Depth-First, Breadth-First, and Bidirectional Heuristic Rush Hour Solvers

Miles E. Mercer
CPSC 450
Fall 2024

Summary

This project aimed to develop an intelligent Rush Hour solver. Rush Hour, a game traditionally played on a 6×6 square grid, is proven PSPACE-complete when on an arbitrarily large scale [1]. I implemented two naïve algorithms: depth-first and breadth-first search, as well as bidirectional heuristic search [2]. The three algorithms were tested on their time to solve the original 40 puzzles included with the game, as well as the length of their solutions (judged by total car displacement). While breadth-first search outperformed bidirectional heuristic search in both categories (significantly by time on a few specific puzzles), the bidirectional heuristic search method may scale better to larger Rush Hour puzzles.

1. ALGORITHM SELECTED

Rush Hour is a puzzle played on a square grid with cars that are 1 unit wide and either 2 or 3 units long. A car may only move according to its orientation on the board (i.e., cars cannot turn). The goal is to move the cars such that the *target car* can leave the grid through an exit on the perimeter of the board. Bidirectional heuristic search, broadly speaking, involves developing two heuristic functions, one to estimate the number of moves separating a given state from a terminal state, and another to estimate the number of moves separating a given state from the initial state. Then, two searches begin: one from the initial state and another from the terminal state set. The algorithm chooses which of the two to iterate based on which has a smaller set of currently reachable positions, and states are prioritized that minimize the sum of the distance from their root and their heuristic value. The algorithm terminates when both loops visit the same state [2]. The source article for this approach did not consider Rush Hour as an application (though it did solve the similar 15 puzzle) so much of my work had to do with adapting the approach to Rush Hour. Provided is the pseudocode I developed to find the set of feasible win states as well as that for the solver algorithm.

Algorithm: FeasibleWinStates
In: The initial board state (dimension N, exit position (x, y), and $cars$ list)
Out: A set of feasible winning board states

1. *begin*
2. | $k \leftarrow$ number of cars in initial state
3. | $A \leftarrow$ an array of empty sets of boards of length n
4. | Add a board with just the target car at the exit position to $A[0]$
5. | *for* $i \leftarrow 1$ to $k - 1$ *do*
6. | | $c \leftarrow$ copy of initial state's $cars[i]$
7. | | *if* c is horizontal *do*
8. | | | $c.x \leftarrow c.length - 1$
9. | | *else do*
10. | | | $c.y \leftarrow c.length - 1$
11. | | $f \leftarrow$ total length of cars in line with and in front of c
12. | | $b \leftarrow$ total length of cars in line with and behind c
13. | | *for* $d \leftarrow b$ to $N - c.length - f$ *do*
14. | | | *for each* board state in $A[i - 1]$ *do*
15. | | | | Try to add c to the board
16. | | | | *if* successful *do*
17. | | | | | Add resulting state to $A[i]$
18. | | | Move c forward
19. | $w \leftarrow A[\mathrm{k} - 1]$
20. | *for each* board in w *do*
21. | | *for each* car in board *do*
22. | | | Try to move car forward and backward and see if new state found
23. | | *if* no new states are found *do*
24. | | | Remove board from w
25. | *return* w

Algorithm: BidirectionalHeuristicSearch
In: The initial board state (as above)
Out: A sequence of board states from the initial board to a win state

1. *begin*
2. | $w \leftarrow$ *FeasibleWinStates(*initial state*)*
3. | $V_s =$ set of board states visited from initial position
4. | $V_t =$ set of board states visited from a win state
5. | Add initial state to V_s
6. | $V_t \leftarrow w$
7. | $d_s =$ map from board states to distances from initial state
8. | $d_t =$ map from board states to distances from a win state
9. | Add initial state $\rightarrow 0$ to d_s
10. | *for each* state in w *do*
11. | | Add state $\rightarrow 0$ to d_t
12. | $h_f =$ map from board states to forward heuristic values
13. | $h_b =$ map from board states to backward heuristic values
14. | Calculate initial state's heuristic value and add to h_f
15. | Calculate win states' heuristic values and add to h_b
16. | $p_s =$ map from states to their parents from initial state
17. | $p_t =$ map from states to their parents from a win state
18. | *while* both sides can reach new states and there does not exist a state b such that $b \in V_s \wedge b \in V_t$ *do*
19. | | *if* more states are reachable from initial state than win states *do*
20. | | | $b \leftarrow$ state minimizing $d_s[b] + h_f[b]$
21. | | | Add b to V_s
22. | | | Add $b \rightarrow b$'s parent to p_s
23. | | *else do*
24. | | | $b \leftarrow$ state minimizing $d_t[b] + h_b[b]$
25. | | | Add b to V_t

26. | | | Add $b \rightarrow b$'s parent to p_t
27. | s = list of board states
28. | Find $b \rightarrow$ initial state path from p_s, reverse it, and add to s
29. | Find $b \rightarrow t$ path from p_t and append to s
30. | *return* s

We'll use Breadth-First Search (BFS) for comparison in our complexity analysis. On a graph $G = (V, E)$, BFS has time complexity $O(|V| + |E|)$. Let a Rush Hour game be an $N \times N$ square grid containing k cars. Since cars can only be 2 or 3 units long, regardless of N, k is $O(N^2)$. A car can occupy $O(N)$ different positions, so the total number of distinct board states is $O(N^k)$ or $O(N^{N^2})$. Given a board state, you can reach a different board state by moving one of k cars either forward or backward, meaning each board state is connected to $O(k)$ or $O(N^2)$ different states. Thus, if we conceptualize a Rush Hour game as a graph, where vertices are distinct board states, $|V|$ is $O(N^{N^2})$. With each vertex having $O(N^2)$ connections, $|E|$ is $O(N^{N^2} \cdot N^2) = O(N^{N^2+2})$. Therefore, BFS on Rush Hour has time complexity $O(N^{N^2} + N^{N^2+2}) = O(N^{N^2+2})$.

We know that the total number of board states is $O(N^k) = O(N^{N^2})$. For a board state to be a winning state, the target car must be at the exit, so the number of winning states w is $O(N^{k-1}) = O(N^{N^2-1})$ in the worst case. The complexity of the *FeasibleWinStates* algorithm will be overshadowed by the complexity of *BidirectionalHeuristicSearch* (BDHS), so we omit that complexity analysis. We assume a worst case where the search algorithm explores every possible board state to find a solution. In this case, the outer loop makes $O(N^{N^2})$ iterations. The forward heuristic function needs to make $k = O(N^2)$ car comparisons to w win states, giving h_f time complexity $O(N^2 w)$. The backward heuristic function needs to make k comparisons to one initial state, giving h_b time complexity $O(N^2)$. We assume the algorithm makes an equal number of iterations in the forward and backward directions. While this is not necessarily true of any implementation, we assume that newly discovered reachable states only have their heuristic function calculated once before being put into a priority queue, which has enqueuing and dequeuing complexity $O(\log n)$. Each iteration adds a reachable state to a visited set, which unlocks $O(k) = O(N^2)$ new reachable states. Given iteration number i, we assume that $|V_s| = 1 + \frac{i}{2}$ and $|V_t| = w + \frac{i}{2}$. Denoting the number of non-visited states reachable from the start and win states as R_s and R_t respectively, $|R_s|$ is $O\left(k|V_s| - \frac{i}{2}\right) = O\left(k\left(1 + \frac{i}{2}\right) - \frac{i}{2}\right)$ and $|R_t|$ is $O\left(k|V_t| - \frac{i}{2}\right) = O\left(k\left(w + \frac{i}{2}\right) - \frac{i}{2}\right)$. Forward iterations must deque a reachable state at $O(log|R_s|) = O\left(\log\left(N^2\left(1 + \frac{i}{2}\right) - \frac{i}{2}\right)\right)$, then calculate $O(k)$ new h_f values at $O(N^4 w)$, and enqueue those $O(k)$ states at $O(k \log|R_s|) = O\left(N^2 \log\left(N^2\left(1 + \frac{i}{2}\right) - \frac{i}{2}\right)\right)$. The sum of these processes has complexity $O\left((N^2+1)\log\left(N^2\left(1 + \frac{i}{2}\right) - \frac{i}{2}\right) + N^4 w\right)$. Likewise, the backward direction contains these same three processes at complexity $O\left((N^2+1)\log\left(N^2\left(w + \frac{i}{2}\right) - \frac{i}{2}\right) + N^4\right)$. Therefore, across all $O(N^{N^2})$ iterations of the outer loop, BDHS has total complexity

$$O\left(\sum_{i=0}^{N^{N^2}-1} \frac{1}{2}\left[(N^2+1)\log\left(N^2\left(1+\frac{i}{2}\right)-\frac{i}{2}\right) + N^4 w + (N^2+1)\log\left(N^2\left(w+\frac{i}{2}\right)-\frac{i}{2}\right) + N^4\right]\right)$$

$$= O\left(\frac{1}{2}\left((N^2+1)\,log\left(2^{-N^{N^2}}(N^2-1)^{N^{N^2}}\left(\frac{2N^2 w}{N^2-1}\right)^{N^{N^2}}\right) + (N^2+1)\,log\left(2^{-N^{N^2}}(N^2-1)^{N^{N^2}}\left(\frac{2N^2}{N^2-1}\right)^{N^{N^2}}\right) + N^{N^2+4}(w+1)\right)\right)$$

$$= O\left(N^{N^2+4} w\right) = O\left(N^{N^2+4} \cdot N^{N^2-1}\right) = O\left(N^{2N^2+3}\right)$$

2. IMPLEMENTATION

All implementation for this project was completed in Java. The Board class contains an ArrayList of Car objects and performs much of the computational load of the three solvers. Cars can perform basic functionality like evaluating whether they intersect with other cars. They also have overridden HashCode and Equals functions which allow them to be compared for equality. Boards can validate car positions, compare themselves to other boards, and attempt and perform car moves as well as car additions. Boards also have overridden HashCode and Equals functions, allowing them to be compared and, most importantly for my implementation, stored in sets. The DepthFirstSolver class contains a Deque stack instantiated with a LinkedList as well as a HashSet to track visited board states. It performs DFS with these two data structures similarly to our DFS implementation and returns its certificate as an ArrayList. The BreadthFirstSolver is similar, except that it uses its Deque as a queue rather than a stack. It also keeps a HashMap to track which boards are which boards' parents in the traversal. It returns its certificate as a LinkedList.

The BidirectionalHeuristicSolver has a far more complex data model. HashSets track which board states have been visited from each end of the traversal as well as which states are reachable. HashMaps are used to associate board states with distances as well as heuristic values. In addition to the HashSet, reachable states are stored in a PriorityQueue, with an underlying Comparator for each traversal direction. Feasible win states are computed using an ArrayList of HashSets. The feasibleWinStates function was also the most difficult element of the BidirectionalHeuristicSearch implementation. While I was

figuring out the logic, the algorithm would switch cars around and eliminate valid winning states. It took me a long time to figure out the front- and back-padding logic as well as ensuring that winning states are reachable from non-winning states.

Describe the test cases you used to ensure the implementation was working correctly. This shouldn't be the actual test code. Instead, you should generally describe the tests you developed, the purpose of each test, what aspects of the algorithms each test evaluated, and any other relevant information for judging the quality of your tests. Include tables and figures as needed. For instance, a table listing each test with additional information and figures of graphs used for the test cases. Figures must have a caption and fit within a single column (see Table 1 and Figure 1). You can refer to figures by number, e.g., "Figure 1 shows ...".

The majority of unit testing targeted the Board and Car classes, which handled vital functions for the solver classes. Car intersection testing was a major element of unit testing, as well as Board validation. All Solver classes had to perform 10 common unit tests. Many of them were seemingly trivial, putting the target car at the exit or one space away, or one space away with a single obstacle, etc. Three of the unit tests were impossible to solve, the simplest and most complex of which are pictured below.

Figure 1. A simple impossible Solver test

Figure 2. A more complex impossible Solver test

Additionally, special attention in unit testing was paid to the feasibleWinStates function. One primary concern was puzzles that have a great number of winning states but only one that could be reached without reaching another first. The unit test to evaluate that identification is shown below.

Figure 3: A board with multiple win states but only one possible one

3. PERFORMANCE TESTS

For performance testing, I used the 40 original puzzles included with the Rush Hour game. Each solver solved each puzzle 10 times and were recorded on the average time to find a solution as well as the length of that solution. In future research, I would be curious to see if an algorithm could be developed to generate challenging puzzles at higher values of N, but that was not accomplished within the scope of this project.

4. EVALUATION RESULTS

The average solution times of the three algorithms are plotted below.

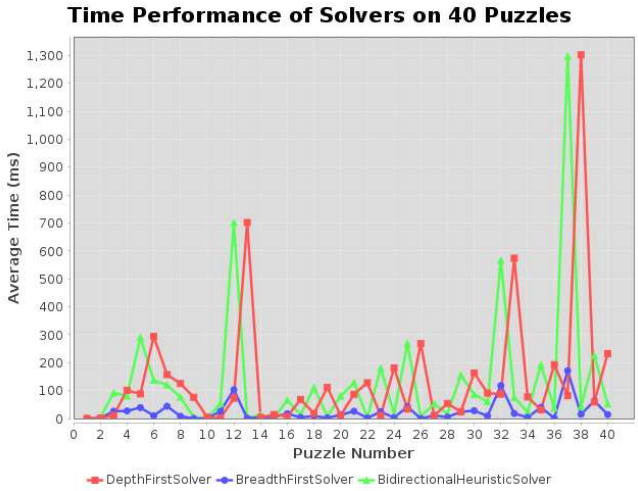

Figure 4. Average time on each puzzle by three solvers

As can be seen above, the breadth-first solver was the only consistently efficient algorithm. Occasionally, the depth-first and bidirectional heuristic solvers had comparable performance, but they also suffered massive spikes in solution time. There could be

a number of possible reasons for these spikes. Depth-first search was likely less efficient when it searched "rabbit holes" or board states that led to lots of other possible board states but not a winning state. The bidirectional heuristic model likely suffered spikes when a problem had a significant number of feasible winning states, bogging down the forward heuristic function.

Shown below are the average solution lengths from the three algorithms.

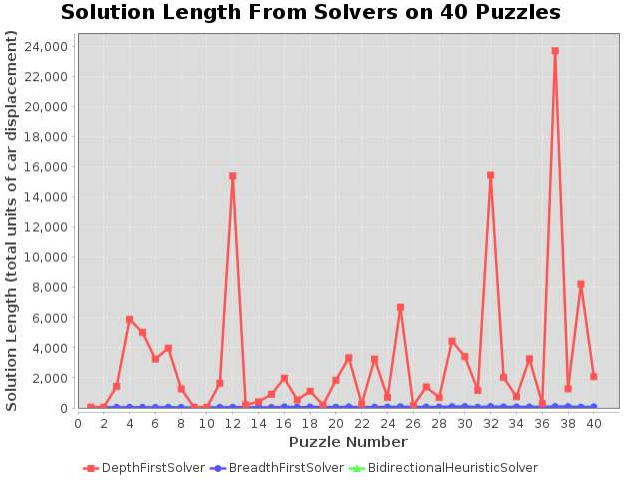

Figure 5. Solution lengths for each puzzle from three solvers

The depth-first solver is a clear outlier in this category of evaluation. The potential advantage of brute-force algorithms like depth-first search (which may be referred to as a "backtracking" algorithm in this case) is that they traverse the state space quickly and with minimal logic, but when they come upon a solution, it is by chance and could occur at any point during an aimless traversal of game states.

Below is the result of the same test without the depth-first solver.

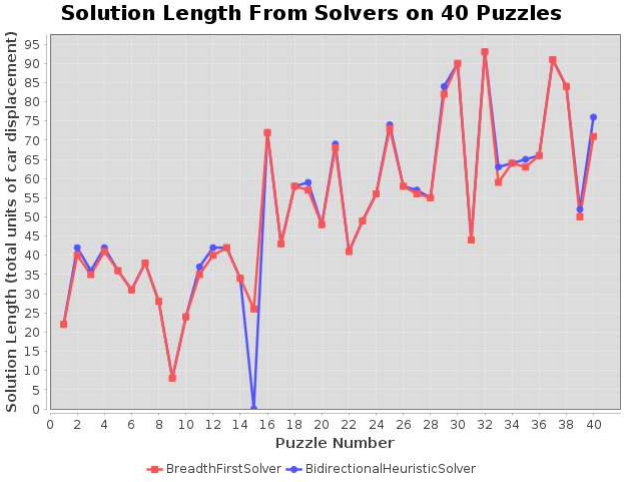

Figure 6. Solution lengths for each puzzle excluding DFS

We can see here that bidirectional search was fairly comparable to BFS on solution length. Unlike BFS, this implementation of bidirectional heuristic search is not guaranteed to find a shortest solution to a puzzle, but it's prioritization of states that seem the most likely to lead directly to a win state allows it to find efficient solutions. It would appear from this graph that the bidirectional heuristic search mistakenly identified puzzle 15 as impossible. It's not clear from this test alone why that was the case. The algorithm was efficient on that puzzle, so it was likely an early failure, perhaps a logical error in its generation of feasible winning states. Regardless, it seems clear that breadth-first search is optimized for both efficiency and solution length on the original Rush Hour puzzles.

5. REFLECTION

It is certainly disappointing in a project like this to invest time in a complex algorithm and find it less optimized than a straightforward approach. And in fact, on a 6×6 Rush Hour game, it makes sense that this approach was over-engineered for such a comparatively small state space. It would be interesting to see how bidirectional heuristic search and breadth-first search compare on larger puzzles with exponentially larger state spaces. Generating these boards in a non-trivial way, however, is no simple task. It also seems that my forward heuristic function was entirely too complex to be more efficient than logicless brute force. Future development may focus on finding more efficient heuristic approximations as well as tuning meta-parameters to favor the more efficient of the two heuristic functions.

Bidirectional heuristic search bears a striking resemblance to Dijkstra's algorithm. In fact, a simpler bidirectional search algorithm can be used in the shortest weighted path problem as an alternative to Dijkstra's. The algorithm is also generalizable to other puzzles with a quantifiable heuristic. This is shown in [2] with the 15 puzzle.

6. RESOURCES

I was first introduced to Rush Hour as a PSPACE-complete computational problem in [1]. While the article is only mentioned briefly in the Summary, it presents a brilliant proof of the problem's PSPACE-completeness by developing gadgets that reduce SAT to Rush Hour. It is certainly a fascinating read. The bidirectional heuristic search algorithm was first published in [2], which provided not only a description of the algorithm, but pseudocode for multiple applications, making it of vital importance in Section 2 and incredibly informative for Section 3.

7. REFERENCES

[1] G. W. Flake and E. B. Baum, "Rush Hour is PSPACE-complete, or 'Why you should generously tip parking lot attendants,'" *Theoretical Computer Science*. Available: https://www.sciencedirect.com/science/article/pii/S0304397501001736?via%3Dihub (accessed Nov. 5, 2024).

[2] I. Pohl, "Bi-Directional and Heuristic Search in Path Problems [Thesis]," SLAC Report SLAC-R-104, *OSTI OAI (U.S. Department of Energy Office of Scientific and Technical Information)*, OSTI ID: 1453875, Contract No. AC02-76SF00515, May 1969, doi: https://doi.org/10.2172/1453875