Algorithms and Computational Complexity

Problem 1: Given a flow network $G = (V, E)$, a maximum bottleneck path between two vertices u and v is a path between u and v that allows the most flow to go through from u to v.

Give an algorithm that finds a maximum bottleneck path between two given vertices. Prove the correctness and analyze the running time of your algorithm.

Proof. The maximum bottleneck path (MBP) from s to any vertex v can be recursively defined by:

$$MBP_s(v) = \max\{\min_{(u,v)\in E}\{MBP_s(u),\ c(u,v)\}\}$$

where $MBP_s(s) = 0$. Since MBP_s is non-negative ($c(u,v) \geq 0$ in a flow network), we can modify Dijkstra's shortest path algorithm to store the MBP instead of the distance.

Algorithm 1: $MBP(G = (V, E),\ c,\ s,\ t)$

```
foreach edge in E do
    edge.mbp ← −∞
    edge.π   ← NIL
s.mbp ← 0

Q ← MAX-heap priority queue based on .mbp values of all v ∈ V
S ← {}

while Q != {} do
    u = ExtractMax(Q)
    S = S ∪ {u}

    foreach v ∈ Adj[u] do
        bottleneck ← min{u.mbp, c(u, v)}
        if v.mbp < bottleneck then        // non-equality ensures simple π-path
            v.mbp ← bottleneck
            v.π   ← u
            IncreaseKey(Q, v, bottleneck)

path = [ ]
curr ← t
while curr.π != NIL do
    path.prepend(curr)
    curr = curr.π
return path
```

Correctness: Let $\beta_s(t)$ be the MBP of node t from node s. We will prove that our algorithim

(1) Sets $t.mbp = \beta_s(t)$ at the time t in inserted into S.

(2) Prints a path that achieves $\beta_s(t)$ if one exists.

We will prove (1) with induction on the size of S. So if $|S| = 1$, then $s \in S$ (since s is always the first vertex extracted) and $s.mbp = 0 = \beta_s(S)$.

Now, notice that during the i^{th} iteration of the while loop, $|S| = i$ at line 9. So suppose that when $|S| = i$, $u \in S \Rightarrow u.mbp = \beta_s(u)$. We need to show that on the $(i+1)^{th}$ iteration, when t is extracted and appended to S, $t.mbp = \beta_s(t)$.

If $t.mbp = -\infty$, then line 12 did not execute for t, and thus, there was no vertex $u \in S$ such that $(u, t) \in E$. So there is no path from s to t and $\beta_s(t) = -\infty = t.mbp$ as wanted.

Otherwise, if $t.mbp > -\infty$, then there exist node $u \in S$ such that $t.mbp = \min\{\beta_s(u), c(u, t)\}$ and $t.\pi = u$. Since the path $(s, ..., u, t)$ is just one possible path, $t.mbp \leq \beta_s(t)$.

Suppose the optimal path is $(s, ..., p, t)$ where $p \neq t.\pi$ (since otherwise, we found the optimal path and we are done). If $p \in S$, then $\beta_s(t) = \min\{\beta_s(p), c(p, t)\}$ by induction hypothesis. But when p was examined, line 12: $t.mbp < \beta_s(t)$ must have evaluated as <u>false</u> (since $p \neq t.\pi$). Therefore $t.mbp = \beta_s(t)$ as wanted.

Now suppose $p \notin S$. Let q (possibly p) be the first element not in S in optimal path $(s, ..., q, ..., p, t)$. Well, t is only extracted if $t.mbp \geq q.mbp$ for all $q \notin S$ by ExtractMax(). So, $\beta_s(t) = \min\{\beta_s(q), \beta_s(p), w(p, t)\}$ which implies $\beta_s(t) \leq \beta_s(q) = q.mbp$ (since the optimal predecessor of q must already been analyzed since q the first element not in S). Thus

$$\beta_s(t) \leq B_s(q) = q.mbp \leq t.mbp \leq \beta_s(t)$$

and therefore $\beta_s(t) = t.mbp$ in all possible cases.

Now we prove (2). If $\beta_s(t) = -\infty$, then $t.\pi = NIL$ (since they are assigned simultaneously in lines 2-3 and 13-14. Thus, line 18 never runs and nothing is printed.

Otherwise, $\beta_s(t) = t.mbp = \min\{u.mbp, c(u, t)\}$ and the path relies on (u, t) and $\beta_s(u)$. Rightly so, t and $t.\pi = u$ is prepended to a list, and the loop repeats until s is prepended, returning a valid path.

Termination is guaranteed since the predecesor path is acyclic (nodes not in S are given predecesors in S) and s acts as the "root" node in the resulting tree made up of $(u.\pi, u)$ edges.

Runtime: The for-loop in lines 1-3 runs $|E|$ times and the while-loop in lines 18-20 runs at most $|V|$ times (since the path of predecessors must be simple).

Creating a max-fibonacci heap in line 5 takes $O(|V|)$ since we need to do $|V|$ inserts which take $O(1)$ amortized time.

We do $|V|$ ExtractMax operations in $O(\log n)$ amortized time and thus lines 7-8 runs in $O(|V| \log |V|)$. The inner-loop goes through all edges going outward from all vertices, which is simply at worst $|E|$. And since all operations in that loop take constant time (including IncreaseKey at $O(1)$ amortized time), the entire runtime of lines 7 to 15 is $|V| \log |V|$.

Therefore the entire runtime of the algorithim is $O(|E| + |V| \log |V|)$. $\square$

Problem 2: Show that 2-SAT is in P.

Proof. Let $X = \{x_1, ...x_n, \neg x_1, ..., \neg x_n\}$ be a set of literals.

Let $\mathcal{C} = C_1 \wedge ... \wedge C_k$ where C_i is a disjunction of two literals.

We will show that 2-SAT $= \{\langle \mathcal{C} \rangle \mid \mathcal{C}$ is satisfiable$\}$ belongs to P.

<u>Idea:</u> For any clause $(a \vee b)$ to be satisfied, then $\neg a \Rightarrow b$ and $\neg b \Rightarrow a$

Create a directed graph $G = (V, E)$. Set $V = X$ and $(a \vee b) \in \mathcal{C} \Rightarrow (\neg a, b), (\neg b, a) \in E$.

Let $1 \leq i \leq n$. We will first prove the statement:

$$\mathcal{C} \text{ is unsatisfiable} \iff \exists \text{ path } x_i \to \neg x_i \text{ and } \neg x_i \to x_i \text{ in } G \text{ for some variable } x_i.$$

($\Leftarrow$) Suppose that a path $x_i \to \neg x_i$ and $\neg x_i \to x_i$ exists in G. For sake of contradiction, suppose there exist an assignment $f : X \to \{0, 1\}$ such that $f(x_i) = 1 - f(\neg x_i)$ where $\mathcal{C}$ is satisfied.

Well, for any edge (a, b) in the path, $f(a) = 1 \Rightarrow f(b) = 1$. This is true since edge (a, b) means there is a clause $(\neg a \vee b) \in \mathcal{C}$ and this clause must be satisfied. Thus, for any path $c \to d$, $f(c) = 1 \Rightarrow f(d) = 1$.

Now, if $f(x_i) = 1$, then $f(\neg x_i) = 1$. This is a contradiction. Thus, $f(x_i) = 0$. But that means $f(\neg x_i) = 1$. But since there's also a path from $\neg x_i \to x_i$, then $f(x_i) = 1$, another contradiction. Thus, the existence of a valid assignment f is absurd.

($\Rightarrow$) Now, suppose there are no such two paths $x_i \to \neg x_i$ and $\neg x_i \to x_i$. Consider:

Algorithm 2: Satisfy(G=(V,E))

$X \leftarrow V$
$True \leftarrow [\,]$
$False \leftarrow [\,]$

foreach $x_i \in X$ **do**
    if $\exists$ path $x_i \to \neg x_i$ **then**
        $False.add(x_i)$
        $X.remove(x_i)$
    else
        foreach x_j reachable from x_i **do**
            $True.add(x_j)$ `// This includes` x_i
            $False.add(\neg x_j)$
            $X.remove(x_j, \neg x_j)$

return T, F

To prove that this assignment of True and False is well-defined, we first prove a lemma:

Lemma 1: If there is a path from $a \to \neg b$ in G, then there is also a path from $b \to \neg a$. This is true by the construction of the edges: if $(\neg c, d) \in E$ then $(\neg d, c) \in E$.

Looking at lines (9 - 12), no $x_j, \neg x_j$ pair are both reachable to an x_i, or otherwise, by Lemma 1, there will be a path from $x_j \to \neg x_i$ and consequently a path $x_i \to \neg x_i$, contradicting the "else" condition. (This also guarantees that for all such $x_k \to \neg x_k$ subpaths imply $x_k \in False$.)

But what if there's a path starting from x_i that sets x_j to True, and another starting from x_i' that sets $\neg x_j$ to be True? That's also absurd since that means there exist paths $x_i \to x_j$, $x_i' \to \neg x_j$ and by Lemma 1, $x_j \to x_i'$. Those paths together means x_i can reach $\neg x_j$ and x_j which we already proved is impossible.

Looking at lines (5-7), no $x_i, \neg x_i$ pair can be both assigned False because of our initial assumption that $x_i \to \neg x_i$ and $\neg x_i \to x_i$ cannot both exist.

The last possibility to consider is when x_j is assigned False in lines 5-7 and $\neg x_j$ is also assigned False but in lines 9-12. Well the former implies existence of path $x_j \to \neg x_j$ and the latter implies existence of path $x_i \to x_j$ for some x_i, again meaning both x_j and $\neg x_j$ are both reachable from x_i, violating the "else" condition that $x_i \to \neg x_i$ does not exist.

Therefore, the algorithim that assigns all variables a value is well-defined. Furthermore, since each edge represents every $(\neg a \vee b)$ clause, our True assignments ensure that every clause in $\mathcal{C}$ is satisfied.

Therefore, $\mathcal{C}$ is unsatisfiable $\iff \exists$ path $x_i \to \neg x_i$ and $\neg x_i \to x_i$ in G.

What's left is to find a polynomial-time algorithim that correctly decides if $\mathcal{C}$ is satisfiable or not.

We can check if $\mathcal{C}$ is unsatisfiable by confirming the existence of the two paths using a BFS on G. This is at worst $\mathcal{O}(|V| + |E|) = \mathcal{O}(|X| + |\mathcal{C}|)$ which is polynomial to our problem size.

To solve for a satisfying assignment, we can run the algorithim described above. At worst, we do $\mathcal{O}(|X|)$ BFS searches and thus the overall runtime is $\mathcal{O}(|X|^2 + |X||\mathcal{C}|)$ which is also polynomial to our problem size.

Therefore, 2-SAT $\in P$. $\square$

Problem 3: Consider the following DisjointHamiltonianPaths decision problem.

- Input: Graph $G = (V, E)$ (may be directed or undirected).
- Output: Does G contain at least two edge-disjoint Hamiltonian paths?

Prove DisjointHamiltonianPaths is NP-Complete.

Proof. First we show DisjointHamiltonianPaths $\in NP$.

When we are given two sequences of vertices of length n, we can verify that each vertex in the sequence is unique, and that for any two consecutive vertices u and v, $(u, v) \in E$. This verification can be done in polynomial time.

Next, we will show that HamPath $\leq_p$ DisjointHamiltonianPaths.

Say we have an instance of HamPath, $G = (V, E)$. Construct a new graph $H = (U, F)$ where:

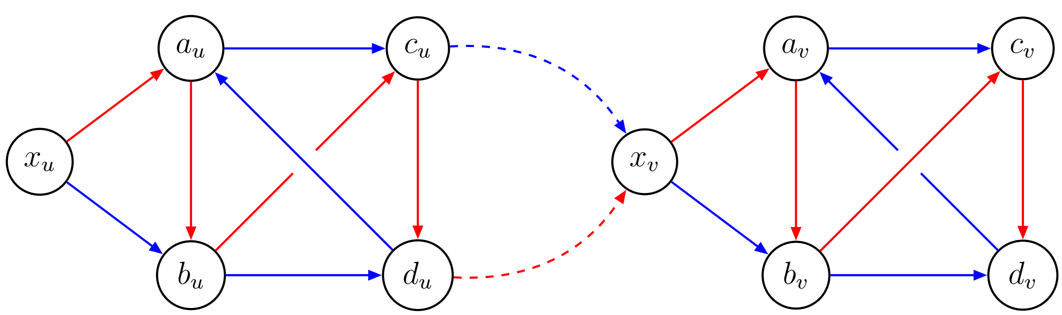
(1) For each $v \in V$, create a gadget h_v with 5 nodes $a_v, b_v, c_v, d_v, x_v \in U$
and 8 edges: $(a_v, b_v), (a_v, c_v), (b_v, c_v), (b_v, d_v), (c_v, d_v), (d_v, a_v), (x_v, a_v), (x_v, b_v) \in F$.

(2) For each $(u, v) \in E$, create 2 more edges $(c_u, x_v), (d_u, x_v) \in F$

Below is an example for $G = \{\{u, v\}, \{(u, v)\}\}$ which clearly has a Hamiltonian path.

G :

H :



The solid edges are defined in (1), while the dotted ones defined are in (2). Coloured in red and blue are two edge-disjoint Hamiltonian paths. Notice that the reduction is polynomial and that the existence of Hamiltonian paths holds even if G was undirected.

Now we will prove $G \in \textsc{HamPath} \iff H \in \textsc{DisjointHamiltonianPaths}$

($\Rightarrow$) Suppose $G \in \textsc{HamPath}$.

Each gadget h_v essentially doubles the entry and exit points of node v. So for any edge (v_i, v_{i+1}) in Hamiltonian path $\{v_1, ..., v_n\}$, we have two edge-disjoint paths $x_{v_i} \to d_{v_{i+1}}$ and $x_{v_i} \to c_{v_{i+1}}$ that traverses all nodes in h_{v_i} and $h_{v_{i+1}}$. String all paths together and we have two edge-disjoint Hamiltonian paths in H. Thus $H \in \textsc{DisjointHamiltonianPaths}$.

($\Leftarrow$) Suppose $H \in \textsc{DisjointHamiltonianPaths}$

We will prove the following lemma:

Lemma 2: Any Hamiltonian path in H that enters gadget h_v must exit exactly once. That is, any Hamiltonian path immediately traverses all five nodes of h_v.

In the directed case, there is only 1 entry point, x_v for each gadget h_v and so, a Hamiltonian path must traverse all nodes once it enters, or else some nodes will not be included in the path.

In the undirected case, we can enter h through x, c, or d. Since that is an odd number of nodes, a Hamiltonian path can't "skip" some of the nodes of h when it enters the gadget, or else it will get "trapped" when it re-enters.

A special case to consider (in both directed and undirected cases) is when the Hamiltonian path starts in a non-entry point of gadget h_v and ends a non-exit point of the same h_v. That implies the existence of a Hamiltonian cycle, and our lemma still holds.

An immediate consequence of Lemma 2 is that a Hamiltonian path through H can be described as a permutation of all gadgets h (with at most the first and last gadget repeated). And since each gadget corresponds to a vertex in V, G must also have a hamiltonian path corresponding to that permutation. Thus, $G \in \textsc{HamPath}$.

Therefore, $\textsc{HamPath} \leq_p \textsc{DisjointHamiltonianPaths}$.

And since $\textsc{HamPath}$ is a known NP-Hard problem, $\textsc{DisjointHamiltonianPaths}$ is also NP-hard and therefore NP-complete. $\square$