AI2615 算法设计与分析 2020-2021 春季学期

Lecture 9 – Dynamic Programming 2

2021 年 4 月 23 日

Lecturer: 张驰豪 *Scribe:* 陶表帅

In this lecture, we will continue studying dynamic programming.

1 Shortest Path Visiting at Most k Vertices

Suppose we are given an undirected weighted graph $G = (V, E, w)$ and we want to transmit a message from a vertex s to another vertex t. During transmission, a packet loss may occur when passing an intermediate vertex. Therefore, we would like to shorten the path while making sure the path does not visit too many intermediate vertices. This motivates the following problem.

Problem 1. Given an undirected weighted graph $G = (V, E, w)$, two vertices $s, t \in V$ and a positive integer $k \in \mathbb{Z}^+$, find a path from s to t with minimum length that visits at most k vertices (excluding s and t).

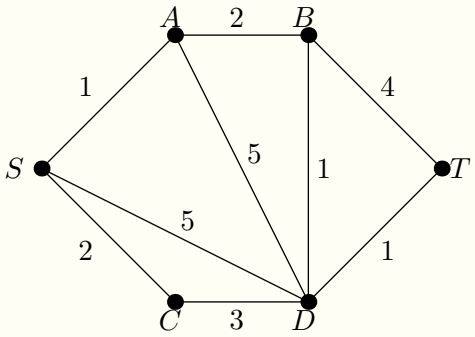

Figure 1: Find a path from S to T with minimum length that visits at most $k = 2$ intermediate vertices.

In the example in Fig. 1, the shortest path from S to T is $S \to A \to B \to D \to T$ with length 5. However, if we restrict that at most $k = 2$ intermediate vertices can be visited, the shortest path is then $S \to D \to T$ or $S \to C \to D \to T$ with length 6.

This variant of the shortest path problem can still be solved by dynamic programming. Let $d(j, u)$ be the length of the shortest path from s to u that visits *exactly* j nodes. If no such path exists, we set $d(j, u) = \infty$. The solution we would like to output is then $\min_{j=0,1,\ldots,k} d(j, t)$.

For initial conditions, we have $d(0, s) = 0$ and $d(0, u) = \infty$ for each $u \neq s$. To build a recurrence relation for $d(j, u)$, noticing that the last vertex visited by an s-u path before reaching u must be one of u's neighbors,

it suffices to consider all of u's neighbors:

$$d(j,u) = \min_{v:\{v,u\}\in E} \{d(j-1,v) + w(v,u)\}.$$

The algorithm is presented in Algorithm 1.

Algorithm 1 Finding an s-t path with minimum length visiting at most k intermediate vertices

Input: $G = (V, E, w)$, $s, t \in V$ and $k \in \mathbb{Z}^+$

Output: the length of the shortest s-t path visiting at most k intermediate vertices

1: $d(0,s) \leftarrow 0$ and $d(0,u) \leftarrow \infty$ for each $u \neq s$
2: **for** $j = 1, \ldots, k$:
3: **for** each $u \in V$:
4: $d(j,u) \leftarrow \min_{v:\{v,u\}\in E} \{d(j-1,v) + w(v,u)\}$
5: **endfor**
6: **endfor**
7: **return** $\min_{j=0,1,\ldots,k} d(j,t)$

The time complexity for Algorithm 1 is $O(k(n+m))$, this is because each vertex and each edge are visited exactly k times.

This algorithm works when the graph contains negatively-weighted edges or even negatively-weighted cycles. In particular, the constraint of at most k intermediate vertices prevents the path from infinitely looping around a negatively-weighted cycle or indefinitely going back and forth on a negatively-weighted edge.

2 All Pair Shortest Path, Floyd-Warshall Algorithm

In Lecture 5, we have studied Bellman-Ford algorithm that outputs the distance from a fixed starting vertex s to each vertex $u \in V$. In this lecture, we will consider the *all pair shortest path* problem, that requires to output the distance between all pairs of vertices.

Problem 2 (All Pair Shortest Path). Given an undirected weighted graph $G = (V, E, w)$ with $w(e) > 0$ for each $e \in E$, output the distance from u to v for all vertex pairs (u, v).

Of course, we can implement Bellman-Ford algorithm for each starting vertex $u \in V$. Since each implementation of the algorithm requires $O(nm)$ time, the overall time complexity is $O(n^2 m)$. In this lecture, we will learn a dynamic programming based algorithm, *Floy-Warshall algorithm*, that runs in $O(n^3)$ time. Suppose the n vertices are labelled $1, \ldots, n$. Let $d(k,u,v)$ be the length of the shortest path from u to v such that all the intermediate vertices are from the vertex set $\{1, \ldots, k\}$. In particular, $d(0,u,v)$ is the length

of the shortest u-v path such that all the intermediate vertices are from $\emptyset$, which means that there cannot be any intermediate vertices. If no such u-v path exists, we set $d(k,u,v)=\infty$. We need to find $d(n,u,v)$ for each $(u,v)\in V\times V$.

For initial conditions, we have

$$d(0,u,v)=\begin{cases}0 & \text{if } u=v\\ w(u,v) & \text{if } \{u,v\}\in E\\ \infty & \text{otherwise}\end{cases}.$$

To build a recurrence relation for $d(k,u,v)$, we consider two cases: 1) vertex k is on the u-v path and 2) vertex k is not on the u-v path. For case 1), we can break down the problem to two sub-problems: finding a shortest path from u to k using vertices $\{1,\dots,k-1\}$, and finding a shortest path from k to v using vertices $\{1,\dots,k-1\}$. Notice that, since the graph does not contain negatively-weighted edges, any shortest u-v path will not visit a vertex more than once (i.e., each shortest u-v path is a simple path). Therefore, in the two sub-problems above, we can safely assume that vertex k is not an intermediate vertex on the shortest u-k path and the shortest k-v path. For case 2), we know that the shortest u-v path only uses vertices from $\{1,\dots,k-1\}$. Putting together, we have

$$d(k,u,v)=\min\{d(k-1,u,k)+d(k-1,k,v),d(k-1,u,v)\}.$$

The algorithm is shown in Algorithm 2.

Algorithm 2 Floyd-Warshall algorithm

FLOYD-WARSHALL($G=(V,E,w)$)

set $d(0,i,j)\leftarrow\infty$ for each $i=1,\dots,n$ and each $j=1,\dots,n$
set $d(0,i,i)\leftarrow 0$ for each $i=1,\dots,n$
set $d(0,i,j)\leftarrow w(i,j)$ and $d(0,j,i)\leftarrow w(i,j)$ for each $\{i,j\}\in E$
for $k=1,\dots,n$:
    for $i=1,\dots,n$:
        for $j=1,\dots,n$:
            $d(k,i,j)\leftarrow\min\{d(k-1,i,k)+d(k-1,k,j),d(k-1,i,j)\}$
        endfor
    endfor
endfor
return $\{\text{dist}(i,j)=d(n,i,j)\}_{i=1,\dots,n;j=1,\dots,n}$

The time complexity for Floyd-Warshall algorithm is clearly $O(n^3)$.

Optimizing Algorithm 2. Algorithm 2 can be further optimized. The memory complexity for the algorithm is $O(n^3)$ for storing a 3-dimensional array d. This can be improved to $O(n^2)$. Notice that the

2-dimensional array $d(k,\cdot,\cdot)$ only depends on $d(k-1,\cdot,\cdot)$, and we will not start to work on $d(k,1,1)$ until all of $\{d(k-1,i,j)\}_{i=1,\dots,n;j=1,\dots,n}$ are finalized. Therefore, the first argument of $d(\cdot,\cdot,\cdot)$ is redundant. Line 7 of the algorithm can be simplified to

$$d'(i,j) \leftarrow \min\{d'(i,k)+d'(k,j), d'(i,j)\}. \tag{1}$$

A few details are needed to justify this. Suppose we are at the k-th iteration of the for-loop at Line 4. For each i and j, the nature of the two inner nested for-loops indicates that $d'(i,j)$ is updated exactly once in the k-th iteration. Consider any fixed i and j. Before executing (1), $d'(i,j)$ corresponds to $d(k-1,i,j)$ in the original algorithm (as $d'(i,j)$ has not been updated in the k-th iteration yet). However, $d'(i,k)+d'(k,j)$ may corresponds to one of the four values:

$$d(k-1,i,k)+d(k-1,k,j), d(k,i,k)+d(k-1,k,j), d(k-1,i,k)+d(k,k,j) \text{ or } d(k,i,k)+d(k,k,j), \tag{2}$$

depending on whether the values for $d'(i,k)$ and $d'(k,j)$ have already been updated in the k-th iteration. The correctness of the implementation in (1) is based on that the four values in (2) are always equal. To see this, it suffices to show $d(k-1,i,k)=d(k,i,k)$ and $d(k-1,k,j)=d(k,k,j)$. Notice that the shortest path from i to k cannot have k as an intermediate vertex, for otherwise k is visited twice (once in the middle of the path and once at the end) and we know that a shortest path on a graph with positively weighted edges cannot visit a vertex twice. Therefore, the shortest path from i to k using intermediate vertices from $\{1,\dots,k-1\}$ is the same as the shortest path from i to k using intermediate vertices from $\{1,\dots,k\}$, which implies $d(k-1,i,k)=d(k,i,k)$. For the same reason, $d(k-1,k,j)=d(k,k,j)$.

Algorithm 3 Floyd-Warshall algorithm (optimized)

FLOYD-WARSHALL($G=(V,E,w)$)

1: set $d(i,j)\leftarrow\infty$ for each $i=1,\dots,n$ and each $j=1,\dots,n$
2: set $d(i,i)\leftarrow 0$ for each $i=1,\dots,n$
3: set $d(i,j)\leftarrow w(i,j)$ and $d(j,i)\leftarrow w(i,j)$ for each $\{i,j\}\in E$
4: **for** $k=1,\dots,n$:
5: **for** $i=1,\dots,n$:
6: **for** $j=1,\dots,n$:
7: $d(i,j)\leftarrow\min\{d(i,k)+d(k,j), d(i,j)\}$
8: **endfor**
9: **endfor**
10: **endfor**
11: **return** $\{\text{dist}(i,j)=d(i,j)\}_{i=1,\dots,n;j=1,\dots,n}$

The optimized version of Floyd-Warshall algorithm is shown in Algorithm 3, where we have used $d(\cdot,\cdot)$ instead of $d'(\cdot,\cdot)$ in the analysis above.

Negatively Weighted Edges and Directed Graphs. The all pair shortest path problem is ill-defined if we are working on undirected graphs while allowing negatively weighted edges. As long as there is an negatively weighted edge on a path from u to v, the distance between u and v is $-\infty$, as we can go back and forth indefinitely on the negatively weighted edge.
The same algorithm works for directed graphs with the same analyses. For directed graphs, the algorithm works even if there are negatively weighted edges, as long as there is no negatively weighted cycle.

3 Traveling Salesman Problem

Suppose a salesman needs to visit n cities given by the n vertices on a complete undirected weighted graph, where the weight of the edge $\{i, j\}$ is the distance between city i and city j. Starting from city 1, the salesman visits each of the cities $2, \ldots, n$ exactly once, and then goes back to city 1. The *traveling salesman problem (TSP)* asks for such a route with minimum total distance traveled.

Problem 3 (Traveling Salesman Problem (TSP)). Given a complete weighted undirected graph G such that $w(\{u, v\}) > 0$ for each $u, v \in V$ with $u \neq v$, find a cycle of n vertices with minimum weight.

A simple reduction from the Hamiltonian path problem can show that TSP is NP-hard. Moreover, the same reduction can show that, for any factor F that may depend on the graph parameters, the existence of a polynomial time F-approximation algorithm for TSP would imply P = NP. That is, TSP is NP-hard to approximate to within any finite factors. See the last section of Lecture 7 for the concepts of approximation algorithms and hardness-of-approximation.
If we are not restricted to polynomial time algorithms, the brute-force search algorithm has time complexity $O((n-1)!)$, as there are $(n-1)!$ cycles in a complete graph with n vertices. We will learn a dynamic programming algorithm that runs in $O(n^2 2^n)$ time. To see that $n^2 2^n = o((n-1)!)$, notice that $\log((n-1)!) = O(n \log n)$ since $\left(\frac{n-1}{2}\right)^{\frac{n-1}{2}} < (n-1)! < (n-1)^{n-1}$ and $\log(n^2 2^n) = 2\log n + n \log 2 = O(n)$.
Suppose the vertices are labelled as $1, \ldots, n$ and we are starting at vertex 1. A natural attempt is to use the ideas from Sect. 1. Let $d(k, i)$ be the length of the shortest path from 1 to i that contains exactly k intermediate vertices. It is plausible that

$$d(k, i) = \min_{j \in V} \{d(k-1, j) + w(j, i)\}.$$

However, this equation is wrong, because the shortest path from 1 to j that contains exactly $k-1$ intermediate vertices may already contain vertex i (in particular, a shortest path here may visit a vertex more than once due to the restriction that exactly $k-1$ intermediate vertices must be used). In addition, we are unable to fix the equation if we define $d(k, i)$ in the above-mentioned way: we need to know if i is contained in the shortest path from 1 to j, and we need to store the intermediate vertices on the path; $d(k, i)$, on the other hand, only stores *the number of* the intermediate vertices.

To store the set of intermediate vertices on the shortest path, we define $d(S, i)$ to be the length of the shortest path from 1 to i such that the set of intermediate vertices is *exactly* S. For the initial condition, we have $d(\emptyset, i) = w(1, i)$ for each $i = 2, \ldots, n$. The solution output is $\min_{i=2,\ldots,n} \{d(V \setminus \{1, i\}, i) + w(i, 1)\}$, where i enumerates all possible vertices right before going back to vertex 1. It is also easy to find the recurrence relation:

$$d(S, i) = \min_{j \in S} \left\{ d(S \setminus \{j\}, j) + w(j, i) \right\}.$$

A natural topological order for the dynamic programming is by the size of S, as $d(S, i)$ only depends on those $d(S', j)$ with $|S'| = |S| - 1$. The algorithm is presented in Algorithm 4.

Algorithm 4 A dynamic programming algorithm for TSP

Input: $G = (V, E, w)$ such that $\{i, j\} \in V$ and $w(\{i, j\}) > 0$ for all $i, j \in V$ with $i \neq j$

Output: the minimum length of a cycle of n vertices.

1: **for** $k = 0, 1, \ldots, n-2$:
2: **for** each $S \in \binom{V \setminus \{1\}}{k}$: // $\binom{V \setminus \{1\}}{k}$ is the collection of all subsets of $V \setminus \{1\}$ with size k
3: **for** each $i \notin S \cup \{1\}$:
4: $d(S, i) \leftarrow \min_{j \in S} \left\{ d(S \setminus \{j\}, j) + w(j, i) \right\}$
5: **endfor**
6: **endfor**
7: **endfor**
8: **return** $\min_{i=2,\ldots,n} \{d(V \setminus \{1, i\}, i) + w(i, 1)\}$

The time complexity for Algorithm 4 is $O(n^2 2^n)$. To see this, the two nested for-loops at Line 1 and 2 enumerate all possible subsets of $V \setminus \{1\}$, and there are 2^{n-1} of them. The for-loop at Line 3 enumerates all vertices outside $S \cup \{1\}$, and there are $O(n)$ of them. Finally, executing Line 4 requires $O(n)$ time, as we need to enumerate all vertices in S.

The reason that Algorithm 4 is faster than the brute-force search is that Algorithm 4 only cares about the *set* of the intermediate vertices S without caring about the *order* of the vertices along the path, while the brute-force search, by enumerating all possible paths, takes care of the order of vertices on each path. The dynamic programming technique helps us to avoid enumerating the orders of the intermediate vertices.

4 Independent Sets on Trees

Definition 4. Given an undirected graph $G = (V, E)$, an *independent set* is a set of vertices $S \subseteq V$ such that $(u, v) \notin E$ for any $u, v \in S$.

Problem 5 (Independent Set on a Tree). Given an undirected tree $G = (V, E)$, find an independent set with the maximum size (number of vertices).

The independent set problem in general graphs is known to be NP-hard. In this section, we will consider the problem with the special case that the graph is a tree. This problem can be solved by dynamic programming. If a root has not been specified, we specify an arbitrary vertex of G as the root. In a rooted tree, for a vertex u, let T_u be the subtree rooted at u, let $\mathcal{C}(u)$ be the set of all children of u, and let $\mathcal{GC}(u)$ be the set of all grandchildren of u.

Proposition 6. *For any two vertices u and v such that T_u and T_v has no overlapping, if S_u is an independent set in T_u and S_v is an independent set in T_v, then $S_u \cup S_v$ is also an independent set.*

Proof. The tree structure ensures there is no edge between any vertex in T_u and any vertex in T_v. The proposition follows immediately. □

Let $F(u)$ be the size of the largest independent set in the subtree T_u. We need to find $F(r)$ for the root r. For initial conditions, we have $F(u) = 1$ for each leaf u. To build a recurrence relation for $F(u)$, we consider two cases: 1) u is selected in the independent set, and 2) u is not selected.
For case 1), we have $F(u) = 1 + \sum_{i \in \mathcal{GC}(u)} F(i)$. The selection of u disables the selection of any of u's children. Therefore, we need to look for an independent set in the subtree rooted at each of u's grandchildren. Letting S_i be a maximum independent set in T_i, Proposition 6 implies that $\bigcup_{i \in \mathcal{GC}(u)} S_i$ is an independent set. Since vertex u is not adjacent to any vertex in this independent set, $\{u\} \cup \bigcup_{i \in \mathcal{GC}(u)} S_i$ is also an independent set, which has size $1 + \sum_{i \in \mathcal{GC}(u)} F(i)$. In addition, this must be a maximum independent set in T_u subject to that u is selected: for any independent set S_u in T_u with $u \in S_u$, we have $|S_u| = 1 + \sum_{i \in \mathcal{GC}(u)} |S_u \cap V(T_i)|$, and $S_u \cap V(T_i)$ is an independent set in T_i (where $V(T_i)$ is the set of all vertices in the subtree T_i); since S_i is a maximum independent set in T_i, we have $|S_u \cap V(T_i)| \le |S_i| = F(i)$. Thus, $|S_u| \le 1 + \sum_{i \in \mathcal{GC}(u)} F(i)$, indicating that $\{u\} \cup \bigcup_{i \in \mathcal{GC}(u)} S_i$ is a maximum independent set (subject to that u is selected).
For case 2), we have $F(u) = \sum_{i \in \mathcal{C}(u)} F(i)$: letting S_i be the maximum independent set in T_i, Proposition 6 implies that $\bigcup_{i \in \mathcal{C}(u)} S_i$ is an independent set, which has size $\sum_{i \in \mathcal{C}(u)} F(i)$; it is also easy to see this is a maximum independent set in T_u subject to that u is not selected (by the same arguments above).
Putting together, we have the recurrence relation

$$F(u) = \max\left\{1 + \sum_{i \in \mathcal{GC}(u)} F(i), \sum_{i \in \mathcal{C}(u)} F(i)\right\}.$$

A natural topological order for implementing this dynamic program is the bottom-up order: we first find $F(u)$ for all the leafs, then we go to the second last level, and so on. We leave the remaining details for an exercise. The time complexity of this algorithm is $O(n)$: for each vertex u, the value $F(u)$ is used at most twice throughout the algorithm, one for u's parent, and one for u's grandparent.

4.1 A Greedy Algorithm

Problem 5 can also be solved by the following simple greedy algorithm: initialize $S \leftarrow \emptyset$; iteratively add all the leafs of G to S and remove all vertices in G that are in S or adjacent to S, until all vertices in G are removed. This algorithm also runs in $O(n)$ time, as each vertex is removed exactly once.

Exercise 7. Prove that the greedy algorithm always outputs a maximum independent set.

4.2 On Graphs That Are Close to Trees, Treewidth

We have seen that the NP-hard problem of finding a maximum independent set is polynomial time solvable on trees. What if the graphs we are dealing with are "very close to" trees? For example, the graph shown in Figure 2 is very close to a tree: if we remove the edge $\{G, H\}$, it becomes a tree.

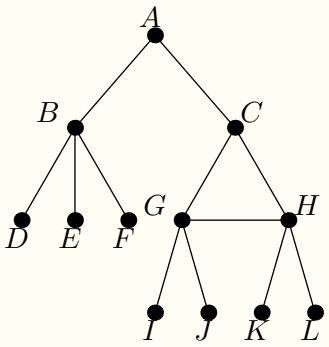

Figure 2: A graph that is close to a tree

A trick here is to contract the triangle $\{C, G, H\}$ to a single super-vertex, say, vertex S, so that S is a child of A and S is the parent of vertices I, J, K, L. In the case of independent set, there can be four different states for vertex S: none of C, G, H is selected, C is selected, G is selected, and H is selected. When we are dealing with the recurrence relation for $F(S)$, we merely need to discuss four cases. Instead of having I, J, K, L as four leafs that are children of G and H as it is in Figure 2, in the analysis below, to gain more generality, we assume that G and H are roots of two giant subtrees. It is easy to verify that

$$F(S) = \begin{cases} \sum_{i \in \mathcal{C}(G)} F(i) + \sum_{i \in \mathcal{C}(H)} F(i) & \text{if none of } C, G, H \text{ is selected} \\ 1 + \sum_{i \in \mathcal{C}(G)} F(i) + \sum_{i \in \mathcal{C}(H)} F(i) & \text{if } C \text{ is selected} \\ 1 + \sum_{i \in \mathcal{GC}(G)} F(i) + \sum_{i \in \mathcal{C}(H)} F(i) & \text{if } G \text{ is selected} \\ 1 + \sum_{i \in \mathcal{C}(G)} F(i) + \sum_{i \in \mathcal{GC}(H)} F(i) & \text{if } H \text{ is selected} \end{cases}.$$

By removing the first case from our consideration (notice that the first case is dominated by the second case), we have

$$F(S) = \max\left\{1 + \sum_{i \in \mathcal{C}(G)} F(i) + \sum_{i \in \mathcal{C}(H)} F(i), 1 + \sum_{i \in \mathcal{GC}(G)} F(i) + \sum_{i \in \mathcal{C}(H)} F(i), 1 + \sum_{i \in \mathcal{C}(G)} F(i) + \sum_{i \in \mathcal{GC}(H)} F(i)\right\}.$$

In general, when working on problems with graphs, if the graph is close enough to a tree such that we can contract certain groups of vertices to super-vertices, we can still apply dynamic programming. In particular, if each super-vertex corresponds to a bounded constant number of original vertices, the number of possible states for each super-vertex is also bounded by a constant. Hence, the dynamic programming algorithm is still efficient.

The *treewidth* of a graph measures how much the graph looks like a tree. The graph is converted to a tree where each vertex in the tree corresponds to a bundle of vertices in the original graph, subject to some conditions which are not discussed in this lecture. The treewidth is then the number of vertices in the largest bundle, minus 1. We are not going into the formal definition of treewidth. We will instead give some examples.

Examples for graphs with small treewidth:

- any tree has treewidth 1, and a graph with treewidth 1 is a tree;
- any cycle has treewidth 2;
- the graph in Fig. 2 has treewidth 2;
- any *series-parallel graph* has treewidth 2; in fact, a graph has treewidth at most 2 if and only if all of its *biconnected components* are series-parallel graphs (search on the Internet for those italic words if you are interested in this).

Examples for graphs with large treewidth:

- a complete graph with n vertices has treewidth $n-1$;
- a $n \times n$ *grid* has treewidth n.

To work on problems (especially NP-hard problems) on graphs that are known to have a small treewidth, we can convert the graph to a tree and apply dynamic programming as discussed above.

5 Color Coding and Finding Simple Paths of Length $k-1$

Definition 8. Given a graph, a *simple path* is a path that visits no vertex more than once.

Problem 9. Given an undirected graph $G = (V, E)$ and a positive integer k, decide if G contains a simple path of length $k-1$.

This problem is NP-hard, which can be proved by a simple reduction from the Hamiltonian path problem. A brute-force search algorithm requires $O(n^k)$ time. We will see this problem can be solved in $2^{O(k)} \cdot (n+m)$ time, which is much faster when k is small.

We first define a problem of finding a colorful path of length k where each vertex is colored by a set of colors $\{1, \ldots, k\}$. We show that the problem can be solved by dynamic programming. Then we use the solution for this problem to solve Problem 9.

5.1 Finding Simple Colorful Paths of Length $k-1$

Problem 10. Given an undirected graph $G = (V, E)$ and assign a color from the set of k colors $\{1, \ldots, k\}$ to each vertex, letting $c : V \to \{1, \ldots, k\}$ be the color assignment, decide if the graph contains a simple path of length $k-1$ such that the k vertices on the path are assigned distinct colors.

We say that a path satisfying the description in the problem above is *k-colorful*.
This problem can be solved by dynamic programming. Given $S \subseteq \{1, \ldots, k\}$ and $u \in V$, let $F(S, u)$ be the Boolean value which is **true** if and only if there is a path of length $|S| - 1$ such that

- u is one endpoint of the path, and
- the colors of the $|S|$ vertices on the path are from S, and all the $|S|$ vertices on the path have distinct colors.

For the initial conditions, we have $F(\{c(u)\}, u) =$ **true** and $F(S, u) =$ **false** if $u \notin S$. We would like to output $\bigvee_{u \in V} F(\{1, \ldots, k\}, u)$. The recurrence relation can be easily found by looking at u's neighbors:

$$F(S, u) = \begin{cases} \textbf{false} & \text{if } c(u) \notin S \\ \bigvee_{v:(v,u)\in E} F(S \setminus \{c(u)\}, v) & \text{otherwise} \end{cases}.$$

A valid topological order is by the ascending order of $|S|$, since $F(S, u)$ is related to some $F(S', v)$ with $|S'| = |S| - 1 < |S|$. To analyze the time complexity, for each fixed S, we need to find $F(S, u)$ for each vertex u, and we need to search for all of u's neighbors when working on $F(S, u)$. Therefore, for each fixed S, each vertex is visited once, and each edge is visited twice. Since we need to enumerate $2^k - 1$ possible S, the overall time complexity is $O\left(2^k(n + m)\right)$.

Exercise 11. Suppose we want to find the *number* of colorful paths of length k. Adapt the dynamic programming algorithm above to solve this problem.

5.2 Solving Problem 9

After solving Problem 10, Problem 9 can be solved by a simple randomized algorithm. The algorithm repeatedly finds a random color assignment to G and see if there is a k-colorful path using the algorithm in the previous sub-section. If a k-colorful path is found, the algorithm outputs that G contains a simple path of length $k-1$. If no k-colorful path has been found after sufficiently many repetitions, the algorithm outputs that G does not contain a simple path of length $k-1$.
If the truth is that G does not contain a simple path of length $k-1$, the algorithm always outputs the correct answer. Otherwise, there is a positive probability that the algorithm outputs the wrong answer, which happens when the color assignment in each repetition does not make the length-k simple path colorful. However, this probability becomes small after sufficiently many repetitions. It then remains to decide how many repetitions are required to reduce the error probability to an acceptably small value.

Suppose the graph contains a simple path of length $k-1$. A random color assignment will assign k distinct colors to the k vertices on the path with probability $\frac{k!}{k^k} > e^{-k}$. Thus, with probability more than e^{-k}, the algorithm outputs the correct answer for one repetition. If we set the number of repetition to, say, e^{2k}, the error probability is below

$$\left(1 - e^{-k}\right)^{e^{2k}} \leq \left(e^{-e^{-k}}\right)^{e^{2k}} = e^{-e^{k}},$$

which is already extremely small (we have used the inequality $1 + x \leq e^{x}$).
The overall time complexity for this algorithm is $e^{2k} \cdot O\left(2^k(n+m)\right) = 2^{O(k)} \cdot (n+m)$.
This algorithm can even be derandomized, by using a technique called *universal Hashing*.

5.3 Fixed-Parameter Tractable

Given a problem where n denotes the input length, and a parameter k of this problem, if there is an algorithm that solves this problem with time complexity $O(f(k) \cdot n^c)$ for some constant c and some functions $f(\cdot)$, we say that the problem is *fixed-parameter tractable* with respect to the parameter k (here, $f(\cdot)$ do not need to be polynomially bounded). Notice that fixed-parameter tractability only makes sense after specifying a parameter. In this section, we have seen that the problem of deciding if a graph contains a simple path of length $k-1$ is fixed-parameter tractable with respect to the parameter k. In the previous section, we see that most graph problems are fixed-parameter tractable with respect to the treewidth.