COMP 3711 – Design and Analysis of Algorithms
2019 Spring Semester – Written Assignment # 3 – Solution Sketch
Distributed: April 8, 2019 – Updated April 18, 2019
Due: April 24, 2019

Your solutions should contain (i) your name, (ii) your student ID #, and (iii) your email address

Notes:

- Follow the guidelines on doing your own work and avoiding plagiarism given on the class home page.
  In particular ***don't forget to acknowledge individuals who assisted you, or sources where you found solutions.*** Failure to do so will be considered plagiarism.
- Write clearly and follow the submission guidelines on the class web page. Use white, unwatermarked paper, e.g., no student society stationary.
  - If handwritten, solutions should be single sided, start a new page for every problem, be single column and leave space between consecutive lines and more space between paragraphs.
  - If typed, try to use an equation editor, e.g., latex , the equation editor in MS-WORD or whatever the equivalent is in whatever typesetting system you are using
- This assignment is due by 23:59 on April 24, 2019 in BOTH hard AND soft copy formats. A hard copy should be deposited in one of the two COMP3711 assignment collection boxes outside of room 4210. A soft copy for our records in PDF format should also be submitted via the online CASS system. See the Assignment 1 page in Canvas for information on how to submit online.
- The default base for logarithms will be 2, i.e., $\log n$ will mean $\log_2 n$. If another base is intended, it will be explicitly stated, e.g., $\log_3 n$.
- Fixed typos: There were two typographical errors that were fixed in this update.
  - Problem 3: In original version, the example adjacency list of $(1, 1)$ contained $(0, 2)$. This was corrected to $(1, 2)$.
  - Problem 2, page 4, condition b. Was originally stated as $Gap(s_i, e_i) \leq M$. This was corrected to $Gap(s_i, e_i) \geq 0$.
    Equation on last line. Was originally stated as $\min\{OPT(i) : Gap(i + 1, n) \leq M\}$.
    This was corrected to $\min\{OPT(i) : Gap(i + 1, n) \geq 0\}$.

Problem 1: Optimal Binary Search Trees [15 pts]

Consider the following input to the Optimal Binary Search Tree problem (it is presented in the same format as the example powerpoint file posted on the lecture page):

i	1	2	3	4	5	6	7	8
a_i	A	B	C	D	E	F	G	H
$f(a_i)$	5	7	1	8	4	6	2	3

a) **Fill in the two tables below. As in the example powerpoint only the entries with $i \leq j$ need to be filled in. We have started you off by filling in the $[i, i]$ entries.**

i/j	1	2	3	4	5	6	7	8
1	5							
2		7						
3			1					
4				8				
5					4			
6						6		
7							2	
8								3

Table 1: Left matrix is $e[i, j]$.

i/j	1	2	3	4	5	6	7	8
1	1							
2		2						
3			3					
4				4				
5					5			
6						6		
7							7	
8								8

Right matrix is $root[i, j]$.

b) Draw the optimal Binary Search Tree (with 8 nodes) and give its cost.

Solution:

(a)

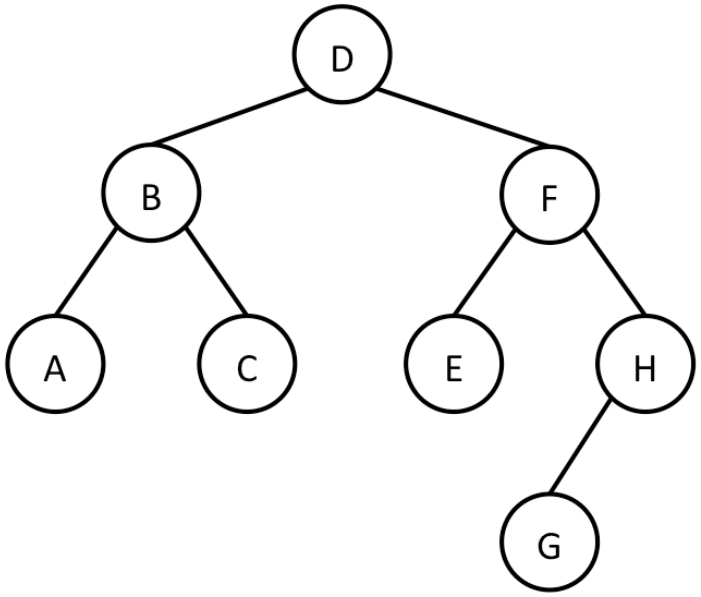
i/j	1	2	3	4	5	6	7	8
1	5	17	19	36	48	64	70	81
2		7	9	25	33	49	55	66
3			1	10	18	34	40	49
4				8	16	32	38	46
5					4	14	18	26
6						6	10	18
7							2	7
8								3

Table 2: Left matrix is $e[i, j]$.

i/j	1	2	3	4	5	6	7	8
1	1	2	2	2	2/4	4	4	4
2		2	2	4	4	4	4	4
3			3	4	4	4	4	6
4				4	4	4/5	4/5/6	6
5					5	6	6	6
6						6	6	6
7							7	8
8								8

Right matrix is $root[i, j]$.

(b) Cost is 81. Tree is below.



Problem 2: Dynamic programming for pretty printing [25 pts]

Consider the problem of neatly formatting a paragraph on a page for printing or displaying. The input text is a sequence of n words of lengths $\ell_1, \ell_2, \ldots, \ell_n$, measured in characters. We want to print this paragraph neatly on lines that hold a maximum of M characters each. As an example, consider the problem of laying out the nonsense phrase, *It is the time for all wise good people and very exceptional friends to hasten quickly and silently home* on a line of length 29. Here are two possible layouts.

```
12345678901234567890123456789
It is the time for all wise
good people and very
exceptional friends to hasten
quickly and silently home
```

```
12345678901234567890123456789
It is the time for all
wise good people and very
exceptional friends to hasten
quickly and silently home
```

Note that the first layout has $2, 9, 0, 4$ spaces remaining at the end of, respectively, its 1st, 2nd, 3rd and 4th lines, while the second layout has $7, 4, 0, 4$ spaces remaining. The second one 'looks" neater (less ragged) than the first.

The criterion we use for measuring "neatness" is as follows: The input will be a sequence $\ell_1, \ell_2, \ldots, \ell_n$ where ℓ_i is the length of word i.

Note that if a given line contains words i through j and we leave exactly one space between words, the number of extra space characters remaining at the end of the line is

$$Gap(i, j) = M - j + i - L(i, j) \quad \text{where} \quad L(i, j) = \sum_{k=i}^{j} \ell_k.$$

The penalty associated with using this line will be the cost

$$C(i, j) = (Gap(i, j))^2 .$$

The cost of a layout will be the sum, over all lines except the last, of the cost of the line. The reason for not charging for the last line is that, in printing, the last line is allowed to be ragged.

Consider the example that started this problem. $M = 29$ and the remainder of the input is the list of the lengths of the 19 words in the paragraph:

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
ℓ_i	2	2	3	4	3	3	4	4	6	3	4	11	7	2	6	7	3	8	4

Two possible ways (there are others) of placing these words on four lines are (see below for definitions of s_i, e_i)

	Layout 1					Layout 2			
	s_i, e_i	$L(s_i, e_i)$	$Gap(s_i, e_i)$	$C(s_i, e_i)$		s_i, e_i	$L(s_i, e_i)$	$Gap(s_i, e_i)$	$C(s_i, e_i)$
Line $i = 1$	$1, 7$	21	2	4		$1, 6$	17	7	49
Line $i = 2$	$8, 11$	17	9	81		$7, 11$	21	4	16
Line $i = 3$	$12, 15$	26	0	0		$12, 15$	26	0	0
Line $i = 4$	$16, 19$	22	4			$16, 19$	22	4	

The cost of Layout 1 is $2^2 + 9^2 + 0^2 = 85$ while the cost of Layout 2 is $7^2 + 4^2 + 0^2 = 65$ so Layout 2 is much better (less ragged). Note that the costs of their last lines are *not* included in the costs of the paragraphs.

A *Layout* will be defined as a partition of the words into lines. Lines will not be allowed to overflow (be longer than M). A layout will be formally given by a sequence s_i, e_i, for $i = 1 \ldots j$. These are the starting and ending word indices of each line and satisfy:

(a) $s_1 = 1$, $e_j = n$

(b) $\forall i = 1, \ldots, j, \quad s_i \leq e_i$ and $Gap(s_i, e_i) \geq 0$.
(Line i contains a continuous set of words and can not overflow)

(c) $\forall i = 1, \ldots, j-1, \quad s_{i+1} = e_i + 1$
(line $i + 1$ starts immediately after the last word of line i.)

Note that j, the number of lines, is NOT fixed in advance and may range anywhere between 1 and n.

The problem is, given M and the ℓ_i as input, to find a layout that minimizes the cost, $\sum_{i=1}^{j-1} C(s_i, e_i)$ of the paragraph. For example, for the input above, Layout 2 is the best possible layout.

Give a dynamic-programming algorithm to solve this problem for n words. The input will be the value M and the list $\ell_1, \ldots, \ell_n$ of word lengths. Your algorithm should report the optimum cost and the layout (list out the s_i, e_i) that achieves this.

Analyze the running time and space requirements of your algorithm. Full credit will only be given for an $O(n^2)$ time algorithm. Anything that runs worse than that will only receive partial credit.

Design Hint: Modify the problem so that the cost also includes the cost of the last line, define $OPT(i)$ to be the minimum cost of solving this modified problem restricted to just the first i words and find a DP recurrence for $OPT(i)$. Then show that the solution to the original problem satisfies $\min\{OPT(i) : Gap(i + 1, n) \geq 0\}$.

Solution:

A) Start by setting $L[i] = \sum_{1 \le i} \ell_i$ *with* $L[0] = 0$. *The* $L[i]$ *can be precomputed in total* $O(n)$ *time.*

Once this is done, $L(i,j) = L[j] - L[i-1]$ *can be calculated in* $O(1)$ *time. This means that* $Gap(i,j)$ *and* $C(i,j)$ *can also be calculated in* $O(1)$ *time.*

B) Consider a line on which word j *is the last word. Recall from the description that word* j' *can legally be the first word on that line if and only if* $Gap(j',j) \ge 0$. *For later description, we set*

$$\mathbf{c_j} = \min\{\mathbf{j' \le j} \,:\, \mathbf{Gap(j',j) \ge 0}\}. \tag{1}$$

Note that each c_j *can be calculated in* $O(n)$ *time so all of the* c_j *can be calculated in* $O(n^2)$ *total time.*

Note: Using methods seen in the tutorial ALL of the c_j could actually be calculated in $O(n)$ total time but this would not improve the running time of our algorithm so we do not describe the details.

(C) The generic step for calculating $OPT(j)$ *is to consider the structure of the last line, that one that ends with word* j. *This can be any line* $j', \ldots, j$ *with* $c_j \le j' \le j$. *Once* j' *is fixed, the penalty imposed for the last line is* $C(j',j)$. *The minimum cost that can be imposed on the previous lines is* $OPT(j'-1)$ *(note that this is still valid if* $j' = 1$ *because we set* $OPT(0) = 0$*.) So the cost is*

$$OPT(j'-1) + C(j',j)$$

In $O(n)$ *time, we check this value for every possible* j' *to find the one that yields the minimum value. That will be the one defining* $OPT(j)$*:*

$$\mathbf{OPT(j)} = \min_{\mathbf{c_j \le j' \le j}} (\mathbf{OPT(j'-1)} + \mathbf{C(j',j)}) \tag{2}$$

D) The final layout for all n *words has words* $j, j+1, \ldots n$ *on the last line for some* $c_j \le j \le n$. *The second to last line therefore ends with word* $j-1$. *Since no penalty is given for the last line, the best cost for such a solution is* $OPT(j-1)$. *This tells us that the actual optimum solution is*

$$\text{Final-OPT} = \min_{\mathbf{c_n \le j \le n}} \mathbf{OPT(j-1)}. \tag{3}$$

Thus, given a completed $OPT(j)$ *table we can find the best optimal layout using (3) in* $O(n)$ *time.*

Code for the full algorithm is on the next page.

% (A) Precalculate the $L[j]$
1. $L[0] = 0.$
2. *For* $j = 1$ *to* n *do*
3. *For* $L[j] - L[j-1] + \ell_j$

% Define constant time evaluation functions based on $L(i,j)$
4. *Define* $Gap(i,j) = M - j + i - (L[j] - L([i-1])$
5. *Define* $C(i,j) = (Gap(i,j))^2$

% (B) Precalculate the c_j
6. $c_j = j$
7. *While* $(c_j - 1 > 0)$ *AND* $(Gap(c_j - 1, j) \geq 0)$
8. $c_j = c_j - 1$

% (C) Evaluate $OPT(j) = \min_{c_j \leq j' \leq j} (OPT(j'-1) + C(j',j))$
9. $OPT(0) = 0$
10. *For* $j = 1$ *to* n *do*
11 $OPT(j) = OPT(j-1) + C(j-1,j)$
12. $Break(j) = j$
13. *For* $t = j - 1$ *downto* c_j *do*
14. *If* $OPT(t-1) + C(t,j) < OPT(j)$
15. $OPT(j) = OPT(t-1) + C(t,j)$
16. $Break(j) = t$

% (D) Find Final-OPT $= \min_{\mathbf{c_n} \leq \mathbf{j} \leq \mathbf{n}} \mathbf{OPT(j-1)}$
17. Final-OPT $= OPT(n-1)$
18. Final-Break $= n$
19. *For* $t = n - 1$ *downto* c_n *do*
20. *If* $OPT(t-1) <$ Final-OPT
21. Final-OPT $= OPT(t-1)$
22. Final-Break $= t$

% (E) Print lines (first word, last word) in reverse line order
23. *Print* (Final-Break, n)
24. $t =$ Final-Break $- 1$
25. *While* $t \neq 0$ *Do*
26. *Print* $(Break(t), t)$
27. $t = Break(t) - 1$

Note that $Break(j)$ *stores the value* j' *(first word in the line that ends with* j*) that minimizes (2) and* Final-Break *stores the value* j' *(first word in the last line) that minimizes (3).*

Finally, note that the algorithm only uses extra space to store the $L[i]$, c_j and $OPT(j)$ value, so it only requires $O(n)$ additional space.

Marking Note: To get full credit it was necessary to distinguish between the last line (which had no associated penalty cost) and the preceeding lines.

It was also necessary to document your pseudocode.

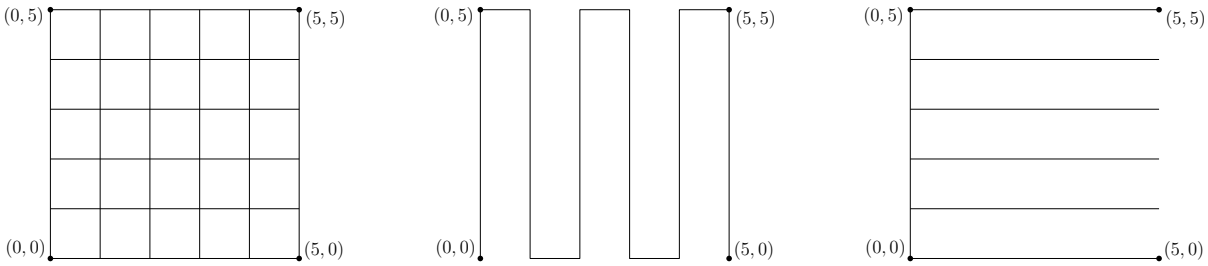
Problem 3: BFS/DFS [20 pts]

In this problem you will have to describe the depth and breadth first search trees calculated for particular graphs. Let graph G_n be the $n \times n$ grid containing the n^2 points (i, j), $i = 0, \ldots, n-1$, $j = 0, \ldots, n-1$. Figure (A) illustrates G_6. Each point is connected to four neighbors: The one below it, the one above it, the one to its left and the one to its right. Note that some points only have two or three neighbors, e.g., the four corner points only have 2 neighbors and that the edge points (aside from the corners) each have three neighbors. The adjacency list representation used is

$$(i, j) : (i, j-1) \to (i, j+1) \to (i-1, j) \to (i+1, j).$$

For example, the adjacency list representation for $(1, 1)$ in G_6 is

$$(1, 1) :\to (1, 0) \to (1, 2) \to (0, 1) \to (2, 1).$$



Some nodes will only have two or three neighbors; their adjacency lists should be adjusted appropriately. For example

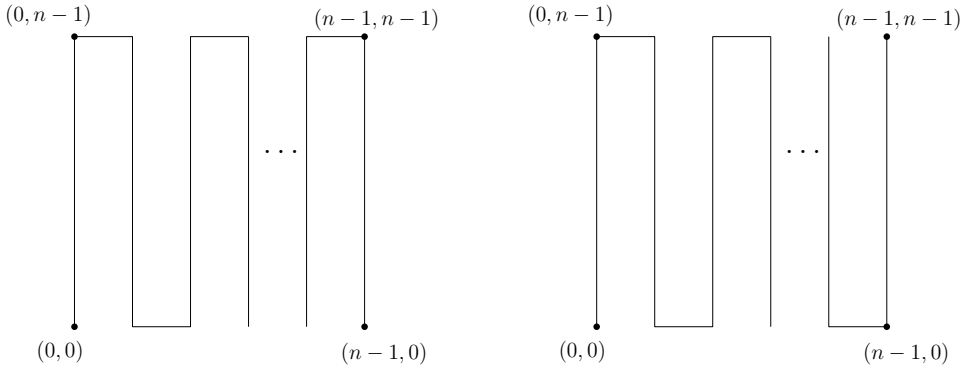
$$(0, 0) :\to (0, 1) \to (1, 0) \quad \text{and} \quad (i, 0) :\to (i, 1) \to (i-1, 0) \to (i+1, 0)$$

for $i = 1, \ldots, n-2$.

In what follows *Describe the tree* means (i) list the edges in the tree and (ii) sketch a diagram that illustrates how the tree looks.

(a) Describe the tree produced by Depth First Search run on G_n starting at root $s = (0, 0)$. Figure (B) illustrates the tree produced for G_6.

Solution: The diagram below illustrates the tree (which is a path).



n even $\qquad\qquad$ n odd

Note that there is a small distinction between the cases n even and n odd.

The edges in the tree (path) are

- *Vertical edges:*
  For every $i = 0, 1, \ldots, n-1$
  For every $j = 0, 1, \ldots, n-2$
  all edges on the i'th vertical line: $\big((i, j)\,(i, j+1)\big)$.
- *Horizontal edges*
  - *n even: the edge $\big((n-2, n-1),\ (n-1, n-1)\big)$ and*
    For $i = 0, 1, 2, \ldots (n/2) - 2$, the edges
    $\big((2i, n-1),\ (2i+1, n-1)\big)$ and $\big((2i+1, 0),\ (2i+2, 0)\big)$
  - *n odd*
    For $i = 0, 1, 2, \ldots ((n-1)/2) - 1$, the edges
    $\big((2i, n-1),\ (2i+1, n-1)\big)$ and $\big((2i+1, 0),\ (2i+2, 0)\big)$

(b) Describe the tree produced by Breadth First Search run on G_n starting at root $s = (0, 0)$. Figure (C) illustrates the tree produced for G_6.

Solution:

This has the exact same structure as G_6.

- *The leftmost vertical line:*
  For every $j = 0, 1, \ldots, n-2$, the edge $\big((0, j),\ (0, j+1)\big)$
- *Every horizontal line:*
  For every $j = 0, 1, \ldots, n-1$
  For every $i = 0, 1, \ldots, n-2$, the edge $\big((i, j),\ (i+1, j)\big)$

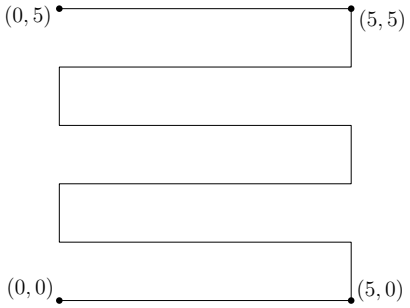
For the rest of this problem keep G_n the same but change its adjacency list representation to

$$(i, j) :\to (i + 1, j) \to (i, j - 1) \to (i - 1, j) \to (i, j + 1).$$

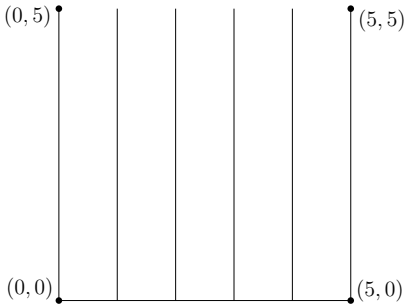
(c) Now, show the tree produced by Depth First Search run on G_6 starting at root $s = (0, 0)$ using this new adjacency list representation.

(d) Now, do the same for the tree produced by Breadth First Search run on G_6.

Solution for (c) and (d).



DFS

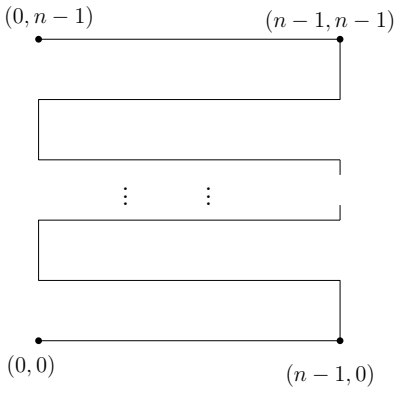


BFS

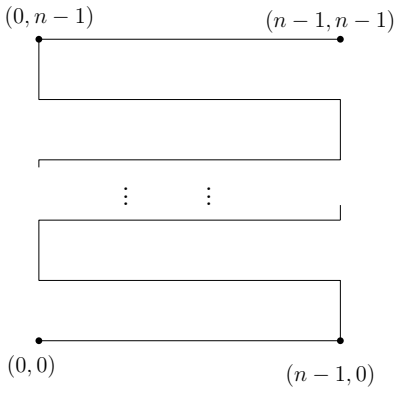
(e) Finally, describe the tree produced by Depth First Search and Breadth First Search run on G_n using this new adjacency list representation.

Solution:

The diagram below illustrates the DFS tree (which is a path).



DFS: n even



DFS: n odd

Note that there is a small distinction between the cases n even and n odd.

The edges in the tree (path) are

- *Horizontal edges:*
  For every $j = 0, 1, \ldots, n-1$
  For every $i = 0, 1, \ldots, n-2$
  all edges on the j'th horizontal line: $\big((i, j)\,(i+1, j)\big)$.
- *Vertical Edges*
  - *n even: the edge $\big((n-1, n-2),\, (n-1, n-1)\big)$ and*
    For $j = 0, 1, 2, \ldots (n/2) - 2$, the edges
    $\big((n-1, 2j),\, (n-1, 2j-1)\big)$ and $\big((0, 2j+1),\, (0, 2j+2)\big)$
  - *n odd*
    For $j = 0, 1, 2, \ldots ((n-1)/2) - 1$, the edges
    $\big((n-1, 2j),\, (n-1, 2j+1)\big)$ and $\big((0, 2j+1),\, (-0, 2j+2)\big)$

The BFS tree has the exact same structure as for G_6.

- *The bottomost horizontal line:*
  For every $i = 0, 1, \ldots, n-2$, the edge $\big((i, 0),\, (i+1, 0)\big)$
- *Every vertical line:*
  For every $i = 0, 1, \ldots, n-1$
  For every $j = 0, 1, \ldots, n-2$, the edge $\big((i, j),\, (i, j+1)\big)$

Marking Note. For full credit it was necessary, in parts (a) and (d), to explicitly distinguish between the even and odd cases, at least in your diagrams.

Problem 4: Cycle Finding [15 pts]

In class, we learned how to check if an undirected graph has a cycle in $O(|V|)$ time. That algorithm either returned a cycle or told us that the graph is a tree. For this problem the input is an undirected graph $G = (V, E)$ *and* a specified vertex $v \in V$. Design an $O(|E| + |V|)$ time algorithm that returns

- NO: if the graph does not contain a cycle containing v.
- YES: if the graph does contain a cycle containing v. In this case, it also prints out any simple cycle containing v.

Your algorithm should be given using well-documented pseudocode. You need to prove correctness and the fact that it has a $O(|E| + |V|)$ running time.

Your algorithm and proof may explicitly use any statement or algorithm we taught in class or the tutorial as long as you explicitly reference what you are using.

Let G_v be the connected component of G containing v. Note that any cycle containing v must be fully contained in G_v so we only need to check if G_v contains such a cycle.

Recall the procedure $DFS-visit(v)$ taught in class. This runs DFS starting from node v. It generates a DFS tree T_v of G_v T_v that is rooted at v.

Since T_v is a tree, every $w \in T_v$ defines a unique path $P_v(w)$ in T_v from w to v.

Now consider every edge (u_i, v) in the adjacency list of v. By definition, all such u_i are in G_v and thus T_v. There are two possible cases

(a) There exists some i such that $u_i.p \neq v$.
In this case, path $P_v(u_i)$ and edge (u_i, v) form a cycle.

(b) For all i, $u_i.p = v$.
In this case, we claim that G_v contains no cycle containing v.
If such a cycle existed, then it would contain two edges (v, u_i) and (v, u_j) with $u_i \neq u_j$. This implies the existence of a path from u_i to u_j not containing v. Without loss of generality assume u_i precedes u_j on v's adjacency list. Then u_j would have appeared in the DFS tree rooted at u_i and $u_j.p \neq v$, contradicting the assumption.

This immediately gives an algorithm

```
1.   Run DFS − visit(v).          % This constructs DFS tree T_v rooted at v

2.   For every u ∈ Adj(v) do
3.        If u.p = v then              % Case (a) on previous page
4.           Print " A cycle through v Exists. It is"
5.           Print (v, u)
6.           While u ≠ v do
7.                print (u, u.p))
8.                u = u.p
9.           Exit procedure

% If Procedure wasn't exited then ∀u ∈ Adj(v), u.p = v
%    and Case (b) on previous page
10.  Print "No Cycle through v Exists"
```

Line 1 of the algorithm used $O(|E| + |V|)$ time to run DFS. Lines 2-9 are run once for every $u \in Adj(v)$ so they take, in total $O(|Adj(v)|) = O(|E|)$ time.

The entire algorithm therefore takes only $O(|E| + |V|)$ time.

Marking Note: For full credit it was necessary to understand that you were only working in the *connected component containing* v. As an example, if you just wrote "run DFS starting at v", you would have been wrong. We had two algorithms, DFS and DFS-visit. DFS ran through all vertices, starting DFS-visit if v was still white.

It was also necessary to explain both directions of the if and only if proof. That is, you needed to prove that if a backedge to v exists then a cycle containing v exists but you also needed to prove that if no backedge exists, no such cycle exists (or something equivalent).

Problem 5: MST Checking [25 pts]

Let G be a connected undirected graph with distinct weights on the edges. Recall that such a graph has a *unique* MST.

Given an edge e of G, can you decide whether e belongs to the MST in $O(|E|)$ time? If you compute the MST and then check whether e belongs to the MST, this would take $O(|E| \log |V|)$ time. To design a faster algorithm, you will need the following theorem:

Edge $e = (u, v)$ does not belong to the MST if and only if there is a path from u to v that consists of only edges cheaper than e.

1. Prove this theorem.
   Caveat: your proof may explicitly use any statement or algorithm we taught in class or the tutorial as long as you explicitly reference what you are using.
2. Describe and prove the correctness and running time of the $O(|E|)$-time algorithm. Hint. Use BFS or DFS.
   Caveat: again, you may use any statement or algorithm we taught in class or the tutorial as long as you explicitly reference what you are using.

Solution:

$\Rightarrow$: *First we prove the "only if" part of the statement by assuming that edge $e = (u, v)$ does not belong to the MST.*

Consider what happens during the running of Kruskal's algorithm.

- *Since e does not belong to the MST, e is not added to the MST by the algorithm.*
- *This can only happen if, at the time e is being checked, it created a cycle, i.e., there was a path connecting u and v composed of the previously added edges,*
- *But, since all of these edges are cheaper than e this is a path from u to v that consists of only edges cheaper than e.*

$\Leftarrow$ *Now we prove the "if" part of the statment by assuming that there is a path P from u to v that consists of only edges cheaper than e. Suppose this path is edges*

$$P = e_1 e_2 \dots e_i$$

where $e = (u_i, v_i)$ with $u = u_1$, $v = u_k$ and $\forall 1 \leq i < k$, $v_i = u_{i+1}$.

Again consider the running of Kruskal's algorithm. Since the e_i are all cheaper than e, they are all checked before e is checked. At the time e_i is checked two possibilities can occur

– *e_i is added to the MST. In this case set $P_i = e_i$*

– *e_i is not added to the MST because the MST already contained a path P_i connecting u_i to v_i. Since all of the edges in the path have weight cheaper than e_i they are also cheaper than e*

Note that by construction the first vertex of P_1 is u, the last vertex of P_k is v and, for all $1 \leq i < k$, the last vertex of P_i is $v_i = u_{i+1}$ which is also the first vertex of P_{i+1}. Thus, the concatenation of the paths

$$P = P_1 P_2 \dots P_k$$

is a path P in the MST *connecting u and v composed of edges cheaper than e. This means that adding e to the MST would create a cycle, so e is NOT in the MST.*

MARKING NOTE:
For the "if" part it was NOT correct to write that

"if there is a path (P) from u to v that consists of only edges cheaper than e
then
that path (P) will be in the MST ".

It is quite possible that P is NOT in the MST so that statement is incorrect and would have points deducted.

What needed to be proved was that if

"if there is a path P from u to v that consists of only edges cheaper than e'
then that implies that
there is a path P' from u to v that consists of only edges cheaper than e THAT IS IN THE MST".

(b) To use this theorem to check whether e belongs to the MST, we just delete all edges heavier than e and e itself, and then check whether u and v are still connected, using either BFS or DFS. This takes time $O(V + E) = O(E)$, since we have $E \geq V - 1$ as the graph is connected.