Examples of Graph Modelling

Tony Gong

ITEE
University of Queensland

In lectures we have been studying a number of graph algorithms, with the focus this week being on DFS (which when modified slightly also does topological sorting) and strongly-connected components.

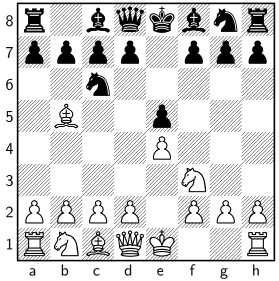
In the ensuing slides we will look at a few "real-life" problems (some more contrived than others) that can be modelled with a graph and solved using graph algorithms. The process essentially boils down to identifying:

1. where the graph is, answering questions like what are the vertices, what are the edges, are the edges directed, is the graph acyclic, etc; and
2. which graph algorithm(s) to apply to solve the given problem.

You are encouraged to try to solve each problem yourself before looking at the solutions.

Knight Placement

Suppose we were given a configuration of a chessboard, e.g.



Given a knight on the board, say the black knight on g8, the problem is to decide it is able to reach every unoccupied square on the board via only unoccupied squares (i.e. we can't take) if every other piece on the board are frozen in place.

Knight Placement

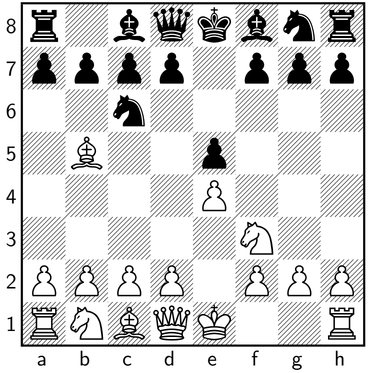
We will model the graph in the following way:

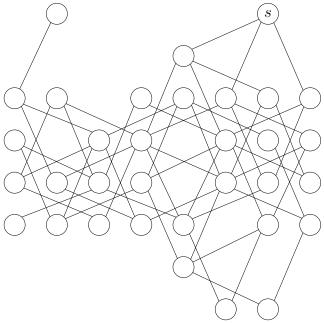
- Vertices: each unoccupied square on the board is a vertex, additionally the square the target knight is on is also a vertex.
- Edges: add an undirected edge for every pair of vertices u and v that are a knight's move apart.

The problem then is to check that the graph is connected, which we can do using DFS.

Knight Placement

Using the example shown earlier, the graph looks like:

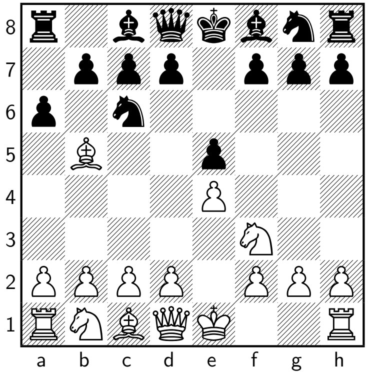


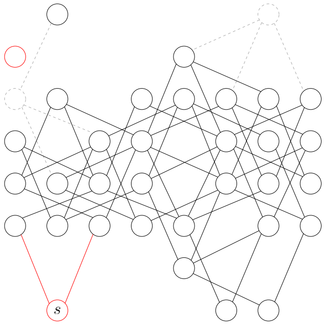


At the end of running DFS on this graph with *s* being our source node, every node in the graph would be coloured black and hence is reachable from *s*, thus the answer is yes.

Knight Placement

After the move **3. . . a6**, if we consider then the white knight on `b1`:





The graph is now disconnected with 3 connected components, and thus the answer is no.

Pick-Up Sticks

Pick-Up Sticks is a game where a bundle of sticks are randomly arranged in a pile and each player takes turn attempting to remove sticks without moving any of the others.

Suppose we are given a description of a pile of sticks (which are numbered from 1 to N) in the form of a list of pairs (a, b) which say that stick a is propped up by stick b. Note that a and b can never be equal and we cannot have (a, b) and (b, a) simultaneously (in terms of a relation, this means irreflexive and asymmetric).

Because it is difficult to remove sticks that lie underneath other sticks, we are interested in knowing if there is an order in which we can remove all of the sticks such that every stick we remove does not prop up any other.

Pick-Up Sticks

The graph can be modelled as follows:

- Vertices: each stick is a vertex.
- Edges: for each pair (a, b) encoding that a is on top of b, add the directed edge (a, b).

At this point we would like to topological sort the vertices because that would give us an ordering of the sticks from "top-to-bottom"... but is our graph guaranteed to be free of cycles?

Pick-Up Sticks

The answer is no! Even though a cycle in the form (u, v) and (v, u) is forbidden by asymmetry, it is possible to have a cycle of the form say (u, v), (v, w) and (w, u) (you should convince yourself that this arrangement of three sticks is physically possible).

Thus it is not always possible to find the required ordering, and there is an ordering iff the graph is free of cycles. One algorithm to solve this problem is then the following:

1. Use DFS for cycle detection, and if there are cycles in the graph then we know there is no such ordering.
2. If we know the graph is free of cycles, then run topological sort and this will give us the required order.

Dominoes

Suppose someone has laid out a configuration of dominoes to be toppled. Again the dominoes are numbered from 1 to N and we have again a list of pairs (a, b) that specify if domino a falls then it will topple b. As with the previous question we may assume irreflexivity and asymmetry (although these won't actually help simplify the problem).

The problem is to minimise the number of dominoes we need to manually push over such that every single domino falls over.

Dominoes

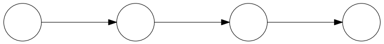
Hopefully the way the graph is modelled is obvious by this point:

- Vertices: each domino is a vertex.
- Edges: for each pair (a, b) add a directed edge (a, b).

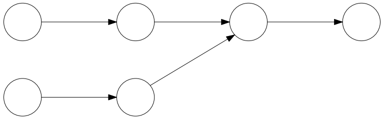
The more subtle part of this question is figuring out how to solve it. Note that as with the previous part, our directed graph may contain cycles.

Dominoes

It's often useful to use some simple examples to gain insight into the generalised version of the problem. In the simplest case where we just have a string of dominoes:



The solution is 1 because all we have to "push" the domino on the far left and that is all. Let's consider a slightly more complicated case:

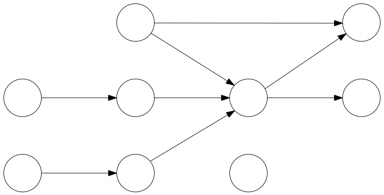


Here we need to push 2 dominoes.

Why are these two cases easy to solve?

Dominoes

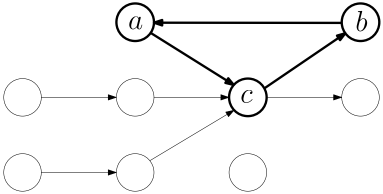
Both of the cases presented previously were easy to solve because they do not contain cycles. If the graph is a DAG, regardless of how complicated it may be:



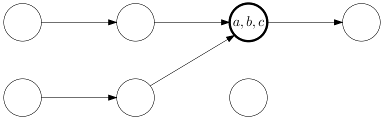
The problem reduces to counting the number of vertices with an in-degree of 0. In the above case the answer is 4.

Dominoes

In the case where we do have cycles, such as in the following:



Notice that if any of *a*, *b* and *c* gets toppled (either directly or indirectly) then all three will fall. This means that we can condense the three vertices into one:



Dominoes

So what we really want to do is turn our directed graph (possibly with cycles) into a DAG by condensing certain components of the graph into a single vertex.

Recall that in the proof of why the strongly-connected components algorithm is correct we made use of the SCC graph G^{SCC}, which is a DAG obtained by condensing each SCC into a vertex. This is precisely what we need for this question because each SCC acts essentially like a single vertex where if any vertex in an SCC is toppled then the entire SCC falls as well. We can now give an algorithm for this problem:

1. Build G^{SCC}. We do this by first identifying the SCCs, creating a vertex for each SCC and for each edge in G that connects two SCCs, add the appropriate edge into G^{SCC}.
2. Count the number of vertices with 0 in-degree in G^{SCC}.

As a final remark, observe that G^{SCC} is identical to the original graph G when it is a DAG. In other words, every vertex in a DAG forms a SCC by itself (try to prove this!).

Hopefully these three problems provided some insight into the process of reframing problems as graph problems and reassured you that the algorithms we have been studying are indeed useful!