to find a minimal-move solution. We also ran Dijkstra's algorithm to find a minimal-slide solution. In each case, we found that the solution on the back of the card is, in fact, a minimal-move solution, though not always a minimal-slide solution. In Table 1 we list minimal-move results for twelve representative cards-three each from beginner, intermediate, advanced and expert. Table 2 contains minimal-slide results for the same twelve cards.

Table 1: Minimal-move results

Card Number	Minimum Moves	Graph Diameter	Graph Vertices	Dead Ends	Winning Edges
2	8	26	20691	9293	1304
6	9	20	2912	1884	194
8	12	13	949	412	3
14	17	33	45591	18720	29838
17	24	31	2191	1243	127
19	22	22	474	218	16
21	21	22	254	126	5
25	27	35	9010	3459	277
29	31	32	4323	1968	30
32	37	47	651	249	84
36	44	55	3489	1214	236
40	51	60	3432	1381	203

On a 600-MHz computer, solving all 40 minimal-moves problems (breadth-first search) required a total time of about 30 seconds, whereas the 40 minimal-slide problems (Dijkstra's algorithm) were solved in about five minutes. To find all 40 all-moves graphs took more than 90 minutes.

In Table 1, the graphs consist of all vertices that can be reached from the initial vertex, excluding winning vertices and vertices that can only be reached by passing through a winning vertex. In other words, we assume that a player would stop when a winning vertex has been reached. The *dead ends* are those vertices that have no continuing path forward and have not reached a winning vertex. The *winning edges* column lists the number of ways that winning vertices can be reached from one of the *graph vertices*. Note that this is not the number of winning vertices, but instead the number of edges from non-winning vertices to winning vertices. In terms of the game, the winning edges column gives the number of moves that put the red car into a winning position.

Table 3 contains results for the all-moves graphs. We list the number of placements that must be tried in order to exhaustively construct the all-moves graph (*all-moves placements*) and the number of vertices (*all-moves vertices*) and edges (*all-moves edges*) in the resulting graph. For the 40 cards examined, the number of

Table 2: Minimal-slide results

Card Number	Minimum Slide	Moves	Vertices
2	14	8	3913
6	18	9	2363
8	22	15	951
14	34	18	17203
17	47	28	2205
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

vertices in the all-moves graph varies between 2.87 and 5.96 times the number of edges. These results clearly show the benefit of not precomputing the entire all-moves graph.

Why are some *Rush Hour* configurations easy to solve whereas others are much more difficult? From Table 1, for example, it is not apparent that knowledge of the graphs gives us much insight into the answer to this question. It would seem that a larger diameter, more vertices, and more dead ends would make the problem harder, whereas more winning edges might tend to make the puzzle easier. However, the minimum number of moves appears to be the only reliable indicator of difficulty. Some clever combination of graph parameters might yield a good measure of difficulty, but we have not found such a combination.

Table 3: All-moves graphs

Card Number	All-moves Placements	All-moves Vertices	All-moves Edges
2	6.00×10^6	22139	125902
6	3.00×10^6	4500	19308
8	4.05×10^6	952	3234
14	5.86×10^7	82169	506070
17	2.40×10^7	4092	17659
19	7.50×10^4	842	3247
21	3.20×10^4	656	2542
25	1.50×10^8	20748	95222
29	7.20×10^6	4524	19763
32	4.32×10^5	805	2308
36	1.50×10^7	7135	29698
40	3.60×10^7	4805	18729

6. Conclusion

Binary Arts also produces a game called *Railroad Rush Hour*®, which is played on a 7×7 grid and includes several 2-long and 3-long railroad pieces and a pair of 2×2 "luggage racks."

This variant of the game is significantly more difficult than *Rush Hour* and the number of board configurations can also be far greater. However, the computational methods discussed in this paper should suffice to find a minimal-move solution of any *Railroad Rush Hour* configuration. The minimal-slide problem is more difficult but should also be manageable.

Another challenging problem is to find the most difficult possible *Rush Hour* initial configuration, as measured by the minimum number of moves—or slides—required to win. Computational methods that do not assure a minimal solution, but require less work, would also be worth pursuing. Other possible problems include different board shapes, unusual vehicles, or allowing diagonal placement of vehicles. Finally, playing the game on a torus or Klein bottle would add an interesting twist.

References

[1] [5] R.J. Wilson; *Introduction to Graph Theory, 3rd. edit.*, Longman Scientific & Technical (1985).

[2] [2] E.W. Dijkstra and W.H.J. Feijen; *A Method of Programming*, Addison-Wesley (1988).

[3] [4] R. Perlman; *Interconnections: Bridges, Routers, Switches and Internetworking Protocols, 2nd. edit.*, Addison-Wesley (2000).

[4] [1] T. Corman, C. Leiserson, and R. Rivest; *Introduction to Algorithms*, MIT Press (1990).

[5] [3] M. Fredman and R. Tarjan; Fibonacci heaps and their uses in improved network optimization algorithms, *J. ACM*, **34**, 596–615 (1987).

Received: November 25, 2000
Revised: January 6, 2001