COMPSCI 311: Introduction to Algorithms **Spring 2020**

Homework 1

Released 1/26/2020 Due 2/10/2020 11:59pm in Gradescope

Instructions. You may work in groups, but you must write solutions yourself. List collaborators on your submission. Also list any sources of help (including online sources) other than the textbook and course staff.

If you are asked to design an algorithm, please provide: (a) the pseudocode or precise description in words of the algorithm, (b) an explanation of the intuition for the algorithm, (c) a proof of correctness, (d) the running time of your algorithm and (e) justification for your running time analysis.

Submissions. Please submit a PDF file. You may submit a scanned handwritten document, but a typed submission is preferred. Please assign pages to questions in Gradescope.

1. **(30+5 points) Variations on Stable Matching**

   (a) (10 points) Consider a variant of stable matching in which at every point, either a *free* college *or* a *free* student can propose. As in the Gale-Shapley algorithm, proposals are done going down in the preference list, so that a proposer cannot repeat a proposal to the same partner. Show that this algorithm always terminates with a perfect matching, but not necessarily a stable one.[1]

   In this algorithm, each college and student has a preference list of n elements as in Gale-Shapley algorithm, and will run through the list with no backturn. Also, one college only matches one student. So, a college and a student will propose to at most n students/colleges, means at most n^2 proposals can occur, means the number of cases is finite, so it will terminate.

   1: A C B A: 3 2 1
   2: B C A B: 1 3 2 $1A \Rightarrow B1 discard \Rightarrow B3 \Rightarrow 2B discard \Rightarrow 2C$, so result is $1A, 2B, 3C$
   3: A C B C: 3 2 1

   Here, we have pair $3A$ as unstable match, because 3 perfers A to C, and A perfers 3 to 1.

   (b) (5 points extra credit) Consider colleges A B, C and students 1, 2, 3, with preference lists:
   1: A B C A: 2 3 1
   2: B C A B: 3 1 2
   3: C A B C: 1 2 3

   How many of the six perfect matchings are stable? How many stable matchings can be obtained by a version of part (a) where colleges and students alternate proposing (any party can start)?

   $1A, 2B, 3C$ stable, $1A, 2C, 3B$ unstable on pair $3A$, $1B, 2A, 3C$ unstable on pair $2C$, $1B, 2C, 3A$ stable, $1C, 2A, 3B$ stable, $1C, 2B, 3A$ unstable on pair $1B$ So, three of the six perfect matchings are stable.

   If start from college, matching $1A, 2B, 3C$ will be obtained, if start from students, matching $1C, 2A, 3B$ will be obtained, because for colleges and students, their first ranking has no conflict, so they will always select the first on their rankings, so the total number is 2.

   (c) (10 points) Now allow proposals also from matched parties. That is, on any turn, any college or student may propose to the next partner on its preference list if this is better than the current match (if any), and if the proposal is accepted, both the proposer's and the acceptor's former partner (if any) become free. Does the algorithm always terminate? Will it always produce a perfect matching? Will it always produce a stable matching? (Hint: a college c might not get a student s when s already has a better match, but be offered by s later if s becomes unmatched. Look at $n = 3$ first).

[1] We are no longer confident that the procedure with only free parties proposing always yields a perfect matching. We have also failed to find an example where it doesn't. You will get full credit for exhibiting an example where this procedure fails to find a stable matching. There may be extra credit available if a student or students resolve the question of whether the matching from the procedure is always perfect.

The colleges and students will only try to match on the partners which has not been proposed on their lists. So, the result is, still for each college and students, they will match at most n proposals, gives $O(n^2)$ which is finite, so the program will terminate.

Suppose that some student or college is not matched upon termination, meaning this student / college has never been matched. This means there is one college / student that also remains unmatched. Because the selection is double sided, it infers that these two college-student have run through the perference lists but not matched, and also they did not receive any proposal, these two conditions are contradictory, so it will always produce a perfect matching.

Suppose P, Q is an unstable pair, each perfers other to the current partner Q', P' in the result S. By defination, we know that the last matches for P, Q are Q', P' which forms $(P, Q'), (P', Q)$, so there are two cases.

Case 1: P, Q has never proposed to each other. Because everyone runs the perference list from the most prefer one to the least prefer one, before get to $(P, Q'), (P', Q)$, they must have tried to propose each other. It is contradictory to the premises

Case 2: P, Q has proposed to each other at least from one side Suppose P proposed first, Q rejected P, means Q perfers P' to P, which is contradictory to the premises It is also true on the reversed version So, this algorithm will produce a stable matching

(d) (10 points) Consider now an algorithm that starts with an arbitrary perfect matching of colleges to students. As long as the matching is not stable, choose an instability (c, s) and eliminate it, by matching c with s, and the former partners of c and s with one another. Show that this algorithm does not always terminate with a stable matching. (Hint: an example with $n = 3$ suffices).

2. **(15 points) Big-O.** For each function $f(n)$ below, find (1) the smallest *integer* constant H such that $f(n) = O(n^H)$, and (2) the largest *positive real* constant L such that $f(n) = \Omega(n^L)$. Otherwise, state that H or L do not exist. All logarithms have base 2. Your answer should consist of: (1) the correct value of H, (2) a proof that $f(n)$ is $O(n^H)$, (3) the correct value of L, (4) a proof that $f(n)$ is $\Omega(n^L)$.

(a) $f(n) = \frac{n \log n}{\log(\log n)}$

$\log(\log n) > 1, n \log n < n^2$, so $f(n) = \frac{n \log n}{\log(\log n)} <= n^2 = g(n)$ for all $n >= 3$, so $f(n) = O(n^2), H = 2$ $\log(\log n) < \log n$ so $f(n) = \frac{n \log n}{\log(\log n)} >= n$ for all $n >= 0$, so $f(n) = \Omega(n), L = 1$

(b) $f(n) = \sum_{k=1}^{n} \sqrt{k} \cdot \sqrt{n-k}$

$f(n) = n\sqrt{k} \sum_{k=1}^{n} \sqrt{n-k}$

(c) $f(n) = \sum_{k=1}^{n} k \cdot 2^{-k}$

$\lim_{n \to \infty}$

$f(n) = \sum_{k=1}^{n} \frac{k}{2^k} = \frac{1}{2} + \frac{2}{4} + \frac{3}{8} + \frac{4}{16} + \cdots$

$= \left(\frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \cdots\right) + \left(\frac{1}{4} + \frac{1}{8} + \frac{1}{16} + \cdots\right) + \cdots$

$= 1 + \frac{1}{2} + \frac{1}{4} + \cdots = 2$ So, $f(n) = 1$ for any $n >= n_0$ with n_0 is a positive integer. So, $f(n) = \Theta(1)$, means $f(n) = O(1), f(n) = \Omega(1)$, so $H = L = 0$

(d) $f(n) = \sum_{k=1}^{n} k \log k$

$f(n) <= \sum_{k=1}^{n} k^2 = \frac{n(n+1)(2n+1)}{6}$ for all $n >= 0$, and $\frac{n(n+1)(2n+1)}{6} = \frac{n^3}{3} + \frac{n^2}{2} + \frac{n}{6} <= n^3$, so $f(n) = O(n^3), H = 3$ $f(n) >= \sum_{k=1}^{n} k = \frac{n(n+1)}{2}$ for all $n >= 5$, and $\frac{n(n+1)}{2} = \frac{n^2}{2} + \frac{n}{2} <= n^2$, so $f(n) = \Omega(n^2), L = 2$

(e) $f(n) = \sum_{k=1}^{n} \frac{n-k}{k}$

$f(n) = \sum_{k=1}^{n} \frac{n}{k} - 1 = -n + \sum_{k=1}^{n} \frac{n}{k}$ $\sum_{k=1}^{n} \frac{n}{k} = n + \frac{n}{2} + \frac{n}{3} + \cdots + 1 > n + \frac{n}{2} + \frac{n}{2} + \cdots + \frac{n}{\log_2 n+1} = n + n + n + \cdots + n = n^{\log_2 n+1}$. Because $\log_2 n+1$ is diverge, there is no k such that $f(n) = O(n^k)$, then there is no H. However, when $n = 1$, there is $n^1 = n$. Because $\log_2 n + 1$ grows in $(0, \infty)$, for $n >= 1$, we have $n^{\log_2 n+1} >= n$, so there exists a $\Omega(n)$, and $L = 1$

3. **(15 points) Asymptotics.** Let $n = k^2$ for some positive integer k. A Sudoku solution is an $n \times n$ array of numbers, each in the set $\{1, ..., n\}$, such that the same number does not occur (1) twice in a

row, (2) twice in a column, and (3) twice in a $k \times k$ square "box", where the whole array is divided into n such boxes. Given an array as input, we want to determine whether it is a Sudoku solution.

(a) (5 points) Give an $\Theta(n^3)$ time algorithm to solve this problem.

```
input A as Sudoku solution
for i in 1 to n:
  for j in 1 to n:
    elem = A[i, j]
    for h in 1 to n: // check rows
      if elem == A[l, j] and h != i:
        return false
      endif
    endfor
    for w in 1 to n: // check columns
      if elem == A[i, w] and w != j:
        return false
      endif
    endfor
    u, l = int(i / k) + 1, int(j / k) + 1 // determines the upper left of the box
    for u1 in 1 to k: // as k^2=n, here has n cases
      for l1 in 1 to k:
        if elem == A[u + u1, l + l1] and u + u1 != i and l + l1 != j:
          return false
        endif
      endfor
    endfor
  endfor
endfor
return true
```

We can see that in the two nested $\Theta(n)$ loops, there are three separated $\Theta(n)$ loops, so the total complexity should be $\Theta(3n^3) = \Theta(n^3)$

(b) (5 points) Give an $\Theta(n^2)$ time algorithm to solve this problem.

```
input A as Sudoku solution
set C, R, K as three n*n arraies filled with booleans false
for i in 1 to n:
  for j in 1 to n:
    elem = A[i, j]
    if C[i, elem] == true: // check has same element in column
      return false
    endif
    C[i, elem] = true
    if R[j, elem] == true: // check has same element in row
      return false
    endif
    R[j, elem] = true
    u, l = int(i / k) + 1, int(j / k) + 1 // determines the upperleft of the box
    if K[(u-1)*k+l, elem] == true: // check if same element in box
      return false
    endif
    K[(u-1)*k+1, elem] = true
  endfor
```

```
endfor
return true
```

This algorithm only uses 2 nested $\Theta(n)$ loops, and inside the loops are just indexing, which are $\Theta(1)$, so the complexity of the algorithm is $\Theta(n^2)$

(c) (5 points) Give an $\Omega(n^2)$ lower bound for this problem, by showing that any algorithm that makes fewer than n^2 queries to entries of the input array cannot be correct for all possible inputs.

If we want to check every element which are randomly distributed in the array, we need to access one by one. When we want to get the information about the numbers in a column/row/box, we need to read all the elements as there is no restriction on the values. So, to access n columns/rows/boxes which have n elements in them, at least $n \times n$ operations are required, give $\Omega(n)$

4. **(20 points) Paths of Particular Lengths** Let G be an undirected graph, and s and t be arbitrary vertices of G. We are interested in the possible lengths of (non necessarily) simple paths from s to t.

(a) (10 points). Design an algorithm that determines whether there is a path from s to t with an odd number of edges. Do the same for an even number of edges. Argue that your algorithm is correct and analyze its running time.

```
procedure GetAdj(A, c)
  input A as adjacency matrix which is nxn
  input c as coordinate we want the neighbors
  set result as a array
  for i = 1 to n:
    if A[c, i] == 1:
      result.add(i)
  return result
endprocedure

procedure PathSearch(A, s, seekType)
  input A as adjacency matrix which is nxn
  input s, e as start, end index
  input seekType as the goal // 1 is odds, 2 is even
  set T as a n length array // memory the type, in which 0 means unvisited,
             //1 means reachable by odd path, 2 means even,
             //and 3 means both by odd and even
  set V as a nxn matrix // stores visited connections
  set Q as a queue
  set typeMap as HashTable = {1: 2, 2: 2, 3: 3}
        // odd and even alternates, but both will be kept
  Q.push(s)
  neighbors = GetAdj(A, s)
  if neighbors.length == 0: // if no connection, it must be false
    return false
  endif
  T[s] = 2 // if there is connection, it must can go back in 2 steps
  set index = 0
  while Q.length > 0 && index < 3 * sum(A): // maximum sum(A) adjacencies, 3 changes each
    index++
    curr = Q.pop()
    neighbors = GetAdj(A, curr)
    currType = T[curr]
    for neighbor in neighbors:
      neighborType = T[neighbor]
      T[neighbor] |= typeMap[currType]
        // combine the neighbor node type and current reachable type
      if neighborType != T[neighbor]: // if change node type, recalculate its paths
        for i in 1 to N:
          V[i, neighbor] = 0
        for i in 1 to N:
          V[neighbor, i] = 0
      if V[curr, neighbor] != 1: // connection not visited
        Q.push(neighbor)
      endif
      V[curr, neighbor] = 1 // set adjacency visited
      if e == a && (T[a] & seekType) == seekType:
        // is neighbor end node and reachable by wanted condition
        return true
      endif
    endfor
    // Check missing poinsts
    if Q.length == 0:
      for i in 1 to N:
        for j in 1 to N:
          if i != j:
            if V[i, j] == 0 && T[i] != 0 && A[i, j] == 1:
              Q.push(i)
            else if V[j, i] == 0 && T[j] != 0 && A[j, i] == 1:
              Q.push(j)
            endif
          endif
        endfor
      endfor
    endif
  endwhile
  return false // found nothing
endprocedure
```

The above algorithm can be used for searching for either odds or even paths.

Basically, this algorithm records whether a node can be reached by odds path or even path, and induce their adjacencies. In the algorithm, if the reachablility of a node changes, all the edges that connect to it will be set to unvisited. Therefore, the nodes that connect to it can also be updated. Because the visited edges will not be visited again unless the reachablility of the nodes connect to it changed, the algorithm will terminate, because a node can either be reached by odd, even or both number of steps in a connected subgraph, so it will final reach a stable status and no more nodes will be put into the queue.

Because when there is no element in the queue, it will check whether there are unexplored edges in the graph, so the algorithm will definite reach the whole graph. Also because whenever a node changes, its edges will be unvisited, all the changes will be updated to the graph. Actually, if the algorithm is not set to terminate at the end point, it will result in the reachablility of all the nodes.

The complexity of the algorithm is $O(|V|^2|E|)$, because in the worst case, it will reach all the edges, and will search the nodes need to update for each iteration.

(b) (10 points). Design an algorithm that determines whether there is a path from s to t whose length is divisible by 3. Argue that your algorithm is correct and analyze its running time. (Hint: It might be useful to also consider paths whose length is 1 or 2 modulo 3).

```
procedure GetAdj(A, c)
  input A as adjacency matrix which is nxn
  input c as coordinate we want the neighbors
  set result as a array
  for i = 1 to n:
    if A[c, i] == 1:
      result.add(i)
  return result
endprocedure

procedure PathSearch(A, s)
  input A as adjacency matrix which is nxn
  input s, e as start, end index
  set T as a n length array // memory the type, in which 0 means unvisited,
             //1, 2, 4 means path length mod 3 = 0, 1, 2
  set V as a nxn matrix // stores visited connections
  set Q as a queue
  set modMap = {0: 1, 1: 2, 2: 4}
  set typeMap as HashTable = {1: 2, 2: 4, 4: 1
                              1|2:2|4, 1|4:2|1, 2|4:4|1,
                              1|2|4:1|2|4}
      // modulos are cycling
  Q.push(s)
  neighbors = GetAdj(A, s)
  if neighbors.length == 0: // if no connection, it must be false
    return false
  endif
  T[s] = 4|1 // if there is connection, it must can go back in 2 steps
  set index = 0
  while Q.length > 0 && index < 3 * sum(A): // maximum sum(A) adjacencies, 3 changes each
    index++
    curr = Q.pop()
    neighbors = GetAdj(A, curr)
    currType = T[curr]
    for neighbor in neighbors:
      neighborType = T[neighbor]
      T[neighbor] |= typeMap[currType]
        // combine the neighbor node type and current reachable type
      if neighborType != T[neighbor]: // if change node type, recalculate its paths
        for i in 1 to N:
          V[i, neighbor] = 0
        for i in 1 to N:
          V[neighbor, i] = 0
      if V[curr, neighbor] != 1: // connection not visited
        Q.push(neighbor)
      endif
      V[curr, neighbor] = 1 // set adjacency visited
      if e == a && (T[a] & modMap[0]) == modMap[0]:
        // is neighbor end node and reachable by wanted condition
        return true
      endif
    endfor
    // Check missing poinsts
    if Q.length == 0:
      for i in 1 to N:
        for j in 1 to N:
          if i != j:
            if V[i, j] == 0 && T[i] != 0 && A[i, j] == 1:
              Q.push(i)
            else if V[j, i] == 0 && T[j] != 0 && A[j, i] == 1:
              Q.push(j)
            endif
          endif
        endfor
      endfor
    endif
  endwhile
  return false // found nothing
endprocedure
```

As it is the same algorithm with that in a) only with different search mark method, so it will terminate, and has the complexity $O(|V|^2|E|)$ because of the same reason.

In the algorithm, each node is marked as can be reached by mod 3 = 0, 1, 2 steps and all the combinations of them, and the markers will expand to the whole subgraph which contains the start point until stable or meet the constraint of the goal path.

5. **(20 points) (Non)overlapping Intervals**. We must schedule n jobs, each with a starting time s_i and a finishing time f_i. We have a set of constraints of the following form, each about a pair of jobs:

   - job i finishes before job j starts
   - jobs i and j at least partially overlap.

   We want to assign each of the n starting and finishing times a *distinct* integer value from 1 to $2n$, satisfying these constraints. Design an algorithm that produces such an assignment or reports that this is not possible. Argue that your algorithm is correct and analyze its running time.

   For a array of n jobs, the max number of constraints is $f(n) = \frac{n(n-1)}{2}$, because these two constraints cannot on a pair of same jobs, because overlap means start before finish. For $f(2) = 1$, there is only (j_1, j_2) pair, and $f(n+1) = f(n) + n = \frac{n(n+1)}{2}$, because the new added job will only have one pair with each jobs added before.

   Consider an algorithm which makes the jobs alternatively overlap each other, for example, [(1,3),(2,5),(4,7),(6,8)] as the start/end time of job a, b, c, d, here $(a, b), (b, c), (c, d)$ overlaps each other, and $(a, c), (a, d), (b, d)$ the previous ones finish jobs before the latter ones start. The total number of pairs is $6 = \frac{4*3}{2} = f(4)$. Here guess that this method will meet the maximum constraint pairs.

   Base: $n = 2, f(2) = 1$ for [(1,3),(2,4)], set ith job as J_i ordered by ending time

   Hypothesis: $f(n) = \frac{n(n-1)}{2}$

   Proof: for $f(n+1)$, consider it starts before the J_n ends after J_{n-1} ends, and ends after the J_n ends. So, the J_n will overlap with the new job, and all the jobs except the J_n end before J_{n+1}. So, totally $1 + (n-1) = n$ constraint pairs will be added, $f(n+1) = f(n) + n = \frac{n(n+1)}{2}$, satisfies the hypothesis.

   With this method, we can always get the maximum number of constraint pairs. Also, when $n = 1$, the only situation is (1,2), which does not meet any of the constraint, so $f(1)$ is not possible

```
input n as number of jobs
if n < 2:
  print("n must be larger than 1")
  return null
endif
```

```
if n == 2:
  return [(1, 3), (2, 4)]
endif
set result as list with length n
result.add((1, 3))
for i = 2 to n:
  start = result[i - 2][1] + 1 // start after the latest 2nd overlap one
  result.add((start, min(start + 3, 2 * N)))
      // make sure 2 timestamps can be inserted into the job
      // and it will not exceeds 2 * N
endfor
return result
```

The complexity of this algorithm is $O(n)$, obviously because except the loop from 2 to n, other operations only takes $O(1)$

6. **(0 points).** How long did it take you to complete this assignment?