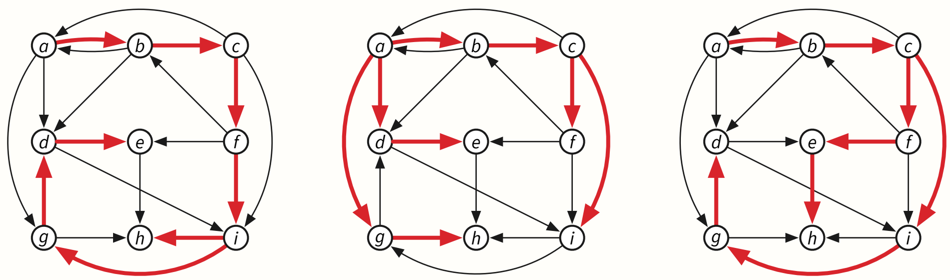
CS/ECE 374 A ✧ Fall 2019
Midterm 2^{Y} Problem 1 Solution

Clearly indicate the following structures in the directed graph G drawn below, or write NONE if the indicated structure does not exist.

1. A depth-first search tree rooted at a.

Solution: Here are three of *many* correct solutions:



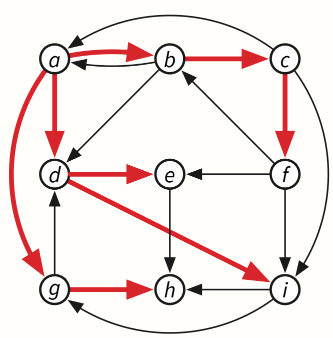
■

Rubric: 2½ points. These are not the only correct solutions.

- No credit if the reported graph is not a spanning tree rooted at vertex a.
- −1 for each misplaced edge, compared to the closest correct solution

2. A breadth-first search tree rooted at a.

Solution: The correct solution is unique!



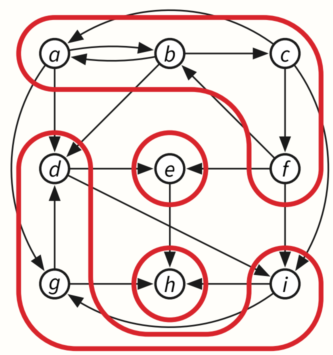
■

Rubric: 2½ points

- No credit if the reported graph is not a spanning tree rooted at vertex a.
- −1 for each misplaced edge

3. Circle each strong component.

Solution:



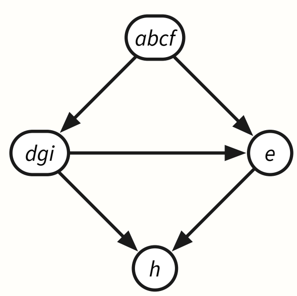
■

Rubric: 2½ points

- −1 for not circling the individual vertices e and h.
- No credit if circled subgraphs omit any vertex other than e or h.
- −1 for circling subgraphs ab and bcf, either instead of or in addition to $abcf$.
- No credit if any circled subgraph is not strongly connected.

4. Draw the strong-component graph of the example graph G.

Solution:



■

Rubric: 2½ points

- No credit if the reported graph is undirected or contains a directed cycle.
- −1 for each misplaced, omitted, or extra vertex or edge.
- No penalty for omitting the vertex labels.

CS/ECE 374 A ✧ Fall 2019
Midterm 2[Y] Problem 2 Solution

Eggsy has a map of the city in the form of an undirected graph G, whose vertices represent intersections and whose edges represent streets between them. A subset of the vertices are marked to indicate that the corresponding intersections are lit. Every edge e has a non-negative length $\ell(e)$. The graph has two special nodes s and t, which represent Eggsy's work and home, respectively. Assume s and t are lit.

Describe an algorithm that computes the shortest path in G from s to t that visits at most k unlit vertices, or correctly reports that no such path exists.

Solution: We construct a new *directed* graph $G' = (V', E')$ as follows:

- $V' = V \times \{0, 1, \ldots, k\}$ — Each vertex (v, i) represents Eggsy reaching vertex v after visiting exactly i unlit vertices (including v, if v is unlit).
- E' contains two types of directed edges:
  - Edges into lit vertices: $\{(u,i)\to(v,i) \mid uv \in E \text{ and } v \text{ is lit}\}$
  - Edges into unlit vertices: $\{(u,i)\to(v,i+1) \mid uv \in E \text{ and } v \text{ is unlit}\}$
- Each edge $(u,i)\to(v,j)$ has weight $\ell(uv)$.

We need to compute the minimum distance from $(s,0)$ to any vertex of the form (t,i). We can compute all such distances by running Dijkstra's algorithm once from $(s,0)$, after which we can find the minimum distance in $O(k)$ additional time. The resulting algorithm runs in $O(E' \log V') = O(kE \log(kV)) = \boldsymbol{O(kE \log V)}$ ***time***. ■

Rubric: 10 points: standard graph reduction rubric. −1 for vanishing k into the $O()$ notation.

CS/ECE 374 A ✧ Fall 2019
Midterm 2 Problem 3 Solution

An undirected graph $G = (V, E)$ is ***bipartite*** if its vertices can be partitioned into two subsets L and R, such that every edge in E has one endpoint in L and one endpoint in R. Describe and analyze an algorithm to determine, given an undirected graph G as input, whether G is bipartite. *[Hint: Every tree is bipartite.]*

Solution: Let G be the input graph. The algorithm runs in three phases:

- First, compute a spanning tree T of G using whatever-first search, starting at an arbitrary vertex s. (If G is disconnected, consider each component of G separately, and return TRUE if and only if every component is bipartite.)
- Color s black. Using a preorder traversal of T, color every vertex except s the opposite color of its parent.
- Finally, scan through all edges in G by brute force; if any edge joins two vertices with the same color, return FALSE, and if all edges join vertices of both colors, return TRUE.

This algorithm runs in $O(V + E)$ time. ■

Solution: Let G be the input graph. The algorithm runs in two phases:

- Perform a breadth-first search of the graph, starting at an arbitrary source vertex s, labeling every vertex v with its shortest-path distance $dist(v)$ from s. (If G is not connected, consider each component of G separately, and return TRUE if and only if every component is bipartite.)
- Then for every edge uv in the graph, if $dist(u) = dist(v)$, return FALSE; if every edge joins vertices with two different distances, return TRUE.

This algorithm runs in $O(V + E)$ time.

This algorithm relies on a specific property of shortest-path distances in *unweighted*, *undirected* graphs: For every source vertex s and every edge uv, the distance to from s to u and the distance from s to v differ by at most 1. ■

Solution: The following variant of depth-first search either colors every vertex black or white, so that every edge has one endpoint of each color, or fails because the input graph is not bipartite.

```
2COLOR(G):
  for all vertices v
      v.color ← NONE
  for all vertices v
      if v.color = NONE
          2COLORDFS(v, WHITE)
```

```
2COLORDFS(v, mycolor):
  if mycolor = WHITE
      yourcolor ← BLACK
  else
      yourcolor ← WHITE
  v.color ← mycolor
  for each edge vw
      if w.color = mycolor
          fail gracefully
      else if w.color = NONE
          2COLORDFS(w, yourcolor)
```

The algorithm runs in $O(V+E)$ time. ■

Solution: Assume without loss of generality that the input graph $G=(V,E)$ is connected. (Otherwise, consider each component of G separately, and return TRUE if and only if every component of G is bipartite.) Construct a new undirected graph $G'=(V',E')$ as follows:

- $V' = V \times 0, 1$
- $E' = \{(u,0)(v,1) \mid uv \in E\} \cup \{(u,1)(v,0) \mid uv \in E\}$

G' has exactly twice as many vertices and twice as many edges as G. Now perform a whatever-first search in G' starting at an arbitrary vertex. If this search marks every vertex of G', then report that G *is not* bipartite; if at least one vertex remains unmarked, report that G *is* bipartite. This algorithm runs in $\boldsymbol{O(V+E)}$ ***time***.

Suppose G is not bipartite. Then G contains an odd cycle $v_1 \to v_2 \to \cdots \to v_{2k+1} \to v_1$, and therefore H contains the cycle

$$(v_1,0)\to(v_2,1)\to\cdots\to(v_{2k+1},0)\to(v_1,1)\to(v_2,0)\to\cdots\to(v_{2k+1},1)\to(v_1,0).$$

Because G is connected, G contains a path from v_1 to every other vertex u of G. It follows that H contains a path from $(v_1,0)$ to every vertex (u,i) of H, obtained either by following some $v_1 \rightsquigarrow u$ path in G directly, or by going around the cycle to $(v_1,1)$ and then following some $v_1 \rightsquigarrow u$ path in G. We conclude that H is connected.

On the other hand, suppose G is bipartite. Color the vertices of G alternately black and white. Consider any two vertices u and v of G. Any path between $(u,0)$ and $(v,0)$ must have even length; but any path between a black vertex and a white vertex must have odd length. Thus, $(u,0)$ and $(v,0)$ lie in the same component of H if and only if u and v have the same color. Similarly, $(u,0)$ and $(v,1)$ lie in the same component of H if and only if u and v have different colors. We conclude that H consists of two disjoint copies of G, one with $0=$ white and $1=$ black, the other with $0=$ black and $1=$ white. ■

Rubric: 10 points = 8 points for the algorithm (if recursive: 2 point for base case + 6 points for recursive case) + 2 point for time analysis. These are not the only correct solutions. No penalty if the algorithm works only for connected graphs. A proof of correctness is not required. Max 7 points for an algorithm that runs in $O(E \log V)$ time; max 4 points for any slower correct algorithm.

CS/ECE 374 A ✧ Fall 2019
Midterm 2$^{\curlyvee}$ Problem 4 Solution

Describe an efficient algorithm that computes the largest number of students that Satya can host for testing in a sequence of n rooms without using three consecutive rooms. The input to your algorithm is an array $S[1..n]$, where each $S[i]$ is the number of students that can fit in room i. (See the question handout for more details.)

Solution: For any index i and any integer $c \in \{0, 1, 2\}$, let $\mathit{MaxStudents}(i, c)$ denote the maximum number of students that can be hosted for testing in rooms i through n, assuming exactly c rooms immediately before room i are already being used. We need to compute $\mathit{MaxStudents}(1, 0)$.

This function can be described by the following recurrence:

$$\mathit{MaxStudents}(i, c) = \begin{cases} 0 & \text{if } i > n \\ \mathit{MaxStudents}(i+1, 0) & \text{if } c = 2 \\ \max\left\{ \begin{array}{c} S[i] + \mathit{MaxStudents}(i+1, c+1) \\ \mathit{MaxStudents}(i+1, 0) \end{array} \right\} & \text{otherwise} \end{cases}$$

We can memoize this function into an array $\mathit{MaxStudents}[1..n, 0..2]$, which we can fill by decreasing i in the outer loop and considering c in any order in the inner loop. The resulting algorithm runs in $\boldsymbol{O(n)}$ ***time***. ■

Solution: For any index i, let $\mathit{MaxStudents}(i)$ denote the maximum number of students that can be hosted for testing in rooms i through n, assuming room $i-1$ is *not* occupied. We need to compute $\mathit{MaxStudents}(1)$.

This function can be described by the following recurrence:

$$\mathit{MaxStudents}(i) = \begin{cases} 0 & \text{if } i > n \\ S[n] & \text{if } i = n \\ S[n-1] + S[n] & \text{if } i = n-1 \\ \max\left\{ \begin{array}{c} \mathit{MaxStudents}(i+1) \\ S[i] + \mathit{MaxStudents}(i+2) \\ S[i] + S[i+1] + \mathit{MaxStudents}(i+3) \end{array} \right\} & \text{otherwise} \end{cases}$$

We can memoize this function into an array $\mathit{MaxStudents}[1..n]$, which we can fill by decreasing i in $\boldsymbol{O(n)}$ ***time***. ■

Rubric: 10 points: standard dynamic programming rubric. These are not the only correct solutions.

CS/ECE 374 A ✧ Fall 2019
Midterm 2$^{\curlyvee}$ Problem 5 Solution

Describe and analyze an algorithm to compute the number of maximal points in in a given point set P in $O(n \log n)$ time. The input to your algorithm is a pair of arrays $X[1..n]$ and $Y[1..n]$ containing the x- and y-coordinates of the points in P. (See the question handout for more details and an example.)

All model solutions assume the given points have distinct x- and y-coordinates.

Solution (sort and scan backward): The following algorithm runs in ***$O(n \log n)$ time***; the running time is dominated by the initial sort.

```
NumMaximal(X, Y):
  Sort X and permute Y to match    ⟨⟨= Sort the points from left to right⟩⟩
  ⟨⟨Scan right to left, counting new maximum y-coordinates⟩⟩
  count ← 0
  maxy ← −∞
  for i ← n down to 1
      If Y[i] > maxy
          count ← count + 1
          maxy ← Y[i]
  return count
```

■

Solution (sort and scan forward): The following algorithm runs in ***$O(n \log n)$ time***; the running time is dominated by the initial sort.

```
NumMaximal(X, Y):
  Sort X and permute Y to match    ⟨⟨= Sort the points from left to right⟩⟩
  ⟨⟨Scan left to right⟩⟩
  S ← new stack          ⟨⟨maintain staircase of first i points⟩⟩
  Push(S, ∞)             ⟨⟨sentinel⟩⟩
  for i ← 1 down to ∞
      while Y[i] > Top(S)
          Pop(S)
      Push(S, Y[i])
  return Size(S) − 1
```

■

Solution (like quicksort): To simplify notation, assume the points are actually given in a single array $P[1..n]$, where each $P[i]$ stores a point with x-coordinate $P[i].x$ and y-coordinate $P[i].y$. We can convert from two separate coordinate arrays to this array of records in $O(n)$ time by brute force.

The following divide-and-conquer algorithm actually computes an array containing all maximal points in P, sorted by increasing x-coordinate, in ***$O(n \log n)$ time***. The running time is dominated by the initial sort. We can compute the *number* of points in this list in $O(n)$ additional time.

```
STAIRCASE(P[1..n]):
    Sort P by increasing x-coordinate
    QUICKSTAIR(P[1..n])
```

```
QUICKSTAIR(P[1..n]):
    if n = 1
        return P[1]
    ⟨⟨ Recursively construct staircases ⟩⟩
    ⟨⟨ of the left and right halves of P ⟩⟩
    L[1..l] ← QUICKSTAIR(P[1..⌈n/2⌉])
    R[1..r] ← QUICKSTAIR(P[⌈n/2⌉+1..n])
    ⟨⟨ Discard points on the left staircase that lie ⟩⟩
    ⟨⟨ below the first point on the right staircase ⟩⟩
    while L[l].y ≤ R[1].y
        l ← l − 1
    ⟨⟨Copy to an output array and return⟩⟩
    for i ← 1 to l
        M[i] ← L[i]
    for i ← 1 to r
        M[l+i] ← R[i]
    return M[1..l+r]
```

The running time of QUICKSTAIR obeys the standard *mergesort* recurrence $T(n) = 2T(n/2) + O(n)$, so QUICKSTAIR runs in $O(n \log n)$ time. ■

Solution (like mergesort): To simplify notation, assume the points are actually given in a single array $P[1..n]$, where each $P[i]$ stores a point with x-coordinate $P[i].x$ and y-coordinate $P[i].y$. We can convert from two separate coordinate arrays to this array of records in $O(n)$ time by brute force.

The following algorithm actually computes an array containing all maximal points in P, sorted from left to right. Unlike the previous solutions, this algorithm does *not* begin by sorting the points. We can compute the *number* of points in this list in $O(n)$ additional time.

```
MergeStair(P[1..n]):
  if n = 1
      return P[1]

  ⟨⟨ Recursively construct staircases ⟩⟩
  ⟨⟨ of two arbitrary halves of P     ⟩⟩
  A[1..a] ← MergeStair(P[1..⌈n/2⌉])
  B[1..b] ← MergeStair(P[⌈n/2⌉+1..n])

  ⟨⟨Merge the overlapping staircases from left to right⟩⟩
  m ← 0                      ⟨⟨number of output points⟩⟩
  i ← 1; j ← 1
  while i + j < a + b
      if j = b               ⟨⟨we've finished scanning B; copy next point from A⟩⟩
          m ← m + 1;  M[m] ← A[i]
          i ← i + 1
      else if i = a          ⟨⟨we've finished scanning A; copy next point from B⟩⟩
          m ← m + 1;  M[m] ← B[j]
          j ← j + 1
      else if A[i].x < B[j].x    ⟨⟨next unscanned point is in A⟩⟩
          if A[i].y > B[j].y        ⟨⟨and not hidden by next point in B⟩⟩
              m ← m + 1;  M[m] ← A[i]
          i ← i + 1
      else ⟨⟨A[i].x > B[j].x⟩⟩    ⟨⟨next unscanned point is in B⟩⟩
          if B[j].y > A[i].y        ⟨⟨and not hidden by next point in A⟩⟩
              m ← m + 1;  M[m] ← B[j]
          j ← j + 1
  return M[1..m]
```

The running time of MergeStair obeys the standard mergesort recurrence $T(n) = 2T(n/2) + O(n)$, so MergeStair runs in $\boldsymbol{O(n \log n)}$ ***time***. ■

Rubric: 10 points = 7 points for algorithm + 3 points for time bound. These are not the only correct solutions. Max 5 points for an $O(n^2)$-time algorithm; max 3 points for anything slower. No penalty for misbehavior when x- or y-coordinates are not distinct. These are not the only correct solutions.