CPSC 320 2018W2: Assignment 1

Members:

1. Email: **c0t2b@ugrad.cs.ubc.ca**, Student number: **79507828**

"All group members have read and followed the guidelines for groupwork on assignments in CPSC320. As part of those rules, when collaborating with anyone outside my group, (1) I and my collaborators took no record but names (and GradeScope information) away, and (2) after a suitable break, my group created the assignment I am submitting without help from anyone other than the course staff."

1 SMP and Gale-Shapley

1. Let $\#E_k$ be the number of employers matched at the end of the kth iteration of the While loop on some execution of the G-S algorithm. If there are at least $k+1$ iterations of the While loop, must it be the case that $\#E_k < \#E_{k+1}$?

   No, at the iteration $k+1$ the number of employers matched may be increased by 1, but it can also remains the same $\#E_k$.

   Suppose the employer from the $k+1$ iteration e_{k+1} prefers some applicant a that has already been matched with some employer e, there are two possible cases:

   (a) **Case 1:** a prefers e_{k+1} over e, then the applicant a will be matched with e_{k+1} and employer e will now be unmatched, so the number of employers matched at the end of $k+1$ iteration will be the same as the previous kth iteration, because there were a new match but one unmatch on the iteration $k+1$.

   (b) **Case 1:** a does not prefers e_{k+1} over e, then the applicant a will still be matched with e, and e_{k+1} will remain unmatched, so the number of employers matched at the end of $k+1$ iteration will be the same as the previous kth iteration, because there were no new matches on the iteration $k+1$.

2. Let Set-A_k be the set of applicants matched at the end of the kth iteration of the While loop on some execution of the G-S algorithm. If there are at least $k+1$ iterations of the While loop, must it be the case that Set-$A_k \subseteq$ Set-A_{k+1}?

   Yes, Set-A_k may be a subset of Set-A_{k+1} if not the same set as Set-A_{k+1}.

   On each iteration we have only two possible cases:

   (a) **Case 1:** the applicant a that will be matched on the kth iteration was not considered before by another employer, he will be added to the Set-A_k. So Set-A_k will be a subset of Set-A_{k+1}

   (b) **Case 2:** the applicant a was previously matched with another employer, so according to his preferences he is either gonna be matched with employer e or remain with is matched pair, so the Set-A_{k+1} will be exactyl the same as Set-A_k.

3. Consider a variant of SMP in which each of n employers has exactly **two** positions, and the number of applicants is $2n$. As a function of n, how many valid solutions are there?

   It would be $2n$ spaces available for n applicants, so the number of valid solutions will be $2n * n$ that can be translated to $2n^2$.

2 Progressing Towards Goodness in Gale-Shapley

Prove the following claim, from part 8.1 of the Stable Matching Problem Part II worksheet. You could use a proof by induction, with a structure that is modeled on the proof of problem 5.2 of the worksheet.

Claim: At the end of every iteration k of the While loop of Algorithm G-S, every employer e has only considered applicants that it ranks at least as high as best(e). Moreover, if e has considered best(e) on or before iteration k, then e is matched with best(e) at the end of iteration k.

Suppose at iteration k, employer e considers applicant a

1. **Base case:** If $k = 1$, at the end of this iteration M has only 1 pair (e, a). The Algorithm G-S matches the current employer e with his first preference. So e has only considered a, wich ranks at least as high as best(e), because best(e) is equal to a.
2. **Case 1:** a is unmatched: At the end of the iteration e will be paired with a. Applicant a is the best(e) because it has not been match by a previous employer and there are no applicants in e preference list that rank higher than a and are a stable matching with e. Also previous applicants that e has considered will rank at least as high as a on e's preference list.
3. **Case 2:** a is matched with e' but prefers e over e': At the end of the iteration e will be paired with a. The applicant a is best(e) because the pair (a, e) is a stable matching and there is not another applicant, with a stable matching pair, whom e ranks higher than a.
4. **Case 3:** a is matched with e' and do not prefer e over e': At the end of the iteration e will be unpaired. The applicant a is not best(e) because the pair (a, e) is not a stable matching. So e do not considered best(e), therefore it won't be matched with any applicant on the kth iteration.

The iteration must match with one of the 4 cases above, and all preserve the claim, that every employer has only considered applicants that rank at least as high as best(e), and if e considered best(e) on the iteration k it will end the iteration by being match with best(e).

3 SMP with Identical Preference Lists

For positive integers n, let I_n be the instance of SMP with n employers and n applicants such that every employer has the same preference list, and also every applicant has the same preference list, namely:

$$e_i : a_1, a_2, \ldots, a_n \qquad\qquad a_i : e_1, e_2, \ldots, e_n.$$

Let S be the (infinite) set of instances $\{I_n, n > 0\}$.

1. Write down the instance I_2. (No justification needed.)

$$e_1 : a_1, a_2 \qquad\qquad a_1 : e_1, e_2$$

$$e_2 : a_1, a_2 \qquad\qquad a_2 : e_1, e_2$$

2. Show a good solution for the instance I_2. (No justification needed.)

$$e_1 - - - a_1$$

$$e_2 - - - a_2$$

3. Prove that for any $n > 0$, in the instance I_n we have best$(e_i) = a_i$, for $1 \le i \le n$.
All employers in I_n instances will have their preference list sorted as the order of the applicants, and this also applies to the preference of the applicants. Given that order of the elements in the preference lists, the best(e_i) is always the applicant a_i. Applicant a_{i-1} would be ranked higher than a_i in e_i's preference list, but it would not be an stable matching between e_i and a_{i-1}, because the applicant will already be matched with employer e_{i-1} which is ranked higher than e_i on a_{i-1}'s preference list. Therefore the best(e_i) would be always a_i, for $1 <= i <= n$

4 Faster or Slower

Suppose that some algorithm A has running time $f(n)$ and that algorithm B has running time $g(n)$, on all inputs of size n. Assume that f and g are functions $\mathbb{N} \to \mathbb{N}^+$, and that $\lim_{n\to\infty} f(n)$ and $\lim_{n\to\infty} g(n)$ are both infinity. Explain whether each statement in parts 2 and 3 below is true or false. Part 1 is already done for you.

1. For some choice of $g(n)$ with $g(n) \in \Omega(f(n)\log n)$:
   (a) A is faster than B on all sufficiently large inputs.
   SOLUTION True. Choosing $g(n) = f(n)(\lceil \log_2 n \rceil + 1)$ satisfies the condition that $g(n) \in \Omega(f(n)\log n)$. For this choice, $g(n) > f(n)$ for all n, and so B is slower than A on all inputs.
   (b) A is slower than B on all sufficiently large inputs.
   SOLUTION False. For all choices of g with $g(n) \in \Omega(f(n)\log n)$, we have that $g(n) > f(n)$ for sufficiently large n. So B is slower than A on all sufficiently large inputs.
   (c) A is faster than B on some inputs, and slower than B on other inputs.
   SOLUTION True. Let $n_1 < n_2$ be such that $f(n_1) > 1$ and $f(n_2) > 1$. Choose $g(n) = 1$ for $n \le n_2$ and $g(n) = f(n)\lceil \log_2 n \rceil$ for $n > n_2$. Then $g(n) \in \Omega(f(n)\log n)$, $g(n) > f(n)$ for all $n > n_2$, and $g(n) < f(n)$ for n_1 and n_2. So B is faster than A on inputs n_1 and n_2, while being slower than A on all inputs of size greater than n_2.

2. For some choice of g such that $g(n) \in \Theta(f(n))$:
   (a) A is faster than B on all sufficiently large inputs.
   True, choosing $g(n) = 2f(n)$ satisfies the condition of $g(n) \in \Theta(f(n))$. For this choice $f(n) < g(n)$ is true because if we replace this $g(n)$ with $2f(n)$, we get $f(n) < 2f(n)$ this is reduced to $1 < 2$. So A will be faster than B.
   (b) A is slower than B on all sufficiently large inputs.
   True, choosing $g(n) = \frac{1}{2}f(n)$ satisfies the condition of $g(n) \in \Theta(f(n))$. For this choice $f(n) > g(n)$ is true because if we replace this $g(n)$ with $\frac{1}{2}f(n)$, we get $f(n) < \frac{1}{2}f(n)$ this is reduced to $1 > \frac{1}{2}$. So A will be slower than B.
   (c) A is faster than B on some inputs, and slower than B on other inputs.
   True, let $n_1 < n_2$ be such that $f(n_1) > 1$ and $f(n_2) > 1$. Choosing $g(n) = 2f(n)$ for $n <= n_2$ and $g(n) = \frac{1}{2}f(n)$ for $n > n_2$. This satisfies the condition of $g(n) \in \Theta(f(n))$. Then $f(n) < g(n)$ for n_1 and n_2, and $f(n) > g(n)$ for all $n > n_2$. So A will be faster than B on some inputs n_1 and n_2, and B will be faster on the other inputs that are greater than n_2

3. For some choice of g such that $g(n) \in o(f(n))$:

   (a) A is faster than B on all sufficiently large inputs.
   False, for all choices of g in $g(n)$ we have that $f(n) > g(n)$ for sufficiently large inputs. So A will be slower than B on all sufficiently large inputs.

   (b) A is slower than B on all sufficiently large inputs.
   True, choosing $g(n) = \frac{1}{2}f(n)$ satisfies the condition $g(n) \in o(f(n))$. Then we have that $f(n) > g(n)$, replacing $g(n)$ we have $f(n) > \frac{1}{2}f(n)$ which will be reduced to $1 > \frac{1}{2}$ which is true. Therefore A will be slower than B.

   (c) A is faster than B on some inputs, and slower than B on other inputs.
   False, for all choices of g in $g(n)$ we have that $f(n) > g(n)$ for sufficiently large inputs. Therefore A will never be slower than B because $g(n)$ must satisfy this condition $g(n) \in o(f(n))$

5 Comparing Substrings

Here you'll evaluate running times of algorithms whose input size is expressed using two parameters.

Let $T[1..n]$ and $T'[1..n]$ be strings of length n, over a finite alphabet (For example, T and T' might be over the alphabet $\{A, C, G, T\}$, and represent DNA strands.) A function Match indicates whether or not a letter of T matches a letter of T'. That is, for $1 \leq i, j \leq n$,

$$\begin{aligned} \text{Match}(i,j) \quad &= 1, \quad \text{if } T[i] = T'[j] \\ &= 0, \quad \text{otherwise.} \end{aligned}$$

Fix $k, 1 \leq k \leq n$. For $1 \leq i, j \leq n-k+1$, the *score* of any two length-k substrings $T[i..i+k-1]$ and $T'[j..j+k-1]$ of T and T' respectively is given by

$$\sum_{l=0}^{k-1} \text{Match}(i+l, j+l).$$

Algorithm *Compute-Scores* below computes all scores and stores them in a two-dimensional array called Score. Assume that calls to function Match take $\Theta(1)$ time and that array Score has already been created.

Algorithm *Compute-Scores* $(T[1..n], T'[1..n], k)$
  // T and T' are length-n strings and $1 \leq k \leq n$
  For i from 1 to $n-k+1$
    For j from 1 to $n-k+1$
      // compute score of $T[i..i+k-1]$ and $T'[j..j+k-1]$ and store in Score$[i,j]$
      Score$[i,j] = \sum_{l=0}^{k-1}$ Match$(i+l, j+l)$

1. What terms below describe the worst-case running time of this algorithm? Check all answers that apply. Here, a term $\Theta(f(n,k))$ is correct if for any choice of k in the range $[1..n]$, the algorithm runs in $O(f(n,k))$ time, and for some choice of k in the range $[1..n]$ (where k may be expressed as a function of n), the algorithm runs in $\Omega(f(n,k))$ time.

   ○ $\Theta(k^3)$ ● $\Theta((n-k)^2k)$ ● $\Theta(n^2k)$ ○ $\Theta(n^3)$

2. Modify the algorithm of part 1, to improve the runtime by a factor of k.

Algorithm *Compute-Scores2* $(T[1..n], T'[1..n], k)$
// Initialize scores with the substring score of the first column and row
For i from 1 to $n-k+1$
  Let scoreRow $= 0$
  Let scoreCol $= 0$
  For l from 0 to $k-1$
    scoreRow $+=$ Match$(i+l, l)$
    scoreCol $+=$ Match$(l, i+l)$
  Score$[i][1]$ = scoreRow
  Score$[1][i]$ = scoreCol

// Compute the other scores using first row and column scores
For i from 2 to $n-k+1$
  For j from 2 to $n-k+1$
    Score$[i][j]$ = Score$[i-1][j-1]$ − Match$(i-1, j-1)$ + Match$(i+k-1, j+k-1)$

3. What is the worst-case running time of your algorithm of part 2? Try to find as simple an expression as possible.

$$O(n(n-k))$$

The first section from the algorithm has a runtime of $(n-k)k$, and the second has a runtime of $(n-k)(n-k)$, so the full runtime is $(n-k)(n-k)+(n-k)k$. This can be reduced by:

$$(n-k)(n-k)+(n-k)k$$
$$n^2-2nk+k^2+nk-k^2$$
$$n^2+nk=n(n+k)$$