UNIVERSITY OF WASHINGTON
SCHOOL OF SCIENCE, TECHNOLOGY, ENGINEERING AND MATHEMATICS
DIVISION OF COMPUTING AND SOFTWARE SYSTEMS

ALGORITHM DESIGN AND ANALYSIS

Problem Set 1

Algorithm Analysis, Graphs and Greedy Algorithms

Professor: Clark Olson
Student: Tran Tien Phat
phattt@u.washington.edu

WASHINGTON, FEBRUARY 2024

Contents

Introduction

Problem 1

Is the following statement true or false? If true, give a proof. If false, give a counterexample.

Consider instances of the Stable Matching Problem in which there exists a hospital h and a student s such that h is ranked first on the preference list of s and s is ranked first on the preference list of h. Then in every stable matching S for instances that meet this definition, the pair (h, s) belongs to S.

Solution. The statement is true.

In the Stable Matching Problem, a matching is stable if there does not exist any pair (h', s') such that h' and s' would both prefer each other over their current partners in the matching.

Given that hospital h is ranked first on the preference list of student s and vice versa, suppose there exists a stable matching S where (h, s) does not belong to S. This supposition will be used for establishing a contradiction.

In this case, s is matched with some hospital h' and h is matched with some student s'. Since s prefers h over h' and h prefers s over s', they would both prefer each other over their current partners in the matching, which contradicts the definition of a stable matching.

Therefore, in every stable matching S for instances that meet this definition, the pair (h, s) must belong to S. This proves the statement.

Problem 2

Consider a generalization of the problem that we examined in lecture, where there are m hospitals, each with a certain number of available positions. There are n students graduating in a particular year. Each hospital has a ranking of the students in order of preference, and each student has a ranking of the hospitals in order of preference. Assume that there are more students graduating than there were slots available in the m hospitals.

We want to find a way of assigning each student to at most one hospital, in such a way that all available positions in all hospitals were filled. Some students will not be assigned.

We will say that an assignment of students to hospitals is stable if neither of the following instabilities occurs.

- Instability 1: There are students s and s', and a hospital h, so that
  - s is assigned to h, and
  - s' is assigned to no hospital, and
  - h prefers s' to s.
- Instability 2: There are students s and s', and hospitals h and h', so that
  - s is assigned to h, and
  - s' is assigned to h', and
  - h prefers s' to s, and
  - s' prefers h to h'.

Basically, we have the Stable Matching Problem, except that hospitals may want more than one resident, and not all students will be assigned.

(a) Give an algorithm to find a stable matching. Use pseudocode similar to the book (for example, the Gale-Shapley algorithm on page 6) .

(b) Prove that the algorithm finds a stable matching.

(c) What is the worst-case running time of the algorithm (Big O)?

Solution. Consider a scenario where there exists a set of hospitals, denoted as H. Each hospital, represented by $h \in H$, has a specific number of positions to fill,

denoted by p. The set of all positions to be filled across all hospitals is denoted as P. Mathematically, we can express this as:

$$P = \{(h, i) \mid h \in H, 1 \leq i \leq p_h\}$$

Here, p_h represents the number of positions available at hospital h.

Let S be the set of students, where each student is denoted as $s \in S$. A student can apply to one or more hospitals.

The total number of positions to be filled, represented by the cardinality of P, is denoted by $|P|$, and it is required that $|P| < |S|$, meaning that the total combined number of available positions across all hospitals is less than the total number of students.

Algorithm

Initially, all hospitals in H have no slots occupied, they are considered "free". Similarly, all students $s \in S$ are free and not yet "matched" to any hospital. A student becomes "matched" to a hospital when they accept an offer, which is not necessarily a commitment but rather an initial engagement. A hospital can either have available positions or be deemed "full".

When a slot in a hospital h is matched, the count of free slots decreases by one. Conversely, the number of free slots in a hospital can increase if a student initially accepts an offer from hospital h, then later receives an offer from another hospital h', leading the student to "leave" h for h'. Consequently, the number of free slots in hospital h increases by one. The algorithm is based on the Gale-Shapley algorithm in [1] and is described in detail in Algorithm 1.

Proof

To prove that the algorithm finds a stable matching, we need to show two key properties: completeness and stability.

1. Completeness

    - The algorithm concludes its execution when all hospitals are fully occupied, meaning there are no available positions. This termination condition aligns with our initial assumption that the total number of available positions across all hospitals is less than the total number of graduating students, $|P| < |S|$ or equivalently $\sum_{i=1}^{m} p_i < n$.

Algorithm 1: Stable Matching Algorithm with Multiple Positions

Data: Preference lists of hospitals and students
Result: Stable matching of hospitals to students

foreach *hospital* h **do**
  Initialize h with available positions and no assigned students;
end
while *there exists a hospital* h *with available positions* **do**
  Find the highest-ranked student s in h's preference list;
  if s *is unmatched* **then**
    Assign s to h;
    Decrease the number of available slots in h by one;
  end
  else
    h' is the hospital currently assigned to s;
    if s *prefers* h' *to* h **then**
      s remains committed to h';
    end
    else
      Assign s to h;
      Decrease the number of available slots in h by one;
      Increase the number of available slots in h' by one;
    end
  end
end

- Every student is either matched to a hospital or remains unallocated.

2. Stability

   - Stability is established by examining the conditions under which a student and hospital could be involved in blocking pairs. A blocking pair occurs when both the student and the hospital prefer each other over their current matches.
   - In the algorithm:
     - If a student s is unmatched, they are assigned to their most preferred hospital with available slots. This assignment is stable as there are no existing commitments to be broken.
     - If s is already matched to a hospital h', and s prefers a new hospital h over h', s is reassigned to h and h''s available slots increase. This

reassignment maintains stability by ensuring that s is matched with their most preferred hospital.
  - If s prefers their current match h' over a new hospital h, there is no reassignment, and stability is maintained.

Besides, consider the two types of instabilities individually.

Instability 1

Consider the case where there exist students s and s', and a hospital h, such that:

- s is assigned to h,
- s' is assigned to no hospital, and
- h prefers s' to s.

The algorithm handles this case as follows:

- Before s is assigned to h, h must have proposed to s', since s' is higher in h's preference list. This means that h has expressed its interest in s' before settling for s.
- If s' accepted h's proposal, then s' would be assigned to h and not unmatched. This would make h and s' happy, and s would remain free to look for another hospital.
- If s' rejected h's proposal, then s' must have preferred another hospital h' to h, and h' must have had an available position for s'. This means that s' has found a better match than h, and h' has agreed to take s.
- Therefore, s' cannot be unmatched and preferred by h over s, and Instability 1 is avoided. This ensures that no hospital will have an empty slot while there is a student that it likes more than one of its current students.

The algorithm is trying to avoid a situation where a hospital h has an available position, but there is a student s' who is unmatched and preferred by h over one of its assigned students s. This would be unfair to both h and s', because they could be better off if they were matched together.

Instability 2

Consider the case where there exist students s and s', and hospitals h and h', such that:

- s is assigned to h,
- s' is assigned to h',
- h prefers s' to s, and
- s' prefers h to h'.

The algorithm handles this case as follows:

- Before s' is assigned to h', h must have proposed to s', since s' is higher in h's preference list. This means that h has expressed its interest in s' before settling for s.
- If s' accepted h's proposal, then s' would be assigned to h and not h'. This would make h and s' happy, and h' and s would remain free to look for other partners.
- If s' rejected h's proposal, then s' must have preferred h' to h, which contradicts the assumption that s' prefers h to h'. This means that s' has not found a better match than h, and h' has not agreed to take s'.
- Therefore, s' cannot be assigned to h' and preferred by h over s, and Instability 2 is avoided. This ensures that no pair of students and hospitals will have an incentive to break their current match and form a new one.

The algorithm aims to prevent a scenario where there are two pairs of students and hospitals, denoted as s and h, and s' and h', such that both pairs are unhappy with their current match and would prefer to switch partners. This would be deemed unfair to both pairs, as they could potentially achieve better outcomes if their matches were different.

By addressing both types of instabilities, the algorithm guarantees a stable matching of hospitals to students.

Big O

The key operations of the algorithm include iterating through each hospital (m iterations), finding the highest-ranked student in a hospital's preference list (up to n comparisons for each hospital), and updating the assignments and available slots. We can analyze it as follows:

- The outer loop runs until all hospitals have filled their available positions. In the worst case, this is when all hospitals have assigned all their positions, which is *O(m)*, where m is the number of hospitals.
- Inside the loop, the algorithm may need to go through the list of all students for a given hospital to find the highest-ranked student who is not yet matched. This is *O(n)*, where n is the number of students.

Therefore, the worst-case running time is $O(m \cdot n)$. This is because for each hospital (m), we may need to check each student (n), and these operations could potentially be nested, resulting in a multiplicative effect.

Moreover, when considering the number of available slots in hospitals as an input parameter of the problem, different hospitals can have varying numbers of positions. Let $p_{\max}$ be the maximum number of positions in any hospital, so the worst-case time complexity in this case is refined to $O(m \cdot n \cdot p_{\max})$. This is because each hospital independently goes through its preference list until all its positions are filled. Hence, the time complexity is governed by the hospital with the most positions.

Problem 3

Take the following list of functions and arrange them in ascending order of growth rate. That is, if function $g(n)$ immediately follows function $f(n)$ in your list, then it should be the case that $f(n)$ is $O(g(n))$. You do not need to prove the order is correct. All logarithms are base 2.

- $f_1(n) = n^2 \log n$
- $f_2(n) = n^{5/3}$
- $f_3(n) = 2^{\log n}$
- $f_4(n) = 2^{4n}$
- $f_5(n) = n(\log n)^4$
- $f_6(n) = n^{\log n}$
- $f_7(n) = n \log n$
- $f_8(n) = 10^n$
- $f_9(n) = n^{2.5}$

Solution. The big O notation of each function is:

- $f_1(n) = n^2 \log n = O(n^2 \log n)$
- $f_2(n) = n^{5/3} = O(n^{5/3})$
- $f_3(n) = 2^{\log n} = O(n)$
- $f_4(n) = 2^{4n} = O(2^{4n})$
- $f_5(n) = n(\log n)^4 = O(n(\log n)^4)$
- $f_6(n) = n^{\log n} = O(n^{\log n})$
- $f_7(n) = n \log n = O(n \log n)$
- $f_8(n) = 10^n = O(10^n)$
- $f_9(n) = n^{2.5} = O(n^{2.5})$

For $f_3(n) = 2^{\log n}$, because the base of the logarithm is 2 and the exponentiation is also with base 2, the logarithmic term and the exponential term cancel each other out, resulting in $O(n)$ and represents a linear growth rate.

For $f_7(n) = n \log n$, the growth rate is $O(n \log n)$, which is greater than linear but less than quadratic. Therefore, $f_3(n)$ comes before $f_7(n)$, since $n < n \log n$.

For $f_2(n) = n^{5/3}$, this function represents a polynomial growth, which is faster than any linear or logarithmic growth. The exponent $5/3$ is greater than 1 (which would represent linear growth), but less than 2. Hence, $f_7(n)$ comes before $f_2(n)$.

For $f_5(n) = n(\log n)^4$, grows at a rate proportional to n times the fourth power of the logarithm of n. The polylogarithmic term slows down the growth compared to pure polynomial functions.

For $f_1(n) = n^2 \log n$, this function represents a polynomial term n^2 multiplied by a logarithmic term $\log n$. As n becomes very large, n^2 grows faster than n, and thus $n^2 \log n$ increases faster than $n(\log n)^4$. Therefore, $f_1(n)$ ranks after $f_5(n)$.

For $f_9(n) = n^{2.5}$, this function represents a polynomial growth rate, but with a higher exponent than $f1_(n)$, hence it grows faster.

For $f_6(n) = n^{\log n}$, this function represents an exponential growth, where the exponent itself $(\log n)$ increases with n. Exponential growth is faster than polynomial growth, so $f_6(n)$ is ranked after $f_9(n)$ as $f_6(n)$ grows faster than $f_9(n)$ as n approaches infinity.

For $f_8(n) = 10^n$, this function represents exponential growth with a base of 10. Exponential growth surpasses polynomial and polylogarithmic growth rates.

For $f_4(n) = 2^{4n}$, this function represents exponential growth with a base of 2. The exponent is a multiple of n, which makes the growth rate much faster than the other exponential functions.

$$f_3(n) < f_7(n) < f_2(n) < f_5(n) < f_1(n) < f_9(n) < f_6(n) < f_8(n) < f_4(n)$$

Thus, in ascending order of growth rate, the functions are arranged as follows: $f_3(n), f_7(n), f_2(n), f_5(n), f_1(n), f_9(n), f_6(n), f_8(n), f_4(n)$.

Problem 4

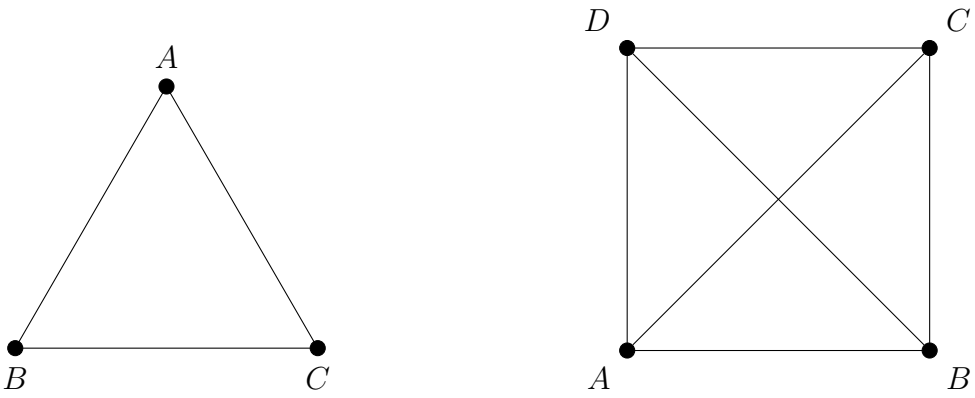
Prove by induction that completely connected undirected graph with n nodes (an edge exists from every node to every other node, but no self-edges) has $\frac{n(n-1)}{2}$ edge for $n > 1$.

Note: Be careful to state what value you are performing induction on and clearly show the basis step, the inductive hypothesis, the inductive conclusion, and where the inductive hypothesis is used to prove the inductive conclusion.

Solution. This is a proof by induction that a completely connected undirected graph with n nodes has

$$\frac{n(n-1)}{2}$$

edges with $n > 1$. Figure 1 illustrates some completely connected undirected graphs.



(a) 3 nodes, 3 edges

(b) 4 nodes, 6 edges

Figure 1: Completely connected undirected graphs with different numbers of nodes and edges.

Induction Base Case

For $n = 2$, a completely connected undirected graph has one edge between the two nodes. This is consistent with the formula

$$\frac{2 \cdot (2-1)}{2} = \frac{2 \cdot 1}{2} = \frac{2}{2} = 1$$

Figure 2: Completely connected undirected graph with two nodes (A and B).

Induction Hypothesis

Assume that the statement is true for some $k > 2$, that is, a completely connected undirected graph with k nodes has

$$\frac{k(k-1)}{2}$$

edges.

Induction Conclusion

Show that the statement is true for $k+1$, that is, a completely connected undirected graph with $k+1$ nodes has

$$\frac{(k+1)(k+1-1)}{2} = \frac{k(k+1)}{2}$$

edges.

Proof

Let G be a completely connected undirected graph with $k+1$ nodes. If we remove any node from G, such as x, we get a completely connected undirected graph with k nodes, which we call G'. By the inductive hypothesis, G' has

$$\frac{k(k-1)}{2}$$

edges. Now, to get G from G', we need to add x back and connect it to every other node in G'. This means we need to add k edges to G' (since in a completely connected undirected graph, each node is connected to every other node). Therefore, the total number of edges in G is

$$\frac{k(k-1)}{2} + k = \frac{k(k-1)+2k}{k} = \frac{k(k-1+2)}{2} = \frac{k(k+1)}{2}$$

, which is the desired result (the inductive conclusion or what must be proven).

Hence, by mathematical induction, the statement is true for all $n > 1$.

Problem 5

Suppose you have interviewed a set of people regarding their pets (all of whom are now passed away). The pets are $P_1, P_2, \ldots, P_n$. The people you have interviewed provide with two types of facts:

- For some i and j, P_i passed away before P_j was born.
- For some i and j, P_i and P_j lived (at least for a while) at the same time.

Owing to fallible memories, these facts may not all be correct. You want to determine if all of the facts can all be true simultaneously.

(a) Find an efficient algorithm to determine whether all of these facts can be true at the same time. A description of the algorithm is sufficient. Full pseudocode is unnecessary.

(b) What is the worst-case running time of the algorithm if there are n pets and m facts?

Hint: If two pets, P_i and P_j, have overlapping life spans, then P_i was born before P_j died and vice versa. These are precedence relationships. What should you use to represent precedence relationships?

Solution. This problem can be solved by using a directed acyclic graph (DAG) to represent the precedence relationships. Each pet can be represented as a node in the graph, and an edge from node P_i to node P_j represents that P_i was born before P_j died.

The algorithm can be described as follows:

1. Create a directed graph G with n nodes, each representing a pet.
2. For each fact, add an edge between the corresponding nodes in the graph G.
    - If P_i passed away before P_j was born, add a directed edge from P_i to P_j.
    - If P_i and P_j lived at the same time, add directed edges in both directions between P_i and P_j.
3. Check if the graph G is a DAG. We can use topological sorting or depth-first search to detect cycles in the graph.
    - If the graph contains a cycle, then there is a contradiction in the facts, and they cannot all be true simultaneously.

- If the graph does not contain a cycle, then the facts can all be true simultaneously.

Algorithm 2: Pet Facts Consistency Algorithm using Topological Sorting

Data: Set of pets $P_1, P_2, \ldots, P_n$ and facts of two types
Result: Check if all facts can be true simultaneously

Create an empty directed graph G;
foreach *fact of the first type (P_i passed away before P_j was born)* **do**
  Add directed edge from P_i to P_j in graph G;
end
foreach *fact of the second type (P_i and P_j lived at the same time)* **do**
  Add directed edge from P_i to P_j in graph G;
  Add directed edge from P_j to P_i in graph G;
end
while *graph G is non-empty* **do**
  Find a node v in G with in-degree 0;
  if *no such node exists* **then**
    return *graph G contains a cycle (conflicting facts)*;
  end
  Remove v and its outgoing edges from G;
end
return *all facts can be true simultaneously*;

The worst-case running time of this algorithm is $O(n+m)$, where n is the number of pets (nodes in the graph) and m is the number of facts (edges in the graph). This is because each node and edge is visited once during the topological sort. If a cycle is detected, the algorithm can terminate early. If no cycle is detected, all nodes and edges will be visited. Therefore, the time complexity is linear with respect to the size of the graph.

Problem 6

You live upon a straight residential street upon which you are trying to provide wireless internet for every house. Each house must be within 100 m of a wireless router to access it. Given an unsorted list of house locations $h_1, h_2, \ldots, h_n$ (distances from the beginning of the street), give an efficient algorithm for placing router locations such that the fewest possible routers is used. They do not need to be at a house location. Prove that your algorithm uses as few routers as possible. Give the worst-case running time of your algorithm.

Solution. To solve this problem, we can use a greedy algorithm that places routers at the minimum distance required to cover all the houses. The algorithm is described as follows:

1. Sort the house locations $h_1, h_2, \ldots, h_n$ in ascending order.
2. Initialize an empty list to store the router locations.
3. Place the first router r_1 at the location of the first house $h_1 + 100$ meters.
    - This ensures that the current house and potentially some of the next houses are within the range of this router.
4. Iterate through the sorted list of house locations.
    - For each house location h_i, check if it is more than 100 m away from the last router location.
    - If it is, place a new router r_{k+1} at the current house location $h_i + 100$ and update the last router location r_k to the new router location r_{k+1}.

The algorithm starts by placing the first router at the initial house location plus 100 meters, ensuring coverage for the first house and potentially some neighboring houses. As it iterates through sorted house locations, it checks if a house is more than 100 meters away from the last router. If so, a new router is placed at the current house location plus 100 meters, extending coverage to subsequent houses. This approach minimizes the number of routers needed while ensuring each house is covered.

Algorithm 3: Router Placement Algorithm

Data: List of house locations $h_1, h_2, \ldots, h_n$
Result: List of router locations

Sort the house locations $h_1, h_2, \ldots, h_n$ in ascending order;
Initialize an empty list L to store the router locations;
Place the first router r_1 at the location of the first house $h_1 + 100$ meters;
$r_k \leftarrow r_1$;
for $i \leftarrow 2$ **to** n **do**
    if $h_i > r_k + 100$ **then**
        Add r_{k+1} at $h_i + 100$ to the list L;
        Update the last router location r_k to the new router location r_{k+1};
    end
end

Greedy Stays Ahead

To prove that this algorithm is optimal, we can use a greedy stays ahead argument. This style of proof works by showing that, according to some measure, the greedy algorithm always is at least as far ahead as the optimal solution during each iteration of the algorithm.

Consider the optimal solution O, and let $r_1, r_2, \ldots, r_m$ be the routers in the optimal solution,where m is the minimum number of routers needed.

Now, consider the solution produced by our greedy algorithm, denoted as G, with routers $s_1, s_2, \ldots, s_n$, where n is the number of routers placed by the algorithm.

We want to show that for every router s_i in G, there exists a corresponding router r_j in O such that $s_i \leq r_j$ (i.e., the placement of routers by the greedy algorithm stays ahead or at the same position as the optimal solution).

The base case is when $i = 1$. In this case, G places the first router at the location of the first house h_1 plus 100 meters. This covers the first house and possibly some of the next houses. Since any solution must place at least one router to cover the first house, G has placed at most as many routers as O.

For the inductive step, assume that the claim holds for some $i \geq 1$. Now consider the $i + 1^{\text{th}}$ router. G places this router at the location of the first house (and plus 100 meters) that is not covered by the previous routers, which is more

than 100 m away from the last router location. This covers the current house and possibly some of the next houses. Since any solution must place at least one router to cover the current house, G has placed at most one more router than O. By the inductive hypothesis, G had placed at most as many routers as O before, so G still has placed at most as many routers as O after placing the $i + 1^{\text{th}}$ router.

Therefore, by induction, the claim holds for any i. It follows that, when all the houses are covered, G has placed at most as many routers as O. Since O is optimal, G must also be optimal.

Contradiction

Assume that there exists an optimal solution that uses fewer routers than our algorithm. We denote the router locations in this optimal solution as $r_1, r_2, \ldots, r_m$ and the router locations in our solution as $s_1, s_2, \ldots, s_n$. Since our solution is greedy, it places the first router as far to the right as possible while still covering the first house. Therefore, $s_1 \geq r_1$.

Now, consider the second router. In the optimal solution, r_2 must be placed to the right of r_1 to cover more houses. Similarly, in our solution, s_2 is placed to the right of s_1 to cover more houses. Since $r_1 \geq s_1$ and both r_2 and s_2 are placed to cover houses to the right of s_1 and r_1 respectively, we have $s_2 > r_2$.

By continuing this argument, we can show that for all i, $s_i \geq r_i$. This means our solution covers at least as many houses with each router as the optimal solution, so it uses no more routers than the optimal solution. Therefore, our solution is optimal.

Big O

The worst-case running time of our algorithm is $O(n \log n)$, where n is the number of houses. This is because we need to sort the house locations in ascending order, which takes $O(n \log n)$ time. The rest of the algorithm takes $O(n)$ time, since we iterate through the sorted list of house locations once.

References

[1] J. Kleinberg and E. Tardos, *Algorithm Design*. USA: Addison-Wesley Longman Publishing Co., Inc., 2005, ISBN: 0321295358.