CSCE 411

Project

Yingtao Jiang

Dr. Anxiao Jiang

Problem 1:

Answer:

The solution is using dynamic programming. For j from 1 to n, get the maximum sum with ending at position j with recursive n times. Then, compared those maximum sums to find the maximum, which takes n times. Finally, trace back to find the start position of the maximum from the ending position, which takes less than n times.

The run time could be minimized to O(n).

```
maxsum(list S)//input be a list S
{
    vector<int> max;//store max sums with ending j
    maxj[0] = S[0];
    for j from 1 to n:
    {
        maxj[j] = max{maxj[j-1]+S[j], S[j]};
    }//get the maximum sum ending at j
    int maxx = max[0];//the maximum of those maximum sum
    int index = 0;
    for i from 0 to n-1:
    {
        if (maxj[i] > maxx)
        {
            maxx = maxj[i];
            index = i;
        }
    }//get the maximum of maximum
    int front;
    int temp = maxx;
    for i = index; i >0; i--:
    {
        temp -= S[i];
        if temp == 0:
            front = i;
    }//trace back to find the start position
    return start position front+1 and end position index+1
}/run time O[n]
```

The tricky part of this problem is that we could use maxj[j] = max{maxj[j-1]+S[j], S[j]}; to find the maximum value if ending at j. This is true because the sequence is consecutive and the maximum could neither be sum with previous or itself.

Problem2:

Answer:

Using dynamic programming. Having a list to record the maximum profit, P[j], to be the maximum profit YuckDonald's can get from 1 to j. j is from 1 to n. Also considering that there should be k distance between two YuckDonalds. So, different with question1, we need to examine whether putting a restaurant at location 'a' is feasible.

```
maxprofit(k, n, pi)
{
    vector P[n];//have a list to record
    //the profit end at position n
    //the size is n
    for i from 1 to n:
    {
        P[i] = pi;
    }//initialize profit for base case
    for a from 0 to n:{
        for b from 1 to a-1{//the maximum profit can be
        //get from any profit in smaller position
        //add a restaurant
            if b-a > __k://check if a restaurant is allowed there
            {
                temp = P[b] + pa;
            }
            else
            {
                temp = P[b];
            }
            if temp >P[a]//check if this specific situation
            //constitute a better solution
            {
                P[a] >temp;
            }
        }
    }
    max = P[1]
    for i from 1 to n
    {
        if max < P[i]
        {
            max = P[i]
        }
    }//get the maximum of maximum
    return max;
}//run time is O(n^2)
Run time is O(n^2), as there is a double for loop related to n.
```

Question 3:

Answer:

Using dynamic programming. The problem of this question is that “a” is a word and “as” is a word. So if we know string S[0] to S[i] is valid, then we need to prove that string S[i+1] to S[j] is a valid string. Because the property of “a” and “as” can all be valid word, we need to check with all previous occasions. Have a list D[n], n represent the substring from 0 to n, where stores starting point of characters that cannot be a word. If the substring is a valid sentence, then its value should be n.

```
issentence(string s, size n)
{
    S[n+1];//a list check if it is true
    //have the first element to be position 0-0
    S[0] = =1;
    sen[n+1][];//a double list to store the
    //sentence compotition of word
    for i from 1 to n{
        for j from 1 to i{
            if (dict(substring(S[j], S[i])))
            //substring of s from position
            //S[j] to S[i]
            {
                S[i] = i;
                sen[i] = sen[j];
                sen[i].push_back(i);
            }//then substring from 1 to i
            //is a sentence
            else if (S[j] > S[i])
            {
                S[i] = S[j]
            }//the least position of i should
            //always be updated to nearest position that
            //there is not a word
        }
    }//O(n^2)
    if (S[n] == n)//to return back to sentence,
    //we coudld use S[n] as it store where a
    //word appears
    //but we need also care about whether the word
    //is used in the final sentence
    {
        //then there is a valid sentence
        for i in sen[n+1]
        {
            insert " " at s[i];
        }
        return s;
    }//return the sentence by tracng back
    else
    {
```

```
        return "there is no sentence"
    }
}
```

Because there is a double loop, so run time O(n^2)

Problem 4:

Answer:

a. There are 8 possible ways, since each column has 4 rows. Like below, where 1 represents a pebble:

	1				1		1
		1				1	
			1				1
				1	1	1	

b. First, we could see that there could not be consecutive two columns with 2 pebbles, so the best solution is to put a column with 2 pebbles and then with a column with 1 pebble repeatedly. So for even columns, there would be 1.5n sum, and for odd columns, there would be 1.5(n-1)+2 sum.
To do this dynamically, we could just start from 1 to n to get the optimal solution.
Do not consider empty type as we can always fill the column with at least 1 pebble.

```
optiplace(n)
{
    T[7];//eight Types
    C[7][3];//Campatibility of 8 types
    //using number 1 to 8 represent the 8 ways to put pebble
    K[n][];//store the optimal placement of length from 1 to n
    K[n].pushback(rand(T[5-7]))//random select a types
    //in T[5-7]
    for k from 1 to n:{
        K[k] = K[k-1];
        K[k].push_back(rand(C[K[n].last]));
    }
    return K[n];//return sub list at n of K
}//run time O(n)
```

Question 5:

Answer:

The idea of this problem is that a palindrome is symmetrical around its circle. So, to solve this question, we need to start from center to outer. Only if the center is a palindrome, a wider string could be a panlindrome.
Suppose we have a set P[i, j], which is the Boolean to determine whether it is a palindrome from position I to j in the string. “n” is the length od the string.
We also need to concern whether this palindrome is odd or even, because

```
longpan(string s)
{
    for i from 0 to n-1:
    {
        P[i,i] = 1;//0 character is always a panlindrome
        P[i,i+1] = 1;//1 character is always a panlindrome
    }
    for i from 0 to n-1
        for j from i+1 to n-1
        {
            P[i, j] = P[i+1, j-1] and (s[i] == s[j])
        }//it is panlindrome as long as its child is
        //a palindrome and s[i] and s[j] is equal
    max = 1;
    for i from 0 to n-1
        for j from i to n-1
        {
            if P[i, j]
            {
                if (max < (j-i))
                {
                    max = j-1;
                }
            }
        }
    return j-i
}//run time O(n^2)
```

Run time is O(n^2), since there is a double for loop in it.

Question 6:

Answer:

First naming the diagonal length from vertex I to j be L(I, j). n > 3. Like the hint, first find the minimum diagonal sum of smaller vertices. Basic case A(I, i) = 0. Let A(I, j) be the minimum cost by vertices I to j. Also notice that the equation of the A(I, j) is minimum of A(I, k)+A(k, j)+L(I, k)+L(k, j) where k is from i+1 to j-1.

```
mintri(n, L(i, j))
{
    for i from 1 to n-1
    {
        A(i, i) = 0;
    }
    for i from 1 to n-1//i is how many vertices included
    {
        for j from 1 to n-i-1//starting vertice
        {//vertices from j to j+i
            for k from 0 to i//k is used to get sub triangulation
            //from j to j+i
            {
                if (A(j, j+i) > A(j, j+k)+A(j+k, k+i)+L(j+i, j+k)+L(j+k, j))
                {
                    A(j, j+i) > A(j, j+k)+A(j+k, k+i)+L(j+i, j+k)+L(j+k, j);
                    //if a minimum exist, replace
                }
            }
        }
    }
    return A(1, n);
}//run time O(n^3)
```

Run time is O(n^3), because there is a triple loop related to n. The tricky part is that we be to add from small brick to large, so that we need to consider triangulation constitute smaller vertices but with different starting points, and then general to the triangulation of n vertices.

Problem 7:

Answer:

Also use dynamic programming. The tricky part of this question is that there is 2 dimensions that need to do dynamic programming. As the previous question, we define A(x, y) to be the maximized c when there is x*y piece. And this could be constituted from smaller piece. For these small pieces, we easily defined them in two categories. First is has the same width with the x*y piece, and the second is has the same length.

```
maxc(X, Y, c(a, b))//c be the price of unit a*b
{
    A(x, y) = 0;//initialized all values in the map
    //to be 0
    for x from 0 to X{
        for Y from 0 to Y{
            if c(x, y) exist
            {
                A(x, y) = c(x, y)
            }//basic case if cuting into one piece
            for i from 0 to x
            {
                if (c(k, y)+c(x-k, y)>A(x, y))
                {
                    A(x, y) = c(k, y)+c(x-k, y)
                }
            }//smaller pieces has same length
            for j from 0 to y
            {
                if(c(x, j)+c(x, y-j)>A(x, y))
                {
                    A(x, y) = c(x, j)+c(x, y-j)
                }
            }//smaller pieces has same width
        }
    }
}//run time O(XY(X+Y+n))
```

Run time is O(XY(X+Y+n)). The reason is that I have (a loop X times)(a loop Y times)(n times to get the price + X times + Y times) as in the pseudo code.

Problem 8:

Answer:

In this question, still use dynamic programming with two variables, I stages and B budget. To solve this question, just have a double for loops to increase from 0 to I stages and from 0 to B budgets. To calculate the best probability at certain stage and budget, we need to have another for loop, and assume we use l budget on this stage, so previous stage has B-m budgets. We try all the possible l budgets and get the optimal maximum probability.

```
maxredun(r[i], c[i], B)
{

    P(i, b);//maximum probability with i steps
    machines(i, B);//machines we used in optimal at certain budgets
    //for all the stages
    //and budget b
    //below is initialization
    for j from 0 to B:
    {
        P(0, i) = 1;
    }
    for j from 0 to i
    {
        P(j, 0) = P(j-1, 0)*r[j-1];
    }
    for (j = 1; j <= i; j++)
    {
        for (k = 1; k <=B; k++)
        {
            maxprob = 0;
            machiens(j, k) = 0;
            for (int l from 0 to k)
            {
                m = l/c[j-i];//number of machine can be used here
                sp = probability equation//stage probability
                left = k- m*c[i-1];
                tp = P(i-1, left)*sp//total probability
                if (maxprob < tp)
                {
                    maxprob = tp;
                    mahcines(j, k) = m;//update the machine redundundcy
                }
            }
            P(j, k) = maxprob//mac probability when j stages
            // and k budget
        }
    }
    redun[i]//list of machines redundency in optimal at budget B
    reb = B;//remaining budget
    for (j from i to 0)
    {
```

```
        redun[j] = machines(i, reb);
        reb -= redun[j]*c[i];
    }
    return redun;//return the machine redundencies at i stages and B budget
}
```

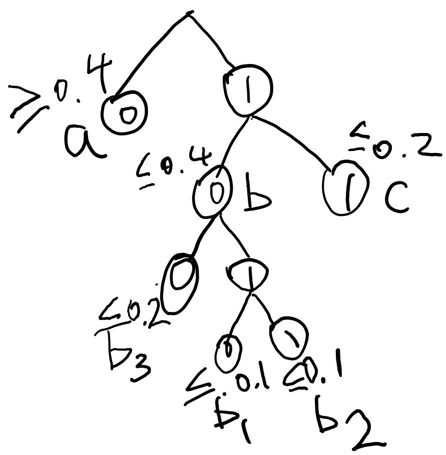
The run time is O(iB^2). "i" is the total stages, and B is the total budget. This is because there is a triple loop in the algorithms that each has i, B, and B times.

Problem 9:

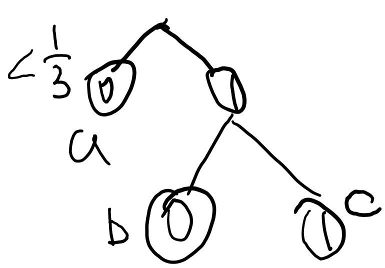
Answer:

If we look at the structure of the Huffman code, we could see that length 1 codeword is best fit for occurrence rate 0.5, and length 2 codeword is best fit for occurrence rate 0.25, length3 codeword is for 0.125 and so on.
To get the optimal Huffman code, we need always put character having more than 0.5 occurrence rate to be 1 code word.

a. Consider the boundary case that we could have three characters, for example a, b, c, with frequency 0.4, 0.4 and 0.2. Then we assign Huffman code to them. One of the optimal Huffman code we could assign is 0, 10, 11. We also know that we could merge and split character frequencies. For example, b, c could merge together (bc) with frequency 0.6 and code word 1. We could also split b into b1, b2, b3 with frequency 0.1, 0.1, 0.2. and we could know here that none of b1, b2, b3 could have length 1 codeword. So this boundary case could be generalized to any characters and frequencies with at least one character has more than 0.4 frequency. So that it is correct.



b. Also considered a boundary case that there is a character "a" <1/3 has codeword 0, and two other character b and c. Then b+c > 2/3. Which means that one of b and c need to be greater than 1/3, which is a contradiction with prerequisite that any characters should smaller than 1/3 and if b is > 1/3, then b is greater than a which means that in the rule of Huffman code b should be assigned to length 1 code word. Like previous question, this case can also be generalized.



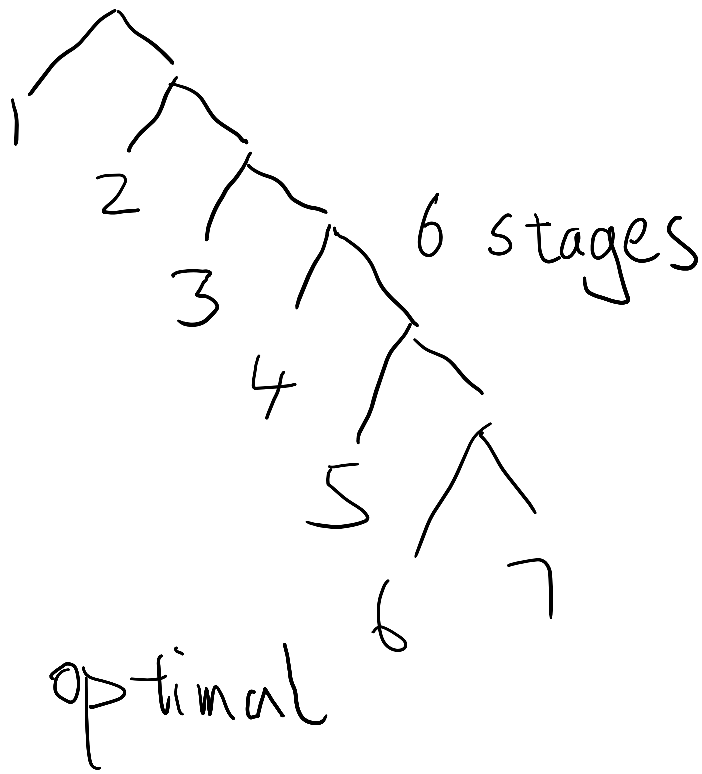
$\because b+c \geq \frac{2}{3}, b, c \geq 0$

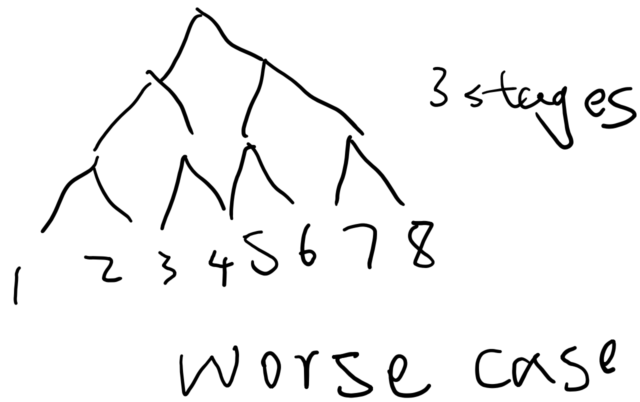
$\therefore b$ or $c \geq \frac{1}{3}$

Question 10:

Answer:

To be efficient expand the codeword length, we could find that the minimum character with a specific codeword length could be 1, and the maximum could be 2^i, where i is the length of the codeword. To get the longest codeword, it means that we need to put as less as characters on each stage/length. Here we could put minimum 1 character on each stage. However, to be a full binary tree, the last stage should always have more than 2 characters. So the maximum code length could be n-1. And the example frequency could be ½, ¼, 1/8, 1/16, 1/32… 1/(2^(n-1)), 1/(2^(n-1)), 1/(2^(n-1)).





Problem 11:

Answer:

Given the tree T(V, E). We can either use DFS or BFS. To do this, we just need to have a list, bool visited[], to record whether a vertex has been visited before. If a vertex is already been visited, then return false.

```
//the first input of the v should be the parent of the tree
sameDFS(vertex v)
{
    if (visited[v])
    {
        return false;//if already visited
        //then there are more edges that touches
        //vertex v
    }
    visited[v] = true;//already visit v
    for every child c of v:
    {
        sameDFS(c)//go deeper to look for childs
    }
    return true
}
```

The run time is O(V+E) for DFS, where V is number of vertices and E is the number of edges. In this question, E = V-1 because it is a perfect matching tree. So the run time could as simplified as O(V).

Question 12:

Answer:

We could have a graph that xn are the vertices and their equality constraints be edges. Then for disquality, we just need to do BFS or DFS to the vertex to see if it connects to vertex it should not connected.

```
//the first input of the v should be the parent of the tree
equality(X[n], C[m])
{
    split C[m] into e[i], ie[j];
    //split constraints by whether is is equality or disequality
    //e[i] is for equality and ie[j] is for unequality
    //i and j is depend
    have a graph with n vertices naming X[n];
    //the graph should not be directed
    for each constraint c in e[i]
    {
        conect verteices according to constraints;
    }
    for each constraint c in ie[j]:
    //suppose the two vertices in the constraint
    //are X[a] and X[b]
    {
        if nor DFS(X[a], X[b])
        //search on DFS for X[a]
        //to see if X[b] is connected to
        //X[a]
        {
            return false;
        }
    }
    return true;
}
DFS (X[a], X[b])
{
    if X[a] == X[b]
    {
        return false
    }
    visited[v] = true;//already visit v
    for every child c of v:
    {
        sameDFS(c, X[b])//go deeper to look for childs
    }
    return true
}
```

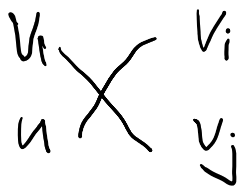
Run time: O(m(m+n)), because we need to check for m constraints and do DFS in running, which running time of DFS is O(V+E) = O(m+n).

Question 13:

Answer:

a. Suppose d1, d2, d3, d4 = (3, 1, 3, 1). Their sum is 8. However, this graph does not exist because when d1 and d3 connects to all other vertices, d2 and d4 should at least be 2 degrees.

b. 1. The relationship is like:



V1 has d1 neighbors; vi has di neighbors; and vj has dj neighbors. Because vj is a neighbor of v1, and vi is not the neighbor of v1, j>I, and di > dj, which means that there exists vertex u that is neighbor of vi but it is not neighbor of vj.

2. We already know a vi, vj, v1, u. If I get out the edges between (V1 and vj) and between (vi, u) and add edges (v1, vi) and (u, vj). Then all the property remains unchanged. So this E' exists.

3. v1 could have edges with both vi and vj, as previous. So it can also have edges with any vi or vj that complies with their property, and v2, v3, …, v_{d1+1} all complies with their property.

c. Having a list D[n] records the degrees. It is not necessary that D[n] need to be sorted. To make the graph exist, we just need to make sure that for every degree, the vertex could actually build an edge. If a vertex va has a degree, then another vertex must connect to va and has an edge, so that this "another" vertex also has a degree. Here we could suppose it connects to anyone vertex, for example, vb. Then a very simple way is that if va has n degrees, then we could have n other vertices' degree minus 1. If finally, when we want to pairs some degrees with other vertices, but there is no other degrees can be used, then there does not exist a graph. Otherwise, exists.

```
existG(D[n], n)
{
    for i from 0 to n
    {
        int temp = D[0];
        D.popfront();//remove the front vertex
        if (temp > sizeof(D))
        {
            return false;
        }//not enough verteices to be paired
        for j from 0 to temp
        {
            while D[j] == 0
            {
                remove D[j];
            }//the vertex D[j] is totally paired
            if D[j] does not exist
            {
                return flase
```

```
            }//there is not enough vertices
            //to be paired
            D[j]--;
        }//pairs the vertices
    }
    return true;
}//run time O(n^2)
```

Run time is O(n^2). The reason is that there is a double for loop of n and d. The maximum of d is n, so the run time is O(n^2). The algorithms used the solution to pair all the vertices. If all the vertices can be paired with their degrees, then there is a graph. If not, then the graph does not exist.

Problem 14:

Answer:

Suppose that we already know the basic Huffman coding. Here it is very similar for prefix-free encoding. In Huffman coding, we use fi for ith word to get the li, and fi is constant for ith word. To minimize the cost, we assigned words with bigger fi with smaller li. In prefix-free encoding, the fi and ci is constant for ith word, hence fi*ci is also constant. Here we could take fi*ci to look for li as in Huffman coding we use fi to look for li. We just need to sort the word from bigger value of fi*ci to smaller value of fi*ci. Then we could assign code to it.

Problem 15:

Answer:

The idea is very simple: letting the customer with smaller service time to be served first, then others could have less waiting time in average. So in the code we need just to sort the customer with ti from small to large.

```
order(i, t[i])//i customers with t[i] service time
{
    list order[i] = 1, 2, 3...n;//this is used to remembered the order
    sort(t[i], order[i], 0, n);//this sorting algorithms
    //when sorting t[i], it also moves order[i]
    //and order[i] is remembered
    return order[i]
}
```

We do not need to compute exactly how much time it costs. This algorithm is optimal because greedy method is used and it is true for this question.
Suppose the sorting algorithm cost O(nlogn), so that this algorithm costs O(nlogn)

Problem 16:

Answer:

1. The reason is that any edge connects two vertices, which means that an edge always gives a degree to two vertices. For |E| edges it connects totally 2|E| times, which also means that there are total 2|E| degrees for all the vertices in graph G(V, E).
2. Suppose that there are odd number of vertices whose degree is odd. Then for the total degrees of vertices whose degree is odd, it is also odd because odd*odd is odd. And the total degree of vertices whose degree is even is even, because multiples of even is always even. Then the total numbers of degrees of a Graph is odd as even plus odd is

odd. This is contradictory with what we got in the first question. So there cannot be odd number of vertices whose degree is odd.

3. If consider indegree and outdegree separately, it is not hold because the sum of indegree or outdegree cannot always be even. However, if consider degrees as a whole, then there is no difference for this statement.

Problem 17:

Answer:

Have the graph G(V, E) and edge e(u, v). If there is a circle about u, then if we do DPS from u, it will finally go back to u. To examine if there is a circle about e, then we just need to take out e(u, v), and then we do DFS from v to see if it finally will to back to u. Run time of this algorithm is O(V+E) because DFS cost most of the time.

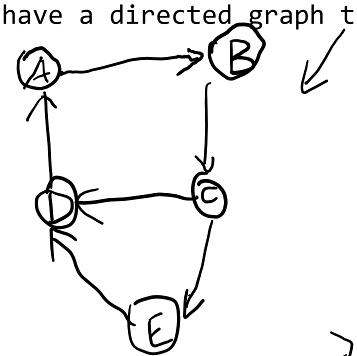
```
ifcircle(G(V, E), e(u, v))//suppose e exists
{
    take out e(u, v);
    for every childs c of v:
    {
        if ch == u//check if
        //reach u
        {
            return false
        }
        if(!DFS(c))//do DFS
        // and if any loop in this
        //return false
        //return false back
        {
            return false
        }
    }
}
DFS(c)//dfs search
{
    if no child
    {
        return true
    }
    for every childs ch of c:
    {
        if ch == u
        {
            return false
        }
        if(!DFS(c))
        {
            return false
        }
    }
    return true
```

```
}
Run time is O(V+E).
```

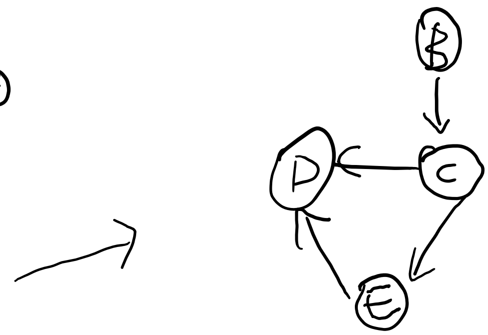
Problem 18:

Answer:

a. If we run DFS search on the graph, then have a list of every vertices visited by the time is get visited. Suppose there is such list [a,g,e,r,n,x], where a is visited first and x is visited last. If we removed the last visited vertex x, we could still do DFS on the remaining vertices because they do not depend on edges with x to connect with others, which means that there are still connected to each other. We could remove any vertex that we last visited and the remaining vertices will keep connected.
b. Suppose we have a directed graph that strongly connected as below:

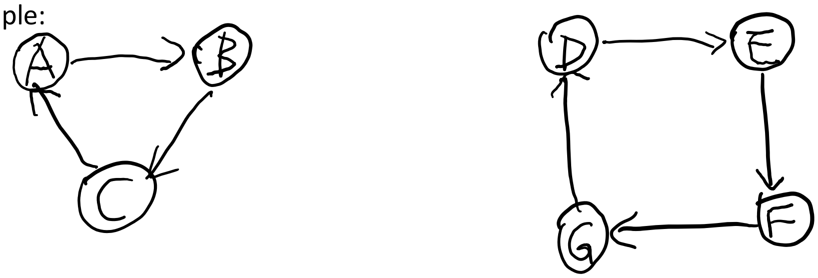


If we moved A, it becomes:

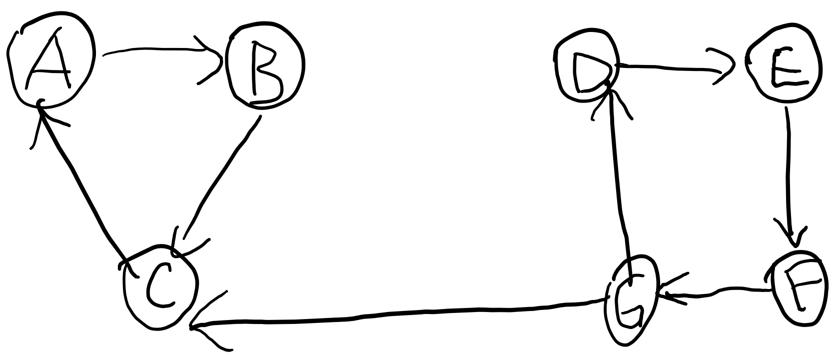


Which is not strong connected because there is no path from D to B.

c. It is true because adding one edge can only provide a tunnel with 1 direction. Once go through that edge, it cannot go back.
For example:



The above is a graph with 2 strongly connected components. If add an edge e(G, C), it still cannot made the whole graph strong connected, because C cannot go to G. It cannot be strongly connected by adding only 1 edge.



Problem 19:

Answer:

Use depth first search (DFS) to get as many paths as possible which start from s and contains t. In this search, if t is found, then change to another path to check if there is t on the path.

```
//a directed acyclic graph
//every vertice has .path which means how many paths
//go through this vertice can reach t
int stot(s, t)//start vertice s and end vertice t
{
    if s == t
        return 1 //if s is t, then find one path
    else if s has childs{
        for each children c of s{
            //search each child if there is t
            //and search child's child
            //until there is no child
            s.path = s.path + stot(c, t)
        }
        return s.path//return the total number of paths found
    }
    return 0//if no t is found, then this path return 0
    //which means that this path cannot find t
}//this is a depth first search because we use for loop to
//get from the top vertice to the bottom vetice and then
//change to another edge by return (go back)
//so the run time is O(V+E)
//where V means vertices and E means edges
//which is linear
```

Problem 20:

Answer:

This question, in other word, is asking to examine whether a directed acyclic graph's any adjacent two vertices are directed in a list after topological sort. This linearization is unique.

```
ifonce(G(V, E))
{
    topological sort(G)
    c[|V|]//this is the list we got after the topological sort
    for i from 1 to |V|-1
    {
        if e(c[i], c[i+1]) does not exist
        {
            return false
        }//if teo adjecent verteces is not connected, then it is false
    }
    return true
}
```

Run time is G(V+E). This is because topological sort is changed from DFS, which has run time O(V+ E), and the for loop cost O(V).

Problem 21:

Answer:

In problem 16, we already know that for any undirected graph, there must be even number of vertices whose degrees are odd. Suppose we have such graph G(V, E), who has 2n vertices with odd degrees, and these vertices now can paired into n pairs. According to Euler tour, a graph can form a Euler tour if and only if there are no vertices that have odd degrees. If we add an edge between each pair of odd vertices, then all vertices have even degrees, which formed an Euler tour. A Euler tour means that there is a graph circle that can travel all the vertices and return back to the origin vertex and at the same time travel any edges only once. If we get out of the edges we added to the graph, then we still have path between each pair of vertices with unique edges but we just cannot go back without the same path. So the statement is correct.

Problem 22:

Answer:

If T is a shortest path tree, then its path is always smaller than or equal to any path we random have in the graph G from vertex s to any vertex, supposing this unknown vertex is v. Then how to build a path in linear time? We could use a path from s to vertex u plus an edge from u to v to build a path, and this path's weight should be bigger or equal to the path in the T from s to v. Because we only compare weight here, so we could just have a list, called W[V-1] to remember the weight. And compared W[u]+L[e(u, v)] and W[v].

```
bool shorestpath(G(V, E), T(V, E'))
{
    L[e(u, v)]//store the weight on every edge
    RUN BFS(T) from s, and have a list W[v] that stored weight from
    vertex s to vertex v
    //note that this distance, inthereotical,
    //should be shortest distance

    for every edge e(u, v) on T:
    {
        if (W[u] + L(e(u, v)) < W[v])
        {
            return false;//this path should not be smaller
            //than T tree
        }
    }
    return false
}//run time O(V+E)
```

Run time is O(V+E) because we do BFS in the algorithm, and the for loop cost O(E'), which is smaller than O(E). This algorithm is true because it carefully and fully checks all the possible situation that could made T not the shortest. It is efficient because we use dynamic programming in it.

Problem 23:

Answer:

To do this, we need to find the shortest cycle of every vertices in the graph G(V, E). To find the shortest cycle, we can use the Dijkstra's algorithm to find path from u to v and path from v to u, which connects to form the shortest cycle, which cost O(V^2). And we need to do for every node, so total cost would be O(V^3)

```
shortestlength(G(V, E))
{
    vertex set Q
    dist[V][V]//distance from one vertex to another vertex
    //initialized to infinity
    for every vertices v of graph G
    {
        while Q not NULL
        {
            get vertex u in Q that dist[v][u] is min
            get out of u from Q

            for every neighbor n of u:
            {
                temp = dist[v][u] + length(u, n)//length(u, v)get the weight of
                //edge e(u, v)
                if temp < dist[v][n]
                {
                    dist[v][n] = temp
                }
            }
        }
    }//O(|V|^3) find the shortest path
    shortest = dist[1][2]+dist[2][1]//initialize shorest cycle distance
    for every vertices v of G
        for every vertices u of G except v
        {
            if shortest > dist[v][u] + dist[u][v]//formed a cycle
            {
                shortest = dist[v][u] + dist[u][v]
                //remembered the shortest
            }
        }
    return shortest
}//Run time O(|V|^3)
```

Run time is O(|V|^3), because it has a triple loop about vertices V. This algorithm is efficient because it get every cycle in the graph, which is complete.

Problem 24:

Answer:

Guess we have two vertices u, v. To find the shortest path from u to v and also pass through v0, we could simplify this question by finding the shortest path from u to v0 and the shortest path from v0 to v. So here we need to use Dijkstra's algorithm 2 times, and then do plus two lengths together.

```
shortestlength(G(V, E))
{
    vertex set Q
    dist[V][V]//distance from one vertex to another vertex
    //initialized to infinity
    v0 here is represented by v
    while Q not NULL
    {
        get vertex u in Q that dist[v][u] is min
        get out of u from Q

        for every neighbor n of u:
        {
            temp = dist[v][u] + length(u, n)//length(u, v)get the weight of
            //edge e(u, v)
            if temp < dist[v][n]
            {
                dist[v][n] = temp
            }
        }
    }
    restore Q back
    while Q not NULL
    {
        get vertex u in Q that dist[u][v] is min
        get out of u from Q

        for every neighbor n of u:
        {
            temp = dist[u][v] + length(n, u)//length(u, v)get the weight of
            //edge e(u, v)
            if temp < dist[n][u]
            {
                dist[n][v] = temp
            }
        }
    }
    list shorest[V][V]//store the shortest paths
    for every vertex s of first half of V, except v0,
    and for every vertex t of second half of V, except v0
    //this for loop can do in only one for loop
    {
        shorest[s][t] = dist[s][v0] + dist[v0][t]
```

```
        shorest[t][s] = dist[t][v0] + dis [v0][s]
    }
}//Run time O(|V|^2)
```
Run time is O(|V|^2), since there is Dijkstra's algorithm in it. The algorithm is complete and efficient.

Problem 25

Answer:

This question add the vertex costs, so we just need to add vertex cost in Dijkstra's algorithm and set the initial cost[s] be c_s.

```
shortestlength(G(V, E), s, l(e(u, v)), c(v))
{
    vertex set Q
    cost[V]//distance from one vertex to another vertex
    //initialized to infinity
    cost[s] = c(s)
    while Q not NULL
    {
        get vertex u in Q that cost[v][u] is min
        get out of u from Q
        for every neighbor n of u:
        {
            temp = cost[u] + c(n) + l(e(u, n))//length(u, v)get the weight of
            //edge e(u, v)
            //here add the vertex costs
            if temp < cost[n]
            {
                cost[n] = temp
            }
        }
    }
    return cost;
}//Run time O(|V|^2)
```
The run time is O(|V|^2) , because of the Dijkstra's algorithm we used here. The algorithm is complete and efficient because of Dijkstra's algorithm and it calculate total cost of each vertex with vertices costs.

Problem 26:

Answer:

a. Since the cycle is totally negative weight, then Σw= Σrc- Σp < 0, which means that Σrc < Σp. However, r^* = Σp/ Σc, and c is positive, then r < r^*
b. The same prove as sub-question a. Σw= Σrc- Σp > 0, so Σrc < Σp, which means Σr < Σp/ Σc. However, r^* = Σp/ Σc, and c is positive, then r > r^*
c. R is the maximum ratio of profit and cost of some edge. And the question, in other word, needs us to find the cycle with maximum weight, and then make sure the cycle is larger than R-€ or not.

```
maxweight(G(V, E), accuracy ac, R, p, c)
{
    for every vertex v in G
    {
        find the largest cycle from v to virtual
        //the exact calculation could be like problem 23
        //which get the shortest cycle
        //but here I got the maximum cycle
        find the ratio r = sum(p)/sum(c) in cycler[V]// of each bertex v
        if (r >= R-ac)
        {
            return this cycle
        }
    }//run time O(|V|^3)
}
```

Run time is O(|V|^3) as in question 23, as we calculate all the maximum path start from any vertex. The algorithm is efficient and complete.

Problem 27:

Answer:

a. CLIQUE-3 is not NP because the question can be solved in polynomial time as in sub question d. This is because g is restricted to be smaller than or equal to 4, because that there are maximum 3 degrees each vertex. Then we could solve it in polynomial time.
b. Well, to prove CLIQUE-3 is NP-complete, we need to do reduction from CLIQUE to CLIQUE-3 instead of reduction from CLIQUE-3 to CLIQUE. All cases of CLIQUE should need can be transformed to CLIQUE-3 in polynomial time. So it is incorrect.
c. Because CLIQUE-3 restrict the degree to be at least 3, then a vertex can in maximum connects to 3 other vertices. Then the maximum size of a clique in a CLIQUE-3 is 4, rather than defined in the problem ">=|V|-b". And the transform will not successful because that of VC-3 what limits degree to 3, this does not limit the transform to be CLIQUE-3, but rather it leads to CLIQUE.
d. CLIQUE-3 is not NP-complete. For CLIQUE-3 graph G(V, E), we need to check does G contain a clique of size g. Here we could just check for every vertex. If a vertex has g-1 degrees, then check its neighbors whether they are mutually connected, which cost g constant time (maximum 4).

```
Clique3(G(V, E), g)
{
    for every vertex v in g:
    {
        if v has g-1 neighbors
        {
```

```
            bool indi = true//indicator whether it is a clique
            have a neighbor list nei[]//that
            //include v and its neighbors
            for every neighbors n of v
            {
                if !(n has g-1 neighbors and all the neighbors are in the nei[])
                {
                    indi = false
                }
            }
            if indi is true//there is a clique at v with size g
                return true;
        }
    }
    return false
}//take O(|V|*g) time, g is <= 4, so run time is O(|V|)
```

This algorithm is efficient because it exhausted have all the situation.

Problem 28:

Answer:

This question can be solved be reducing EXACT 3SAT problem.

1. NP. If we know an answer of a list a variables, then to verify its correctness, we just need to bring it into the set of clauses to see whether its output is 1 or 0 (true or false), which cost O(1) theoretically.
2. NP-completeness. We need to do a reduction from EXACT 3SAT problem to EXACT 4SAT problem. Let's consider a clause in EXACT 3SAT, for example, (a v b v c). This can be transformed to form of 4SAT, (a v b v c) = (a v b v c v d) ∧ (a v b v c -d), where d is added randomly. For every clause of EXACT 3SAT with 3 literals, it can be transformed to two clauses with 4 literals each in liner time. Then EXACT 3-SAT is true if and only if its transformation EXACT 4-SAT is true, which proves the NP-completeness of EXACT 4-SAT problem.

Problem 29:

Answer:

1. NP. K-Spanning Tree problem is NP. If a correct answer a spanning tree T is provided, then to verify its correctness, we just need to go through every vertex of T and check whether its degrees are <= k, which cost O(|V|).
2. NP-Completeness of 2-Spanning Tree. A 2-Spanning Tree is just a path, because it takes 1 degree to directed in the vertex and only 1 degree remaining for directed out. So 2-Spanning Tree problem is a path problem. We could find that 2-spanning Tree problem is just an undirected Hamiltonian path problem, which asks whether there is a path in an undirected path that visits each vertex exactly once. Because Hamiltonian path problem is NP-complete, then 2-spanning tree is NP-complete.
3. NP-Completeness of k-Spanning Tree, k >2. We need to do reduction from 2-Spanning Tree to k-Spanning tree problem. There are a lot of ways to do this reduction. Let's consider a simple way, which is 2-spanning tree reduced to 3-spanning tree. This 2-spanning tree is from G(V, E). From this graph, we add a vertex and an edge to each vertex in the G. these vertex are only connect to only one originally vertex, then we have a graph G'(V+V, E+V). Then for every problem of 2-spanning tree of G is the same as the 3-spanning tree problem of G'. 2-spanning tree of G is true if and only if 3-spanning tree of G' is true. Then k-spanning tree is NP-complete.

Problem 30:

Answer:

1. NP. If provided V1’ and V2’, then we could two graphs G1’ and G2’, and we could verify if these two graphs are identical by checking vertices and edges one by one in polynomial time.
2. NP-Complete. First, there is a reduction from Subgraph Isomorphism Problem to Clique problem. Subgraph Isomorphism asks whether G1 has a subgraph that is identical to G2. The corresponding is this: whether a graph G1’, which is a subgraph of G1, is identical to a graph G2, which is a clique of size |V|-b of G1. This means that Subgraph Isomorphism Problem is also NP-complete. Then we need to prove that Subgraph Isomorphism Problem can be reduced to this Maximum Common Subgraph problem. The transformation is easy: whether a graph G1’, which is a subgraph of G1, is identical to a graph G2’, which is a subgraph of G2. (The subgraph here only deletes vertices and their edges on I them). Subgraph Isomorphism is true if and only if Maximum Common Subgraph is true. So that Maximum Common Subgraph is NP-complete.

Problem 31:

Answer:

1. NP. KITE problem is NP. Consider a subgraph G’ of G with 2g nodes is given. Then to verify its correctness, we just need to determine how many edges a vertex have and get rid of those with only 1 edge and then verify if the remaining graph is a clique of G.
2. NP-Complete. Clique problem can be reduced to KITE problem. Suppose we have a clique problem that graph G(V, E) and find whether a g clique exists. Then to transform to KITE problem, we could add 1 vertex with 1 edge to connect to every vertex of G and constitute a graph G’(V+V, E). Then we could solve KITE problem on this graph G’ which is to find a subgraph contains 2g nodes and have a clique of g nodes. A clique problem is true if and only if its corresponding KITE problem is true.

Problem 32:

Answer:

a. NP. Suppose we have a graph G(V, E), and there is a feedback arc set E’, which then reduced the graph to be G’. First, we do topological sort on G’, which then we could check whether G’ is acyclic. Then E’ is verified. Thus, it is NP.
b. As said in the problem, then there is always a cycle between w’j, wi, w’i, and wj for every (vi, vj) belongs to E. Consider G has a vertex cover S, then remove all the edges wi to w’I in G’ that belongs to the vertices in S. Then This is no cycle in G’ except cycle like because the necessary edges to make vertex cover working are been removed. Then vertex cover cannot worked to connect each other.
c. Suppose G’ has a feedback arc k of b, this feedback arc can always simplified to a feedback arc k’ containing only edges like e(v, v’). The reason is that removing edge e(v, v) is just could have the purpose of removing any edge like e(u, v) and e(v, u). Then the nodes in the set k’, constitutes a vertex cover of k’ in graph G. This is because k’ has the ability to disconnect any edges between random vertex u and random vertex v in G’, which means that k’ connects all the edges in graph G.

Then, FAS is NP-Complete.

Problem 33:

Answer:

The thing we can do approximation here is that for nodes i, j, and k, e(I, j)+ e(j, k) > e(I, k).. By using this property, we can approximate a minimum cost and made sure that the cost is smaller than minimum cost. Suppose we have such minimum Steiner tree with cost C, then go through the whole tree in a cycle would cost 2C. Then because of the triangular property, we can seen that go through any edge connected the vertex v and u of V' could cost less than go through another 2 edges that formed a triangle together. Then when we go back, we would go through the same triangle. By general this trick to every vertex v of V', we could see that the minimum-cost tree that includes the vertices V' could be approximate to 2C. Algorithm is like below:

```
appro3steiner(G(V, E), V1)//v1 is the terminal nodes v'
{
    do DFS on G and remember cost of each edges cost(u, v)
    for every pairs (u, v) of V1
    {
        the cost is equal to the total cost of any path from u to v
        //because of the triangle property
        //so cost between terminal nodes has to be smaller than
        //the total cost of path
        remembered the cost in a list C(u, v)
    }
    use above costs to find a minimum spanning tree
    //this can be done in greedy algorithm
    //in run time O(m logn)
    //or more optimal O(m) according to Wikipedia
    returned the order of the spanning tree on V1
}//O(n^2)
```

Run time is approximate to O(V^2), because the exact algorithm is not given in the for loop, but I believed that we have to cost another O(V) to get a minimum path. This algorithm is efficient, because we minimized cost to all vertices terminal and then use minimum spanning tree algorithm to find a minimum spanning tree about V'.

Problem 34:

Answer:

a. When k = 2, this problem is simplify to that after removing a set of edges, the DFS search start from terminal vertex s cannot find another terminal vertex t (the minimum s-t cut problem). To solve this problem efficiently, we could exhaust find every path from s to t, which can be done in polynomial time. Then we could decide which and how many edges to be cut. The decision-making process can also be done in polynomial time. This is one way to solve this problem, which may not be the most efficient, but this proves it can be done in polynomial time.
b. We could have an approximation algorithm with ratio at 2 by using the minimum s-t cut problem above. For each terminal node si, do min s-t cut problem with the remaining terminal nodes as one node.

```
appro3mutiway(G, si)
{
    for every node u of si
    {
        collapse S\u into a single node v
        min s-t cut(u, v);//function of min s-t cut
        remembered the cutting edges in list E[]
```

```
    }
    return E[]
}//the reason why the approximation is 2 is that
//the total cuts that seperate each nodes from the
//rest of the graph is half of the optimal
//multiway cut, because that any edge is adjecent
//to 2 components, then a cut in an optimal multiway cut
//could be the cuts in 2 terminal nodes
//so the ratio is close to 2
```

Problem 35:

Answer:

a. Let xi represents whether actor i is chosen, xi is either 1 or 0. Let yj represents whether investor j is chosen, yj is also either 1 or 0. Then the profit is:

$$\sum_{j}^{m} y_j p_j - \sum_{i}^{n} x_i s_i$$

"The goal is to maximize this profit", then this problem can be expressed as:

Max:

$$\sum_{j}^{m} y_j p_j - \sum_{i}^{n} x_i s_i$$

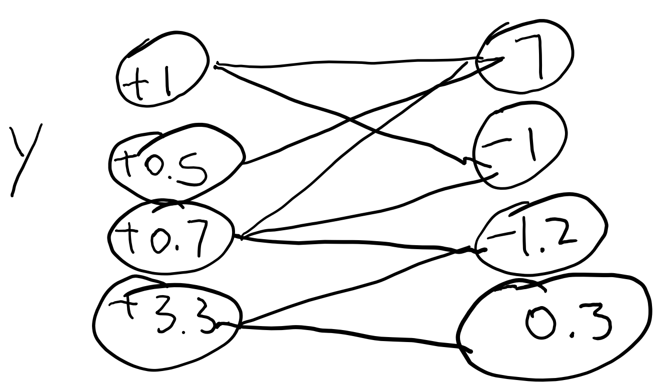
subject to:

$$\forall i \in L_j$$

$$\mathrm{x_i}, \mathrm{y_j} = 0 \; or \; 1, \forall \mathrm{i}, \forall \mathrm{j}$$

Also need to fulfill the requirement that if yj = 1, then xi = 1, for $i \in L_j$

b. This problem asks to find whether there must exist a maximum profit. We could notice that for each y we choose, there must be some x be chosen, so we could have a bipartite graph that one side is y and another side is x. The edges between them means that they need to be both 1 or 0 at the same time. Then we could find that there is many cycle in the graph which constitutes the possible set of the investor and actor. We also noticed that for any y chosen, its neighbors must be chosen, then its neighbors' neighbors could be chosen to maximize the profit. We then add weights to each node such that profits are well-illustrated, investment to y and charge for x. Then, to solve this problem, we could just start from each y and find the maximum flow problem solution (maximum flow can be solve in polynomial time) that maximized the profit. Then we could get a maximum profit. The sample model is like the graph below, where numbers in the circle are the profits. We could start from +1 in y first and fo the maximum flow problem. Do the maximum flow on every node of y. Because we could have a matrix about xi and yj in a matrix that only has 0 and 1, which constitutes an unimodular matrices, this polyhedron is integral. Thus there is an integral optimal solution.



NOTE: Thank you instructor, TA and grader for your help and work. Wish we all have a good luck!