CS 170 Spring 2016 — Efficient Algorithms and Intractable Problems — Alessandro Chiesa and Umesh Vazirani — HW Solution 4

Due February 15, 11:59pm

1. (15 pts.) Short Questions

For these questions, either provide a short justification for your answers (one or two sentences should be enough) or provide a counterexample (please make your counterexample as small as possible, for the readers' sake).

(a) We want to use the FFT to multiply two polynomials of degree 62 and 83, respectively. What would be the size n of the FFT needed? What is the corresponding ω? What is ω^{64}? (It is a very simple complex number.)

Solution: We need to evaluate these polynomials at enough points so that the resulting *product* polynomial can be uniquely represented. The degree of the product of these polynomials will be $d_1 + d_2 = 145$, so we need to evaluate at at least 143 points.

Because the FFT requires us to use an nth root of unity such that n is a power of 2, we take the next-largest power of 2, $\boxed{n = 256}$.

To get ω we compute the *primitive* nth root of unity, $\boxed{\omega = e^{\frac{i2\pi}{256}} = e^{\frac{i\pi}{128}}.}$

Now, we have $\omega^{64} = \left(e^{\frac{i\pi}{128}}\right)^{64} = e^{\frac{i\pi}{2}}$, so $\boxed{\omega^{64} = i.}$

(b) Let G be a dag and suppose the vertices $v_1, \ldots, v_n$ are in topologically sorted order. If there is a path from v_i to v_j in G, are we guaranteed that $i < j$?

Solution: $\boxed{Yes}$ Consider a path $v_{i_1} \to \cdots \to v_{i_k}$. Since the graph is in topological order, we can conclude that $i_j < i_{j+1}$ for $j = 1, \ldots, k-1$, so by transitivity, $i_1 < i_k$.

(c) Let $G = (V, E)$ be any strongly connected directed graph. Is it always possible to remove one vertex (as well as edges to and from it) from the graph so that the remaining directed graph is strongly connected?

Solution: $\boxed{No}$ Consider the graph with 3 nodes a, b and c and edges $a \to b$, $b \to c$ and $c \to a$.

(d) Consider the array `prev` computed by Dijkstra's algorithm ran on a connected undirected graph, starting at any vertex s. Is the graph formed by the edges $\{u, \texttt{prev}(u)\}$ a tree?

Solution: $\boxed{Yes}$ There are no cycles because $\texttt{prev}(u)$ is deleted from the heap before u, therefore there cannot be another path from $\texttt{prev}(u)$ to u.

(e) Give an example where the greedy set cover algorithm does not produce an optimal solution.

Solution: This question is removed from the homework.

2. (15 pts.) All roads lead to Rome

You are the chief trade minister under Emperor Caesar Augustus with the job of directing trade in the ancient world. The Emperor has proclaimed that *all roads lead to (and from) Rome*; that is, all trade must go through Rome. In particular, you are given a strongly connected directed graph $G = (V, E)$ with positive edge weights, and there is a particular node $v_0 \in V$ (Rome). Give an efficient algorithm for finding shortest path between *all pairs of nodes*, with the one restriction that these paths must all pass through v_0 (Rome). Make your algorithm as efficient as you can, perhaps as fast as Dijkstra's algorithm.

(a) Give the efficient algorithm.

Solution:

Main Idea:

We want to run an initial computation after which we can compute the shortest distance from any vertex to another quickly. To do that we first run Dijkstra's to find the shortest paths from v_0 (Rome) to all other nodes, then find the shortest paths from all other nodes to v_0. The latter is done by reversing the direction of edges of the graph and running Dijkstra's starting from v_0, since a shortest path from v_0 to u in the reversed graph is a shortest path from u to v_0 in the original graph.

Pseudocode:

Assume we have access to a procedure DIJKSTRA(G, v) finding shortest paths starting from v, which returns two arrays `dist` and `prev` as described in the textbook.

1: $\texttt{dist}_{\text{from}}, \texttt{prev}_{\text{from}} \leftarrow$ DIJKSTRA(G, v_0)
2: $G' \leftarrow$ Reverse all edge directions of G
3: $\texttt{dist}_{\text{to}}, \texttt{prev}_{\text{to}} \leftarrow$ DIJKSTRA(G', v_0)

Proof of Correctness:

A shortest path from v_0 to u in the reversed graph is a shortest path from u to v_0 in the original graph (with the direction reversed). This is because reversing all edges also reverses the direction of all paths.

The correctness of the full algorithm comes from the correctness of Dijkstra's algorithm and the fact that the shortest path from s to t passing through v_0 is the combination of the shortest path from s to v_0 and v_0 to t.

Runtime:

This algorithm has the same runtime as Dijkstra's, which is $O((|V| + |E|) \log |V|)$ if a binary heap is used for the priority queue.

(b) Occasionally, Augustus will ask you for the (smallest) distance between two vertices. You want to do this as quickly as possible, so that Augustus does not have your head.

This is called a *distance query*: Given a pair of vertices (u, v), give the the distance of the shortest path that passes through v_0. Describe how you might store the results such that you require $O(|V|)$ storage and from your data structure you can compute the result in $O(1)$ time. For your answer, a clear description of the data structure and its usage is sufficient.

Solution:

Using the arrays saved in the above algorithm, to query the shortest path from u to v passing through v_0, return $\texttt{dist}_{\text{to}}[u] + \texttt{dist}_{\text{from}}[v]$, where u, v are used here to mean the corresponding indices of the nodes. This is a constant time operation and the storage is linear in $|V|$.

(c) On the other hand, the traders need to know the paths themselves.

This is called a *path query*: Given a part of vertices (u, v), give the shortest path itself, that passes through v_0. Describe how you might store the results such that you require $O(|V|)$ storage and from your data structure you can compute the result in $O(|V|)$ time. Again, a clear description of the data structure and its usage is sufficient.

Solution:

To query (u, v), first follow the pointers from u to v_0 in the array $\texttt{prev}_{\text{to}}$. This gives the path from u to v_0. Then then follow the pointers from v to v_0 in the array $\texttt{prev}_{\text{from}}$. This gives the reversed sequence of vertices in shortest path from v_0 to v. Return both sequences of vertices with the second sequence reversed. Both the query and storage are linear in $|V|$.

3. (15 pts.) Unique Shortest Path

Shortest paths are not always unique: sometimes there are two or more different paths with the minimum possible length. Show how to solve the following problem in $O((|V|+|E|)\log|V|)$ time.

Input: An undirected graph $G=(V,E)$; edge lengths $l_e > 0$; starting vertex $s \in V$.

Output: A Boolean array `usp[·]`: for each node u, the entry `usp[u]` should be `true` if and only if there is a *unique* shortest path from s to u. (Note: `usp[s] = true`.)

Solution:

Main Idea:
Suppose there are two different shortest paths from s to u. These two paths can either share the same last edge (the edge ending at u), or not. If they do, this can be detected by modifying Dijkstra's to detect if a node has been added to the known region previously with the same distance. If not there must be two different shortest paths to u's parent, which can be detected by propagating (for every edge (a,b)) `usp`(a) to `usp`(b) if DECREASEKEY(H,v) is called.

Pseudocode:
This can be done by slightly modifying Dijkstra's algorithm. The array `usp[·]` is initialized to `true` in the initialization loop. The main loop is modified as follows (lines 7-9 are added):

```
1: while H is not empty do
2:     u = DELETEMIN(H)
3:     for all (u,v) ∈ E do
4:         if dist(v) > dist(u) + l(u,v) then
5:             dist(v) = dist(u) + l(u,v)
6:             DECREASEKEY(H, v)
7:             usp[v] = usp[u]
8:         else if dist(v) = dist(u) + l(u,v) then
9:             usp[v] = false
```

Proof of Correctness:
By Dijkstra's proof of correctness, this algorithm will identify the shortest paths from the source u to the other vertices. For uniqueness, we consider some vertex v. Let p denote the shortest path determined by Dijkstra's algorithm. If there are multiple shortest paths, then take another path $p' \neq p$ and it will either share the same final edge (w,v) (for some vertex w) as p, or they have different final edges from p. In the former case, there must be multiple shortest paths from u to w. Using an inductive argument, which supposes that `usp` is already set correctly for all vertices that are closer than v, this will be detected in the first conditional statement when the algorithm explores from w and updates the distance to v by taking edge (w,v), as it will detect that there are multiple shortest paths to w. In the latter case, if the last edges of the two shortest paths are (w_1,v) and (w_2,v), then since edge lengths $l_e > 0$, both w_1 and w_2 must be visited before v, thus the algorithm will detect the existence of multiple shortest paths with the second conditional statement. The base case is true because there is a unique shortest path from the source s to itself.

Runtime:
The runtime analysis follows that of Dijkstra's, and will run in the required time when a binary heap is used for the priority queue.

4. (20 pts.) Arbitrage

Shortest-path algorithms can also be applied to currency trading. Suppose we have n currencies $C = \{c_1, c_2, \ldots, c_n\}$: e.g., dollars, Euros, bitcoins, dogecoins, etc. For any pair i, j of currencies, there is an exchange rate $r_{i,j}$: you can buy $r_{i,j}$ units of currency c_j at the price of one unit of currency c_i. Assume that $r_{i,i} = 1$ and $r_{i,j} \geq 0$ for all i, j.

(a) The Foreign Exchange Market Organization (FEMO) has hired Oski, a CS170 alumnus, to make sure that it is not possible to generate a profit through a cycle of exchanges; that is, for any currency $i \in C$, it is not possible to start with one unit of currency i, perform a series of exchanges, and end with more than one unit of currency i. (That is called *arbitrage*.) Give an efficient algorithm for the following problem: given a set of exchange rates $r_{i,j}$ and two specific currencies s,t, find the most advantageous sequence of currency exchanges for converting currency s into currency t. We recommend that you represent the currencies and rates by a graph whose edge lengths are real numbers.

Solution:

Main Idea:

We represent the currencies as the vertex set V of a complete directed graph G and the exchange rates as the edges E in the graph. Finding the best exchange rate from s to t corresponds to finding the path with the largest product of exchange rates. To turn this into a shortest path problem, we weigh the edges with the negative log of each exchange rate. Since edges can be negative, we use Bellman-Ford to help us find this shortest path.

Pseudocode:

1: **function** BESTCONVERSION(s,t)
2: $G \leftarrow$ Complete directed graph, c_i as vertices, edge lengths $l = \{-\log(r_{i,j}) \mid (i,j) \in E\}$.
3: `dist,prev` $\leftarrow$ BELLMANFORD(G,l,s)
4: **return** Best rate: $e^{-\texttt{dist}[t]}$, Conversion Path: Follow pointers from t to s in `prev`

Proof of Correctness:

To find the most advantageous ways to converts c_s into c_t, you need to find the path $c_{i_1}, c_{i_2}, \cdots, c_{i_k}$ maximizing the product $r_{i_1,i_2} r_{i_2,i_3} \cdot \cdots \cdot r_{i_{k-1},i_k}$. This is equivalent to minimizing the sum $\sum_{j=1}^{k-1}(-\log r_{i_j,i_{j+1}})$. Hence, it is sufficient to find a shortest path in the graph G with weights $w_{ij} = -\log r_{ij}$. Because these weights can be negative, we apply the Bellman-Ford algorithm for shortest paths to the graph, taking s as origin. The correctness of the entire algorithm follows from the proof of correctness of Bellman-Ford.

Runtime:

Same as runtime of Bellman-Ford, $O(|V|^3)$ since the graph is complete.

(b) In the economic downturn of 2016, the FEMO had to downsize and let Oski go, and the currencies are changing rapidly, unfettered and unregulated. As a responsible citizen and in light of what you saw in lecture this week, this makes you very concerned: it may now be possible to find currencies $c_{i_1}, \ldots, c_{i_k}$ such that $r_{i_1,i_2} \times r_{i_2,i_3} \times \cdots \times r_{i_{k-1},i_k} \times r_{i_k,i_1} > 1$. This means that by starting with one unit of currency c_{i_1} and then successively converting it to currencies $c_{i_2}, c_{i_3}, \ldots, c_{i_k}$ and finally back to c_{i_1}, you would end up with more than one unit of currency c_{i_1}. Such anomalies last only a fraction of a minute on the currency exchange, but they provide an opportunity for profit.

You decide to step up to help out the World Bank. Given an efficient algorithm for detecting the presence of such an anomaly. You may use the same graph representation as for part (a).

Solution:

Main Idea:

Just iterate the updating procedure once more after $|V|$ rounds. If any distance is updated, a negative cycle is guaranteed to exist, i.e. a cycle with $\sum_{j=1}^{k-1}(-\log r_{i_j,i_{j+1}}) < 0$, which implies $\prod_{j=1}^{k-1} r_{i_j,i_{j+1}} > 1$, as required.

Pseudocode: This algorithm takes in the same graph constructed in the previous part.

1: **function** HASARBITRAGE(G)
2: `dist,prev` $\leftarrow$ BELLMANFORD(G,l,s)
3: `dist`* $\leftarrow$ Update all edges one more time

4: **return** True if for some v, $\texttt{dist}[v] > \texttt{dist}^*[v]$

Proof of Correctness:
Same as the proof for the modification of Bellman-Ford to find negative edges.

Runtime:
Same as Bellman-Ford, $O(|V|^3)$.

Note:
Both questions can be also solved with a variation of Bellman-Ford's algorithm that works for multiplication and maximizing instead of addition and minimizing.

5. (20 pts.) Hyperloop construction

Elon Musk has perfected Hyperloop technology, and would like to shorten travel time between Palo Alto and Hawthorne. There is already a network of (two-way) roads $G = (V, E)$ connecting a set of cities V. Each road $e \in E$ has a (nonnegative) travel time $\ell(e)$ needed to cross it.

(a) Mr. Musk has enough cash in his checking account to finance the construction of *one* Hyperloop route, out of a set H of possible routes. Like normal roads, each Hyperloop route h would connect two cities (in the same set V) and have a travel time $\ell(h)$. As Mr. Musk's assistant, you are asked which of these Hyperloop routes, if added to the existing network G, would result in the maximum decrease in the driving time between two fixed Palo Alto (s) and Hawthorne (t). Design an efficient algorithm for this problem.

In particular, the problem is:

Input: an undirected graph $G = (V, E)$, a set of candidate edges H (between the same set of vertices V), with both $e \in E$ and $h \in H$ have nonnegative edge lengths $\ell(\cdot)$, and two vertices s, t

Output: the edge $h_{best} \in H$ such that adding it to G reduces the distance from s to t as much as possible.

Clarifications: There might not be an $s-t$ path in G, but there is definitely an $s-t$ path in $G \cup h$ for at least one $h \in H$. Edges in H may connect the same two vertices as edges in E, and their lengths could be greater or smaller than the parallel already-existing edge.

Solution:

Main Idea:
This is similar to the question "All roads lead to Rome", we find the shortest paths from s to all nodes, as well as shortest paths from all nodes to t, then we can use the information stored to find in constant time the travel time between s and t by adding any $h \in H$.

Pseudocode:
Note that on line 4, since the edges are undirected, for an edge $h \in H$ connecting nodes u and v, both (u, v) and (v, u) will be considered.

1: **function** BESTHYPERLOOP(G, H, l, s, t)
2: $\quad \texttt{dist}_s \leftarrow$ DIJKSTRA(G, s)
3: $\quad \texttt{dist}_t \leftarrow$ DIJKSTRA(G, t)
4: $\quad$ **return** $\arg\min\{\texttt{dist}_s[u] + l((u, v)) + \texttt{dist}_t[v] \mid (u, v) \in H\}$

Proof of Correctness:
If the distance between s and t decreases with the addition of $h = (u, v)$, the new shortest path from s to t will be either (1) the concatenation of the shortest path from s to u, the edge (u, v) and the shortest path from v to t, or (2) concatenation of the shortest path from s to v, the edge (u, v) and the shortest path from u to t. The length of this path will be $\min(d(s,u) + \ell(u,v) + d(v,t), d(s,v) + \ell(u,v) + d(u,t))$, where $d(x, y)$ denotes the distance from x to y in the original graph G (without adding any new edge). If none of the edges in H reduces the shortest path between s and t, we can return any edge in H.

Runtime:
This algorithm involves constructing and running Dijkstra's on a graph of size $(2|V|, 2|E|+|H|)$. When using a binary heap for Dijkstra's, the total running time is $O((|V|+|E|+|H|)\lg|V|)$.

Alternate Solution:
Same solution as next part, with $n = 1$.

(b) Mr. Musk is feeling more generous, and is willing to bestow even more Hyperloop routes upon California! Rather than paying for one Hyperloop route, he is now willing to pay to build n of them. The rest of the problem is the same as before: design an efficient algorithm to find the set of n Hyperloop routes that will decrease the driving time between Palo Alto and Hawthorne the most.

Note: It's likely that extending your algorithm from part (a) to this problem will result in an exponentially slow algorithm. When we encounter a problem that has extra features that a standard algorithm (Dijkstra's in this case) cannot handle, our choices include using the standard algorithm as a subroutine in a more complex algorithm; modifying the standard algorithm to take into account the extra features; or modifying the problem instance to incorporate additional information so that the standard algorithm will return the correct result. In this case, we recommend you try the last strategy: construct a graph such that every node also includes information about how many Hyperloop routes have been taken to reach it.

Solution:

Main Idea:
Suppose we have a directed graph with nodes partitioned into disjoint sets A and B, and for any edge $h \in H$ between A and B, h only goes from A to B. Then any path from a node in A to a node in B must pass through exactly one edge in H. This gives us a useful tool to force a shortest path to take exactly one edge from a set H. To force the shortest path to take exactly k edges from H, we make $k+1$ copies/layers of G with directed edges corresponding to the edges in H going from layer i to layer $i+1$. We use this to find the best $k \leq n$ set of edges to add.

Pseudocode:

function BESTHYPERLOOPEDGES(G, H, l, s, t, n)
    $G' \leftarrow$ Directed graph from $n+1$ copies of G with edges $(u_i, v_{i+1}), (v_i, u_{i+1})$ for all $0 \leq i < n$, $(u,v) \in H$
    `dist, prev` $\leftarrow$ DIJKSTRA(G', s_0)
    Find k that minimizes shortest path between s_0 and t_k
    return Hyperloop routes in shortest path between s_0 and t_k

Proof of Correctness:
Dijkstra's algorithm will correctly find the shortest path in G'. Since there are $n+1$ layers in the new directed graph and layer i is connected to layer $i+1$ but not vice-versa, the shortest path from layer 0 to i must pass through exactly i edges from H. This allows us to search through all paths using up to n hyperloop routes. Similar to before, if none of the edges in H reduces the shortest path between s and t, or if the best path takes less than n hyperloop routes, we can add arbitrary hyperloop routes until we get n routes in total.

Runtime:
Let $E' = E \cup H$. Then the runtime is Dijkstra's runtime on graph G', which is of size $O(n(|V|+|E'|))$. If a binary heap is used, it is $O(n(|V|+|E'|)\log|nV|)$.

6. (15 pts.) Proof of correctness for greedy algorithms

This question guides you through writing a proof of correctness for a greedy algorithm. A doctor's office has n customers, labeled $1, 2, \ldots, n$, waiting to be seen. They are all present right now and will wait until the

doctor can see them. The doctor can see one customer at a time, and we can predict exactly how much time each customer will need with the doctor: customer i will take $t(i)$ minutes.

(a) We want to minimize the average waiting time (the average of the amount of time each customer waits before they are seen, not counting the time they spend with the doctor). What order should we use? You do not need to justify your answer for this part. (Hint: sort the customers by ____)

Solution: Sort the customers by $t(i)$, starting with the smallest $t(i)$.

(b) Let $x_1, x_2, \ldots, x_n$ denote an ordering of the customers (so we see customer x_1 first, then customer x_2, and so on). Prove that the following modification, if applied to any order, will never increase the average waiting time:

- If $i < j$ and $t(x_i) \geq t(x_j)$, swap customer i with customer j.

(For example, if the order of customers is $3,1,4,2$ and $t(3) \geq t(4)$, then applying this rule with $i = 1$ and $j = 3$ gives us the new order $4,1,3,2$.)

Solution:

First observe that swapping x_i and x_j does not affect the waiting time customers $x_1, x_2, \ldots, x_i$ or customers $x_{j+1}, x_{j+1}, \ldots, x_n$ (i.e., for customers x_k where $k \leq i$ or $k > j$). Therefore we only have to deal with customers $x_{i+1}, \ldots, x_j$, i.e., for customer k, where $i < k \leq j$. For customer x_k, the waiting time before the swap is

$$T_k = \sum_{1 \leq l < k} t(x_l),$$

and the waiting time after the swap is

$$T'_k = \sum_{1 \leq l < i} t(x_l) + t(x_j) + \sum_{i < l < k} t(x_l) = T_k - t(x_i) + t(x_j).$$

Since $t(x_i) \geq t(x_j)$, $T'_k \leq T_k$, so the waiting time is never increased for customers $x_{i+1}, \ldots, x_j$, hence the average waiting time for all the customers will not increase after the swap.

(c) Let u be the ordering of customers you selected in part (a), and x be any other ordering. Prove that the average waiting time of u is no larger than the average waiting time of x—and therefore your answer in part (a) is optimal.

Hint: Let i be the smallest index such that $u_i \neq x_i$. Use what you learned in part (b). Then, use proof by induction (maybe backwards, in the order $i = n, n-1, n-2, \ldots, 1$, or in some other way).

Solution:

Let u be the ordering in part (a), and x be any other ordering. Let i be the smallest index such that $u_i \neq x_i$. Let j be the index of x_i in u, i.e, $x_i = u_j$ and k be the index of u_i in x. It's easy to see that $j > i$. By the construction of x, we have $T(x_i) = T(u_j) \geq T(u_i) = T(x_k)$, therefore by swapping x_i and x_k, we will not increase the average waiting time. If we keep doing this, eventually we will transform x into u. Since we never increase the average waiting time throughout the process, u is the optimal ordering.

7. (Bonus 0 pts.) Two-vertex Disjoint Paths

(Note: Only solve this problem if you want an extra challenge. We will not grade this problem.)

Find a polynomial-time algorithm to solve the following problem:

Input: A dag G, and vertices s_1, s_2, t_1, t_2.

Question: Does there exist a path from s_1 to t_1 and a path from s_2 to t_2, such that no vertex appears in both paths?

Your algorithm should have running time $O(|V|^c)$ for some constant c (e.g., $c = 4$ or something like that).

Solution:

Linearize the graph, and label each vertex with the index in which it appears in the topologically sorted list (e.g., the first vertex in the list is labelled 1, and so on).

Make two copies of the graph, and imagine the following solitaire game. Initially there's a marker in the first copy of the graph, placed on the vertex s_1, and there's a marker in the second copy of the graph, placed on the vertex s_2. At each turn, you are allowed to move *one* of the two markers along an edge, as long as the move obeys both of the following rules:

1. Only the marker on the "lower-numbered" vertex is allowed to move. For instance, if the first marker is sitting on a vertex labeled 7 and the second marker is sitting on a vertex labeled 5, you must move the second one (not the first).
2. You are not allowed to move a marker onto a copy of the same vertex that the other marker is sitting on.

Your goal is to get the marker in the first graph to t_1 and the marker in the second graph to t_2.

Can you win this solitaire game? We will prove that you can win, if and only if the answer to the original question is "yes."

Can we build an algorithm to check whether it's possible to win the game? Yes, we can, by constructing a new graph $G^* = (V^*, E^*)$ whose vertices represent the location of both markers, and where edges represent how the markers can move in a single turn of the solitaire game. We can then use depth-first search in G^* to determine whether the vertex $\langle t_1, t_2 \rangle$ is reachable from the vertex $\langle s_1, s_2 \rangle$. The running time is then $\Theta(|V^*| + |E^*|) = \Theta(|V^2| + |V||E|) = \Theta(|V||E|)$.

Details. Let's make this more precise. Define $l(v)$ to be the label on vertex v. In other words, if the topological sort returns $v_1, v_2, \ldots, v_n$, then we'll define $l(v_1) = 1$, $l(v_2) = 2$, and so on.

Define the graph G^* as follows. Its vertex set is

$$V^* = \{\langle v, w \rangle : v, w \in V, v \neq w\},$$

so each vertex of G^* is a pair of distinct vertices from G. Its edge set is

$$\begin{aligned} E^* = &\{(\langle v, w \rangle, \langle v', w \rangle) : l(v) < l(w), (v, v') \in E\} \cup \\ &\{(\langle v, w \rangle, \langle v, w' \rangle) : l(v) > l(w), (w, w') \in E\}. \end{aligned}$$

We can see how this corresponds to the solitaire game introduced informally above. We run DFS in G^*, starting from the vertex $\langle s_1, s_2 \rangle \in V^*$. If $\langle t_1, t_2 \rangle$ is reachable, then we answer "yes" to the original question, otherwise we answer "no." All that remains is to prove that this algorithm is correct. We do this next.

Proof of correctness.

Lemma 1 *Suppose there exists a path $s_1 = a_1 \to a_2 \to \cdots \to a_m = t_1$ and a path $s_2 = b_1 \to b_2 \to \cdots \to b_n = t_2$ in G, such that no vertex appears in both paths. Then there exists a path from $\langle s_1, s_2 \rangle$ to $\langle t_1, t_2 \rangle$ in G^*.*

Proof: Define $v_1, \ldots, v_{n+m}$ and $w_1, \ldots, w_{n+m}$ iteratively, as follows. First, $v_1 = a_1$ and $w_1 = b_1$. Also, if $l(v_i) < l(w_i)$, then define v_{i+1} to be the next vertex that appears in the path $a_1 \rightsquigarrow a_m$ immediately after v_i, and define $w_{i+1} = w_i$. Alternatively, if $l(v_i) > l(w_i)$, then symmetrically define $v_{i+1} = v_i$ and define w_{i+1} to be the next vertex that appears in the path $b_1 \rightsquigarrow a_n$ immediately after w_i.

Note that this sequence is well-defined. Because the paths $a_1 \rightsquigarrow a_m$ and $b_1 \rightsquigarrow a_n$ have no vertex in common, we are guaranteed $v_i \neq w_i$ for all i, and consequently $l(v_i) \neq l(w_i)$ (this can be proven by induction on i). Also, $v_{n+m} = a_m$ and $w_{n+m} = b_n$, so this path ends at $\langle t_1, t_2 \rangle$, as claimed. □

Lemma 2 *Suppose that there exists a path from $\langle s_1, s_2\rangle$ to $\langle t_1, t_2\rangle$ in G^*. Then there exists a path $s_1 \rightsquigarrow t_1$ and a path $s_2 \rightsquigarrow t_2$ in G, such that no vertex appears in both paths.*

Proof: Let the path in G^* be

$$\langle s_1, s_2\rangle = \langle v_1, w_1\rangle \to \langle v_2, w_2\rangle \to \cdots \to \langle v_q, w_q\rangle = \langle t_1, t_2\rangle.$$

Define $a_1, \ldots, a_m$ to be the sequence of vertices output by the following algorithm:

1. For $i := 1, 2, \ldots, q$:
2. If $i = 1$ or $v_i \neq v_{i-1}$: output v_i.

In other words, $a_1, \ldots, a_m$ is the result of removing all repeated vertices from the sequence $v_1, \ldots, v_q$. Similarly, define $b_1, \ldots, b_n$ to be the result of removing all repeated vertices from the sequence $w_1, \ldots, w_q$.

Obviously, $a_1 = v_1 = s_1$ and $b_1 = w_1 = s_2$. Since each edge $\langle v_i, w_i\rangle \to \langle v_{i+1}, w_{i+1}\rangle$ changes either the first component or second component (but not both), we find that $m + n = q$ and therefore $a_m = v_k = t_1$ and $b_n = w_k = t_2$, so this yields a path from s_1 to t_1 in G and a path from s_2 to t_2 in G.

Moreover, we can prove that these two paths are vertex-disjoint. Suppose there is some vertex x that is visited by both paths. We will show that this implies a contradiction. Let i be the smallest index where either $v_i = x$ or $w_i = x$; such an i must exist. Suppose without loss of generality it is $v_i = x$ (the other case is symmetrical). Then we know $w_i \neq x$, since by the definition of V^*, $\langle x, x\rangle \notin V^*$.

Since the labels on the vertices of any path in G are strictly increasing, we know that the labels $l(v_1), \ldots, l(v_{i-1})$ are all $< l(x)$. For a similar reason, $l(w_i) < l(x)$. Let j be the last index such that $l(w_j) < l(x)$. We see that $j \geq i$, and

$$l(w_1) \leq \cdots \leq l(w_j) < l(x) = l(v_i).$$

Therefore, none of $w_1, w_2, \ldots, w_j$ are equal to x.

Also, the same equation tells us that at states $\langle v_i, w_i\rangle, \ldots, \langle v_j, w_j\rangle$, the second marker is the one that moves, so in fact $v_i = \cdots = v_j$. Now consider the transition $\langle v_j, w_j\rangle \to \langle v_{j+1}, w_{j+1}\rangle$. We've seen that $l(w_j) < l(x) = l(v_j)$, so the second marker is the one that moves, and $v_{j+1} = v_j$. Moreover, since j was the last index such that $l(w_j) < l(x)$, we must have $l(w_{j+1}) > l(x)$ (it cannot be equal, because that would violate the second rule of the solitaire game, or equivalently, because that would take you to $\langle x, x\rangle$, which is not an element of V^*). Now since the levels of the vertices on a path are strictly increasing, it follows that all of $w_{j+1}, w_{j+2}, \ldots, w_q$ have labels larger than $l(x)$, i.e., none of $w_{j+1}, w_{j+2}, \ldots, w_q$ are equal to x.

All in all, we have shown that x does not appear in the sequence $w_1, \ldots, w_q$. This means that x does not appear in the sequence $b_1, \ldots, b_n$, either. However, this contradicts our initial assumption that $b_k = x$.

Therefore, the only remaining possibility is that there is no vertex that is visited by both paths. □