بسم اللّه الرّحمن الرّحیم

دانشگاه صنعتی اصفهان ــ دانشکدهٔ مهندسی برق و کامپیوتر
(نیم‌سال تحصیلی ۲۰۲۲)

طراحی الگوریتم‌ها

حسین فلسفین

Single-source shortest-paths problem

Given a graph $G = (V, E)$, we want to find a shortest path from a given source vertex $s \in V$ to each vertex $v \in V$. The algorithm for the single-source problem can solve many other problems, including the following variants:

1. *Single-destination shortest-paths problem: Find a shortest path to a given destination vertex t from each vertex v. By reversing the direction of each edge in the graph, we can reduce this problem to a single-source problem.*

2. *Single-pair shortest-path problem: Find a shortest path from u to v for given vertices u and v. If we solve the single-source problem with source vertex u, we solve this problem also. Moreover, all known algorithms for this problem have the same worst-case asymptotic running time as the best single-source algorithms.*

3. *All-pairs shortest-paths problem:* *Find a shortest path from u to v for every pair of vertices u and v. Although we can solve this problem by running a single-source algorithm once from each vertex, we usually can solve it faster (با الگوریتم‌های فصل قبل).*

Dijkstra's algorithm solves the single-source shortest-paths problem on a weighted, directed graph $G = (V, E)$ for the case in which all edge weights are nonnegative. We assume that $w(u, v) \geq 0$ for each edge $(u, v) \in E$.

We often wish to compute not only shortest-path weights, but the vertices on shortest paths as well. Given a graph $G = (V, E)$, we maintain for each vertex $v \in V$ a predecessor $v.\pi$ that is either another vertex or NIL. Dijkstra's algorithm sets the π attributes so that the chain of predecessors originating at a vertex v runs backwards along a shortest path from s to v. Thus, given a vertex v for which $v.\pi \neq NIL$, the procedure $\mathrm{PRINT-PATH}(G, s, v)$ will print a shortest path from s to v.

```
PRINT-PATH(G, s, v)
if v == s
    print s
elseif v.π == NIL
    print "no path from" s "to" v "exists"
else PRINT-PATH(G, s, v.π)
    print v
```

Relaxation

Dijkstra's algorithm uses the technique of relaxation. For each vertex $v \in V$, we maintain an attribute $v.d$, which is an upper bound on the weight of a shortest path from source s to v. We call $v.d$ a shortest-path estimate. We initialize the shortest-path estimates and predecessors by the following $\Theta(|V|)$-time procedure:

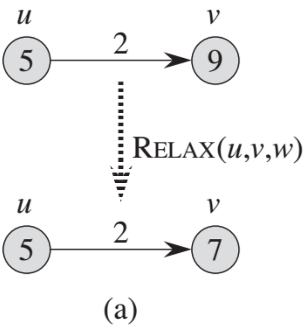
INITIALIZE-SINGLE-SOURCE(G, s)

```
1  for each vertex v ∈ G.V
2      v.d = ∞
3      v.π = NIL
4  s.d = 0
```

After initialization, we have $v.\pi = NIL$ for all $v \in V$, $s.d = 0$, and $v.d = \infty$ for $v \in V - \{s\}$.

The process of relaxing an edge (u, v) consists of testing whether we can improve the shortest path to v found so far by going through u and, if so, updating $v.d$ and $v.\pi$. A relaxation step may decrease the value of the shortest-path estimate $v.d$ and update v's predecessor attribute $v.\pi$. The following code performs a relaxation step on edge (u, v) in $O(1)$ time:

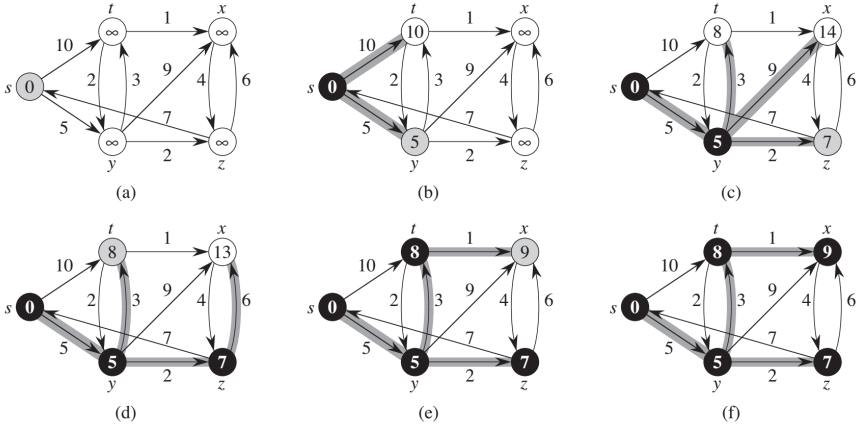
```
RELAX(u, v, w)
1  if v.d > u.d + w(u, v)
2       v.d = u.d + w(u, v)
3       v.π = u
```



Relaxing an edge** (u, v) **with weight** $w(u, v) = 2$. **The shortest-path estimate of each vertex appears within the vertex. (a) Because** $v.d > u.d + w(u, v)$ **prior to relaxation, the value of** $v.d$ **decreases. (b) Here,** $v.d \leq u.d + w(u, v)$ **before relaxing the edge, and so the relaxation step leaves** $v.d$ **unchanged.

Dijkstra's algorithm maintains a set S of vertices whose final shortest-path weights from the source s have already been determined. The algorithm repeatedly selects the vertex $u \in V - S$ with the minimum shortest-path estimate, adds u to S, and relaxes all edges leaving u. In the following implementation, we use a min-priority queue Q of vertices, keyed by their d values.

The execution of Dijkstra's algorithm: *The source s is the leftmost vertex. The shortest-path estimates appear within the vertices, and shaded edges indicate predecessor values. Black vertices are in the set S, and white vertices are in the min-priority queue $Q = V - S$.*



```
DIJKSTRA(G, w, s)
INITIALIZE-SINGLE-SOURCE(G, s)
S = ∅
Q = G.V
while Q ≠ ∅
    u = EXTRACT-MIN(Q)
    S = S ∪ {u}
    for each vertex v ∈ G.Adj[u]
        RELAX(u, v, w)
```

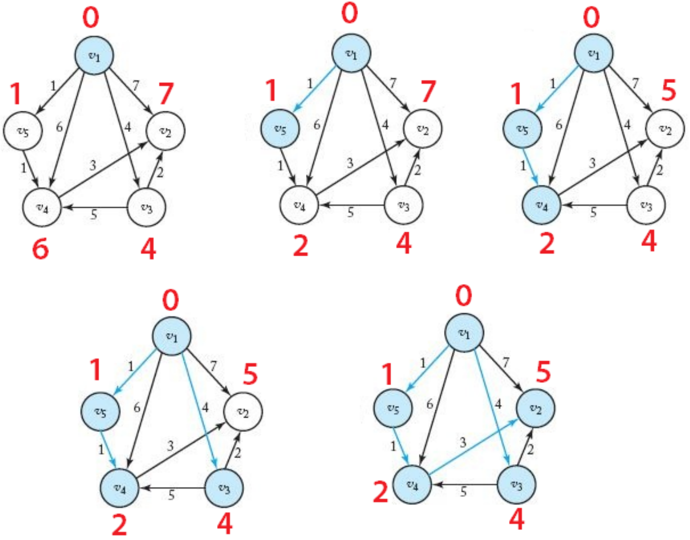
Because Dijkstra's algorithm always chooses the "lightest" or "closest" vertex in $V - S$ to add to set S, we say that it uses a greedy strategy.

Line 1 initializes the d and π values in the usual way, and line 2 initializes the set S to the empty set. Line 3 initializes the min-priority queue Q to contain all the vertices in V. Each time through the while loop of lines 4–8, line 5 extracts a vertex u from $Q = V - S$ and line 6 adds it to set S. (The first time through this loop, $u = s$.) Then, lines 7–8 relax each edge (u, v) leaving u, thus updating the estimate $v.d$ and the predecessor $v.\pi$ if we can improve the shortest path to v found so far by going through u. Observe that the algorithm never inserts vertices into Q after line 3 and that each vertex is extracted from Q and added to S exactly once, so that the while loop of lines 4–8 iterates exactly $|V|$ times.

Greedy strategies do not always yield optimal results in general, but it can be shown that Dijkstra's algorithm does indeed compute shortest paths. It can be shown that each time it adds a vertex u to set S, we have

$$u.d = \text{The length of the shortest path from } s \text{ to } u.$$

Dijkstra's algorithm calls INITIALIZE-SINGLE-SOURCE and then repeatedly relaxes edges. Moreover, relaxation is the only means by which shortest path estimates and predecessors change.



در کدام شکل ریلکس کردن یال بی اثر است؟

The set-covering problem

An instance $(X, \mathcal{F})$ of the set-covering problem consists of a finite set X and a family $\mathcal{F}$ of subsets of X, such that every element of X belongs to at least one subset in $\mathcal{F}$:

$$X = \bigcup_{S \in \mathcal{F}} S.$$

We say that a subset $S \in \mathcal{F}$ covers its elements. The problem is to find a minimum size subset $\mathcal{C} \subseteq \mathcal{F}$ whose members cover all of X:

$$X = \bigcup_{S \in \mathcal{C}} S.$$

We say that any $\mathcal{C}$ satisfying the above equation covers X.

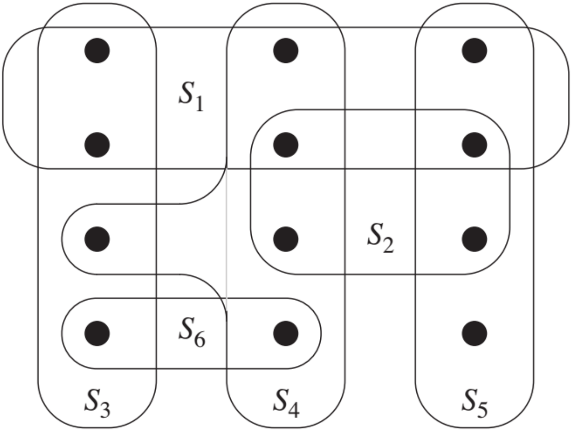
The set-covering problem abstracts many commonly arising combinatorial problems. As a simple example, suppose that X represents a set of skills that are needed to solve a problem and that we have a given set of people available to work on the problem. We wish to form a committee, containing as few people as possible, such that for every requisite skill in X, at least one member of the committee has that skill.

The set-covering problem is NP-hard.

یک الگوریتم تقریبی مبتنی بر راهبرد حریصانه

GREEDY-SET-COVER$(X, \mathcal{F})$

$U = X$
$\mathcal{C} = \emptyset$
while $U \neq \emptyset$
    select an $S \in \mathcal{F}$ that maximizes $|S \cap U|$
    $U = U - S$
    $\mathcal{C} = \mathcal{C} \cup \{S\}$
return $\mathcal{C}$



The greedy algorithm: $\{S_1, S_4, S_5, S_6\}$ ***or*** $\{S_1, S_4, S_5, S_3\}$
A minimum-size set cover: $\{S_3, S_4, S_5\}$

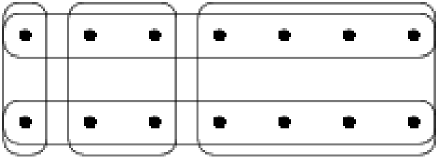
☞ GREEDY-SET-COVER is a polynomial-time $\rho(n)$-approximation algorithm, where $\rho(n) = H(\max\{|S| : S \in \mathcal{F}\})$ and $H(d) = \sum_{i=1}^{d} \frac{1}{i}$.

☞ GREEDY-SET-COVER is a polynomial-time $(\ln |X| + 1)$-approximation algorithm.

. . . .

There is a standard example on which the greedy algorithm achieves an approximation ratio of $\log_2(n)/2$. The set X consists of $n = 2^{k+1} - 2$ elements. The set $\mathcal{F}$ consists of k pairwise disjoint sets $S_1, S_2, \ldots, S_k$ with sizes $2, 4, 8, \ldots, 2^k$ respectively, as well as two additional disjoint sets T_0 and T_1, each of which contains half of the elements from each S_i. On this input, the greedy algorithm takes the sets $S_k, S_{k-1}, \ldots, S_1$, in that order, while the optimal solution consists only of T_0 and T_1.

An example of such an input for $k = 3$:



The graph coloring problem

☞ A vertex coloring of G is a map $f : V(G) \mapsto S$, where S is a set of distinct colors; it is proper if adjacent vertices of G receive distinct colors of S. This means that if $uv \in E(G)$, then $f(u) \neq f(v)$.

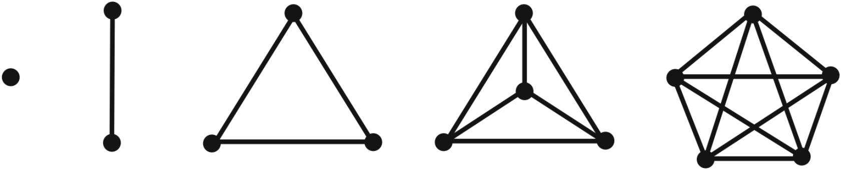
☞ The chromatic number $\chi(G)$ of a graph G is the minimum number of colors needed for a proper vertex coloring of G. G is k-chromatic if $\chi(G) = k$.

☞ A k-coloring of a graph G is a vertex coloring of G that uses at most k colors.

☞ A graph G is said to be k-colorable if G admits a proper vertex coloring using at most k colors.

یادآوری

A simple graph G is said to be complete if every pair of distinct vertices of G are adjacent in G. If all the vertices of G are pairwise adjacent, then G is complete. A complete graph on n vertices is a K_n.

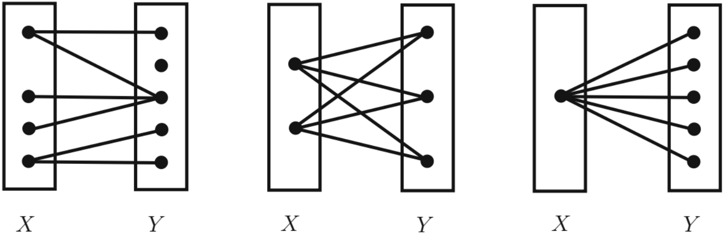


K_1 K_2 K_3 K_4 K_5

یادآوری

A graph is bipartite if its vertex set can be partitioned into two nonempty subsets X and Y such that each edge of G has one end in X and the other in Y. The pair (X, Y) is called a bipartition of the bipartite graph. The bipartite graph G with bipartition (X, Y) is denoted by $G(X, Y)$.

A simple bipartite graph $G(X, Y)$ is complete if each vertex of X is adjacent to all the vertices of Y. If $G(X, Y)$ is complete with $|X| = p$ and $|Y| = q$, then $G(X, Y)$ is denoted by $K_{p,q}$. A complete bipartite graph of the form $K_{1,q}$ is called a star.



A bipartite graph | The graph $K_{2,3}$ | The star graph $K_{1,5}$

It is clear that $\chi(K_n) = n$. Further, $\chi(G) = 2$ if and only if G is bipartite having at least one edge. In particular, $\chi(T) = 2$ for any tree T with at least one edge (since any tree is bipartite).

$$\chi(C_n) = \begin{cases} 2, & \text{if } n \text{ is even}, \\ 3, & \text{if } n \text{ is odd}. \end{cases}$$

(A cycle of length k is denoted by C_k.)

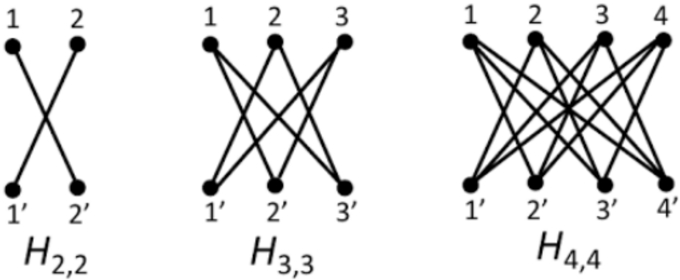
The graph coloring problem is to find $\chi(G)$ as well as the partition of vertices induced by a $\chi(G)$-coloring. The graph coloring problem is NP-hard.

یک الگوریتم حریصانهٔ ساده

One obvious way to color a graph G with not too many colors is the following greedy algorithm: starting from a fixed vertex enumeration $v_1, v_2, \ldots, v_n$ of G, we consider the vertices in turn and color each v_i with the first available color—e.g., with the smallest positive integer not already used to color any neighbor of v_i among $v_1, v_2, \ldots, v_{i-1}$.

اگر گراف کامل یا دور فرد باشد، الگوریتم حریصانه فوق جواب بهینه را برمی‌گرداند.

حال گراف کامل دوبخشی $K_{n,n}$ که رئوس واقع در یکی از بخش‌های آن $x_1, x_2, \dots, x_n$، و رئوس واقع در بخش دیگر $y_1, y_2, \dots, y_n$ هستند را درنظر گرفته و فرض کنید که یال‌های $x_i y_i$ از مجموعهٔ یال‌های این گراف حذف شده‌اند. گراف حاصل را $H_{n,n}$ بنامید. (این گراف را معمولاً **crown graph** می‌نامند.)



حال (در گراف $H_{n,n}$)، اگر ترتیبی که برای رنگ آمیزی رئوس درنظر گرفته می‌شود به‌شکل

$$x_1, x_2, \ldots, x_n, y_1, y_2, \ldots, y_n$$

باشد، آنگاه تنها به ۲ رنگ برای رنگ‌آمیزی نیاز داریم (بهترین رنگ‌آمیزی ممکن)؛ اما اگر ترتیبی که برای رنگ‌آمیزی مدنظر قرار می‌گیرد به‌صورت

$$x_1, y_1, x_2, y_2, \ldots, x_n, y_n$$

باشد به n رنگ نیاز داریم. (چرا؟)

پس نسبت جواب حاصل از الگوریتم به جواب بهینه برابر با

$$r(s_a) = \frac{f(s_a)}{f(s^*)} = \frac{n}{2}$$

است که مطلوب نیست.