CSC373H1

Weidong A.
Eric B.
Rohan D.
Felix Z.

Assignment 1

February 24, 2015

CSC373 Assignment 1 Eric Bannatyne
24 February 2015 1000468451

The following algorithm updates the given minimum spanning tree T of G, to produce a new minimum spanning tree T_1 for G_1.

```
UPDATE-MST($V, E, w, T, e_1, w_1$)
 1  $T_1 = T \cup \{e_1\}$
 2  $D$ = DFS tree produced by DFS on $T_1$ starting from $u$, including information about back edges // CLRS p. 610
 3  $e$ = NIL
 4  $weight = 0$
 5  // Find the (unique) back edge of $D$.
 6  // This must have $u$ as an endpoint since $e_1$ is in the cycle and DFS was started at $u$.
 7  for $x$ in $u.neighbours$ // Neighbours in $T_1$
 8      if $\{x, u\}$ is a back edge of $D$
 9          $e = \{x, u\}$
10          $weight = w(e)$
11          break
12  // Traverse up along the cycle in the DFS tree until the root $u$ is reached,
13  // keeping track of the maximum-weight edge.
14  while $x \neq u$
15      if $w(\{x, x.parent\}) > weight$ // $x.parent$ in $D$
16          $e = \{x, x.parent\}$
17          $weight = w(e)$
18      $x = x.parent$
19  $T_1 = T_1 - \{e\}$
20  return $T_1$
```

Correctness

On a high level, this algorithm updates T by inserting the new edge $e_1 = \{u, v\}$ into T. This produces exactly one cycle in the graph $T \cup \{e_1\}$ (Result given on Piazza). The algorithm then finds and removes the maximum-weight edge e from the cycle, to produce a new tree $T_1 = T \cup \{e_1\} - \{e\}$. This is, in fact, a spanning tree, since the removed edge e is on a cycle, meaning that neither of the endpoints of e become isolated vertices when e is removed. We will show that T_1 is in fact a minimum spanning tree.

By definition, we have $w(T_1) = w(T \cup \{e_1\} - \{e\}) = w(T) + w(e_1) - w(e)$. However, since e is a maximum-weight vertex on its cycle in $T \cup \{e_1\}$ and e_1 lies on that cycle, we have $w(e_1) \leq w(e)$, which implies that $w(T) \geq w(T_1)$.

To show that the spanning tree that this algorithm produces is indeed a minimum spanning tree of G_1, suppose that T_1 is not a MST. Then since G_1 is connected, there must be some MST T_1' for G_1 such that $w(T_1') < w(T_1)$. We have two cases to consider, depending on whether or not T_1' contains e_1.

If $e_1 \notin T_1'$, then T_1' must be a spanning tree for G, which means that $w(T) \leq w(T_1')$. However, since we established that $w(T_1) \leq w(T) \leq w(T_1')$, this contradicts our assumption that $w(T_1') < w(T_1)$. Therefore T_1 must also be a minimum spanning tree.

Now suppose that $e_1 \in T_1'$. Removing $e_1 = \{u, v\}$ from T_1' must disconnect the tree, such that $T_1' - \{e_1\}$ contains exactly two connected components $A = (V_A, E_A)$ and $B = (V_B, E_B)$, such that $u \in V_A$ and $v \in V_B$. Let C be the unique cycle contained in $T \cup \{e_1\}$. It will be helpful to prove the following lemma.

Lemma 1. *There is some edge $e' = \{a, b\} \in C - \{e_1\}$ such that $a \in V_A$ and $b \in V_B$.*

Proof. Since C is a cycle, $C - \{e_1\}$ must be a connected subgraph of T which is a chain of the form

$$u = w_1 \longleftrightarrow w_2 \longleftrightarrow \ldots \longleftrightarrow w_k = v,$$

where "$\longleftrightarrow$" denotes "is adjacent to (in $C - \{e_1\}$)". Since $V_A \cap C$ and $V_B \cap C$ form a partition of the vertices included in C, and we know that $u \in V_A$ and $v \in V_B$, there must be some i such that $w_i \in V_A$ and $w_{i+1} \in V_B$. Choosing $e' = \{w_i, w_{i+1}\}$ completes the proof. $\square$

This means that, if we remove e_1 from T_1', there must be some edge e' in $C - \{e_1\}$ such that $T_1' - \{e_1\} \cup \{e'\}$ is a spanning tree of G. Since $e' \in C$, we know that $w(e') \leq w(e)$, where e is the edge that the algorithm chose to remove from the cycle when producing T_1. Thus, we have

$$\begin{aligned} w(T) &\leq w(T_1') - w(e_1) + w(e') \\ &< w(T_1) - w(e_1) + w(e') \\ &\leq w(T_1) - w(e_1) + w(e) \\ &= w(T_1 - \{e_1\} \cup \{e\}) \\ &= w(T). \end{aligned}$$

Thus, $w(T) < w(T)$, which is a contradiction. Therefore T_1 must be a minimum spanning tree of G_1.

Running Time

We now analyze the running time of UPDATE-MST. Performing depth-first search to obtain the DFS tree D requires $\Theta(|V| + m)$ steps, where m is the number of edges in $T \cup \{e\}$. However, since T is a spanning tree of G, it contains $|V| - 1$ edges, so $m = |V|$, and so this step really only requires $\Theta(|V|)$ time.

The rest of the algorithm proceeds by examining the neighbours of u in $T \cup \{e_1\}$ to find a back edge, and then traversing a single cycle of $T \cup \{e_1\}$, both of which are bounded above by $O(|T|) = O(V|)$ operations, as before. Therefore UPDATE-MST runs in $\Theta(|V|)$ time in the worst case, an improvement over using the standard algorithms to produce a new minimum spanning tree from scratch.

Question 1b

Consider the following algorithm:

```
Update-MST($V, E, w, T, e_0, w_0$)
 1  if $e_0 \in T$
 2      // Suppose $e_0 = (u_0, v_0)$
 3      for $v \in V$
 4          Augment $v$ to keep track of a representative vertex.
 5      Run DFS on $T - \{e_0\}$ starting from $u_0$, where for each encountered vertex $u$, set $u$'s representative to $u_0$
 6      Run DFS on $T - \{e_0\}$ starting from $v_0$, where for each encountered vertex $v$, set $v$'s representative to $v_0$
 7      best-weight = $\infty$
 8      best-node = NIL
 9      for $e \in E - \{e_0\}$
10          // suppose $e = (u, v)$
11          if representative($u$) $\neq$ representative($v$) and $w(e) <$ best-weight
12              best-weight $\leftarrow w(e)$
13              best-node $\leftarrow e$
14      if best-weight = $\infty$
15          return NIL
16      else
17          return $T - \{e_0\} \cup$ best-node
18  else
19      return $T$
```

0.1 Runtime Analysis

If the removed edge e_0 is not in T, then T is still a valid MST for G, and the algorithm returns T which is $O(1)$. Now let's consider the case where $e_0 \in T$.

1. Augmenting all vertices in V to keep track of a representative involves allocating $O(n)$ memory, which is takes $O(n)$ time.

2. Since T is an MST, removing an edge e_0 from T creates two disjoint trees, say T_1 and T_2, such that $|T_1| + |T_2| = T - 1$. Then running DFS on T_1 and T_2 and setting representatives (which is possible to implement in constant time) on lines 7-8 has a total runtime of $O(n)$.

3. Each loop of the for loop starting on line 9 takes a constant number of operations (accessing the representatives, comparing values, and setting values), which means that the entire for loop is $O(m)$

Hence, the total worst case runtime is $O(n + m)$ which in practice is faster than $O(m \log^* n)$.

0.2 Proof of Correctness

We limit our attention to the case in which $e_0 \in T$.

If there is no edge in E which connects the disjoint trees created by the deletion of e_0 in T (V_1 and V_2 in the algorithm), then removing e_0 also causes a disconnection in G since by definition this means that the edges in E but not in T do not connect V_1 and V_2. In this case, it is impossible to construct a spanning tree, and so there is no solution, which is what is returned in the algorithm in this case (it returns NIL if it iterates through the edges and cannot find an edge connecting V_1 and V_2).

Removing e_0 from T yields two connected components C_1 and C_2, since T was a minimum spanning tree (property). The algorithm simply finds the minimum cost edge e between the two connected components, and adds this to $T - \{e_0\}$ as its result. Suppose, for the sake of contradiction, that the resulting tree $T_1 := T - \{e_0\} \cup \{e\}$ is not a minimum spanning tree. Then there must exist some other tree T' (which is minimum) such that $w(T') < w(T)$. Consider the following cases:

1. Case 1: $e \notin T'$
   Then there must be some other edge e' which connects a vertex from one of the edges in C_1, call it u', to a vertex from one of the edges in C_2, call it v' (otherwise there would be a disconnection and T' would not be spanning). Now, by adding e_0 and removing e' from T', we obtain a tree $T'' \in E$ such that $w(T'') = w(T') - w(e') + w(e_0)$. We know that $w(T') = w(T) - w(e_0) + w(e)$ and, since based on the algorithm e is cost edge connecting a vertex from one of the edges in C_1 to a vertex from one of the edges in C_2, we also know that $w(e) \leq w(e')$. Then $w(T'') = w(T) - w(e_0) + w(e) - w(e') + w(e_0)$ where $w(e) - w(e') \leq 0$, which means that $w(T'') \leq w(T)$, leading to a contradiction since T is an MST for G.

2. Case 2: $e \in T'$
   Then at least one of the subtrees in T' corresponding to C_1 and C_2 (i.e. the subtrees that contain the same vertices as in C_1 and C_2), denote them as C_1', C_2' have a weight less than its counterpart. Suppose without loss of generality that $w(C_1)' < w(C_1)$. Then $w(C_1' \cup C_2 \cup e_0) < w(C_1 \cup C_2 \cup e_0) = w(T)$ and we have a contradiction since this gives a tree $T'' \subseteq E$ with a weight lower than T which is an MST for G.

Since either case leads to a contradiction, there does not exist a tree T' where $w(T') < w(T)$. Hence, T is an MST.

Question 2

Solution. **Step 0**: Recursive substructure. In order to find the optimum path, we must find the maximum potential amount of gold we can get from drilling to each block. In order to do this for a given block (i, j), we need to look at the potential amount of gold we get for the 3 blocks above it from which we can then drill down to (i, j). The base case is the surface layer, where the potential amount of gold is just the amount of gold in the block if the hardness of the block is at most d, and 0 otherwise.
Step 1: Array definition.

For ease of notation, we define two $m \times n$ arrays A and D, so that $A[i, j]$ is the maximum amount of gold obtainable at location (i, j) in the mine, while $D[i, j]$ represents the amount of hardness of the drill remaining at position (i, j) by following the same path. If the remaining drill hardness is not enough to harness position (i, j) of the mine, we set $A[i, j] = D[i, j] = -\infty$. Notice that at the first row $(i = 1)$, the remaining drill hardness is just the original drill hardness d.

Step 2: Array recurrence.

$$D[1, j] = \begin{cases} d - H[1, j], & \text{if } H[1, j] \le d \\ -\infty, & \text{otherwise.} \end{cases}$$

$$A[1, j] = \begin{cases} G[1, j], & \text{if } H[i, j] \le d \\ -\infty, & \text{otherwise.} \end{cases}$$

For $i > 1$ and $1 \le j \le n$, define X as the set of *indices* such that $x \in X$ means $j - 1 \le x \le j + 1$ (it's a valid previous "x" co-ordinate) and $H[i, j] \le D[i - 1, x]$ (enough drill-hardness remained at that slot to mine the current square). If $X = \varnothing$, we define $D[i, j] = A[i, j] = -\infty$. Otherwise, set y as,

$$\operatorname*{argmax}_{x \in X} A[i - 1, x],$$

i.e. y is the "x" co-ordinate of the previous valid square at which we had the most gold. Then we define $D[i, j]$ as,

$$D[i, j] = D[i - 1, y] - H[i, j].$$

Similarly, we define $A[i, j]$ as,

$$A[i, j] = A[i - 1, y] + G[i, j].$$

Step 3: Iterative algorithm. This populates A, D (and S so we can reconstruct the optimum solution).

```
OPTIMUM-GOLD($G, H, d$):
let $A[1..m, 1..n]$ and $D[1..m, 1..n]$ be new $m \times n$ matrices. // the arrays defined above.
let $S[1..m, 1..n]$ be a new $m \times n$ matrix // stores how we got to $(i, j)$
for $i = 1$ to $m$:
    for $j = 1$ to $n$:
        $A[i, j] = -\infty$
        $D[i, j] = -\infty$
        if $i == 1$:
            if $H[i, j] \le d$: // sufficient drill hardness
                $A[i, j] = G[i, j]$
                $D[i, j] = d - H[i, j]$
                $S[i, j] = j$
        else :
            $best = -\infty$
            for $x = j - 1$ to $j + 1$: // possible parent slots
                if $1 \le x \le n$ and $A[i - 1, x] \ge best$ and $H[i, j] \le D[i - 1, x]$: // we can mine $(i, j)$
                    $best = A[i - 1, x]$
                    $A[i, j] = A[i - 1, x] + G[i, j]$
                    $D[i, j] = D[i - 1, x] - H[i, j]$
                    $S[i, j] = x$
return $(A, S)$
```

Step 4: Reconstructing the optimum solution.

To reconstruct an optimum solution, we simply need to first find the slot of the mine that yields the most total gold (while using at most the original drill hardness d). The maximum amount of gold is easily found, since we simply have to scan A for its largest value, and this also gives us the corresponding position (i, j) – the last element of the optimum path. Once we have found this position, we simply use S to go backwards (up the mine) and find the previous element of the path, until we reach the top of the mine, and this gives us the desired optimum path. The procedure RECONSTRUCT-OPTIMUM-PATH(A, S) implements precisely this idea.

```
RECONSTRUCT-OPTIMUM-PATH$(A, S)$:
$k = 0$ // length of the optimum path
$maxGold = 0$
$j' = 0$
for $i = 1$ to $m$:
    for $j = 1$ to $n$:
        if $A[i, j] > maxGold$:
            $maxGold = A[i, j]$
            $k = i$
            $j' = j$
let $J$ be a new $k$ element array // the entire $k$-element path
for $i = k$ to 1: // populating $J$ backwards
    $J[i] = j'$
    $j' = S[i, j']$ // the previous element of the path
return $J$
```

♦

CSC373 Winter 2015 Assignment # 1

Question 3 (Weidong An)
Algorithm: Suppose there are n CPOs(denoted by $c_1, ..., c_n$) and the examination periods last for k days(denoted by $d_1, ..., d_k$).

- Let P be the set of all examination periods and $P = \{d_1^m, ..., d_k^m\} \cup \{d_1^a, ..., d_k^a\} \cup \{d_1^e, ..., d_k^e\}$. d_j^m indicates "the morning of day d_j". d_j^a indicates "the afternoon of day d_j". d_j^e indicates "the evening of day d_j".
- Let P_j be the set of all examination periods which are available for c_j for $j = 1, ..., n$.
- Based on P_j, create set D_j which contains the dates on which c_j is available for at least one examination period for $j = 1, ..., n$. Each element in D_j has super script j. For example, if c_3 is available for days d_1, d_3, d_5, then $D_2 = \{d_1^2, d_3^2, d_5^2\}$.

Then, implement the following:

1. Create a network flow N with vertices $V = \{s, t\} \cup \{c_1, ..., c_n\} \cup P \cup (\bigcup_{i=1}^n D_i)$ and with edges
   - $E = (\{(s, c_1), ..., (s, c_n)\})$ (with $c(s, c_i) =$ maximum number of examinations that c_i can invigilate)
   - $\cup(\bigcup_{i=1}^n(\bigcup_{d \in D_i}\{(c_i, d)\}))$ (with $c(c_i, d) = 2$)
   - $\cup(\bigcup_{i=1}^n(\{(d, p) | d \in D_i, p \in P_i$ and d, p have the same subscript(i.e. the same date)$\}))$ (with $c(d, p) = 1$)
   - $\cup(\bigcup_{p \in P}\{(p, t)\})$ (with $c(p, t) = \lceil(\text{number of examinations in period } p) \times (1 + 10\%)\rceil$)
2. Find a maximum integer flow f in network N using Edmonds-Karp algorithm.
3. If there is an edge (p, t) with $p \in P$ and $f(p, t) < c(p, t)$, return NIL. Otherwise, set $C_i = \{p | f(d, p) = 1, d \in D_i, p \in P_i\}$ and return $C_1, ..., C_n$.

Runtime Analysis:

- Notice that $|V| \leq nk + 3k + 2$. $|E| \leq n + nk + 3nk + 3k$.
- Since Edmonds-Karp algorithm runs in $O(|V||E|^2)$, it takes $O((nk + 3k + 2)(n + nk + 3nk + 3k)^2) = O(n^3k^3)$ to run Edmonds-Karp algorithm on N.
- It takes $O(|V| + |E|) = O(nk)$ to build network N.
- It takes $O(|E|) = O(nk)$ to build C_i for $i = 1, ..., n$.
- Totally, the algorithm runs in $O(n^3k^3)$ which is in polynomial time.

Justification of Correctness

Claim 1. Every collection of valid sets of examination periods for CPOs $C_1, ..., C_n$ give rise to a flow f in N.
Since $C_1, ..., C_n$ are valid, we have the following:

(1) $f(s, c_i) = |C_i|$ (the number of examination periods that c_i will invigilate)

(2) For $d \in D_i, f(c_i, d)$ = number of examination periods that c_i will invigilate on day d_i and it is no more than 2.

(3) For $d \in D_i, p \in P_i, f(d, p) = 1$ if and only if $(d, p) \in C_i$

(4) For $p \in P, f(p, t)$ = number of CPOs in examination period $p = c(p, t)$

By (4), $|f|$ is maximized. Therefore, every valid collection of sets $C_1, ..., C_n$ gives rise of a maximum flow in N.

Claim 2. Every integer flow in N gives rise to a collection of sets of examination periods for CPOs $C_1, ..., C_n$ (or NIL if it is not possible).

- $C_i = \{p | f(d, p) = 1, d \in D_i, p \in P_i\}$
- Every CPO is within maximum availability because $c(s, c_i)$ = maximum number of examinations that c_i can invigilate
- Every CPO is assigned to no more than 2 examination periods in one day because $c(c_i, d) = 2, d \in D_i$.
- Every CPO is only assigned to examination periods that is available because $(d, p) \notin E$ for $d \in D_i, p \notin P_i$.
- Every examination period has enough CPOs if and only if $f(p, t) = c(p, t)$ for all $p \in P$. Therefore, $C_1, ..., C_n$ exist if and only if there is a flow such that $f(p, t) = c(p, t)$ for all $p \in P$. If such flow f exists, $|f|$ must be maximized.

Therefore, every maximized flow in N gives rise to a collection of sets of examination periods for CPOs $C_1, ..., C_n$ if $f(p, t) = c(p, t)$ for all $p \in P$ otherwise NIL.

By Claim 1 and Claim 2, the algorithm is correct.