RushHour Solver

Team Member(s): Nathan Esau

Algorithm(s): BFS

Languages(s): C++, Python, Java

Jam(s): http://www.mathsonline.org/game/jam.html

Results

The average solve time is about 0.02 seconds per puzzle. Several optimizations were used to improve the solve time. More on this later.

Language	40 Jam Cases	35 Project Cases
C++	0.703 seconds	0.686 seconds
Java	0.903 seconds	0.863 seconds
Python	6.76 seconds	6.34 seconds

Explanation

Solve Algorithm

Use breadth-first search algorithm. Store nodes in a priority queue sorted by distance.

To explain how the algorithm works, I first go through how the algorithm works for a simple maze. Then I make a few modifications to the code for the Rush Hour case.

Case #1: Simple Maze

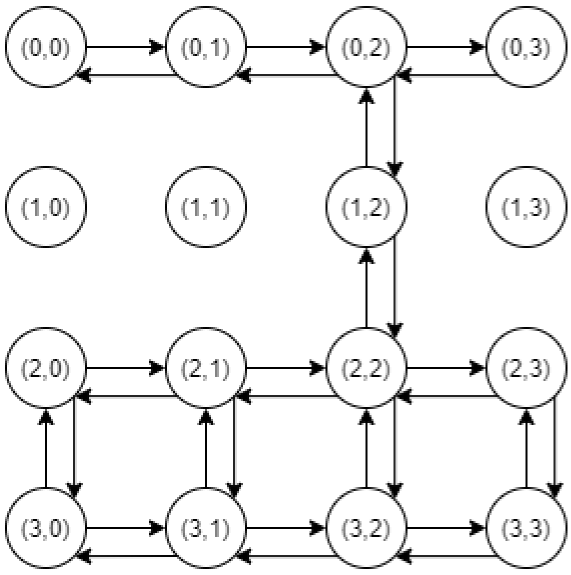
Suppose we have the following maze, where `W` represents a wall and `.` represents a tile you can walk on. We are trying to go from bottom left corner to top right corner.

```
. . . .
W W . W
. . . .
. . . .
```

One of the possible shortest paths is URRRURR:

```
. . _ _
W W | W
_ _ _ .
| . . .
```

Here is the graph for the maze:



Which can also be represented as:

```
graph = {
    (0, 0): {(0, 1), (1, 0)},
    (0, 1): {(0, 2), (0, 0), (1, 1)},
    (0, 2): {(1, 2), (0, 3), (0, 1)},
    (0, 3): {(1, 3), (0, 2)},
    (1, 0): set(),
    (1, 1): set(),
    (1, 2): {(1, 3), (1, 1), (0, 2), (2, 2)},
    (1, 3): set(),
    (2, 0): {(3, 0), (1, 0), (2, 1)},
    (2, 1): {(2, 0), (3, 1), (1, 1), (2, 2)},
    (2, 2): {(1, 2), (3, 2), (2, 3), (2, 1)},
    (2, 3): {(1, 3), (3, 3), (2, 2)},
    (3, 0): {(2, 0), (3, 1)},
    (3, 1): {(3, 0), (3, 2), (2, 1)},
    (3, 2): {(3, 1), (3, 3), (2, 2)},
    (3, 3): {(3, 2), (2, 3)}}
}
```

Which can be solved as follows using Python:

```
def shortest_path(prev, src, target):
    path = []
    u = target
    while u != src:
        path.insert(0, u)
        u = prev[u]
    path.insert(0, src)
    return path
```

```python
def solve(graph, src, target):
    dist = dict((k, float('inf')) for k in graph.keys())
    prev = dict((k, None) for k in graph.keys())
    dist[src] = 0
    pq = [(0, src)]
    while pq:
        distu, u = heapq.heappop(pq)
        if u == target:
            return shortest_path(prev, src, u)
        for v in graph[u]:  # neighbors
            alt = distu + 1
            if alt < dist[v]:
                dist[v] = alt
                prev[v] = u
                heapq.heappush(pq, (alt, v))

# [(3, 0), (2, 0), (2, 1), (2, 2), (1, 2), (0, 2), (0, 3)]
print(solve(graph, (3, 0), (0, 3)))
```

Case #2: RushHour

Graph Representation

First let's discuss the graph representation for the RushHour game. As an example, let's consider Jam 1 from http://www.mathsonline.org/game/jam.html.

```
bb...g
a..c.g
axxc.g
a..c..
e...ff
e.ddd.
```

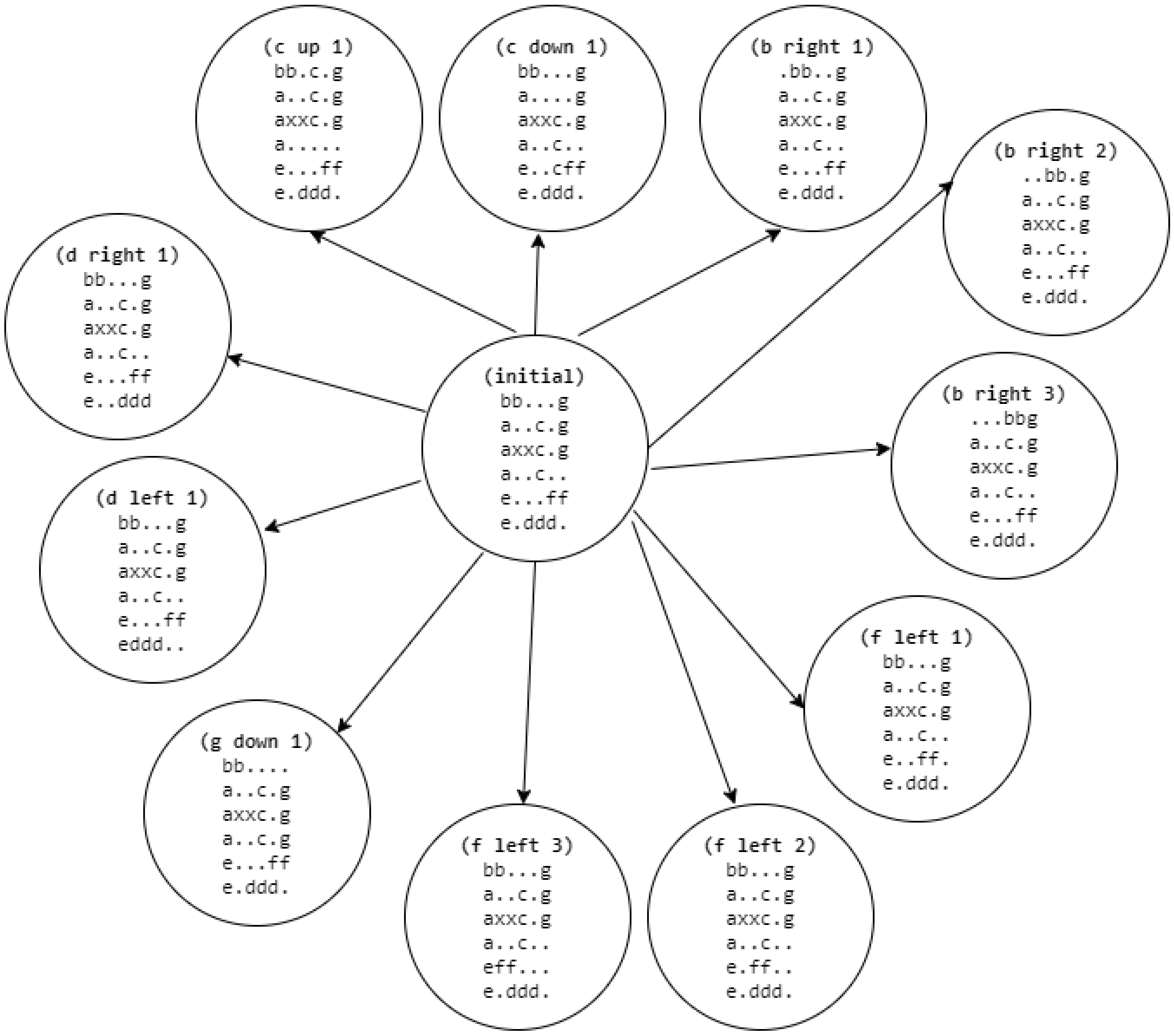
I could represent this as a 2d char array, but it can actual be represented more succinctly as `{"a":1, "b":0, "c":1, "d":2, "e":4, "f":4, "g":0, "x":1}`. These are the left most index for horizontal cars and top most index for vertical cars. Below I will refer to these as the "variable position".

Here is a table summarizing the initial board. The only difference between different "states" is the variable position. The fixed position, size of the car, orientation of the car and name of the car never changes.

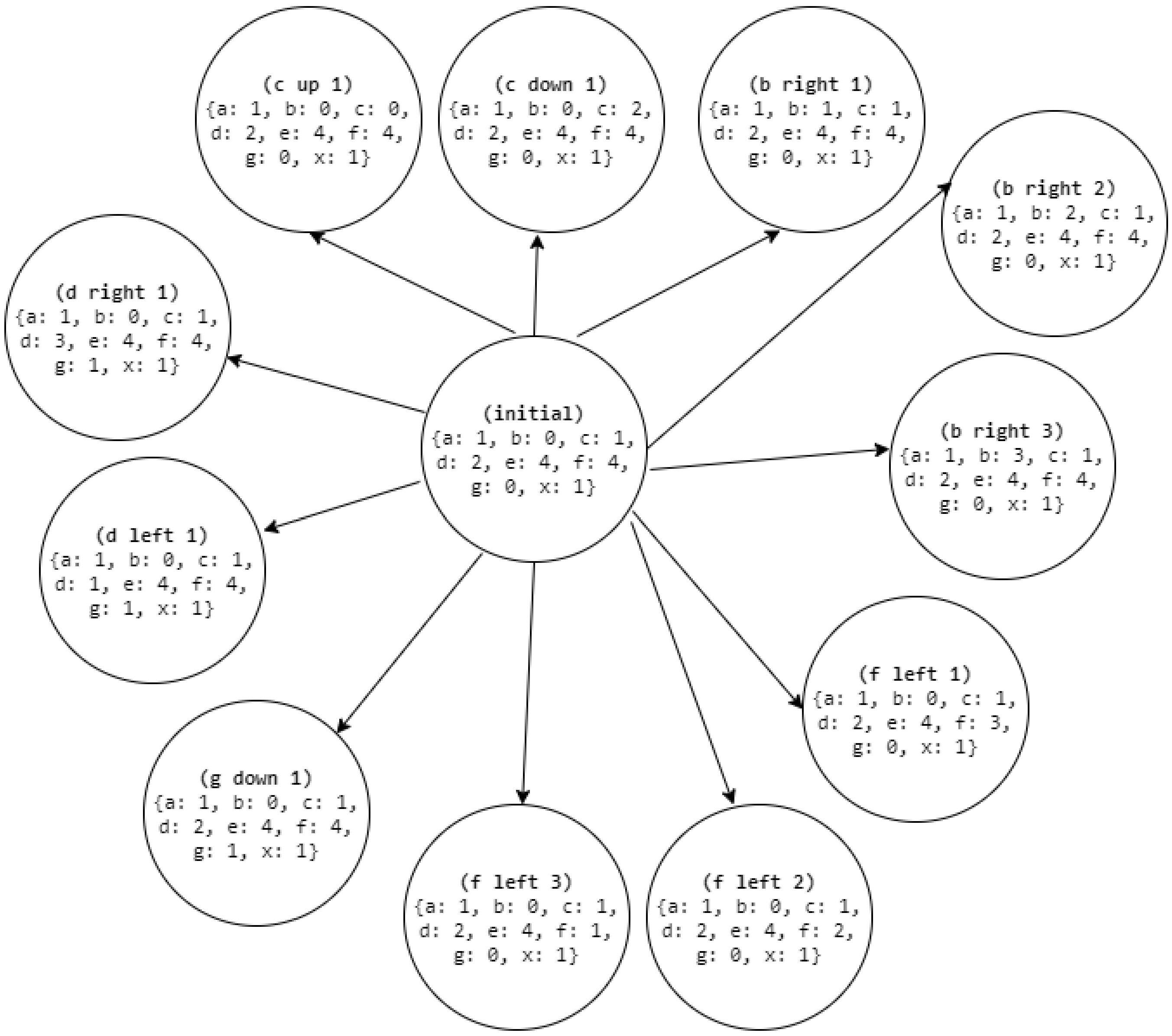
car	orientation	size	fixed position	variable position
a	vertical	3	column 0	row 1
b	horizontal	2	row 0	column 0
c	vertical	3	column 3	row 1
d	horizontal	3	row 5	column 2

car	orientation	size	fixed position	variable position
e	vertical	2	column 0	row 4
f	horizontal	2	row 4	column 4
g	vertical	3	column 5	row 0
x	horizontal	2	row 2	column 1

The initial graph is shown below. The vertex is the initial board and the edges are the boards 1 move away from the initial board. Note that "right 1", "right 2" and so one are all considered to be one move.



We can represent this more succinctly using the variable position as I mentioned above like this:


(c up 1)
{a: 1, b: 0, c: 0, d: 2, e: 4, f: 4, g: 0, x: 1}
(c down 1)
{a: 1, b: 0, c: 2, d: 2, e: 4, f: 4, g: 0, x: 1}
(b right 1)
{a: 1, b: 1, c: 1, d: 2, e: 4, f: 4, g: 0, x: 1}
(b right 2)
{a: 1, b: 2, c: 1, d: 2, e: 4, f: 4, g: 0, x: 1}
(d right 1)
{a: 1, b: 0, c: 1, d: 3, e: 4, f: 4, g: 1, x: 1}
(initial)
{a: 1, b: 0, c: 1, d: 2, e: 4, f: 4, g: 0, x: 1}
(b right 3)
{a: 1, b: 3, c: 1, d: 2, e: 4, f: 4, g: 0, x: 1}
(d left 1)
{a: 1, b: 0, c: 1, d: 1, e: 4, f: 4, g: 1, x: 1}
(f left 1)
{a: 1, b: 0, c: 1, d: 2, e: 4, f: 3, g: 0, x: 1}
(g down 1)
{a: 1, b: 0, c: 1, d: 2, e: 4, f: 4, g: 1, x: 1}
(f left 3)
{a: 1, b: 0, c: 1, d: 2, e: 4, f: 1, g: 0, x: 1}
(f left 2)
{a: 1, b: 0, c: 1, d: 2, e: 4, f: 2, g: 0, x: 1}

Algorithm

Let's make a few modifications to the maze solver from above. Here is a full RushHour solver written in 100 lines of Python code. For the RushHour case, we do not know the entire graph in advance and must figure out neighbors on the fly. Other differences are the use of a Metadata class to store information which is common between different nodes and using a custom solve condition.

To get the neighbors first convert the variable positions such as `{"a":1, "b":0, "c":1, "d":2, "e":4, "f":4, "g":0, "x":1}` back to a 2d grid. Then for each car, iterate through all possible moves and get the variable positions for each of these moves.

Also, the key of our distance/ visited dictionary such as `{"a":1, "b":0, "c":1, "d":2, "e":4, "f":4, "g":0, "x":1}` must be hashed.

```python
class Metadata:
    def __init__(self, grid):
        cl = set([t for r in grid for t in r if t != '.'])
        tl = dict((c, [(i, j) for i in range(6) for j in range(6)
                       if grid[i][j] == c]) for c in cl)
        v = dict((car, tl[car][0][0] != tl[car][1][0]) for car in cl)
        self.grid = grid
        self.cars = cl
        self.orientation = v
        self.size = dict((car, len(tl[car])) for car in cl)
        self.fixed_position = dict((car, tl[car][0][1]
                                    if v[car] else tl[car][0][0]) for car in cl)
        self.node_count = 0

def convert_to_grid(u, metadata):  # convert variable position to grid
    grid = [['.' for _ in range(6)] for _ in range(6)]
    for car in metadata.cars:
        orientation = metadata.orientation[car]
        size = metadata.size[car]
        fp = metadata.fixed_position[car]
        vp = u[car]
        if car == 'x' and vp + size > 6:
            size -= 1
        for d in range(size):
            grid[vp+d if orientation else fp][fp if orientation else vp+d] = car
    return grid

def get_neighbors(u, metadata):
    neighbors = []
    grid = convert_to_grid(u, metadata)
    for car in metadata.cars:
        orientation = metadata.orientation[car]
        size = metadata.size[car]
        fp = metadata.fixed_position[car]
        vp = u[car]
        for np in range(vp-1, -1, -1):
            if orientation and grid[np][fp] != '.' or \
                    not orientation and grid[fp][np] != '.':
                break
            nb = dict((k, v) if k != car else (k, np) for k, v in u.items())
            neighbors.append(nb)
        for np in range(vp+size, 7):
            if np < 6 and orientation and grid[np][fp] != '.' or \
                    np < 6 and not orientation and grid[fp][np] != '.' or \
                    np == 6 and car != 'x':
                break
            nb = dict((k, v) if k != car else (k, np-size+1)
                      for k, v in u.items())
            neighbors.append(nb)
    metadata.node_count += len(neighbors)
    return neighbors

def shortest_path(prev, src, target):
    path = []
    u = target
    uhash = hash(frozenset(u.items()))
    while uhash in prev:
        path.insert(0, u)
        u = prev[uhash]
        uhash = hash(frozenset(u.items()))
    path.insert(0, src)
    return path

def solve(src, metadata):  # no graph or target parameters
    dist = {}
    prev = {}
    dist[hash(frozenset(src.items()))] = 0
    q = [(0, src)]
    while q:
        q = sorted(q, key=lambda it: it[0])
        distu, u = q.pop(0)
        if u['x'] == 5:
            return shortest_path(prev, src, u)
        for v in get_neighbors(u, metadata):  # neighbors
            vhash = hash(frozenset(v.items()))
            alt = distu + 1
            if vhash not in dist or alt < dist[vhash]:
                dist[vhash] = alt
                prev[vhash] = u
                q.append((alt, v))

def print_solution(path, metadata, microseconds):
    print(f"\n{'='*10}\nsolved jam in {microseconds} microseconds\n{'='*10}")
    for i, u in enumerate(path):
        grid = convert_to_grid(u, metadata)
        print(f"\ni = {i}\n" + '\n'.join([''.join(row) for row in grid]))

from datetime import datetime
jam = 'bb...g\na..c.g\naxxc.g\na..c..\ne...ff\ne.ddd.'
grid = [list(line) for line in jam.splitlines()]
cl = set([t for r in grid for t in r if t != '.'])
tl = dict((c, [(i, j) for i in range(6) for j in range(6)
              if grid[i][j] == c]) for c in cl)
v = dict((car, tl[car][0][0] != tl[car][1][0]) for car in cl)
src = dict((car, tl[car][0][0] if v[car] else tl[car][0][1]) for car in cl)
metadata = Metadata(grid)
start = datetime.now()
path = solve(src, metadata)
end = datetime.now()
print_solution(path, metadata, (end - start).microseconds)
```

Here is a side by side comparison of the solve functions:

```
def maze_solve(graph, src, target):
    dist = dict((k, float('inf')) for k in graph.keys())
    prev = dict((k, None) for k in graph.keys())
    dist[src] = 0
    pq = [(0, src)]
    while pq:
        distu, u = heapq.heappop(pq)
        if u == target:
            return shortest_path(prev, src, u)
        for v in graph[u]:  # neighbors
            alt = distu + 1
            if alt < dist[v]:
                dist[v] = alt
                prev[v] = u
                heapq.heappush(pq, (alt, v))

def rushhour_solve(src, metadata):
    dist = {}
    prev = {}
    dist[hash(frozenset(src.items()))] = 0
    q = [(0, src)]
    while q:
        q = sorted(q, key=lambda it: it[0])
        distu, u = q.pop(0)
        if u['x'] == 5:
            return shortest_path(prev, u)
        for v in get_neighbors(u, metadata):  # neighbors
            vhash = hash(frozenset(v.items()))
            alt = distu + 1
            if vhash not in dist or alt < dist[vhash]:
                dist[vhash] = alt
                prev[vhash] = u
                q.append((alt, v))
```

Solution

For Jam 1 from from http://www.mathsonline.org/game/jam.html the number of nodes visited is 11,586. This gives some idea about the size of the graph. For most puzzles the number of nodes ranges between 1,000 to 100,000.

Here is a solution for Jam 1:

```

bb...g        bb...g        bb....        .bb...        abb...
a..c.g        a..c.g        a..c..        a..c..        a..c..
axxc.g  (fL3) axxc.g  (gd3) axxc..  (bR1) axxc..  (aU1) axxc..  (eU1)
a..c..        a..c..        a..c.g        a..c.g        ...c.g
e...ff        eff...        eff..g        eff..g        eff..g
e.ddd.        e.ddd.        e.dddg        e.dddg        e.dddg

abb...        abb...         abb...
a..c..        a..c..         a.....
axxc..  (dL2) axxc..  (cD2)  axx...  (xR3)
e..c.g        e..c.g         e..c.g
eff..g        eff..g         effc.g
..dddg        ddd..g         dddc.g
```

Optimizations

Sorting the priority queue isn't necessary.

- It turns out that we don't need to sort the priority queue for this solver. For instance, we first visit the root, which is distance 0 from the root. Then we visit the direct neighbors distance 1 away. Then we visit the neighbors of the direct neighbors distance 2 away. One of the neighbors distance 2 away will be the original puzzle. But we have already visited this node at distance 0 and don't need to visit it again. Then we will visit the nodes at distance 3. Since there is no backtracking, we always add nodes further and further away from the original root and the nodes at the back of the queue are the furthest away. Therefore, a sort isn't needed.

Using an `int[26]` array to store the variable positions instead of a hash table.

- Although lookup for `{"a":1, "b":0, "c":1, "d":2, "e":4, "f":4, "g":0, "x":1}` is amortized constant, it is not nearly as fast as `[1,0,1,2,4,4,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0]`. If we need to lookup the fixed position for car `a`, we can just use index 0. This change alone gave a 5 times to 10 times speedup.

Lessons

- Less is more. The solve logic is simple and this makes it easy to optimize the code.
- BFS works well for this problem. With BFS, we can easily solve the puzzle in 0.02 seconds or less. With this speed, the algorithm could be written in Javascript and run in browser. The player could get an immediate solution to any puzzle they are stuck on.
- BFS also gives shortest solution, which makes it a desirable algorithm for the solver.
- A hash table is a good data structure for keeping track of visited nodes and parents of nodes. We have a lot of nodes to keep track of and the lookup is O(1*).

Appendix

The appendix contains solutions for the project puzzles as well as some Java code for the main parts of the maze solver and RushHour solver. The full code for the RushHour solver is in the `src` folder.

Solutions

Here are the solutions generated by the solver:

- **A01**: `[AR1,PU1,BU1,RL2,CL3,QD2,OD3,XR3]`
- **A02**: `[EL1,PD2,XR1,AD1,OL3,BU1,CU2,XR3]`
- **A03**: `[PU3,CR2,OU2,AR3,OD3,XR1,BU4,XL1,OU3,AL3,CL3,PD3,OD3,XR3]`
- **A04**: `[OD3,XL1,AU3,XR1,OU3,QL3,RL2,PD3,XR3]`
- **A05**: `[AR1,PU1,RL1,EL3,FL3,QD2,GD1,OD3,XR3]`
- **A06**: `[XL1,EU3,DR1,FU1,RL3,PD2,QD1,OD2,XR4]`
- **A07**: `[CD3,FD1,DD1,BR2,EU1,XR1,AD3,XL1,ED1,BL3,DU1,EU1,XR3]`
- **A08**: `[AL3,BL2,CU1,DU2,EU2,GL1,XR3,IU2,PL1,QL1,OD3,XR1]`
- **A09**: `[XR1,PU3,QL1,DD2,EL1,XR1,AD1,BL1,CL1,FU2,OD1,XR2]`
- **A10**: `[CL1,XR2,OU1,QR2,BD4,AR1,DR1,PU2,QL3,XL2,EU2,FL1,HL1,OD3,CR1,EU2,XR3]`

- **B11**: `[OD3,XL1,BU2,QL1,EU3,QR1,RR1,BD3,XR1,OU3,QL3,PD1,AR3,PU1,QR3,OD3,XL1,BU4,XR1,OU3,QL3,RL3,PD3,ED2,XR3]`
- **B12**: `[BR2,PU1,QL2,CU3,QR2,RR3,PD3,BL1,CU1,XR3,AD1,BL2,PU3,QL1,RL1,OD3,XR1]`
- **B13**: `[DD2,ED1,OU1,FR1,XL3,DU2,GU3,FL2,HL2,CD3,KL1,OD3,BR2,DU1,GU1,XR4]`
- **B14**: `[DU1,EU1,GU2,HL2,IU1,KR3,ID1,HR2,DD3,ED3,XL2,HL2,BD2,CL3,BU2,FU2,XR4]`
- **B15**: `[AL1,BR1,IR1,HR1,QD1,RD1,XL2,EU1,FU1,GL2,OD1,PD1,DR2,EU2,FU2,XR2,QU1,RU1,HL2,IL2,OD1,PD1,XR2]`
- **B16**: `[PD1,DD2,EL2,FD1,XL3,CD1,BR1,PU3,GR2,FD1,QL2,CD3,QR2,FU1,GL2,PD3,XR3,PU3,QL1,OD3,XR1]`
- **B17**: `[BL1,PU2,FU2,GU2,QR3,RR3,DD2,ER1,XR1,AD4,BL1,EL1,XL1,OR2,DU4,ER1,XR1,AU2,QL3,RL3,PD2,FD1,GD1,XR3]`
- **B18**: `[QR2,RR3,PD1,DR3,XL1,BD4,AR1,CR1,XR1,PU3,QL3,OD2,AR3,CR3,OU2,QR3,PD3,XL1,BU4,XR1,PU3,QL3,RL3,OD3,XR3]`
- **B19**: `[DU2,EL2,XL2,AD1,BL1,JU1,FU1,OR2,AD3,ER2,XR2,DD4,BL2,EL2,XL2,AU4,ER2,XR2,DU2,OL2,FD1,XR2]`
- **B20**: `[CD1,BR2,DU2,EU2,QL1,PD1,FL3,CD1,ED1,XR4]`
- **C21**: `[PD2,QR2,XL1,BD4,AR1,XR1,PU3,QL3,OD1,AR3,OU1,QR3,PD3,XL1,BU4,XR1,PU3,QL3,RL3,OD3,XR3]`
- **C22**: `[BU1,FU1,GR1,XL1,AD3,XR1,BD1,OL3,PU1,EL1,HU2,QR2,AD1,DD1,EL2,GR1,PD2,CL3,PU2,HU2,ER3,AU1,DU1,QL3,PD3,XR3]`
- **C23**: `[QL2,DD1,EL1,PD1,OR1,AU1,XL3,AD1,OL1,PU1,ER1,DU1,QR3,CD1,AD1,BL3,AU1,DU2,EL4,PD1,OR1,AU1,CU2,QL1,FL4,PD2,CD1,DD2,XR4]`
- **C24**: `[CU1,DU2,FL1,XL2,AD1,BL1,EU2,GU2,HR3,OR3,AD3,FR2,XR2,CD4,DD4,BL2,FL2,XL2,AU4,FR2,XR2,CU2,OL2,GD1,XR2]`
- **C25**: `[CL1,EU1,OU1,PD1,QR2,XL1,BD4,AR1,DR1,XR1,PU3,QL3,GU3,IL1,QR3,PD3,AL1,DL1,XL1,BU4,HL2,QL2,ED2,OD3,CR1,GU1,XR4]`
- **C26**: `[DU1,ED1,RL1,GU1,HR1,CD3,XR1,AD1,OL3,PU1,RR2,AD3,RL2,DU1,XL1,CU4,XR1,HL1,GD1,RR3,AU3,EU1,FU1,HL3,FD1,RL2,PD3,XR2]`
- **C27**: `[PU2,ER1,OD2,BR2,CR2,DU2,XR1,AD4,XL1,DD2,BL3,CL3,DU1,OU2,EL4,DD1,OD2,CR3,DU2,ER1,XR1,AU3,EL1,FU1,RL3,PD3,OD1,XR3]`
- **C28**: `[BU1,ER1,GU3,EL1,FR2,HU1,RR3,CD1,DL1,AD3,DR1,XR1,OR1,CU4,DL1,XL1,AU3,RL3,FL2,HD1,ER1,GD3,EL2,PD3,OR2,AU1,XR3,AD1,OL1,BU1,XR1]`

- **C29**: `[QU1,DR1,CR1,FR2,ER1,OD3,XL1,AD1,PL1,QU2,DR1,BD3,DL1,QD1,PR1,AU1,XR3,OU3,CL1,EL1,AD4,CR1,XL2,OD1,PL3,QU1,DR1,BU4,DL1,FL1,QD3,XR3]`
- **D30**: `[AR1,DU2,XL1,QU2,FR3,QD2,XR1,DD4,AL1,XL1,QU3,EL3,QD3,BD1,CL1,PU1,FR1,BD1,XR3,QU3,ER1,DU3,EL1,QD3,AR1,DU1,XL3,QU2,BU1,FL4,QD2,BD1,XR3,QU1,RL1,PD2,XR1]`
- **D31**: `[BD1,CL1,PU3,FR1,KU1,HR4,DD1,EL1,KD1,FL1,PD1,CR1,BU1,OD3,XR3,OU3,ER1,DU3,EL1,OD3,AR1,DU1,XL3,BD1,CL1,PU1,FR1,KU1,OU1,HL4,KD1,FL1,PD3,CR1,BU1,OD1,XR4]`
- **D32:** `[BL1,PU3,ER1,GU3,QR1,ID1,DL1,EL2,HU3,QR2,ER2,FR3,RD3,BL1,GD2,HU1,XR3,AD1,BL2,RU3,DR1,IU3,DL1,RD3,BR2,AU1,IU1,XL3,RU2,GU1,FL4,RD2,GD1,XR3,RU1,QL3,GD1,EL1,PD3,XR1]`
- **D33**: `[RL1,PU1,CU1,XR1,HR1,DD1,QR3,AD3,RL2,BU1,XL1,EU3,IR1,AD1,QL3,PD1,DU2,FL3,DD2,QR1,BD1,RR3,EU1,XR1,AU4,FL1,XL1,ED1,RL1,PU1,QR2,IL1,ED3,QL3,CD2,DU1,XR1,AD1,RL2,BU1,HL1,PD3,XR3]`
- **D34**: `[PD1,AR1,BU1,FU3,GR1,KD1,QL1,EU2,DR3,ED2,BD1,AL1,PU1,QR3,KU1,GL1,OD3,XR1,KU3,XL1,OU3,QL3,PD1,AR1,BU1,EU2,DL4,ED2,BD1,AL1,PU1,QR3,OD3,AL2,BU1,FU1,XR3,KD1,AL1,OU3,QL1,PD3,XR1]`
- **D35**: `[CR1,DD1,RR1,XR1,OD1,PL1,AL1,QU1,RR2,EU3,RL1,QD1,AR1,PR1,OU1,RL2,DU1,GR4,DD1,RR1,OD1,PL1,AL1,QU1,RR2,BD3,ED3,RL1,QD1,AR1,PR1,CL2,XL2,OU1,RL2,DU3,RR1,FL1,GL1,QD2,OD1,PL1,DU1,XR3]`

Java code examples

Maze solver:

```java
public class Solver {

    static List<Integer[]> shortest_path(Map<Integer[], Integer[]> prev,
                Integer[] src, Integer[] target) {
        List<Integer[]> path = new ArrayList<>();
        Integer[] u = target;
        while (!u.equals(src)) {
            path.add(0, u);
            u = prev.get(u);
        }
        path.add(0, src);
        return path;
    }

    static List<Integer[]> solve(Map<Integer[], Set<Integer[]>> graph,
                Integer[] src, Integer[] target) {

        Map<Integer[], Integer> dist = new HashMap<Integer[], Integer>();
        Map<Integer[], Integer[]> prev = new HashMap<Integer[], Integer[]>();
        for (Entry<Integer[], Set<Integer[]>> entry : graph.entrySet()) {
            dist.put(entry.getKey(), Integer.MAX_VALUE);
            prev.put(entry.getKey(), null);
        }

        List<Map.Entry<Integer, Integer[]>> q = new ArrayList<>();
        q.add(new AbstractMap.SimpleEntry<>(0, src));

        while (!q.isEmpty()) {
            Collections.sort(q, new Comparator<Map.Entry<Integer, Integer[]>>() {
                @Override
                public int compare(Map.Entry<Integer, Integer[]> a1,
                        Map.Entry<Integer, Integer[]> a2) {
                    return a1.getKey().compareTo(a2.getKey());
                }
            });

            Entry<Integer, Integer[]> entry = q.remove(0);
            Integer distu = entry.getKey();
            Integer[] u = entry.getValue();

            if (u.equals(target)) {
                return shortest_path(prev, src, u);
            }

            for (Integer[] v : graph.get(u)) {
                Integer alt = distu + 1;
                if (alt < dist.get(v)) {
                    dist.put(v, alt);
                    prev.put(v, u);
                    q.add(new AbstractMap.SimpleEntry<>(alt, v));
                }
            }
        }

        return Arrays.asList();
    }

    public static void main(String[] args) {
        Integer[] e00 = new Integer[]{0, 0};
        Integer[] e01 = new Integer[]{0, 1};
        Integer[] e02 = new Integer[]{0, 2};
        Integer[] e03 = new Integer[]{0, 3};
        Integer[] e10 = new Integer[]{1, 0};
        Integer[] e11 = new Integer[]{1, 1};
        Integer[] e12 = new Integer[]{1, 2};
        Integer[] e13 = new Integer[]{1, 3};
        Integer[] e20 = new Integer[]{2, 0};
        Integer[] e21 = new Integer[]{2, 1};
        Integer[] e22 = new Integer[]{2, 2};
        Integer[] e23 = new Integer[]{2, 3};
        Integer[] e30 = new Integer[]{3, 0};
        Integer[] e31 = new Integer[]{3, 1};
        Integer[] e32 = new Integer[]{3, 2};
        Integer[] e33 = new Integer[]{3, 3};

        Map<Integer[], Set<Integer[]>> graph = Map.ofEntries(
            Map.entry(e00, new HashSet<>(Arrays.asList(e01, e10))),
            Map.entry(e01, new HashSet<>(Arrays.asList(e02, e00, e11))),
            Map.entry(e02, new HashSet<>(Arrays.asList(e03, e13, e02))),
            Map.entry(e03, new HashSet<>(Arrays.asList(e13, e02))),
            Map.entry(e10, new HashSet<>()),
            Map.entry(e11, new HashSet<>()),
            Map.entry(e12, new HashSet<>(Arrays.asList(e13, e11, e02, e22))),
            Map.entry(e13, new HashSet<>()),
            Map.entry(e20, new HashSet<>(Arrays.asList(e30, e10, e21))),
            Map.entry(e21, new HashSet<>(Arrays.asList(e20, e31, e11, e22))),
            Map.entry(e22, new HashSet<>(Arrays.asList(e12, e32, e23, e21))),
            Map.entry(e23, new HashSet<>(Arrays.asList(e13, e33, e22))),
            Map.entry(e30, new HashSet<>(Arrays.asList(e20, e31))),
            Map.entry(e31, new HashSet<>(Arrays.asList(e30, e32, e21))),
            Map.entry(e32, new HashSet<>(Arrays.asList(e31, e33, e22))),
            Map.entry(e33, new HashSet<>(Arrays.asList(e32, e23)))
        );

        // [(3, 0), (3, 1), (2, 1), (2, 2), (1, 2), (0, 2), (0, 3)]
        for (Integer[] entry : solve(graph, e30, e03)) {
            System.out.println(Arrays.toString(entry));
        }
    }
```

RushHour solver:

```
public class Solver {

    static Character[][] convertToGrid(Map<Character, Integer> u,
            Metadata metadata) {

        Character[][] grid = new Character[6][6];

        for (int i = 0; i < 6; i++) {
            for (int j = 0; j < 6; j++) {
                grid[i][j] = '.';
            }
        }

        for (Character car : metadata.cars) {
            Boolean orientation = metadata.orientation.get(car);
            Integer size = metadata.size.get(car);
            Integer fp = metadata.fixedPosition.get(car);
            Integer vp = u.get(car);

            if (car.equals('x') && vp + size > 6) {
                size -= 1;
            }
            for (int d = 0; d < size; d++) {
                if (orientation) {
                    grid[vp + d][fp] = car;
                } else {
                    grid[fp][vp + d] = car;
                }
            }
        }

        return grid;
    }

    static List<Map<Character, Integer>> getNeighbors(Map<Character, Integer> u,
            Metadata metadata) {

        var neighbors = new ArrayList<Map<Character, Integer>>();
        Character[][] grid = convertToGrid(u, metadata);

        for (Character car : metadata.cars) {
            Boolean orientation = metadata.orientation.get(car);
            Integer size = metadata.size.get(car);
            Integer fp = metadata.fixedPosition.get(car);
            Integer vp = u.get(car);

            for (int np = vp - 1; np >= 0; np -= 1) {
                if ((orientation && grid[np][fp] != '.')
                        || (!orientation && grid[fp][np] != '.')) {
                    break;
                }

                var neighbor = new HashMap<Character, Integer>();
                for (Entry<Character, Integer> entry : u.entrySet()) {
                    if (entry.getKey().equals(car)) {
                        neighbor.put(entry.getKey(), np);
                    } else {
                        neighbor.put(entry.getKey(), entry.getValue());
                    }
                }

                neighbors.add(neighbor);
            }

            for (int np = vp + size; np <= 6; np++) {
                if ((np < 6 && orientation && grid[np][fp] != '.')
                        || (np < 6 && !orientation && grid[fp][np] != '.')
                        || (np == 6 && !car.equals('x'))) {
                    break;
                }

                var neighbor = new HashMap<Character, Integer>();
                for (Entry<Character, Integer> entry : u.entrySet()) {
                    if (entry.getKey().equals(car)) {
                        neighbor.put(entry.getKey(), np - size + 1);
                    } else {
                        neighbor.put(entry.getKey(), entry.getValue());
                    }
                }

                neighbors.add(neighbor);
            }
        }

        metadata.node_count += neighbors.size();
        return neighbors;
    }

    static List<Map<Character, Integer>> shortest_path(
            Map<Map<Character, Integer>,
            Map<Character, Integer>> prev,
            Map<Character, Integer> src,
            Map<Character, Integer> target) {

        var path = new ArrayList<Map<Character, Integer>>();
        Map<Character, Integer> u = target;
        while (prev.containsKey(u)) {
            path.add(0, u);
            u = prev.get(u);
        }

        path.add(0, src);
        return path;
    }

    static List<Map<Character, Integer>> solve(Map<Character, Integer> src,
            Metadata metadata) {

        var dist = new HashMap<Map<Character, Integer>, Integer>();
        var prv = new HashMap<Map<Character, Integer>, Map<Character, Integer>>();
        dist.put(src, 0);

        List<Map.Entry<Integer, Map<Character, Integer>>> q = new ArrayList<>();
        q.add(new AbstractMap.SimpleEntry<>(0, src));

        while (!q.isEmpty()) {
            Collections.sort(q, new Comparator<
                    Map.Entry<Integer, Map<Character, Integer>>>() {
                @Override
                public int compare(Map.Entry<Integer, Map<Character, Integer>> a1,
                        Map.Entry<Integer, Map<Character, Integer>> a2) {
                    return a1.getKey().compareTo(a1.getKey());
                }
            });

            Map.Entry<Integer, Map<Character, Integer>> item = q.remove(0);

            Integer distu = item.getKey();
            Map<Character, Integer> u = item.getValue();
            if (u.get('x') == 5) {
                return shortest_path(prv, src, u);
            }

            for (var v : getNeighbors(u, metadata)) {
                Integer alt = distu + 1;
                if (!dist.containsKey(v) || alt < dist.get(v)) {
                    dist.put(v, alt);
                    prv.put(v, u);
                    q.add(new AbstractMap.SimpleEntry<>(alt, v));
                }
            }
        }

        return Arrays.asList();
    }
}
```

Converting path to solution format:

```
static String convertPathToInstruction(List<Map<Character, Integer>> path,
        Metadata metadata) {

        String instruction = "";

        for (int i = 1; i < path.size(); i++) {
            var currEntry = path.get(i);
            var prevEntry = path.get(i-1);
            for (Character car : currEntry.keySet()) {
                if (!currEntry.get(car).equals(prevEntry.get(car))) {
                    Integer d = currEntry.get(car) - prevEntry.get(car);
                    Boolean v = metadata.orientation.get(car);
                    String direction =
                        (v && d > 0) ? "D" :
                        (v && d < 0) ? "U" :
                        (d > 0) ? "R" :
                        "L";
                    instruction += String.format("%s%s%d\n", car, direction,
                        (car.equals(metadata.x) && i == path.size() - 1)
                        ? Math.abs(d) - 1 : Math.abs(d)
                    );
                }
            }
        }

        return instruction;
    }
```