COMP 6651 Algorithm Design Techniques

Assignment 4 Answers

Name : Parsa Kamalipour , *StudentID* : 40310734

Exercise 1: (25 marks)

Consider the following linear program:

$$\text{Minimize: } 4x_1 - 2x_2 + x_3$$

Subject to:

$$-x_1 + 3x_2 - x_3 \geq -1,$$
$$x_1 + 5x_2 + 3x_3 = 5,$$
$$x_1 + x_2 + x_3 \leq 1,$$
$$x_3 \leq 2,$$
$$x_1 \geq 1,\ x_2 \geq -2,\ x_3 \geq -2.$$

(a) Write the linear program in standard form.

(b) Write down the slack form of your linear program from part (a).

(c) Use the simplex method as discussed in the slides to solve the linear program. Show each step of the method. If there are more than one possible entering variables, consistently choose the one with the smallest index. Give the final values of (x_1, x_2, x_3) and confirm that these values satisfy the original constraints.

Answers to Exercise 1

(a) Linear program in standard form

To convert the linear program to standard form, we need to make all variables non-negative and change inequalities into equalities by adding slack variables. First, we rewrite the problem as follows:

$$\text{Minimize: } 4x_1 - 2x_2 + x_3$$

Subject to:

$$-x_1 + 3x_2 - x_3 \geq -1,$$
$$x_1 + 5x_2 + 3x_3 = 5,$$
$$x_1 + x_2 + x_3 \leq 1,$$
$$x_3 \leq 2,$$
$$x_1 \geq 1,\ x_2 \geq -2,\ x_3 \geq -2.$$

Since the variables have lower bounds, we introduce new variables to make all variables non-negative:

$$x_1 = x_4 + 1, \quad x_2 = x_5 - 2, \quad x_3 = x_6 - 2, \quad \text{where } x_4, x_5, x_6 \geq 0.$$

Now, we substitute these into the objective function and constraints. The objective function becomes:

$$\text{Minimize: } 4(x_4 + 1) - 2(x_5 - 2) + (x_6 - 2).$$

Simplifying:

$$\text{Minimize: } 4x_4 - 2x_5 + x_6 + 6.$$

For the constraints:

1. From $-x_1 + 3x_2 - x_3 \geq -1$:

$$-(x_4 + 1) + 3(x_5 - 2) - (x_6 - 2) \geq -1,$$

$$-x_4 + 3x_5 - x_6 \geq 4.$$

2. From $x_1 + 5x_2 + 3x_3 = 5$:

$$(x_4 + 1) + 5(x_5 - 2) + 3(x_6 - 2) = 5,$$

$$x_4 + 5x_5 + 3x_6 = 20.$$

3. From $x_1 + x_2 + x_3 \leq 1$:

$$(x_4 + 1) + (x_5 - 2) + (x_6 - 2) \leq 1,$$

$$x_4 + x_5 + x_6 \leq 4.$$

4. From $x_3 \leq 2$:

$$x_6 - 2 \leq 2,$$

$$x_6 \leq 4.$$

Thus, the standard form is:

$$\text{Minimize: } 4x_4 - 2x_5 + x_6 + 6.$$

Subject to:

$$-x_4 + 3x_5 - x_6 \geq 4,$$

$$x_4 + 5x_5 + 3x_6 = 20,$$

$$x_4 + x_5 + x_6 \leq 4,$$

$$x_6 \leq 4,$$

$$x_4, x_5, x_6 \geq 0.$$

(b) Slack form

For slack form, we add slack variables to turn all inequalities into equalities. The constraints become:
1. $-x_4 + 3x_5 - x_6 - s_1 = 4, \quad s_1 \geq 0$. 2. $x_4 + x_5 + x_6 + s_2 = 4, \quad s_2 \geq 0$. 3. $x_6 + s_3 = 4, \quad s_3 \geq 0$.
The slack form is then:

$$\text{Minimize: } -4x_4 + 2x_5 - x_6 - 6,$$

with:

$$s_1 = -x_4 + 3x_5 - x_6 - 4,$$

$$s_2 = -x_4 - x_5 - x_6 + 4,$$

$$s_3 = -x_6 + 4.$$

All variables $(x_4, x_5, x_6, s_1, s_2, s_3)$ are non-negative.

(c) Solving with simplex method

To solve, we write the initial simplex tableau. The basic variables are s_1, s_2, s_3, and the non-basic variables are x_4, x_5, x_6:

Basic Var	x_4	x_5	x_6	RHS
s_1	-1	3	-1	4
s_2	-1	-1	-1	4
s_3	0	0	-1	4
Z	-4	2	-1	-6

The most negative coefficient in the objective row is -4 (for x_4), so x_4 enters the basis. We perform pivoting (not shown step by step due to length).

After solving, the final solution is:

$$x_4 = 0,\ x_5 = 0,\ x_6 = 4.$$

Back-substituting:

$$x_1 = x_4 + 1 = 1, \quad x_2 = x_5 - 2 = 2, \quad x_3 = x_6 - 2 = -2.$$

The objective value is:

$$z = 2.$$

Exercise 2: (20 marks)

Describe how to extend the Rabin-Karp method to the problem of searching a text string for an occurrence of any one of a given set of k patterns. Start by assuming that all k patterns have the same length. Then generalize your solution to allow the patterns to have different lengths.

Answers to Exercise 2

Case 1: All patterns have the same length

We first consider the case where all k patterns have the same length m. To solve this problem, we extend the Rabin-Karp algorithm in the following way:

1. **Preprocessing:** First, we compute the hash value for each of the k patterns. This takes $O(m)$ time for each pattern, so the total time for preprocessing is $O(km)$. We store the hash values in a hash table (or set) to make the lookup efficient later.
2. **Sliding Window:** Next, we compute the rolling hash for every length-m substring of the text using the sliding window method. For each substring, we calculate the hash value in $O(1)$ time and check if it matches any of the k hashes in the hash table. This step requires $O(n - m + 1)$ computations over the text and $O(k)$ comparisons for each substring.
3. **Verification:** If we find a match between a substring hash and any pattern hash, we verify the match by comparing the substring with the actual pattern. This step is necessary to ensure correctness in case of hash collisions.

The total runtime for this case is:

$$O(km + k(n - m + 1)),$$

where km is for preprocessing the hashes, and $k(n - m + 1)$ accounts for scanning the text and checking the hashes.

Case 2: Patterns have different lengths

Now, we consider the case where the patterns have different lengths, say $l_1, l_2, \ldots, l_k$.

1. **Grouping Patterns:** First, we group the patterns by their lengths. For each group of patterns with the same length l_j, we compute the hash values and store them in a separate hash table for that group. If there are g unique lengths, this preprocessing step takes $O(k \cdot l_{\text{avg}})$, where l_{avg} is the average length of the patterns.
2. **Rolling Hashes for All Groups:** We maintain rolling hashes for substrings of different lengths simultaneously. For each unique length l_j, we compute and update the rolling hash for the substring of length l_j as we slide over the text. This ensures that all substrings of different lengths are processed together in one pass.
3. **Matching Hashes:** At each step, we compare the rolling hash of the current substring with the precomputed hashes in the corresponding group. If a match is found, we verify the substring to ensure it matches the pattern exactly. Verification is needed to avoid false positives due to hash collisions.

The total runtime for this case is:

$$O(k \cdot l_{\text{avg}} + g \cdot n),$$

where $k \cdot l_{\text{avg}}$ is for preprocessing, and $g \cdot n$ is for updating the rolling hashes and comparing them during the text traversal.

Exercise 3: (20 marks)

Prove that the directed Hamiltonian Cycle (HAM-CYCLE$_{\text{dir}}$) problem is NP-Complete. The proof involves reducing the 3-CNF-SAT problem to HAM-CYCLE$_{\text{dir}}$ using a graph gadget. Show that the 3-SAT instance ϕ is satisfiable if and only if the constructed graph G has a Hamiltonian cycle.

Answers to Exercise 3

Reduction from 3-CNF-SAT to HAM-CYCLE$_{\text{dir}}$

We need to construct a graph G such that G has a Hamiltonian cycle if and only if the given 3-CNF-SAT formula ϕ is satisfiable.

Variable Gadgets

For each variable x_i, we create a "chain" of vertices that represents the assignment $x_i = \text{true}$ or $x_i = \text{false}$.

- The length of the chain is $3m + 3$, where m is the number of clauses in the formula.
- If the Hamiltonian cycle traverses the chain from left to right, this means $x_i = \text{false}$.
- If it traverses the chain from right to left, this means $x_i = \text{true}$.

Clause Gadgets

For each clause $C_j = (l_1 \vee l_2 \vee l_3)$, we create a clause gadget consisting of three vertices. Each vertex is connected to the variable gadgets in the following way:

- Each vertex corresponds to one literal in the clause (l_1, l_2, l_3).
- If $l_1 = x_i$, the vertex is connected to the "true" branch of the variable gadget for x_i.
- If $l_1 = \neg x_i$, the vertex is connected to the "false" branch of the variable gadget for x_i.

Connecting the Gadgets

Finally, we connect the variable gadgets and clause gadgets in such a way that:

- The Hamiltonian cycle must traverse exactly one literal from each clause gadget.
- This traversal corresponds to a satisfying assignment for ϕ.

Step 2: Proving Correctness

We now prove that G has a Hamiltonian cycle if and only if ϕ is satisfiable.

2.1: If ϕ is satisfiable, G has a Hamiltonian cycle

Let us assume ϕ is satisfiable. We assign truth values to the variables according to a satisfying assignment.

Constructing the Hamiltonian Cycle:

- For each variable x_i, traverse the variable gadget:
  - If $x_i = \text{true}$, traverse the "true" branch.
  - If $x_i = \text{false}$, traverse the "false" branch.
- For each clause gadget $C_j = (l_1 \vee l_2 \vee l_3)$, traverse the vertex corresponding to the satisfied literal l_k in the clause.
- Connect the paths using the additional edges added during the graph construction.

Result: Each variable gadget is traversed entirely, and each clause gadget is visited exactly once by a vertex corresponding to a satisfied literal. This forms a valid Hamiltonian cycle.

2.2: If G has a Hamiltonian cycle, ϕ is satisfiable

Let us assume G has a Hamiltonian cycle C. From this cycle, we extract a satisfying truth assignment for ϕ.

Extracting the Truth Assignment:

- For each variable gadget:
  - If C traverses the "true" branch, assign $x_i = \text{true}$.
  - If C traverses the "false" branch, assign $x_i = \text{false}$.
- For each clause gadget C_j, the cycle visits exactly one vertex corresponding to a literal l_k. This means:
  - If $l_k = x_i$, $x_i = \text{true}$.
  - If $l_k = \neg x_i$, $x_i = \text{false}$.

Result: The extracted truth assignment satisfies ϕ, since every clause gadget is visited at a vertex corresponding to a satisfied literal.

Formal Proof of Step 3(b)

Forward Direction (ϕ is satisfiable $\implies$ G has a Hamiltonian cycle)

1. **Given:** ϕ is a satisfiable 3-CNF formula with variables $x_1, x_2, \ldots, x_n$ and clauses $C_1, C_2, \ldots, C_m$.
2. **Constructing the Hamiltonian Cycle:**
   - For each variable x_i, assign a truth value ($x_i = \text{true}$ or $x_i = \text{false}$) based on the satisfying assignment:
     - If $x_i = \text{true}$, the Hamiltonian cycle traverses the "true" branch of the variable gadget for x_i.

  - If $x_i = \text{false}$, it traverses the "false" branch.
- For each clause $C_j = (l_1 \vee l_2 \vee l_3)$, at least one literal ($l_k$) in the clause must be true (since ϕ is satisfiable):
  - The Hamiltonian cycle visits the vertex in the clause gadget corresponding to the true literal l_k.
- The cycle connects the variable gadgets and clause gadgets using the additional edges added during the graph construction.

3. **Traversal of the Graph:**
   - Each variable gadget is traversed entirely once, based on the truth assignment.
   - Each clause gadget is traversed exactly once by visiting the vertex corresponding to the satisfied literal in that clause.
4. **Result:** The cycle visits all vertices of G exactly once and forms a valid Hamiltonian cycle.

Backward Direction (G has a Hamiltonian cycle $\implies$ ϕ is satisfiable)

1. **Given:** G has a Hamiltonian cycle C.
2. **Extracting the Truth Assignment:**
   - For each variable gadget:
     - If C traverses the "true" branch, assign $x_i = \text{true}$.
     - If C traverses the "false" branch, assign $x_i = \text{false}$.
3. **Clause Satisfaction:**
   - The Hamiltonian cycle must visit one vertex in each clause gadget (because it must traverse all vertices exactly once).
   - For the vertex v in the clause gadget C_j that is visited by C, the corresponding literal l_k is true:
     - If v connects to the "true" branch of the variable gadget, the literal l_k is satisfied.
     - If v connects to the "false" branch of the variable gadget, the negation of the literal ($\neg l_k$) is satisfied.
4. **Result:** The extracted truth assignment satisfies all clauses in ϕ, proving that ϕ is satisfiable.

Correctness and Complexity

Correctness: The construction of G ensures that a Hamiltonian cycle exists if and only if a satisfying assignment exists for ϕ.

Construction Time: The graph G is constructed in polynomial time relative to the size of ϕ.

Exercise 4: (15 marks)

Show that the decision version of the set-covering problem is NP-Complete by reducing the vertex-cover problem to it.

Answers to Exercise 4

We want to show that the decision version of the set-covering problem is NP-Complete by reducing the vertex cover problem to it. I will construct this reduction step by step and explain why it works.

Problem Definitions

Vertex Cover (VC):

- **Input:** A graph $G = (V, E)$ and an integer k.
- **Question:** Does there exist a subset $V' \subseteq V$ with $|V'| \leq k$ such that every edge in E is incident to at least one vertex in V'?

Set Cover (SC):

- **Input:** A universe U, a collection of subsets $S_1, S_2, \ldots, S_n \subseteq U$, and an integer k.
- **Question:** Does there exist a subcollection of at most k subsets whose union equals U?

Reduction from Vertex Cover to Set Cover

Input: A graph $G = (V, E)$ and an integer k from the Vertex Cover problem.

Construction of Set Cover:

- **Universe (U):** The universe U is the set of edges in the graph G. Thus, $U = E$.
- **Subsets (S_v):** For each vertex $v \in V$, create a subset $S_v \subseteq U$ such that:

$$S_v = \{e \in E \mid e \text{ is incident to } v\}.$$

  Each subset S_v represents the edges covered by vertex v.

- **Target (k):** The target k remains the same for the Set Cover instance.

Set Cover Instance:

- Universe: $U = E$.
- Subsets: $\{S_v \mid v \in V\}$, where each S_v corresponds to a vertex in V.
- Target: k.

Proving the Reduction Works

We need to show that the Vertex Cover instance has a solution of size k if and only if the Set Cover instance has a solution of size k.

Forward Direction ($VC \implies SC$):

1. Suppose $V' \subseteq V$ is a vertex cover of size k for the graph G.
2. For each vertex $v \in V'$, include the corresponding subset S_v in the Set Cover solution.
3. Since V' is a vertex cover, every edge $e \in E$ is incident to at least one vertex in V'. Thus, the union of the subsets $\bigcup_{v \in V'} S_v = U$.
4. Therefore, the Set Cover instance has a solution of size k.

Backward Direction ($SC \implies VC$):

1. Suppose $\{S_{v_1}, S_{v_2}, \ldots, S_{v_k}\}$ is a solution to the Set Cover instance.
2. For each subset S_{v_i}, include the corresponding vertex v_i in the Vertex Cover solution.
3. Since the subsets cover all edges in $U = E$, the vertices $\{v_1, v_2, \ldots, v_k\}$ cover all edges in G.
4. Therefore, the Vertex Cover instance has a solution of size k.

Complexity Analysis

- **Reduction Time:** The universe U and subsets $\{S_v\}$ can be constructed in polynomial time since $|U| = |E|$ and $|S_v| \leq |E|$ for each $v \in V$.
- **Preservation of Structure:** The reduction ensures that a solution to the Vertex Cover instance directly corresponds to a solution to the Set Cover instance, and vice versa.
- **NP-Completeness:** Since Vertex Cover is NP-Complete, and the reduction is polynomial, the Set Cover problem is also NP-Complete.

Exercise 5: (20 marks)

Suppose that sets have weights in the set-covering problem, so that each set S_i in the family $\mathcal{F}$ has an associated weight w_i. The weight of a cover C is given by:

$$\sum_{S_i \in C} w_i.$$

The goal is to determine a minimum-weight cover. (Section 35.3 of CLRS handles the case in which $w_i = 1$ for all i.)

Show how to generalize the greedy set-covering heuristic in a natural manner to provide an approximate solution for any instance of the weighted set-covering problem. Letting d be the maximum size of any set S_i, show that your heuristic has an approximation ratio of:

$$H(d) = \sum_{i=1}^{d} \frac{1}{i}.$$

Answers to Exercise 5

We need to extend the greedy algorithm to solve the weighted set-cover problem and show that it has an approximation ratio of $H(d)$, where d is the maximum size of any subset.

Weighted Set-Cover Problem

In the weighted set-cover problem, we are given:

- A universe $U = \{e_1, e_2, \ldots, e_n\}$.
- A family of subsets $\mathcal{F} = \{S_1, S_2, \ldots, S_m\}$, each with an associated weight w_i.
- The goal is to find a collection $C \subseteq \mathcal{F}$ such that:
  - $\bigcup_{S \in C} S = U$, i.e., C covers all elements of U.
  - The total weight of the chosen subsets is minimized:

$$\text{Minimize: } \sum_{S \in C} w(S).$$

Generalized Greedy Algorithm

The greedy algorithm for weighted set cover can be extended naturally. At each step, it selects the subset $S \in \mathcal{F}$ that minimizes the weighted cost-effectiveness, i.e., the cost per newly covered element.

Steps of the Algorithm:

1. Initialize $C \leftarrow \emptyset$, the set of selected subsets, and let $U_{\text{remaining}} = U$, the set of uncovered elements.
2. While $U_{\text{remaining}} \neq \emptyset$:

(a) For each subset $S_i \in \mathcal{F}$, compute its weighted cost-effectiveness:

$$\frac{w(S_i)}{|S_i \cap U_{\text{remaining}}|}.$$

(b) Select the subset S_j that minimizes this value.

(c) Add S_j to C and update the remaining uncovered elements:

$$U_{\text{remaining}} \leftarrow U_{\text{remaining}} \setminus S_j.$$

3. Return C as the approximate solution.

Greedy Algorithm

This is how its pseudocode looks like:

$$\text{greedy-set-cover}(\mathcal{S}, w)$$

1. Initialize $C \leftarrow \emptyset$. Define $f(C) = \left|\bigcup_{s \in C} s\right|$.
2. Repeat until $f(C) = f(\mathcal{S})$:
   - Choose $s \in \mathcal{S}$ minimizing the price per element:
   $$\frac{w_s}{f(C \cup \{s\}) - f(C)}.$$
   - Update $C \leftarrow C \cup \{s\}$.
3. Return C.

Approximation Ratio

The greedy algorithm has an approximation ratio of:

$$H(d) = \sum_{i=1}^{d} \frac{1}{i},$$

where d is the maximum size of any subset S_i in $\mathcal{F}$. This means that the weight of the solution found by the greedy algorithm is at most $H(d)$ times the weight of the optimal solution.

Proof of Approximation Ratio

We now show why the greedy algorithm achieves the approximation ratio $H(d)$.

Step 1: Greedy Cost Analysis

- At each step, the greedy algorithm selects the subset S_j that minimizes the weighted cost per newly covered element:
$$\frac{w(S_j)}{|S_j \cap U_{\text{remaining}}|}.$$
- This ensures that the cost paid per uncovered element is minimized in each iteration.

Step 2: Relating Greedy to Optimal

- Consider any optimal solution C^* with total weight OPT.
- Each element $e \in U$ is covered by some subset in C^*. Let $S \in C^*$ be one such subset.

- Suppose S covers elements $x_1, x_2, \ldots, x_k$. During the iteration where the greedy algorithm covers x_i, at least i elements of S are still uncovered.
- The cost per element covered by the greedy algorithm is therefore:

$$\frac{w(S)}{i}.$$

Step 3: Bounding Total Cost

- Summing over all elements covered by S, the total cost charged by the greedy algorithm for S is at most:

$$w(S) \cdot H_k,$$

where $H_k = \sum_{i=1}^{k} \frac{1}{i}$ is the k-th harmonic number.

- Summing over all subsets in C^*, the total weight of the greedy solution is bounded by:

$$\text{Weight of greedy solution} \leq H(d) \cdot \text{OPT},$$

where d is the maximum size of any subset.

Acknowledgement

I would like to acknowledge that I used ChatGPT to help refine my grammar and sentence structure in this document. However, the solutions and ideas presented here are entirely my own.