Problem Modeling

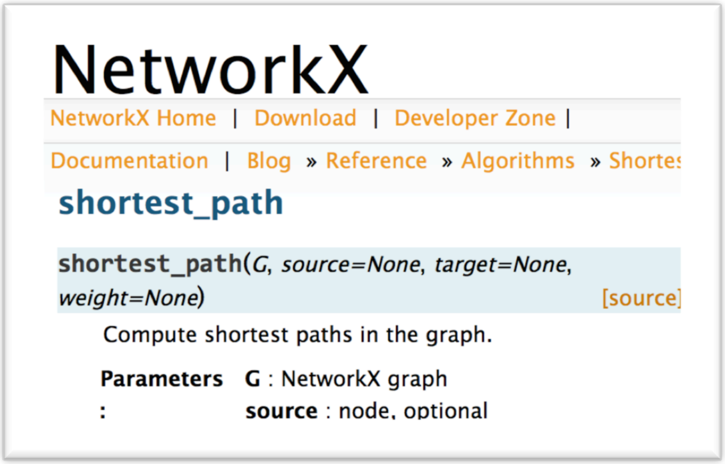
The "Jumping Jim" maze problem given in the assignment was successfully modeled as a NetworkX directed graph. Several cases were used to demonstrate and prove the results. The first, and most simple case, was given in the input file as follows:

```
# Values given          | Node index numbers
# 2 1 1                 | 1 2 3
# 1 2 2                 | 4 5 6
# 1 1 0                 | 7 8 9
```

For this case, it is easy to solve the problem by hand. Starting at the top left cell (2) and seeking the bottom right cell (0) can easily be accomplished by jumping two cells south to (1) represented by node index 7, moving one cell east (1) represented by node index 8, and finally, one more cell east to the goal (0) represented by index 9[1]. This results in a series of moves described as S E E.

By using a graph defined as a set of nodes and edges we can solve this problem numerically utilizing a shortest path algorithm. Graph analysis tools such as NetworkX provide a variety of graph traversal methods including shortest path (Figure 1). The documentation describes the parameters such that a source node (top left in our case) and destination node (bottom right for our case) can be specified. In order to specify the source and destination nodes, a separate matrix holding node indexes is created with the same dimension as the input matrix holding the "jump" values. For the simple example shown above, the source is node index 1 and the destination is node index 9 (a 3 x 3 matrix). For illustration, the following line of code is used in this analysis to find the shortest path from the NetworkX graph.

```
path = nx.shortest_path(G,1,9)
```


NetworkX
NetworkX Home | Download | Developer Zone |
Documentation | Blog » Reference » Algorithms » Shortes

shortest_path

shortest_path(*G*, *source=None*, *target=None*, *weight=None*) [source]

Compute shortest paths in the graph.

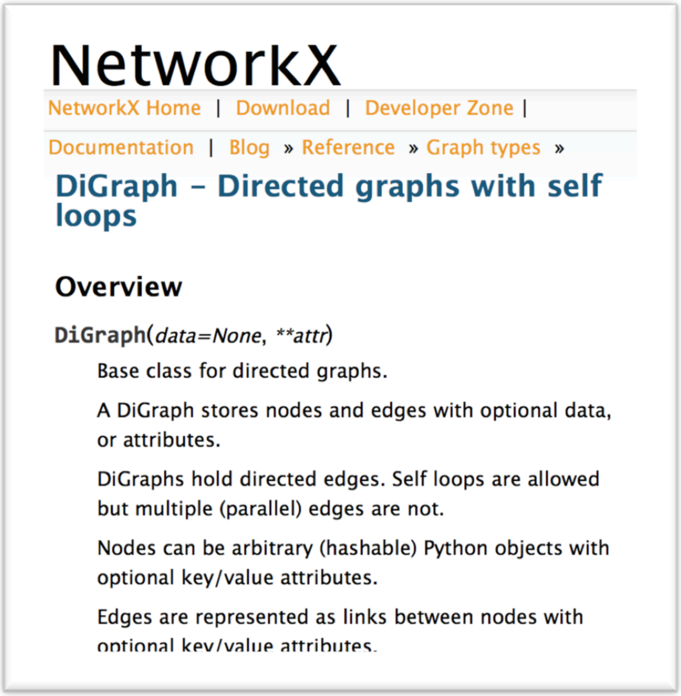
Parameters : **G** : NetworkX graph
source : node, optional

Figure 1: NetworkX shortest_path method

[1] It is assumed that jumps start at the first cell and count the number of cells from that cell. This does not assume that we count the cell we are on.

Before the shortest path method can be used, we must first construct a graph that properly represents the maze problem given. Several graphs were considered, but a directed graph was chosen to model the "Jumping Jim" maze problem. In a directed graph, the edges only point in one direction. This is the correct model for this type of maze because once Jim jumps from on cell to the next he may not be able to jump back to the previous cell. Each additional jump is based on the value given at the node that Jim lands on. Since these node values may only allow Jim to jump at this value, he must proceed based on the number of only the current node and not the predecessor node. Therefor this is a directed graph.

NetworkX provides a DiGraph constructor for creating directed graphs (Figure 2). The self loop option is not used in this project. There are several ways to construct the graph. One way to do this is to use the "add_edge" method provided by the DiGraph class[2].


NetworkX
NetworkX Home | Download | Developer Zone |
Documentation | Blog » Reference » Graph types »

DiGraph – Directed graphs with self loops

Overview

DiGraph(*data=None*, ***attr*)

Base class for directed graphs.

A DiGraph stores nodes and edges with optional data, or attributes.

DiGraphs hold directed edges. Self loops are allowed but multiple (parallel) edges are not.

Nodes can be arbitrary (hashable) Python objects with optional key/value attributes.

Edges are represented as links between nodes with optional key/value attributes.

Figure 2: NetworkX DiGraph type

Directed edges for the maze are determined by looking at the cell Jim occupies, then determining which cells he may be able to jump to based on the value at that cell. For the simple case above, Jim may only jump two cells in the directions of N, S, E, or W. However, if he were to jump north or west he would fall off the maze and this is not allowed. This leaves only the directions of south or east. From index 1, Jim may only jump to 3 or 7. From cell (or node) 1, there are two directed edges possible which are (1,3) and (1,7). The following table shows the list of directed edges used

[2] The "add_edges_from" method was also utilized during code development by passing a list of edges to the DiGraph object. As a note to future programmers and studens of NetworkX, one does not need to add both nodes and edges separately. By adding edges only, the nodes are thus constructed and the graph made complete.

to construct the simple graph. This matrix shows all available directed edges for each cell in the given matrix.

```
directed_edges = [(1, 3),(1, 7),
                  (2, 1),(2, 3),(2, 5),
                  (3, 2),(3, 6),
                  (4, 1),(4,5),(4, 7),
                  (6, 4),
                  (7, 4),(7, 8),
                  (8, 5),(8, 7),(8, 9)]
```

As a result of adding this list of directed edges to the DiGraph object, the graph shown in Figure 3 was constructed. [3]

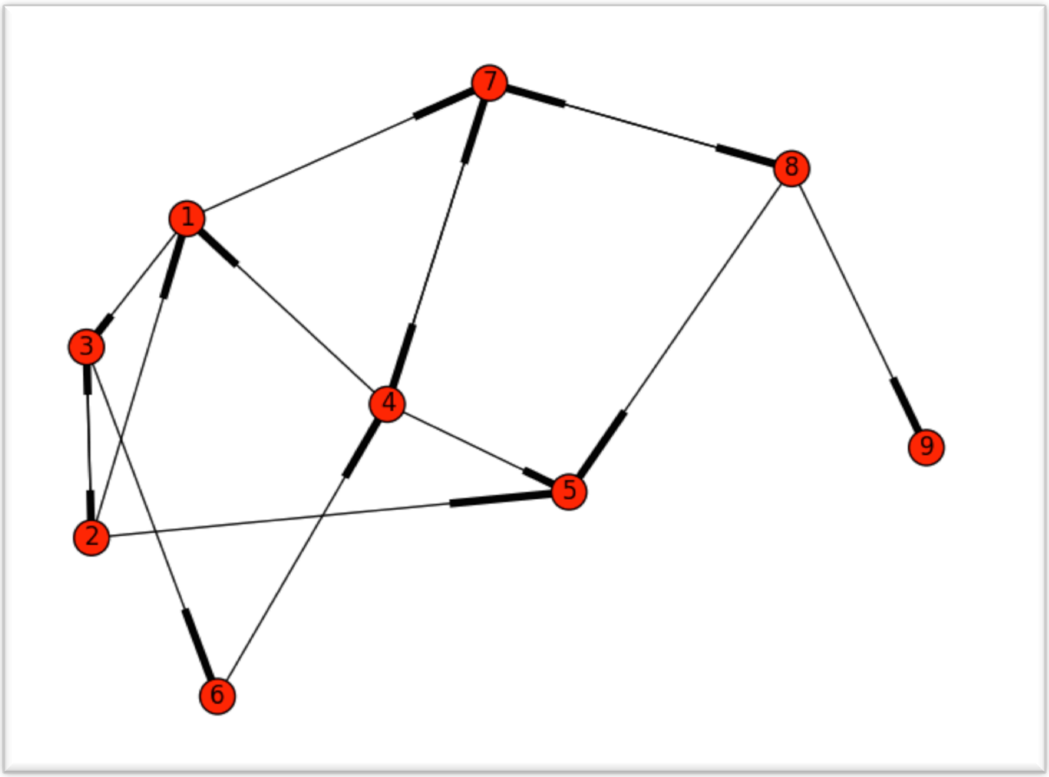


Figure 3: NetworkX directed edge graph for simple case.

Given the graph in Figure 3, the next step is to simply run the "shortest_path" method described in Figure 1 from nodes 1 – 9. Calling "shortest_path" and "shortest_path_length" returns a list based

[3] This graph was generated by utilizing the draw method with Matplotlib http://networkx.lanl.gov/reference/generated/networkx.drawing.nx_pylab.draw.html

on the index numbers described above and an integer describing how many jumps are required by Jim[4].

```
The shortest path by number is:  [1, 7, 8, 9]
The length of shortest path is:  3
```

While this list is of great use, it only relates to the index matrix constructed to solve the problem. For the user of the program, a list of simple N, S, E, W directions are required. In order to translate the shortest path list by number into the directions required by the user, an additional function is needed. This function is shown below.

```
// Translate shortest path list to directions
for i in range(1,len(path)):
    if path[i] > path[i-1]:
        if (path[i] - path[i-1]) % dimension == 0:
            print 'S'
        else:
            print 'E'
    else:
        if (path[i] - path[i-1]) % dimension == 0:
            print 'N'
        else:
            print 'W'
```

The idea here is to look at the two adjacent numbers in the shortest path list and see if the second one is bigger than the first. For example is node 7 bigger than node 1? If so then the direction must either be 'E' or 'S'. The trick is to then to determine the final direction. The question is whether the direction is in the same row of the index table or not. By using the destination value minus the starting value and finding the modulus we can determine if it is in a different row (i.e., a row below). Therefore the line:

```
if (path[i] - path[i-1]) % dimension == 0:
```

looks to see if the destination is immediately North or South of the starting point and chooses based on that decision.

It can easily be shown that this graph model and analysis is true. First take the result from the output:

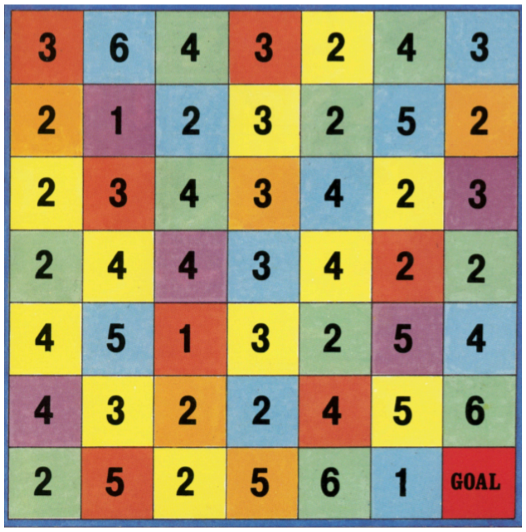
```
S E E
```

Compare this to the index matrix given above. Jim jumps two cells south, then jumps one east, and one more jump east to the destination. In addition to that simple analysis, one can look at the

[4] Other graphs such as a 4x4 matrix of all zeros were tested that did not have a possible path. In the code the analysis is wrapped in a "has_path" if statement to ensure that a path is possible. See http://networkx.github.io/documentation/latest/reference/generated/networkx.algorithms.shortest_paths.generic.has_path.html#networkx.algorithms.shortest_paths.generic.has_path for details provided by this method.

DiGraph in Figure 3. By inspection it is obvious that the shortest path goes from 1 to 7 to 8 to 9 always following directed edges. This provides a second validation that both the model and result is correct.

While the simple case is very interesting and constructive for developing, validating and proving the model, it is important to study the second, and more complex case given in the assignment. The real "Jumping Jim" maze is given in Figure 4.



3	6	4	3	2	4	3
2	1	2	3	2	5	2
2	3	4	3	4	2	3
2	4	4	3	4	2	2
4	5	1	3	2	5	4
4	3	2	2	4	5	6
2	5	2	5	6	1	GOAL

Figure 4: Jumping Jim maze

The simple case allowed for simple inspection of the maze to determine the directed edges show earlier. However, more complex cases such as Jumping Jim would require 49 different edge inspections. Another function was developed to determine the number of directed edges in any maze of dimension n. Figure 4 is a 7 dimension maze (n=7). The potential problems set for this project allows for n up to 50. The following algorithm was developed to determine all of the directed edges possible in any given maze.

```
// Add edges to build NetworkX graph --------------------------------
for row in range(dimension):
    for col in range(dimension):
        # Get value from maze
        value = maze[row,col]
        # Calculate potential destination cells and see if they are on maze
        # Ignore edges that would lead to Jim falling off maze

        # Look North
        if (row-value >= 0):
            G.add_edge(index_array[row,col], index_array[row-value,col])

        # Look South
        if (row+value < dimension):
            G.add_edge(index_array[row,col], index_array[row+value,col])

        # Look East
        if (col+value < dimension):
            G.add_edge(index_array[row,col], index_array[row,col+value])
        # Look West
        if (col-value >= 0):
            G.add_edge(index_array[row,col], index_array[row,col-value])
```

What this algorithm does is simply look in each direction to see the potential cells Jim can jump to. If they fall off the grid then they are ignored. Instead of creating a list of edges and sending the list to the graph object, edges are added one at a time as they are found.

Following the same methods as described above for the simple case, Figure 5 was developed as the graph to solve the Jumping Jim problem.

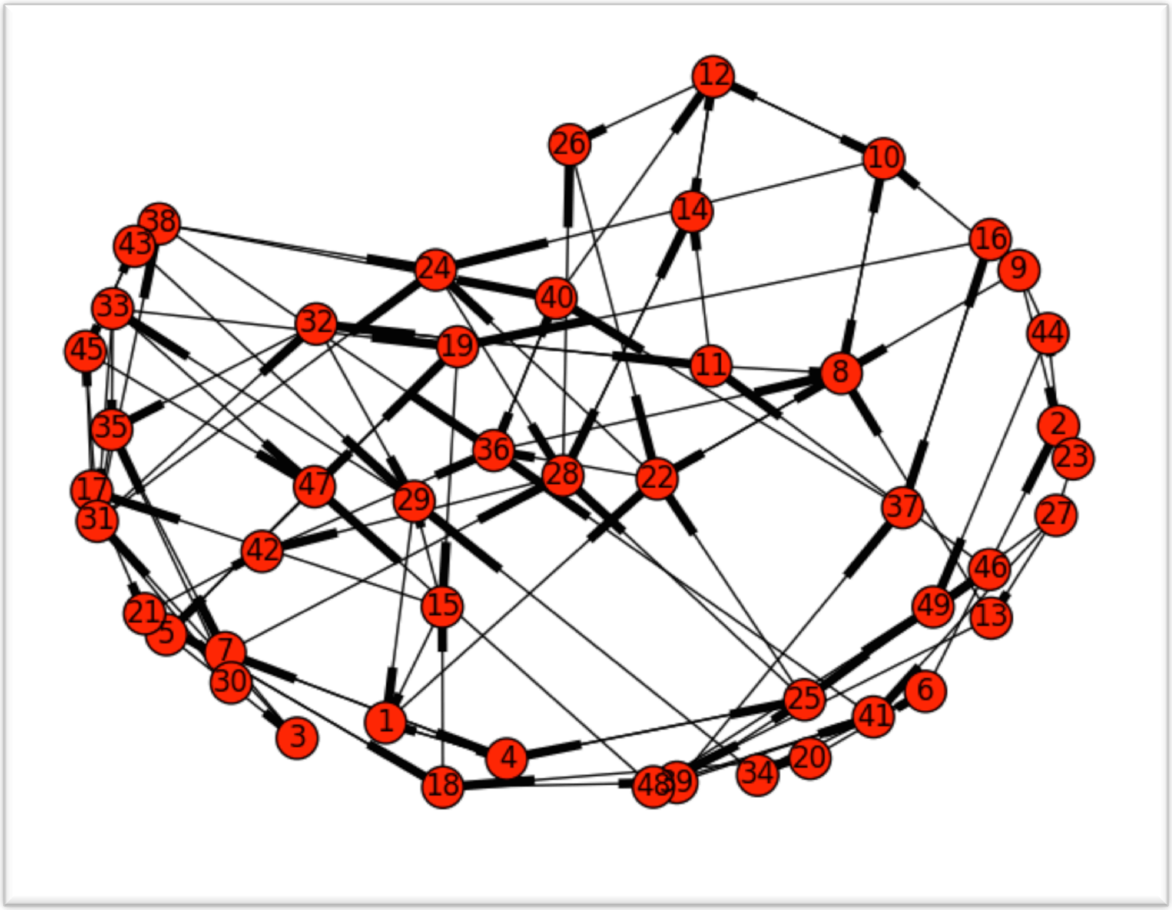


Figure 5: Directed edge graph for Jumping Jim

While this graph is too difficult to analyze by inspection, we again call shortest path and translate the matrix values into directions to give a final result that gets Jim out of the maze in a surprisingly large 18 hops!

```
The shortest path by number is:  [1, 4, 25, 46, 11, 32, 29, 33, 19, 15, 17, 21, 18,
39, 41, 6, 2, 44, 49]
The length of shortest path is:  18

E S S N S W E N W E E W S E N W S E
```