Guarini Solver-Copy1

July 8, 2020

In this notebook, we'll implement a program that we'll solve the Guarini puzzle in blink of an eye!

Before starting to implement, let's agree on the following representation of a configuration. First, let's enumerate the eight essential cells of a 3x3 board as follows:

By a configuration we mean a placement of two white and two black knight on this board. We will represent it as a string of length eight. Its i-th symbol is equal to "*", if there is no knight in the i-th cell of the board. It is equal to "B" or "W", if the i-th cell contains a black or white knight, respectively. See an example below.

We now start creating the graph of all configurations. Below, we iterate through all possible strings of length 8 with two W's and two B's. For this, we iterate through all possible four indices of W's and B's (out of eight possible positions).

```
In [1]: import networkx as nx
        import itertools as it

        G = nx.Graph()

        for wb_indices in it.permutations(range(8), 4):
            configuration = ['*'] * 8
            configuration[wb_indices[0]] = 'W'
            configuration[wb_indices[1]] = 'W'
            configuration[wb_indices[2]] = 'B'
            configuration[wb_indices[3]] = 'B'

            G.add_node("".join(configuration))
```

We then add edges to the graph. For this, we first fill in a list moves: moves[i] are the numbers of cells where a knight can move from the i-th cell.

```
In [2]: moves = [[] for _ in range(8)]
        moves[0] = [4, 6]
        moves[1] = [5, 7]
        moves[2] = [3, 6]
        moves[3] = [2, 7]
        moves[4] = [0, 5]
        moves[5] = [1, 4]
        moves[6] = [0, 2]
        moves[7] = [1, 3]
```

Adding edges to the graph:

```
In [3]: for node in G.nodes():
            configuration = [c for c in node]

            for i in range(8):
                if configuration[i] != "*":
                    for new_pos in moves[i]:
                        if configuration[new_pos] != "*":
                            continue
                        new_configuration = list(configuration)

                        new_configuration[i] = "*"
                        new_configuration[new_pos] = configuration[i]

                        if not G.has_edge("".join(configuration), "".join(new_configuration)):
                            G.add_edge("".join(configuration), "".join(new_configuration))
```

OK, the graph has been cooked! We can now analyze it. Let's first print its number of nodes, number of edges, and number of connected components.

```
In [4]: print(nx.number_of_nodes(G))
        print(nx.number_of_edges(G))
        print(nx.number_connected_components(G))
```

```
420
960
2
```

Well, the fact that the graph has two connected components is not surprising. The eight nodes of a 3x3 board form a cycle. Thus, one connected component contains all configurations where along this cycle two white knights are followed by two black knights, while the other connected components consists of all configurations where the white and black knights are interchanged. Lets now ensure that the configurations "W*B**W*B" and "B*B**W*W" are reachable from "W*W**B*B", while "W*B**B*W" is not.

```
In [7]: assert "W*B**W*B" in nx.node_connected_component(G, "W*W**B*B")
        assert "B*B**W*W" in nx.node_connected_component(G, "W*W**B*B")
        assert "W*B**B*W" not in nx.node_connected_component(G, "W*W**B*B")
```

The number of connected components of the resulting graph of configurations is not the only interesting property! We can now easily find an optimal (i.e., shortest) way of getting from one configuration to another one:

```
In [8]: print(" -> ".join(nx.shortest_path(G, "W*W**B*B", "B*B**W*W")))
```

```
W*W**B*B -> W**W*B*B -> W**WB**B -> WB*WB*** -> *B*WB*W* -> BB*W**W* -> B**W*BW* -> B**WB*W* ->
```

```
In [ ]:
```