INF421 PROGRAMMING PROJECT
NUMBERLINK SOLVER

VINCENT PILAUD
vincent.pilaud@lix.polytechnique.fr

1. Numberlink

Numberlink is a Japanese logic puzzle developed by Issei Nodi and published by Nikoli Co. Ltd., the same company that produced the sudoku and kakuro games. The player is given a grid with some pairs of numbers and is required to connect the different pairs with continuous paths.

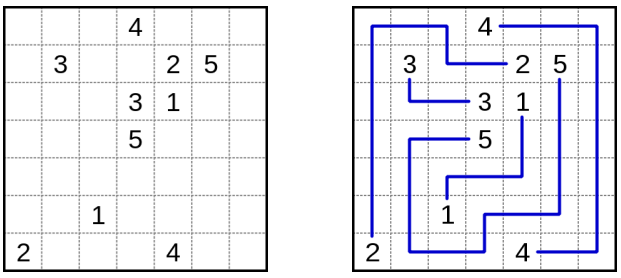

Figure 1. Numberlink solving: A grid (left) and its solution (right). Images from https://en.wikipedia.org/wiki/Numberlink.

The same game has been popularized in a more modern version by the Big Duck Games app Free Flow (see https://www.bigduckgames.com).

The goal of this programming project is to solve and create Numberlink puzzles, using different algorithmic approaches.

2. Solving puzzles

In this section, you will design algorithms to solve a Numberlink problem. We consider a graph $G = (V, E)$ with vertex set V and edge set $E \subseteq \binom{V}{2}$, and we are given a collection $X \subseteq \binom{V}{2}$ of disjoint pairs of vertices of G. A *disjoint path matching* for (G, X) is a collection P of paths in G such that each pair of X is the pair of endpoints of a path of P and the vertex set V is the disjoint union of the paths of P. Finding a disjoint path matching is known to be an NP-complete problem: the special instance with a single pair amounts to finding a Hamiltonian path between two points of a graph, which is NP-complete. In this section, we discuss two computational methods to find a disjoint path matching for small instances (G, X).

2.1. Using a SAT solver.

An instance of the *SAT problem* (SAT for satisfiability) consists in deciding whether a formula over boolean variables $x_1, \dots, x_m$ connected with operators of conjunction $\wedge$ (for "and"), disjunction $\vee$ (for "or") and negation $\neg$ (for "not") is satisfiable. The formula is in *conjunctive normal form* when it is written as a conjunction ($\wedge$) of *clauses*, each of which is a disjunction ($\vee$) of literals (x_i or $\neg x_i$).

To solve a SAT problem, we can use libraries already developed for java, for example Sat4j (see http://www.sat4j.org for the documentation). A minimal example of use of this library is provided at http://www.lix.polytechnique.fr/~pilaud/enseignement/TP/DIX/INF421/1819/.

Given a graph $G = (V, E)$ and a set of k disjoint pairs $X = \{(s_1, t_1), \dots, (s_k, t_k)\} \subseteq \binom{V}{2}$, we can transform the problem to find a disjoint path matching in G connecting the pairs of X into a SAT problem in various possible ways that we need to compare.

2.1.1. *Focus on paths.* For each vertex $v \in V$, each index $i \in [k]$ and position $p \in [n]$, we consider a boolean variable $x_{v,i,p}$ which remembers if the vertex v is the pth vertex along the path i. We also use a phantom vertex $\perp$ so that $x_{\perp,i,p}$ remembers that the path i was finished before position p. These variables should satisfy the following constraints:

- each vertex $v \in V$ appears at a single position in a single path,
- each position $p \in [n]$ in a path $i \in [k]$ is occupied by precisely one vertex (or by $\perp$ if the path finished before position p),
- if a path is finished before position p, it is also finished before position $p+1$,
- for each index $i \in [k]$, consecutive vertices along the path i are adjacent in the graph G,
- for each index $i \in [k]$, the source s_i appears first in the path i, and the sink vertex t_i is followed by $\perp$ in the path i.

Task 1. *Write a method that translates a disjoint path matching problem to a SAT solver using this approach and solves it using the library of your choice. Test on the example of Figure 1, and evaluate the computation time on complete graphs and grid graphs.*

2.1.2. *Focus on edges.* For each edge $e \in E$ and each index $i \in [k]$, we consider a boolean variable $x_{e,i}$ which remembers if the edge e belongs to the path i. These variables should satisfy the following constraints:

- each edge $e \in E$ appears in at most one path,
- each vertex $v \in V$ of a pair (s_i, t_i) of X is contained in precisely 1 edge of path i and none of the other paths,
- each vertex $v \in V$ not in a pair of X is contained in precisely two edges of one of the paths.

Task 2. *Write a method that translates a disjoint path matching problem to a SAT solver using this approach and solves it using the library of your choice. Test on the example of Figure 1, evaluate the computation time on complete graphs and grid graphs, and compare with the previous approach.*

One problem with this method is that we have no way to control that our solution does not contain closed cycles. There are two ways to get rid of cycles:

- include more variables to remember the order of the edges along the paths. This would require $O(n^2)$ variables and would therefore be equivalent to the approach of Task 1.
- each time we find a solution with a cycle C, we add constraints to ensure that the edges of C cannot appear simultaneously in the same path, and we iterate until we find a solution with no cycle.

Task 3. *Using the second solution, adapt your method of Task 2 so that it does not answer with a solution containing closed cycles. Test on the example of Figure 1, evaluate the computation time on complete graphs and grid graphs, and compare with the previous approach.*

2.1.3. *Focus on vertices.* If we are only interested in grid graphs, we could also use a different encoding of the disjoint path matching problem as follows. For each vertex $v \in V$ of the grid, each color $i \in [k]$ and each shape $\sigma \in \{$  ,  ,  ,  ,  ,  , $\}$, we consider a boolean variable $x_{v,i,\sigma}$ which remembers if the path i passes through vertex v with the shape σ. Again, these variables satisfy some elementary constraints and we should be careful to avoid closed cycles.

Task 4. *Write a method that translates a disjoint path matching problem on a grid to a SAT solver using this approach and solves it using the library of your choice. Test on the example of Figure 1, evaluate the computation time on complete graphs and grid graphs, and compare with the previous approaches.*

2.1.4. *Find a second solution.* It will be important in Section 3 to be able to decide whether a puzzle has no, one, or more than one solution.

Task 5. *Adapt your algorithm so that it can tell whether the puzzle has at least 2 solutions (*hint: you might want to run twice the SAT solver*). Is the solution of Figure 1 unique? And if the bottom 4 were in the bottom right corner?*

2.2. **Backtracking algorithm.** Our second method to solve the disjoint path matching problem is to design an adapted backtracking algorithm. The idea is simple: we start with all paths at their source vertices, and at each step we try to extend one path by appending one neighbor of its current last vertex, until all paths reach their target vertices. A priori, the paths can be extended in any order, but there are two things to keep in mind:

- we have to extend in a consistent way so that we do not obtain the same states in two different ways (for example by extending first p_1 and then p_2 or by extending first p_2 and then p_1),
- as we are using a branching algorithm, it is preferable to first make the choices with less options and leave the choices with more options for later (even more since the number of options for future choices might decrease with the first choices that are done).

Therefore, we choose to extend the path whose current last vertex has the fewest possible extensions, and we choose the minimal such path if there are ties.

Task 6. *Implement this backtracking method to solve the disjoint path matching problem. Test on the example of Figure 1 and evaluate the computation time on complete graphs and grid graphs.*

Remark 1. Many improvements can be done to cut the branches of the backtracking algorithm earlier. See for instance the blog post at https://mzucker.github.io/2016/08/28/flow-solver.html. As an extension of your project, you can implement some of these ideas and compare their impact on the computation time.

We are also interested in knowing the number of solutions of a puzzle and in particular if a puzzle has more than one solution.

Task 7. *(i) Adapt your backtracking algorithm so that it counts the number of disjoint path matchings of (G, X). Illustrate on examples of your choice.*

(ii) Adapt your backtracking algorithm so that, for a given k, it returns an empty solution if there are less than k disjoint path matchings of (G, X), and one of these disjoint path matchings if there are more than k.

2.3. **Some optional extensions.** To complete your project, you can:

- discuss and implement further variations as those described on the page https://www.bigduckgames.com and illustrated on Figure 2. You can also invent your own extensions and special rules.
- implement functions that output nice representations of your disjoint path matchings, like the ones in Figures 1 or 2. A basic graphical interface is provided at http://www.lix.polytechnique.fr/~pilaud/enseignement/TP/DIX/INF421/1819/.

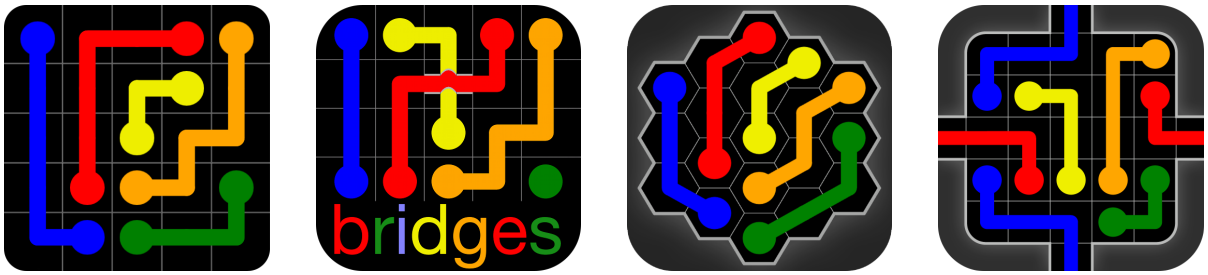

FIGURE 2. Some possible extensions from https://www.bigduckgames.com.

3. Creating puzzles

The goal of the remaining of the project is to create numberlink puzzles. We require that the puzzles have a unique solution and we will try to create puzzles with nice shapes.

3.1. **From a graph.** At the moment, we start from a graph $G = (V, E)$. Our goal is to output a numberlink instance which admits a unique solution but also minimizes the number of pairs in X. For that, we propose to proceed as follows:

(1) Compute an arbitrary partition of the vertices of V into a set P of paths of G, trying heuristically to minimize the number of paths used.
(2) As long as the numberlink problem where the pairs are the endpoints of P admits more than one solution, split an arbitrary path of P into two paths (thus creating one more pair in the numberlink instance).

Task 8. *Write a method that implements this approach to create numberlink puzzles. Evaluate the computation time on grid graphs.*

3.2. **From a picture.** We now want to create puzzles with nice shapes. As you probably have observed, it is quite tedious to create a puzzle with a nice shape.

Task 9. *Given a black and white picture and a chosen resolution, provide a method that creates a numberlink puzzle whose underlying graph is a subset of the grid that looks similar to the initial picture.*

Some binary images and a class to manipulate binary images are provided at http://www.lix.polytechnique.fr/~pilaud/enseignement/TP/DIX/INF421/1819/.

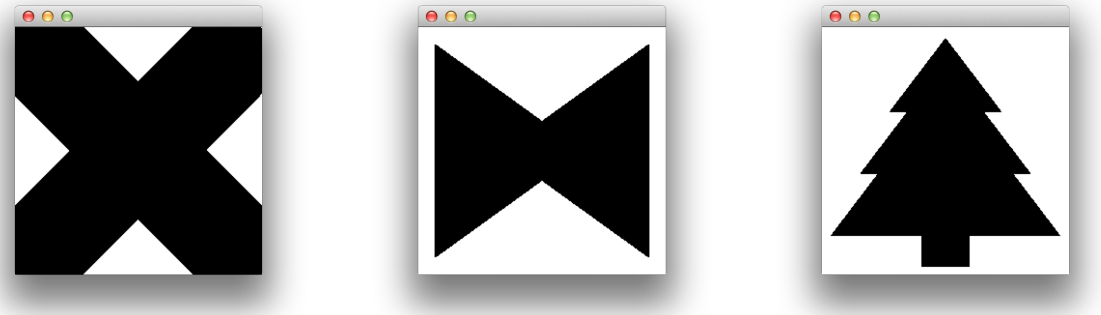

Figure 3. Some binary images to create your own numberlink puzzles.

Remark 2. In general, feel free to invent any extension of the tasks presented in this project (why not a web interface... even if this does not enter anymore in the INF421 phylosophy). Be imaginative, you are a puzzle creator!