BFS Applications

Straight Forward View

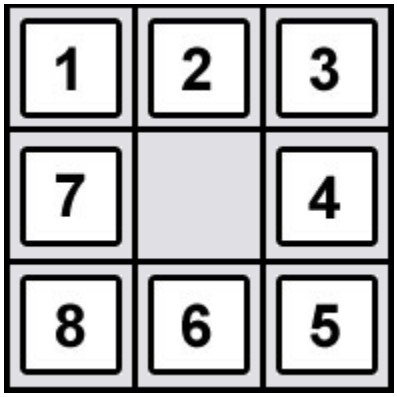
In any graph, a BFS from a single vertex can mark the distance from that vertex to all other vertices, where distance is simply the fewest number of edges one must traverse to get from the starting vertex to the ending vertex.

However, there are many problems where we can represent quite different objects as vertices and edges between vertices as a relationship between the two objects that those vertices represent in such a way that we can solve some interesting problems.

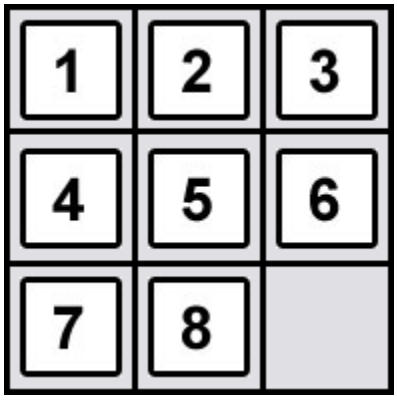
In this lecture, we'll take a look at two single player games. Classically, BFS can be used in many single player games to figure out the fewest number of moves to achieve a winning position.

Eight Puzzle

The Eight Puzzle is as follows, given a 3 x 3 grid filled with 8 tiles labeled one to 8 such as this picture:



Is there a way to slide tiles (a tile can slide into an empty space if it shares an edge with an empty space) to obtain the finishing board configuration shown below? From the position above, we can slide one of the four following tiles into the middle as a single move: 2, 4, 6 or 7.



A cursory analysis shows that there are at most 9! possible board positions, since each permutation of the numbers 1, 2, 3, .., 8, 0 (0 = blank) corresponds to a unique placement of tiles.

Imagine the problem of figuring out the fewest number of moves to solve the puzzle from any starting configuration. One way to attack this problem is a Breadth First Search. Treat each possible board position as a vertex in a graph, and connect two vertices if it's possible to move between the two board positions in a single move (sliding one piece into the open square).

When coding this, we don't actually ever create the graph. We would just keep track of each board position and how many moves it took to get there. For example, from the starting position shown above, the following four board positions are reachable in a single move (each position is separated by vertical bars):

```
1  0  3 | 1  2  3 | 1  2  3 | 1  2  3
7  2  4 | 0  7  4 | 7  4  0 | 7  6  4
8  6  5 | 8  6  5 | 8  6  5 | 8  0  5
```

Thus, our initial queue would have the first starting board position. When we dequeue that position, we would enqueue these four, marking that they are one move from the starting position.

Now, consider dequeing the next board position from the queue, the one on the left of the four above. This in turn will consider three possible positions to enqueue, of which 2 are new positions (notice that if we slide the 2 back to the top center, this was the original starting position. That would be like moving the piece back or going back to the start in a graph search.) These two new positions would get enqueued:

```
0  1  3 | 1  3  0
7  2  4 | 7  2  4
8  6  5 | 8  6  5
```

And our BFS would continue, until either, we reach all possible positions and never get to the finishing position, OR we reach our finishing position.

Since no vertex in this graph has a degree of more than 4, the total number of edges in the graph can't exceed (4 x 9!)/2 < 1,000,000. Thus, a single breadth first search can quickly (under a second) identify all possible positions reachable from a board state as well as the fewest number of moves necessary to reach each of those positions.

Thus, if we want to determine the fewest number of moves to solve one puzzle, we can do this quickly. But what if we wanted to solve many puzzles, say 10,000? In this case, our run-time of 10000 x (2 x 9!) is rather prohibitive. But, one key realization is that all moves are symmetric. Thus, the fewest number of moves from the ending position to a particular starting position is equal to the fewest number of moves from that starting position to the ending position. Thus, we can run ONE BFS from the ending position out to all other positions, and store these as the answers (fewest number of moves) for solving ANY Eight Puzzle Configuration and can they output each result quickly.

Rush Hour

This is a game played on a 6 x 6 grid with some cars. Here is an example of an initial game configuration:

In this picture, the goal would be to drive the red car straight to the right and out the exit. You can slide each of the cars either horizontally or vertically, depending on its current configuration. (So the orange car about the red car slides horizontally as does the purple car near the bottom right. The rest of the cars slide vertically.)

We can use Breadth First Search to find the fewest number of moves possible to slide the red car out. Here, we consider a single move sliding one car any number of spaces in a single direction. For each car, the direction must be either up/down or left/right.

The key idea is that each possible board position is a vertex in a graph and again, we connect each pair of vertices if we can go between the corresponding board positions in a single move. Let's consider storing the situation above in a 6 x 6 integer array. Open squares are 0, the red car is 1, the orange car is 2, the green car is 3, the blue car is 4, the yellow car is 5, the vertical purple car of length 3 is 6, the vertical car at the bottom of length 2 is 7 and the last horizontal car is 8. In this case, the board position above can be represented as:

0 0 0 0 3 5
0 0 2 2 3 5
0 0 1 1 4 5
0 0 6 0 4 0
0 0 6 7 8 8
0 0 6 7 0 0

Car 1 can move back 1 or 2 squares, Car 2 can move back 1 or 2 squares. Car 5 can move down 1 square, and Car 7 can move up one square in a single move.

While the analysis of the total number of board positions is much more difficult here, it turns out that there are relatively few possible achievable positions due to the structure of the cars. (Each car can be in at most 4 or 5 places total, and the relative locations of the cars prevent many of those combinations.) Thus, again, to solve a single puzzle, a BFS can be done from the start state to any end state. (There are many end states, any state with car #1 all the way to the left is a valid end state.)

Implementation Details

The hardest part of implementing solutions to these single player games is how to store the board concisely. For the Eight Puzzle, we can store each board position as a string of characters of length 9, or a single integer of 9 digits, with a leading digit of 0 allowed. Typically, rather than storing the shortest distances in an array, since we don't have "nice numbers" for each board position, we can store shortest distances in a HashMap (in Java). For example, for the Eight Puzzle, when going from the end state to all possible starting states, our HashMap would initially look like this:

```
HashMap<Integer,Integer> distances = new HashMap<Integer,Integer>();
distances.put(123456780, 0);
```

Then, as we generate new neighbors to vertices, we would add those distances to the HashMap. We can use the HashMap to check to see which board positions have been visited as well.

For the eight puzzle problem, our queue can be a queue of integers. When we dequeue what we would do is use the single integer to build a 3 x 3 board, then we would make the moves on that 3 x 3 board to see which positions are reachable next. Then for each of those new positions, we'd encode them as a single integer before enqueuing or storing distances in the HashMap.

Thus, it makes sense for the implementation of these sorts of problems to have two functions:

1) A function that takes in a board and compresses it into something shorter, a String, Integer, or Long.

2) A function that takes in the compressed representation (String, Integer, Long) and returns the usual representation (2D array) that makes move manipulation easy.

Once you have these two functions, you can go back and forth between representations as needed and use the preferred representation for each task.