Inputs:
n m r
s t
<vertices> * means red
<edges> - - or - ->

Red Scare!

October 28, 2020

Oh no! All I wanted was to write a straightforward reachability exercise. But some of the vertices have turned red, giving me cruel ideas for much harder questions!

Problems

In every problem of this exercise, we consider a graph G with vertex set $V(G)$ and edge set $E(G)$. The graph can be directed or undirected. Every graph in this exercise is simple (no multiple edges between any pair of vertices) and unweighted. We fix the notation $n = |V(G)|$ and $m = |E(G)|$.

Every graph comes with two specified vertices $s, t \in V(G)$ called the *start* and *end* vertices, and a subset $R \subseteq V(G)$ of *red* vertices. In particular, R can include s and t. We fix the notation $r = |R|$. In the example graph G_{ex}, we have $s = 0$, $t = 3$, and $R = \{4, 5, 7\}$.

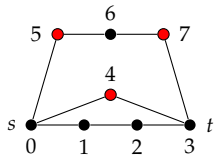

Figure 1: Example graph G_{ex} corresponding to the file `G-ex`.

An *s,t-path* is a sequence of distinct vertices $v_1, \ldots, v_l$ such that $v_1 = s$, $v_l = t$, and $v_i v_{i+1} \in E(G)$ for each $i \in \{1, \ldots, l-1\}$. The *length* of such a path is $l - 1$, the number of edges. Note that this definition requires the vertices on a path to be distinct, this is sometimes called a *simple* path.

The problems we want solved for each graph are the following:

None Return the length of a shortest s,t-path internally avoiding R. To be precise, let P be the set of s,t-paths $v_1, \ldots, v_l$ such that $v_i \notin R$ if $1 < i < l$. Let $l(p)$ denote the length of a path p. Return $\min\{\, l(p) \colon p \in P \,\}$. If no such path exists, return '-1'. Note that the edge st, if it exists, is an s,t-path with $l = 2$. Thus, if $st \in E(G)$ then the answer is 1, no matter the colour of s or t. In G_{ex}, the answer is 3 (because of the path 0, 1, 2, 3.)

Remove red vertices from the graph and use Dijkstra's to find shortest path.

Some Return 'true' if there is a path from s to t that includes at least one vertex from R. Otherwise, return 'false.' In G_{ex}, the answer is 'yes' (in fact, two such paths exist: the path 0, 4, 3 and the path 0, 5, 6, 7, 3.)

Many Return the maximum number of red vertices on any path from s to t. To be precise, let P be the set of s,t-paths and let $r(p)$ denote the number of red vertices on a path p. Return $\max\{\, r(p) \colon p \in P \,\}$. If no path from s to t exists, return '-1'. In G_{ex}, the answer is '2' (because of the path 0, 5, 6, 7, 3.)

Few Return the minimum number of red vertices on any path from s to t. To be precise, let P be the set of s,t-paths and let $r(p)$ denote

the number of red vertices on a path p. Return $\min\{\, r(p) \colon p \in P \,\}$. If no path from s to t exists, return '-1'. In G_{ex}, the answer is 0 (because of the path 0, 1, 2, 3.)

Alternate Return 'true' if there is a path from s to t that alternates between red and non-red vertices. To be precise, a path $v_1, \ldots, v_l$ is *alternating* if for each $i \in \{1, \ldots, l-1\}$, exactly one endpoint of the edge $v_i v_{i+1}$ is red. Otherwise, return 'false.' In G_{ex}, the answer is 'yes' (because of the path 0, 5, 6, 7, 3.)

Requirements

Solved instances. For three of the problems (I'm not telling which), you need to be able to handle *all* instances. For the remaining two problems, you will not be able to solve all instances, but should be able to solve roughly half of them. Your solutions must run in polynomial time. If you do have a polynomial-time implementation that takes more than 1 hour on some instance, just abort it and report this in your report.[1]

[1] However, this should not happen. As a guideline, I have a non-optimised Python implementation that solves all the instances in a single run in half an hour on a low-powered 2012 laptop.

For two of the problems (I'm not telling which), you will not be able to write an algorithm that works for all graphs, because the problems are hard in general. For one of those problems, you should be able to argue for computational hardness with a simple reduction. The other problem will probably mystify you. A sophisticated hardness argument exists in the research literature (but not in the course text book)—you are welcome to try to find it and include a reference in your report. But you are not required to find an explanation, and absolutely not required to come up with the reduction yourself.

Universality. Your algorithms must run in polynomial time on a well-defined class of graphs. "Well-defined class" means something like "all graphs,"[2] "directed graphs," "undirected graphs," "bipartite graphs," "acyclic graphs," "graphs of bounded treewidth," "planar graphs," "expanders" or even a combination of these.

[2] "All graphs" means all simple, loopless, unweighted graphs.

In particular, you are allowed to do something like this:

```
if (isBipartite(G)) then
  ...            # run the Strumpf–Chosa algorithm
else
  print("?!")  # problem is NP-hard for non-bipartite graphs, so give up
```

On the other hand, you are not allowed to base your algorithm on specific knowledge of which graphs are in the `data` directory, For an extreme example, the following would not be allowed:

```
if (filename == "rusty-1-17")) then
  print("14") # solved by hand
else
  print("?")  # no idea what to do
```

Libraries. This exercise focusses on choosing *between* algorithms, not implementing them. Thus, you are *not* required to write these algorithms from scratch. For instance, if you need a minimum spanning tree, and you already have a working implementation of Prim's algorithm, you are free to reuse that. In particular, you are free to use a library implementation of these algorithms. You are also free to use implementations from books or from other people, provided that you are not violating any intellectual property rights. (It goes without saying that you properly cite the source of these implementations in your report.)

You are highly encouraged to use your own implementation of standard graph algorithms that you may have made for some other exercise. If you do this, then separate that implementation in your source code, maybe by leaving it in its own file. Attribute all the original authors in the source code.

Deliverables. Hand in

1. a report; follow the skeleton in `doc/report.pdf`.
2. a text file `results.txt` with all the results, as specified in the report.
3. the programs you have written to answer the questions, including any scripts you need to run these programs on the instances, and a README file that explains how to recreate `results.txt` by running your programs.

Appendix: Gallery of Graphs

This gallery consists of descriptions and drawings of many of the graphs in the `data` directory. Ideally, these descriptions are useful for finding mistakes in your code. In particular, for many of these graphs it is obvious what the correct answers are. Some of the graphs are *random* graph—they have no structure and are pretty boring. Others, such as the Word graphs, have a lot of structure.

Individual graphs

The `data` directory contains a small number of individual graphs, typically of very small size. This includes G_{ex}, a number of small graphs of 3 vertices shown to the right, and an all-red dodecahedron. It is a good idea to use these graphs initially to ensure that your parser works, your graph data structure makes sense, etc. These graphs also serve as an invitation to create more toy examples by hand while you test your code, so ensure that everything works on very small graphs. Keep some good, clear drawings of 'your' graphs around, they help immensely when finding mistakes.

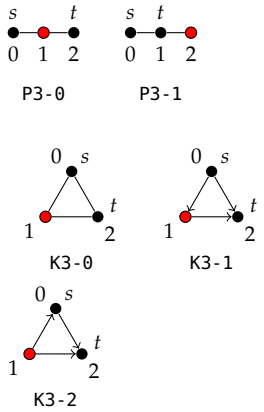

Figure 2: Paths and triangles with various choices of orientation and redness.

Word graphs

In the word graphs, each vertex represents a five-letter word of English. For $k \in \{1, 2\}$, an edge joins u and v if the corresponding words are anagrams, or if they differ in exactly k positions. For instance "begin" and "binge" are neighbours, and so are "turns" and "terns" for $k = 1$.

The word graphs come in two flavours. The *rusty word* graphs are guaranteed to include "begin," "ender," and "rusty." The vertex corresponding to "rusty" is coloured red, no other vertices are red.

The filenames are `rusty-`k`-`n.

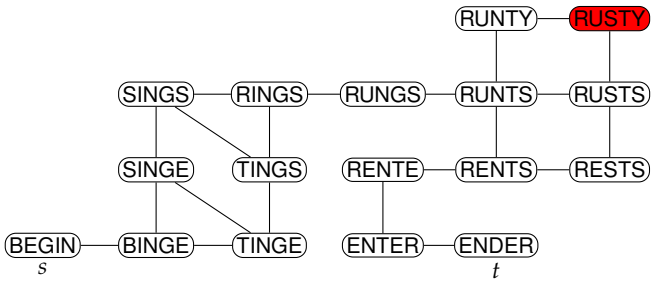

Figure 3: `rusty-1-17`

The *common word* use the same adjacency structure, and always

include 'start' and 'ender.' A word is red if it is uncommon (like 'ender'), just under half the words are uncommon. The filenames for these graphs are `common-`n.

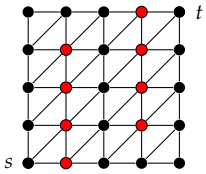

Figure 4: The grid for $N = 5$, represented by `grid-5-0`.

Grids

The Grid graphs consist of N^2 vertices that represent integer coordinates (x, y) for $x, y \in \{0, \ldots, N-1\}$. Each vertex (x, y) is connected to $(x-1, y)$, $(x, y-1)$, and $(x-1, y-1)$, provided that these vertices exist. The red vertices form a maze-like structure in the graph: Every second row is red, except for the top- or bottommost vertex, alternatingly. There is a unique s, t-path avoiding all red vertices, and a shortest alternating path following the diagonal.

Grid graphs of various sizes are represented by `grid-`N`-0`. Each of these graphs comes with two variants. In `grid-`N`-1`, some random red vertices have turned non-red (so there are 'holes' in the hedges). In `grid-`N`-2`, some random non-red vertices have turned red (so some passages are blocked).

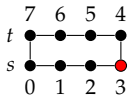

Figure 5: The single-brick wall, `wall-p-1`.

Walls

Bricks are arranged like a wall of height 2. Here are three bricks with overlap 1:

The Wall graphs are a family consisting of N overlapping 8-cycles called *bricks*. The bricks are laid in a wall of height 2, with various intervals of overlap. Each wall has a single red vertex w, the rightmost vertex at the same level as vertex 0. These graphs are interesting instances for finding paths from s to t through the red vertex. The should help you avoid some obvious pitfalls when developing an algorithm for the problem *Some*.

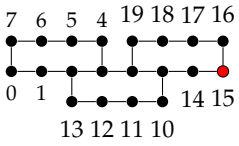

Figure 6: Three bricks with overlap 1, `wall-p-3`.

The Walls with overlap 1, called `brick-1-`N, allow an s, t-path through w.

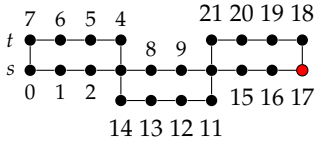

Figure 7: Three bricks with overlap 0, `wall-z-3`.

The Walls with overlap 0, called `brick-0-`N, allow a walk from s to t through w, but this walk will use $N-2$ vertices twice. In particular, such a walk it not a path, and your algorithm for Problem *Some* should not be fooled by it.

The walls with negative overlap, called `brick-n-`N also allow a walk from s to t through w, but this walk would use $N-2$ edges twice. Again, such walk is not a path.

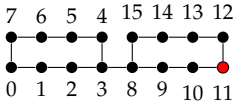

Figure 8: Two bricks with negative overlap, `wall-n-2`.

Ski

SkiFree[3] is an ancient computer game by Chris Pirih, part of the Microsoft Entertainment Pack in 1991, and going back to VT100

[3] Read more at Wikipedia's SkiFree page, the author's site `ski.ihoc.net`, or (amazingly) SkiFree Fan Fiction at `www.fanfiction.net/game/SkiFree/`.

VAX/VMS terminals, ultimately inspired by Activision's *Skiing* for the Atari 2600 console.

Time to show you can handle yourself off-pist. Get from the start to the goal, avoiding the trees, dogs rocks etc. Beware of the dreaded Yeti who is told to lurk at the red nodes of the mountain and will certainly chase you down and eat you if you pass him.

In each level, the player moves down, and either one step left or right. (Some moves are blocked by obstacles.)

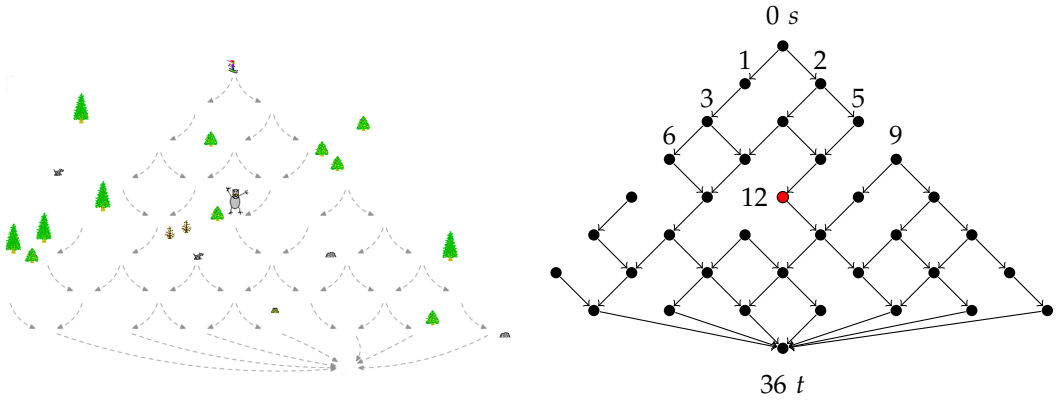

Figure 9: A Ski level and the corresponding graph. This graph is described in `ski-illustration`. Yeti is lurking at node 12. Graphics from `ski.ihoc.net`.

Increasing numbers

Each *Increase* graph is generated from a sequence $a_1, \ldots, a_n$ of unique integers with $0 \leq a_i \leq 2n$. (The random process is this: Pick a subset of size n from $\{0, \ldots, 2n\}$ and arrange the elements in random order.) We set $s = a_1$ and $t = a_n$. Odd numbers are red. There is an edge from a_i to a_j if $i < j$ and $a_i < a_j$.

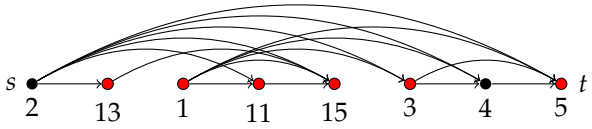

Figure 10: `increase-n8-1.0`

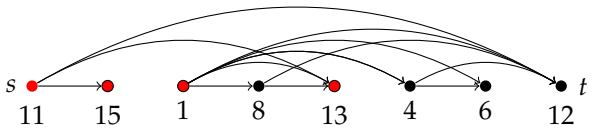

Figure 11: `increase-n8-2.0`