Suggested Solutions to Final Problems

(draft, to be revised)

1. The *next* table is a precomputed table (for $B = b_1 b_2 \cdots b_m$) that plays a critical role in the KMP algorithm. Under what condition regarding $b_1 b_2 \cdots b_i$, $2 \leq i \leq m$, will $next[i]$ get a 0 in the preprocessing? And under what condition can it be safely set to -1 (without missing a potential match when searching for B in another input string)?

   Solution. The value of $next[i]$ is determined by the length of the longest proper prefix of $b_1 b_2 \cdots b_{i-1}$ that is also a proper suffix of $b_1 b_2 \cdots b_{i-1}$. When no such prefix exists, $next[i]$ gets a 0.

   During a search for string B in string A using KMP, when b_j is compared against a_i and the comparison fails, $b_{next[j]+1}$ is tried next against a_i. When $next[j] = 0$, it is b_1 that is compared with a_i. If the comparison fails, then b_1 will be compared against a_{i+1}, according to the case for $next[j]+1 = 0$, i.e., $next[j] = -1$. When $b_1 = b_j$, the comparison between b_1 and a_i is doomed to fail (since $b_1 = b_j \neq a_i$) and the comparison could have been saved. To achieve the saving, we can set $next[j]$ to -1 (instead of 0) when b_j happens to be equal to b_1. □

2. Design an algorithm for finding an Eulerian circuit in an undirected graph. Please present your algorithm in adequate pseudocode and make assumptions wherever necessary. Explain why your algorithm is correct and give an analysis of its time complexity. The more efficient your algorithm is, the more points you will be credited for this problem. (Hint: the discovery of a cycle and that of the Eulerian circuits in individual connected components with the cycle removed, in the induction step, can be interweaved.)

   Solution. To be completed; for now, see solutions from previous TA sessions. □

3. Design an algorithm that, given a weighted directed graph, detects the existence of a negative-weight cycle (the sum of the weights of its edges is negative). Please present your algorithm in adequate pseudocode and make assumptions wherever necessary. Explain why your algorithm is correct and give an analysis of its time complexity. The more efficient your algorithm is, the more points you will be credited for this problem.

   Solution. Floyd's algorithm for all-pair shortest paths can be easily adapted for detecting the existence of a negative-weight cycle.

```
Algorithm Detect_Negative_Weight_Cycle(W);
begin
    {initialization}
    for i := 1 to n do
        W[i,i] := 0;
        for j := 1 to n do
            if (i,j) ∈ E then
                W[i,j] := length(i,j);
                if i = j and W[i,j] < 0 then
                    Print "A negative-weight cycle detected."; Stop
            else W[i,j] := ∞;
```

```
    for m := 1 to n do  {the induction sequence}
        for x := 1 to n do
            for y := 1 to n do
                if W[x,m] + W[m,y] < W[x,y] then
                    W[x,y] := W[x,m] + W[m,y];
                if x = y and W[x,y] < 0 then
                    Print "A negative-weight cycle detected."; Stop
end
```

Suppose a negative-weight cycle exists, looping around vertex v, in the input graph. If it is a self-loop on v, then it will be detected during the initialization stage. Otherwise, the cycle must be the concatenation of a $(< k)$-path from v to k and a $(< k)$-path from k to v, from some $k \leq n$. It will be detected immediately after $W[v,k] + W[k,v]$ has been computed for the iteration of $m = k$ and $W[v,v]$ updated to reflect the current best value (if not happening earlier).

The time complexity of the detection algorithm is clearly $O(n^3)$. □

4. Let $G = (V, E)$ be a connected weighted undirected graph and T be a minimum-cost spanning tree (MCST) of G. Suppose that the cost of one edge $\{u, v\}$ in G is *updated*; $\{u, v\}$ may or may not belong to T. Prove that T is still an MCST of G under any of the following two conditions:

   (a) $\{u, v\}$ belongs to T and its cost decreases or

   (b) $\{u, v\}$ does not belong to T and its cost increases.

   You may assume that the costs of all edges are distinct before and after the cost update to $\{u, v\}$.

   Solution. Claim: for each of those edges not in an MCST, the edge must have the highest cost among the edges in the cycle created if it is inserted into the MCST. This claim can be easily proven by contradiction.

   Part (4a): with its original weight, $\{u, v\}$ is part of the minimum-cost spanning tree T. The preceding claim does not change for T when the cost of $\{u, v\}$ decreases. So, T is still an MCST.

   Part (4b): with its original weight, $\{u, v\}$ is not part of the minimum-cost spanning tree T. Therefore, $\{u, v\}$ has the highest cost among the edges in the cycle created if it is inserted into T. This remains unchanged when the cost of $\{u, v\}$ increases, and hence T is still an MCST. □

5. The most common approach to finding an augmenting path (if one exists) in a network with some given flow is breadth-first search (BFS). Please present such an algorithm in suitable pseudocode.

   Solution.

```
Algorithm Augmenting_Path(G, s, t, c, f);
begin
    put s in Queue;
    while Queue is not empty do
```

```
        remove vertex u from Queue;
        if u = t then
            Print the path (by backtracking the parent values) and stop.
        if u is unmarked then
            mark u;
            for all edges (u, v) with v.parent undefined do
                if f(u, v) < c(u, v) or f(v, u) > 0 then
                    v.parent := u;
                    put v in Queue
    Print "No augmenting path found."
end
```

□

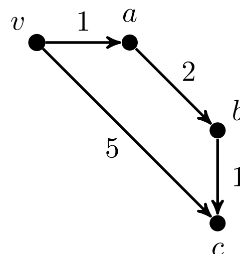
6. Below is an algorithm, based on the dynamic programming approach, for solving the single-source shortest path problem.

```
Algorithm Single_Source_Shortest_Paths(length);
begin
    D[v] := 0;
    for all u ≠ v do
        if (v, u) ∈ E then
            D[u] := length(v, u)
        else D[u] := ∞;
    for l := 2 to n − 1 do
        for all u ≠ v do
            for all u' such (u', u) ∈ E do
                if D[u'] + length[u', u] < D[u] then
                    D[u] := D[u'] + length[u', u]
end
```

Denote by $D^l(u)$ the length of a shortest path from v (the source) to u containing *at most* l edges; particularly, $D^{n-1}(u)$ is the length of a shortest path from v to u (with no restrictions).

In the for loop with index l iterating from 2 to $n-1$, it is possible that, for certain $l = k$, $D[u]$ acquires the value of $D^{k'}(u)$, where $k < k'$. Why? Please explain with an example.

Solution. Consider the following input graph.



Suppose, in the loop "**for** all $u \neq v$ **do**", a, b, and c are processed in this order. When $l = 2$, after $D[b]$ gets updated (since $D[a] + length[a, b] < \infty$) as $D^2(b)$, it becomes immediately available for computing $D[c]$ in the next iteration of "**for** all $u \neq v$ **do**". What we get for $D[c]$ (from $D[b] + length[b, c]$) in that iteration is actually $D^3(c)$, while l is still 2. □

7. Consider designing an algorithm by dynamic programming to determine the length of a longest common subsequence of two strings (sequences of letters). For example, "abbcc" is a longest common subsequence of "abcabcabc" and "aaabbbccc", and so is "abccc".

   (a) Formulate the solution using recurrence relations.

   Solution. Let $LCS(i, j)$ denote the length of a longest common subsequence of $A[1..i]$ and $B[1..j]$, where $A[1..i] = a_1 a_2 \cdots a_i$ and $B[1..j] = b_1 b_2 \cdots b_j$.
   For $i = 0$ or $j = 0$,

$$LCS(0, j) = 0$$
$$LCS(i, 0) = 0$$

   For $i > 0$ and $j > 0$,

$$LCS(i, j) = \max \begin{cases} LCS(i-1, j-1) + 1 & \text{if } a_i = b_j \\ \max(LCS(i-1, j), LCS(i, j-1)) & \text{otherwise} \end{cases}$$

   □

   (b) Present the algorithm in suitable pseudocode, based on the previous recursive formulation. What is the time complexity of your algorithm?

   Solution.

```
Algorithm Longest_Common_Subsequence (A, n, B, m);
    for i := 0 to n do LCS[i, 0] := 0;
    for j := 1 to m do LCS[0, j] := 0;
    for i := 1 to n do
        for j := 1 to m do
            if a_i = b_j then
                x := LCS[i - 1, j - 1] + 1
            y := LCS[i - 1, j];
            z := LCS[i, j - 1];
            LCS[i, j] := max(x, max(y, z))
```

   The time complexity is clearly $O(mn)$. □

8. Every problem in P is polynomially reducible to any other non-trivial problem in P. Why? Please explain. (Note: a decision problem is *non-trivial* if there exists an input such that the answer is *yes* and there also exists an input such that the answer is *no.* In other words, a decision problem is non-trivial if its corresponding language is neither the universe nor the empty set.)

   Solution. Let A be an arbitrary problem in P with U_A as the set of all possible inputs and $L_A \subseteq U_A$ be the corresponding language. Let B be a nontrivial problem in P with U_B as the set of all possible inputs and $L_B \subseteq U_B$ be the corresponding language. L_B is neither the universe U_B nor the empty set. So, there is a $u_2 \in U_B$ such that $u_2 \in L_B$ and there is a $u_2' \in U_B$ such that $u_2' \notin L_B$.

   To establish that L_A is polynomially reducible to L_B, we argue for the existence of a polynomial-time deterministic algorithm AC (mapping from U_A to U_B) such that for every $u_1 \in U_A$, $u_1 \in L_A$ if and only if $AC(u_1) \in L_B$.

   The algorithm AC works as follows. For any input $u_1 \in U_A$, if $u_1 \in L_A$, which can be checked by a deterministic algorithm in polynomial time (since A is in P), then we map u_1 to u_2; otherwise, we map u_1 to u_2'. Clearly, $u_1 \in L_A$ if and only if $AC(u_1) \in L_B$. □

9. In the proof (discussed in class) of the NP-hardness of the 3SAT problem by reduction from the SAT problem, we convert an arbitrary Boolean expression in CNF (input of the SAT problem) to a Boolean expression in 3CNF (where each clause has exactly three literals).

   (a) Please illustrate the conversion by giving the Boolean expression that will be obtained from the following Boolean expression:

   $$(w + \overline{y}) \cdot (v + \overline{w} + x + y + z) \cdot (w + x + \overline{y} + \overline{z}).$$

   Solution.

   $$\begin{array}{l} (w + \overline{y} + x_1) \cdot (w + \overline{y} + \overline{x_1}) \\ (v + \overline{w} + y_1) \cdot (\overline{y_1} + x + y_2) \cdot (\overline{y_2} + y + z) \\ (w + x + z_1) \cdot (\overline{z_1} + \overline{y} + \overline{z}) \end{array}$$

   The three rows correspond respectively to the first, the second, and the third clauses of the original expression. □

   (b) The original Boolean expression is satisfiable. As a demonstration of why the reduction is correct, please use the resulting Boolean expression to show that it is indeed the case.

   Solution. We show that the resulting Boolean expression is satisfiable and any satisfying assignment for it induces one for the original expression.

   Let $v = 1, w = 1, x = 1, y = 1, z = 0, x_1 = 0, y_1 = 0, y_2 = 0, z_1 = 0$. This is a satisfying assignment for the resulting expression. The part of the assignment for v, w, x, y, z is also a satisfying assignment for the original expression.

   In general, for the first row of the resulting expression, the satisfying assignment must make $w + \overline{y}$ equal to 1, as x_1 and $\overline{x_1}$ cannot both be 1, and hence the assignment also makes the corresponding clause $(w + \overline{y})$ in the original expression equal to 1, For the second row of the resulting expression, the satisfying assignment must make one of $v + \overline{w}$, x, and $y + z$ equal to 1, as y_1, $\overline{y_1} + y_2$, and $\overline{y_2}$ cannot all be 1. The assignment therefore makes the corresponding clause $(v + \overline{w} + x + y + z)$ in the original expression equal to 1. Analogously, for the third row. □

10. Solve one of the following two problems. (Note: if you try to solve both problems, I will randomly pick one of them to grade.)

   (a) The independent set problem is as follows.

   > An independent set in an undirected graph is a set of vertices no two of which are adjacent. The problem is to determine, given a graph G and an integer k, whether G contains an independent set with $\geq k$ vertices.

   Prove that the independent set problem is NP-complete.

   Solution. The problem is in NP, as we can guess and select a set of vertices (by nondeterministically choosing, for each vertex, to be in or not in the selection) and check in polynomial time whether the selected set is of size $\geq k$ and is indeed an independent set of the given graph G.

   To prove that it is NP-hard, we demonstrate a polynomial-time reduction from the clique problem, which is known to be NP-hard. An input $(G_1 = (V_1, E_1), k_1)$, which is a pair of a graph and an integer, to the clique problem can be converted to an

input $(G_2 = (V_2, E_2), k_2)$ to the independent set problem in the following manner: To obtain G_2, we simply take the complement of G_1, i.e., $V_2 = V_1$ and $E_2 = \{\{u, v\} \mid u, v \in V_2 \text{ and } \{u, v\} \notin E_1\}$. And, we take k_2 simply to be k_1. This conversion apparently can be done by a deterministic algorithm in polynomial time. We need to show that G_1 has a clique of size $\geq k_1$ if and only if G_2 has an independent set of size $\geq k_2$ $(= k_1)$.

The "only if" part: suppose G_1 has a clique $C \subseteq V_1$ of size $\geq k_1$. C is also a subset of V_2 and $|C| \geq k_1 = k_2$. We claim that C is also an independent set of G_2. This is so, as every pair $\{u, v\}$ of vertices in C are adjacent (i.e., directly connected by an edge) in G_1 and thus become non-adjacent in G_2, which is the complement of G_1.

The "if" part: suppose G_2 has an independent set D of size $\geq k_2$. D is a subset of V_1 and $|D| \geq k_2 = k_1$. We claim that D is also a clique of G_1. This is so, as every pair $\{u, v\}$ of vertices in D are non-adjacent in G_2, which is the complement of G_1, and therefore are adjacent (i.e., directly connected by an edge) in G_1. □

(b) The subset sum problem (a variant of the knapsack problem) is as follows.

> The input is a multiset of numbers $\{a_1, a_2, \cdots, a_n\}$ and another number k. The problem is to determine whether the multiset contains a subset such that the sum of numbers in the subset is exactly k.

Prove that the subset sum problem is NP-complete.

Solution. The problem is in NP, as we can guess and select a set of numbers (by nondeterministically choosing, for each number, to be in or not in the selection) from the input multiset and check in polynomial time whether the sum of the numbers in the selected set is exactly k.

To prove that it is NP-hard, we demonstrate a polynomial-time reduction from the partition problem, which is known to be NP-hard. An input set $X = \{x_1, x_2, \cdots, x_n\}$ of items with an associated function s (assigning a size to each item) to the partition problem can be converted to an input (S, k), where S is a multiset of numbers and k is a number, to the subset sum problem, by taking S to be $\{s(x_i) \mid x_i \in X, \text{ for } 1 \leq i \leq n\}$ and k to be $\frac{1}{2}\Sigma_{i=1}^{i=n} s(x_i)$. This conversion apparently can be done by a deterministic algorithm in polynomial time. We need to show that X can be evenly partitioned, i.e., partitioned into two subsets of the same total size, if and only if S has a subset whose numbers sum to exactly k.

The "only if" part: suppose X can be evenly partitioned and $X_1, X_2 \subseteq X$ are the two subsets of the same total size. Then, obviously the multiset $\{s(x) \mid x \in X_1\}$ is a subset of S whose numbers sum to exactly $\frac{1}{2}\Sigma_{i=1}^{i=n} s(x_i)$.

The "if" part: suppose S has a subset S' whose numbers sum to exactly $k = \frac{1}{2}\Sigma_{i=1}^{i=n} s(x_i)$ and hence the remaining numbers of S also sum to exactly the same amount. Let X' be the set of items whose sizes correspond to the numbers in S'. Then, X' and $X \setminus X'$ form a partition of X and are of the same total size. □

Appendix

- The KMP algorithm (assuming *next*):

```
Algorithm String_Match (A, n, B, m);
begin
    j := 1;  i := 1;
    Start := 0;
    while Start = 0 and i ≤ n do
```

```
        if B[j] = A[i] then
            j := j + 1;  i := i + 1
        else
            j := next[j] + 1;
            if j = 0 then
                j := 1;  i := i + 1;
        if j = m + 1 then Start := i - m
end
```

- Below is a theorem useful for discovering an MCST of a connected weighted undirected graph $G = (V, E)$:

  Let V_1 and V_2 be a partition of V and $E(V_1, V_2)$ be the set of edges connecting nodes in V_1 to nodes in V_2. An edge with the minimum weight in $E(V_1, V_2)$ must be in an MCST of the given G.

- We say that problem/language L_1 is *polynomially reducible* to problem/language L_2 if there exists a conversion algorithm AC satisfying the following conditions:

  1. AC runs in polynomial time (deterministically).
  2. $u_1 \in L_1$ *if and only if* $AC(u_1) = u_2 \in L_2$.

- The clique problem: given an undirected graph $G = (V, E)$ and an integer k, determine whether G contains a clique of size $\geq k$. (A *clique* of G is a subgraph C of G such that every vertex in C is adjacent to all other vertices in C.)

  The clique problem is NP-complete.

- The partition problem: given a set X where each element $x \in X$ has an associated size $s(x)$, is it possible to partition the set into two subsets with exactly the same total size?

  The partition problem is NP-complete.