CSOR 4231, Section 3: Analysis of Algorithms I - Problem Set #3

Kangwei Ling - kl3076@columbia.edu

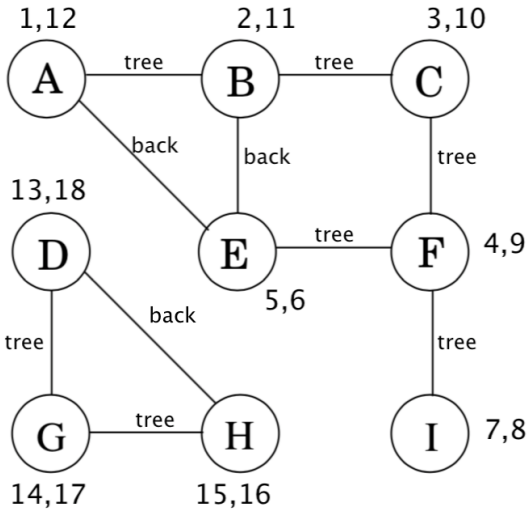
February 15, 2019

Collaborators

Zefeng Liu (zl2715), Kunyan Han (kh2931), Luoyao Hao (lh2913).

1 DPV 3.1

See below.



2 DPV 3.21

We can modify the existing linear time algorithm for finding all strongly connected component algorithm to do this:

1. Each time we find a "sink" SCC (a source in the reversed graph), we run a modified version of DFS on the SCC:

(a) We begin the DFS at any vertex s, it will reach all vertices in this SCC for sure.

(b) We do not use *visited* but rather give each vertex a color (red or black), we give s red. And each time we go on an edge (u, v), we assign red to v if u is black, otherwise black.

(c) At some point, we run in to an edge (u, v) and v have already been colored. If $color(u) \neq color(v)$, we view this as we have already visited v, just backtrack to other edges. If $color(u) = color(v)$, then we can discover an odd length cycle around vertices u, v, we can goto step 2.

(d) If the DFS ends and we have not ran into any edge (u, v) such that $color(u) = color(v)$, we need to try the next "sink" SCC.

2. If we find (u, v) that $color(u) = color(v)$, suppose the path we found when coloring is $s \rightsquigarrow u$ and $s \rightsquigarrow v$, then either the length of these two path is both even or both odd (since they have the same color).

   If they are even, we run DFS from v to find a path $v \rightsquigarrow s$, the path exists because they are in the same SCC. If the length of $v \rightsquigarrow s$ is even, then cycle $s \rightsquigarrow u \rightarrow v \rightsquigarrow s$ is an odd cycle. Otherwise, $s \rightsquigarrow v \rightsquigarrow s$ is an odd cycle.

   If they are odd, we run DFS from v to find a path $v \rightsquigarrow s$. If the length of $v \rightsquigarrow s$ is odd, then cycle $s \rightsquigarrow u \rightarrow v \rightsquigarrow s$ is an odd cycle. Otherwise, $s \rightsquigarrow v \rightsquigarrow s$ is an odd cycle.

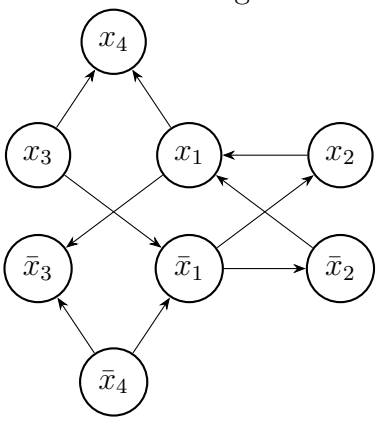
Note that, the modified SCC algorithm in step 1 still runs in linear time, and step 2 also runs in linear time.

3 DPV 3.28

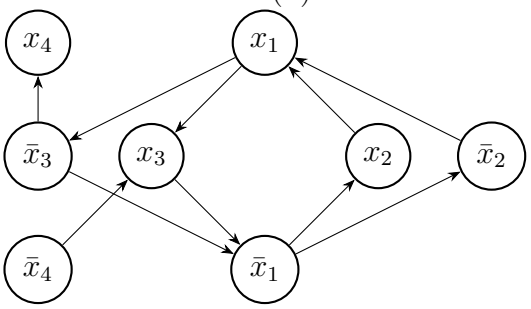
(a) set x_1, x_2, x_3, x_4 to true, true, false, true also satisfy this 2SAT.

(b) $(x_1 \vee x_2) \wedge (x_1 \vee \bar{x}_2) \wedge (\bar{x}_1 \vee \bar{x}_3) \wedge (\bar{x}_1 \vee x_3) \wedge (x_3 \vee x_4)$

(c) For the instance given in the problem:



For the instance in (b):



(d) If G_I has a strongly connected component containing both x and $\bar{x}$ for some variable x. Since they are in a strongly connected component, there's path from x to $\bar{x}$ and vice versa. Let's say $x \rightsquigarrow u_1 \rightsquigarrow \cdots \rightsquigarrow u_i \rightsquigarrow \bar{x}$ and $\bar{x} \rightsquigarrow v_1 \rightsquigarrow \cdots \rightsquigarrow v_j \rightsquigarrow x$ are the paths in the SCC.

From the method that we used to construct the graph G_I, we know that for these two path to exists, the 2SAT must have the following two parts:

$$(\bar{x} \vee u_1) \wedge (\bar{u}_1 \vee u_2) \wedge \cdots \wedge (\bar{u}_i \vee \bar{x}) \tag{1}$$
$$(x \vee v_1) \wedge (\bar{v}_1 \vee v_2) \wedge \cdots \wedge (\bar{v}_j \vee x) \tag{2}$$

To satisfy the 2SAT, both 1 and 2 need to be satisfied. If we assign true to x, thus $\bar{x}$ is false, then in order to satisfy 1, the first clause implies u_1 must be true, then u_2 must be true to satisfy the second clause, keep going on, to the very end, we have that $\bar{x}$ must be true, which is impossible since x is true.

In the same way, if we assign false to x, we cannot satisfy 2.

x must be true or false anyway, in either way, we cannot satisfy both 1 and 2. But they are all part of the 2SAT we are trying to satisfy, therefore, I has no satisfying assignment.

(e) It is obvious that each clause is translated into 2 edges in G_I, to satisfy an instance I is to satisfy every edge of G_I. Here we define that to satisfy an edge is to satisfy the clause it implies by assigning true or false to each side of the edge. To be clear, for any edge (u, v) in G_I, to satisfy this edge means by assigning true or false to u, v, we make $(\bar{u} \vee v)$ evaluate to true. (here u, v can be variables or negations). Note that the other edge $(\bar{v}, \bar{u})$ is satisfied automatically because they are from the same clause.

To see that "if none of G_I's SCCs contain both a literal and its negation, then the instance I must be satisfiable", we must show that we can satisfy every edge in this case.

We can proceed by repeatedly picking a sink strongly connected component S of G_I and make assignment to nodes inside it. The assignment are as follows:

- For each node u in S, if u contains variable that we haven't assigned yet (I call it **fresh node** below), we assign the variable to make u true. That is, if $u = x$, we assign $x = true$, otherwise $x = false$.
- If u contains variable that is already assigned (we meet the $\bar{u}$ one earlier), we leave it be.

Proceed like this will give us the assignment of variables that satisfy the instance I because:

- An edge (u, v) is satisfied if either $v = true$ or $u = false$, since (u, v) implies $(\bar{u} \vee v)$.
- Every edge (u, v) in a SCC(u, v are in the same SCC) is satisfied after we make assignments to its nodes.
  (a) if v is a fresh node, then by assignment, $v = true$, this edge is satisfied.
  (b) if v is not a fresh node, then u is also not one, because (u, v) implies $(\bar{u} \vee v)$, another equivalent edge is $(\bar{v}, \bar{u})$. We must have encountered $\bar{v}$ in an earlier "sink" SCC, and either $\bar{u}$ is in the same SCC with $\bar{v}$ or it is in an even much earlier "sink" SCC. Either cases, $(\bar{v}, \bar{u})$ must already been satisfied, thus (u, v) is satisfied. (See below for how cross SCC edges are satisfied).
- Every cross SCC edge (u, v) (where $u \in S, v \in S'$, S, S' are different SCC in our proceeding order) is satisfied after S, S' have all been processed.
  (a) Since edge (u, v) exists, S' becomes the sink earlier than S. Thus if v is a fresh node, we must have assigned it true. Then (u, v) is satisfied.
  (b) If v is not a fresh node, then just like edges in SCC, the other corresponding edge $(\bar{v}, \bar{u})$ must have been satisfied.

From the explanation above we can conclude that we can satisfy all edges, therefore satisfy the instance I with a set of assignments.

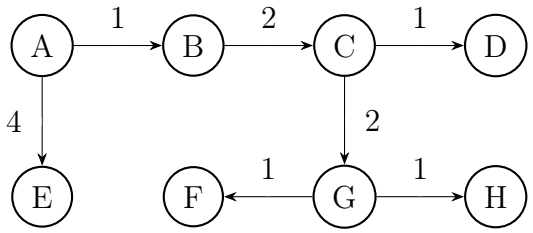
(f) From class and also the textbook, we learnt a linear algorithm to find all strongly connected components of a graph by using a modified DFS algorithm on the graph and its reverse one. We can embed the procedure to give node true/false in (e) for each sinking component. Constant operations is added to process each node, therefore we have a linear time algorithm to solve 2SAT.

4 DPV 4.1

(a) See the table below.

(b) Shortest-path tree:

iteration	A	B	C	D	E	F	G	H
initial	0	∞	∞	∞	∞	∞	∞	∞
1	0	1	∞	∞	4	8	∞	∞
2	0	1	3	∞	4	7	7	∞
3	0	1	3	4	4	7	5	∞
4	0	1	3	4	4	7	5	8
5	0	1	3	4	4	7	5	8
6	0	1	3	4	4	6	5	6
7	0	1	3	4	4	6	5	6
8	0	1	3	4	4	6	5	6



5

(a) we can modify the Dijkstra's algorithm to store the number of edges of it takes to go from the source node to this node via the current shortest path. Refering to Figure 4.8 in the textbook, we edit the testing

```
if dist(v) > dist(u) + l(u, v)
```

to

```
if (dist(v) > dist(u) + l(u, v)) or
(dist(v) == dist(u) + l(u, v) and steps(v) > steps(u) + 1)
```

where `steps(u)` is the number of edges on the current shortest path from source to u.

And when we update `dist(v)`, we also update `steps(v) = steps(u) + 1`

(b) We can modify the weights of the edges to achieve this. Note that a path will be at most contain $|V| - 1$ edges. We can use k bit to represent the number of edges in the path, k is the minimum integer s.t. $x^k - 1 \leq |V| - 1$.

We modify the weight of each edge like this: We left shift the weight by k (equivalent to multiply by 2^k), then add 1 to it.

In this way, shorter path in the original graph is still shorter, since the weight is still dominated by the original weight. And path with same weight sum in original path will be different in lower k bit, the path with fewest edges is shorter in this case.