Scope:
- DPV Chapter 3, Sections 3.1 through 3.3
- DPV Chapter 4, Sections 4.1 through 4.5
- DPV Chapter 5, all sections
- DPV Chapter 6, all sections

Problem 1

(a) Devise an algorithm that, given an undirected graph $G = (V, E)$ and an $e \in E$, determines whether G has a cycle containing e. Your algorithm should run in $O(|V| + |E|)$-time.

(b) Explain why your algorithm is correct.

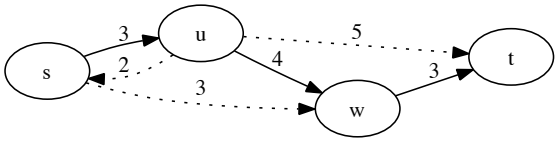
(c) Justify the $O(|V| + |E|)$ runtime.

Problem 2

BACKGROUND. Suppose

(i) $G = (V, E)$ is a directed graph,
(ii) $s, t \in V$,
(iii) $E' \subseteq V \times V$ is a set of edges with $E \cap E' = \emptyset$, and
(iv) each edge $e \in (E \cup E')$ is assigned length $\ell_e > 0$.

You can view V as a set of towns, E as a set of existing roads, and E' as a set of proposed roads.



EXAMPLE. Above, $E = \{ (s, u), (u, w), (w, t) \}$ and $E' = \{(u, s), (s, w), (u, t)\}$ where each edge is labeled with its length. The E'-edge that would most improve the travel distance from s to t is (s, w).

YOUR PROBLEM.

(a) Give a $O(|E'| + (|V| + |E|) \log |V|)$ time algorithm to find which edge $e' \in E'$ is such that if e' is added to E (i.e., road e' is actually build), then the travel distance from s to t decreases the most. If no edge in E' improves the travel distance from s to t, then return NONE.
*(**Hint:** Consider computing G^R, the reverse of G, and running Dijkstra's algorithm from t.)*

(b) Justify the run time of your algorithm.

Problem 3 BACKGROUND. A highway goes from mile-marker 0 to mile-marker 800 with gas stations at mile-markers $g[1], \ldots, g[k]$ where $0 = g[1] < g[2] < \cdots < g[k-1] < g[k] = 800$ and where $g[i] - g[i-1] < 200$ for $i = 2, \ldots, k$. Your car gets 20 miles/gallon and has a 10 gallon gas tank. Starting with a full tank of gas, you want to drive (without running out of gas) from mile-marker 0 to mile-marker 800 ***making as few stops as possible***. Here is an algorithm that finds a sequence of stops for the trip. *(Note that if we are at mile 600+ with a full tank of gas, no more stops are needed.)*

```
mile ← 0;  i ← 1;  L ← the empty list
while (mile < 600) do
    while (g[i] ≤ mile + 200) do i ← i + 1
    // Now, mile ≤ g[i − 1] ≤ mile + 200 < g[i].
    mile ← g[i − 1];  add (i − 1) to the end of L
return L
```

Let $L = [i_1, \ldots, i_m]$ be the list returned by the algorithm (where $i_1 < i_2 < \cdots < i_m$).

YOUR PROBLEMS.

(a) Explain why L is a list of stops that gets us to mile 800 without running out of gas. (I.e., L is a *feasible* list of stops.)

(b) Explain why the algorithm is greedy.

(c) Suppose $L_0 = [j_1, j_2, \ldots, j_\ell]$ (where $j_1 < j_2 < \cdots < j_\ell$) is an optimal list of stops (i.e., a list with as few stops as possible). Explain why $L_0' = [i_1, j_2, j_3, \ldots, j_\ell]$ is an optimal list. *(Q: Is $j_1 > i_1$ possible? Q: Is $j_2 - i_1 > 200$ possible? A picture may be helpful.)*

(d) Suppose L_0 is as in part (c), but we have that that $j_a = i_a$ for $a = 1, \ldots, b$, where $b < \ell$. Explain why $L_0'' = [i_1, i_2, i_3, \ldots, i_b, i_{b+1}, j_{b+2}, \ldots, j_\ell]$ is an optimal list. *(You can use part (c) here.)*

(e) Use part (d) to explain why L is an optimal list. (I.e., L and L_0 have the same number of stops, that is $m = \ell$.) *(Q: In an optimal list of stops, how many stops are there in the 600 to 800 mile-marker range?)*

Problem 4 BACKGROUND. In the game Crawl4Cash there is an $n \times n$ grid of squares with co-ordinates (x, y) for $1 \leq x, y \leq n$. You have a pawn that begins at square (1,1) and moves according to the rules:

Rule 1. A pawn on (x, y) with $x, y < n$, must move to one of $(x + 1, y)$, $(x, y + 1)$, and $(x + 1, y + 1)$.
Rule 2. When the pawn is on square (x, y) with $x = n$ or $y = n$, the game is over.

When a pawn moves on to (x, y), you receive $p(x, y)$ many dollars. (You receive no dollars from starting at (1,1).) We want a dynamic programming algorithm that computes the maximum possible winnings for this game.

YOUR PROBLEMS.

(a) What are the subproblems to solve?

(b) What is the recursive equation for this problem?

(c) Provide pseudo-code for the algorithm.

(d) What is the run-time your algorithm? Justify this run-time.

Some Reference Math Facts

a. $a^m \cdot a^n = a^{m+n}$.

b. $a^{m \cdot n} = (a^m)^n = (a^n)^m$.

c. $a^n \cdot b^n = (a \cdot b)^n$.

d. $\log_a a^n = n$.

e. $a^{\log_a n} = n$.

f. $c \cdot \log_a b = \log_a b^c$.

g. $\log_a(b \cdot c) = (\log_a b) + (\log_a c)$.

h. $\log_a x = (\log_a b) \cdot (\log_b x)$.

i. $a^{\log_b c} = c^{\log_b a}$.

j. $\sum_{k=1}^{n} k = \frac{n \cdot (n+1)}{2}$.

k. $\sum_{k=0}^{n} a^k = \frac{a^{n+1}-1}{a-1}$.

l. $\sum_{k=1}^{\infty} 2^{-k} = 1$.

m. $\sum_{k=1}^{n} k^2 = \frac{n \cdot (n+1) \cdot (2n+1)}{6}$.

n. $\sum_{k=1}^{\infty} k \cdot 2^{-k} = 2$.

o. For $a > 1$ and $b, c > 0$:

$\lim_{n\to\infty} \frac{(\log_a n)^b}{n^c} = 0$.

$\lim_{n\to\infty} \frac{n^c}{(\log_a n)^b} = \infty$.

p. For $a > 1$ and $b, c > 0$:

$\lim_{n\to\infty} \frac{n^b}{a^{c \cdot n}} = 0$.

$\lim_{n\to\infty} \frac{a^{c \cdot n}}{n^b} = \infty$.

q. If $\lim_{n\to\infty} \frac{f(n)}{g(n)} = c$, then:

	$c = 0$	$0 < c < \infty$	$c = \infty$
$f(n) \in O(g(n))$	True	True	False
$f(n) \in \Theta(g(n))$	False	True	False
$f(n) \in \Omega(g(n))$	False	True	True

Dijkstra's Algorithm. GIVEN: $G = (\{1, \ldots, n\}, E)$ and a function len that, for each $u, v \in \{1, \ldots, n\}$:

(i) if $(u, v) \in E$, then $len(u, v) > 0$, and
(ii) if $(u, v) \notin E$, then $len(u, v) = \infty$.

GOAL: For each $i \in \{2, \ldots, n\}$, compute $dist[i]$ = the length of a *shortest* path from 1 to i.

```
function Dijkstra(G, ℓ)
  array dist[1..n]
  ToDo ← {2, ..., n}
  for i ← 2 to n do dist[i] ← len(1, i)
  for i ← 1 to n − 1 do
    choose v ∈ ToDo with minimal dist[v]
    ToDo ← ToDo − {v}
    for each w adjacent to v do
      dist[w] ← min(dist[w], dist[v] + len(v, w))
  return dist
```

A Version of Depth-First Search. $G = (V, E)$ is assumed to be represented by adjacency lists and $V = \{1, \ldots, n\}$. Also, for each vertex u: $parent$[u] ends up with u's parent in the DFS tree and d[u] ends up with u's discovery time.

Global vars: $visited[1..n]$, $parent[1..n]$, $d[1..n]$, $time$

```
procedure dfsInit(G)
  time ← 0
  for u ← 1, ..., n do
    visited[u] ← false;   parent[u] ← nil;   d[u] ← −1
function dfs(G)
  dfsInit(G)
  for u ← 1, ..., n do
    if not (visited[u]) then dfsVisit(u)
  return parent[1..n]
procedure dfsVisit(u)
  visited[u] ← true
  d[u] ← time
  time ← time + 1
  for each v adjacent to u do
    if not(visited[v]) then
      parent[v] ← u;   dfsVisit(v)
```

A Version of Breadth-First Search. $G = (V, E)$ is assumed to be represented by adjacency lists and $V = \{1, \ldots, n\}$. Vertex $s \in V$ is the start of the breadth-first search. For each vertex u: $parent$[u] ends up with u's parent in the BFS tree.

```
function bfs(G,s)
  for u ← 1, ..., n do parent[u] ← nil
  Q ← an empty queue
  enqueue(Q, s);
  while Q is not empty do
    v ← dequeue(Q)
    for each w adjacent to v do
      if parent[w] = nil then
        parent[w] ← v;   enqueue(Q, w)
  return parent[1..n]
```

Answers

Answers for Problem 1

(a) Here is the algorithm:

1. Input $G = (V, E)$ and $(u, v) \in E$.
2. Let $G' = (V, E - \{ (u, v) \})$, i.e., a version of G with edge (u, v) removed.
3. Run $dfsInit(G')$ and then $dfsVisit(G', u)$.
4. **return** $visited[v]$

(b) (u, v) is on a cycle in G if and only if there is a path in G from u to v that does not involve the edge (u, v). If there is such a path, the $dfsVisit(G', u)$ will visit v.

(c) Removing (u, v) from G, running $dfsInit(G')$ and $dfsVisit(G', u)$ all take at worst $O(|V| + |E|)$ time.

Answers for Problem 2.

(a) Suppose $E' = \{ (u_1, v_1), \ldots, (u_k, v_k) \}$.

1. Run Dijkstra's algorithm on G from s to compute $dist[v]$ for each $v \in (V - \{ s \})$
2. Compute G^R
3. Dijkstra's algorithm on G^R from t to compute $dist'[v]$ for each $v \in (V - \{ t \})$
4. $m \leftarrow dist[t]$; $\quad i_{min} \leftarrow 0$
5. **for** $i \leftarrow 1$ **to** k **do**
6. $\quad d \leftarrow dist[u_i] + \ell_{(u_i, v_i)} + dist'[v_i]$
7. $\quad$ **if** $(d < m)$ **then** $m \leftarrow d$; $\quad i_{min} \leftarrow i$
8. **if** $(i_{min} = 0)$ **then return** NONE **else return** $(u_{i_{min}}, v_{i_{min}})$

(b) Computing G^R takes $O(|V| + |E|)$ time. The two runs of Dijkstra's algorithm take $O((|V| + |E|) \log |E|)$ time. The final for-loop is $O(|E'|)$ time. Hence the total is $O(|E'| + (|V| + |E|) \log |E|)$.

t

Answers for Problem 3

(a) Starting at mile 0, the algorithm picks the furthest station it can reach without running out of gas (by assumption there is always such a station), updates the mile marker to this station's mile and repeats the process. Clearly, this will get us to mile 800.

(b) The algorithm always picks the furthest station it can reach from the current position without running out of gas.

(c) Since i_1 is the last gas station you can reach from mile 0 without running out of gas, we must have $g[j_1] \leq g[i_1]$. Since $g[j_2] - g[j_1] \leq 200$ and since $g[j_1] \leq g[i_1]$, it follows that $g[j_2] - g[i_1] \leq 200$. Hence, L_0' is feasible. Since L_0' has the same number of stops as L_0, L_0' is optimal.

(d) This is the same question as **c**, except that all the routes are leaving from stop i_b instead of stop 1.

(e) Clearly, any list feasible stops must include at least one in the 600 to 800 mile range, and as noted above, only one such stop is required. So optimal lists of stops have exactly one stop in the 600 to 800 mile range. So, let $L_0 = [j_1, \ldots, j_\ell]$ be an optimal list. By c+d, we can replace $j_1, j_2, \ldots, j_\ell$ with $i_1, i_2, \ldots, i_\ell$ and the result is still an optimal list. By the arguments of c+d we also have that $j_1 \leq i_1$, $j_2 \leq i_2$, $\ldots$, $j_\ell \leq i_\ell$. Since j_ℓ is in the 600 to 800 range, so must i_ℓ. Hence, i_ℓ must be the end of the L list, i.e., $m = \ell$. But since $m = \ell$, L must be optimal.

Answers for Problem 4.

(a) $C[x, y] =$ the maximum possible winnings if we start on square (x, y), where $x, y \in \{1, \ldots, n\}$.

(b) $C[x, y] = \max\{p(k, \ell) + C[k, \ell] \ :$ for $(k, \ell) = (x + 1, y)$, $(x, y + 1)$, $(x + 1, y + 1)\}$, for $x, y < n$.

(c) Here is the algorithm.

```
function c4c(p)
    for k ← 1, 2, ..., n do C[k, n] ← 0;  C[n, k] ← 0
    for x ← n − 1, n − 2, ..., 1 do
        for t ← n − 1, n − 2, ..., 1 do
            C[x, y] ← max{ as in part (b) }
    return C[1, 1]
```

(d) The max is constant time and the algorithm fills up each entry in a $n \times n$ array. So the run time is $\Theta(n^2)$.