Due 23:59 Oct 23 (Sunday). There are 100 points in this assignment.
Submit your answers (**must be typed**) in pdf file to CourSys
`https://coursys.sfu.ca/2022fa-cmpt-705-x1/`. Submissions received after 23:59 of Oct 23 will get penalty of reducing points: 20 and 50 points deductions for submissions received at $[00:00, 00:10]$ and $(00:10, 00:30]$ of Oct 23, respectively; no points will be given to submissions after 00 : 30 of Oct 23.

1. (Chapter 6 Problem 1 of the text book) 20 points

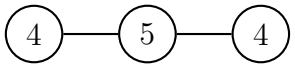
   A subset S of nodes in a graph G is an independent set if no two nodes in S is connected by an edge. Let $G(V, E)$ be a path ($V(G) = \{v_1, v_2, .., v_n\}$ and $E(G) = \{\{v_i, v_{i+1}\}, 1 \leq i < n\}$). Each node v_i of G is assigned a positive integer weight w_i. For a subset $S \subseteq V(G)$, the weight of S is $w(S) = \sum_{v_i \in S} w_i$. Consider the problem of finding an independent set S of G such that $w(S)$ is maximized.

   (a) Give an example to show that the following "heaviest-first" greedy algorithm does not always find an independent set of maximum weight.

   Start with $S = \emptyset$
   while there is a node in $V(G) \setminus S$ **do**
   pick a node v_i of maximum weight and add v_i to S
   delete v_i and its neighbors from $V(G)$
   endwhile
   return S

Answer

These questions can be solved by giving a counter example for each of the algorithms. Taking the queue form the Figure 6.28 from the book (page 312), we can have a graph with weights 4, 5, 4. The greedy algorithm with its heaviest first policy will consider the middle node 5 first and that will be in its independent set and it will be done. Whereas, the set with the first and the last node will have the maximum value for the independent set.



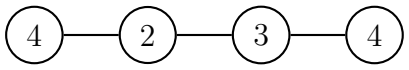
(b) Give an example to show that the following algorithm also does not always find an independent set of maximum weight.

> Let S_1 be the set of all v_i, where i is an odd number
> Let S_2 be the set of all v_i, where i is an even number
> (Note that S_1 and S_2 are both independent sets)
> Determine which of S_1 or S_2 has greater total weight, and return this one

---

Answer

For this we can also give a counter example, which can prove that just by going off the naive assumption of choosing odd or even nodes does not give us the best result. We can consider the following graph where the weights are 4, 2, 3, 4. This would mean that the given algorithm would consider 1 and 3 to be the independent set. Whereas, the independent set with the maximum weight would be the first node and the fourth node.



(c) Give a dynamic programming algorithm that finds an independent set of G with maximum weight. The running time should be polynomial in n, independent of the values of the node weights.

Answer

This can be solved by *inductive reasoning*. If we a node which is in the independent set so its edges will be deleted. A certain node can or cannot be inside the independent set, it cannot be both or none. Now consider that we have a node i is inside the independent set this would mean that we cannot have the node $i-1$ in the independent set so the node $i-2$ can then be considered to be in the independent set or not. Given that we have a set of nodes $\{v_1, v_2, ..., v_i\}$, we can maintain a list of weights W_i, which holds the current cumulative weight of the node. Weight of a certain node is represented by w_i. So this is equivalent to the recursive formulation of:

$$W_i = max(W_{i-1}, w_i + W_{i-2})$$

This means that if we have the node node i is in the independent set then we can recursively check if the node $i-2$ is in the independent set, because $i-1$ cannot be in the independent set (if node i is in the independent set). Otherwise, we check if the node $i-1$ is in the independent set, which also gets passed through the sequence described in the previous sentence. The algorithm looks like this:

We have a list of nodes V[].
Make a new node v[0].
Initialize the list W with zeros.
Loop through the vertices: i = 1...n
    recursively update W[i] = max(W[i−1], V[i] + W[i−2])

Since we can have the value we want in W_n. We can back track through the computations (maximum functions) which would mean that we spent constant time per iteration and there are a total of n iterations, which makes the running time be $O(n)$.

2. (Chapter 6 Problem 2 of text book) 20 points

   For each week i, there is a job h_i with high stress and revenue h_i and a job l_i with low stress and revenue l_i. A team T can complete only one job in week i, either h_i or l_i. A job h_i can be assigned to T in week i if $i = 1$ or no job is assigned to T in week $i - 1$. A job l_i can be assigned to T in week i for every i. Given the jobs $h_1, .., h_n$ and $l_1, .., l_n$, the goal is to either assign a job from h_i, l_i or not assign any job to T for week i such that the total revenue of the assigned jobs is maximized. Example: $h_1 = 5, h_2 = 50, h_3 = 5, h_4 = 1$ and $l_1 = 10, l_2 = 1, l_3 = 10, l_4 = 10$. The optimal solution for the example is no job in week 1, job h_2 in week 2, job l_3 in week 3 and job l_4 in week 4, with total revenue $0 + 50 + 10 + 10 = 70$.

   Give a dynamic programming algorithm to find the maximum revenue of the problem in $O(n)$ time.

Answer

For this question, the team T can take on a high stress job they need not to have a job in the previous week. This is why in the question, the team T does not take any job in the first week and take a high stress job in the second job. In the last two weeks the team takes on low stress jobs as they have a higher revenue value compared to the high paying jobs, which is a no-brainer. We can solve this problem with the help of Dynamic Programming. If we have jobs starting from 1 to i, and we choose a low stress job at i we can have any job till $i - 1$ because low stress jobs do not have a preceding job requirements, but if we ought to choose a high stress job for week i, we can have no job on $i - 1$ which would mean that we can have any job till $i - 2$. So, if we were to see this problem in two parts where we have any job h_i or l_i and we can have the maximum of Low stress job or High stress job till $i-1$ (we can consider LowStress as a function which can be recursively visited) $max(LowStress(i - 1), HighStress(i - 1))$ if job at week i is a low stress job and we can have the maximum of Low stress job or High stress job till $i - 2$ (similarly HighStress can be a recursive function) $max(LowStress(i - 2), HighStress(i - 2))$ if the job at week i is to be a high stress job. So if we combine both of these formulations we get:

$$value(i) = max(l_i + value(i - 1), h_i + value(i - 2))$$

where, *value* is a function which is recursively called. This would achieve a running complexity of $O(n)$ as we just have essentially if conditions and we visit each node at the most once.

---

3. (Chapter 6 Problem 13 of the text book) 20 points

The problem of searching for cycles in graphs arises naturally in financial trading applications. Consider a firm that trades shares in n different companies. For each pair $i \neq j$, they maintain a trade ratio r_{ij}, meaning that one share of i trades for r_{ij} shares of j. Here we allow the rate r to be fractional; that is, $r_{ij} = 2/3$ means that you can trade three shares of i to get two shares of j. A trading cycle for a sequence of shares $i_1, i_2, .., i_k$ consists of successively trading shares in company i_1 for shares in company i_2, then shares in company i_2 for shares i_3, and so on, finally trading shares in i_k back to shares in company i_1. After such a sequence of trades, one ends up with shares in the same company i_1 that one starts with. Trading around a cycle is usually a bad idea, as you tend to end up with fewer shares than you started with. But occasionally, for short periods of time, there are opportunities to increase shares. We will call such a cycle an opportunity cycle, if trading along the cycle increases the number of shares. This happens exactly if the product of the ratios along the cycle is above 1. In analyzing the state of the market, a firm engaged in trading would like to know if there are any opportunity cycles. Give a polynomial-time algorithm that finds such an opportunity cycle, if one exists.

Answer

This is a very interesting problem, as the question states that if the product of ratios is greater than 1, we get an increase in the number of eventual shares. To write this in mathematical form we can say that:

$$\Pi_{i,j \in Cycle} r_{ij} > 1$$

To look at this problem based on intuition, what we mean here is that the numerator of all of the cycle combined is greater than that of the denominator, which means that the company i will end up with higher number of shares than they started out with. To ease this problem and make it solvable in the desired running time complexity we can take the logarithm of the whole equation and the formulation becomes:

$$log[\Pi_{i,j \in Cycle} r_{ij}] > log(1)$$
$$\Sigma_{i,j \in Cycle} log r_{ij} > 0$$

So, in order to detect a loss or a negative cycle we can take negative on both sides of the formulation to make get the formula which we want to avoid and find out in the trading cycle.

$$-[\Sigma_{i,j \in Cycle} log r_{ij} > 0]$$
$$\Sigma_{i,j \in Cycle} - log r_{ij} < 0$$

This problem can be solved now using directed graphs and we can have $-logr_{ij}$ cost on each edge of the directed graph. This would mean that we can naively use a modified version of Dijkstra's Algorithm where the algorithm runs and if it detects that it is stuck in a loop, we can call it a negative cycle, by maintaining a variable for path (which holds the already visited nodes). Dijkstra's Algorithm is good for this because it is prone to getting stuck in negative cycles. But as we already have the assumption that there will be a cycle (in the name trading cycle) in the graph, we just run a for loop over the nodes we have and maintain a sum variable. If the sum variable gives out a negative value, we can be sure that it is a negative cycle. This has a running complexity of $O(n)$.

---

4. 20 points

   Two dynamic programming algorithms Knapsack and Knapsack1 for the knapsack problem are introduced in class. Implement the two algorithms and compare the space and time used by the algorithms. Note that the space efficiency of Algorithm Knapsack can be improved. Your answers to this question should include the the following results:

   (a) The values of $M[i, w]$ for each $i = 0, 1, .., n$ and $w = 0, 1, .., W$ for problem instance $I = \{1, 2, 3, 4, 5\}$, $\{v_1 = 3, v_2 = 6, v_3 = 18, v_4 = 22, v_5 = 28\}$, $\{w_1 = 3, w_2 = 4, w_3 = 5, w_4 = 6, w_5 = 7\}$ and $W = 11$ to show your program for Algorithm Knapsack solves the knapsack problem.

   (b) The contents of $A(j)$ for each $j = 1, .., n$ for the same instance in (a) to show your program for Algorithm Knapsack1 solves the knapsack problem.

   (c) For a few large problem instances, the memory space used by Algorithm Knapsack and the memory space used by Algorithm Knapsack1, brief comparisons on the space used by and running time of the two algorithms. The memory space can be either a measured maximum memory size or the maximum number of array elements used by the algorithms. You can randomly generate problem instances.

   You may include comments to show your tricks to improve the memory space and running time of your implementations. You do not need to submit your program codes.

Answer Knapsack The tabular form for knapsack would be like this:

Item	Value	Weight
1	3	3
2	6	4
3	18	5
4	22	6
5	28	7

Table 1: Knapsack table for Knapsack with capacity of W=11

The Knapsack Algorithm will generate the following table:

Item \Weight	0	1	2	3	4	5	6	7	8	9	10	11
{}	**0**	0	0	0	0	0	0	0	0	0	0	0
{1}	**0**	0	0	3	3	3	3	3	3	3	3	3
{1, 2}	**0**	0	0	3	6	6	6	9	9	9	9	9
{1, 2, 3}	0	0	0	3	6	**18**	18	18	21	24	24	24
{1, 2, 3, 4}	0	0	0	3	6	18	22	22	22	25	28	**40**
{1, 2, 3, 4, 5}	0	0	0	3	6	18	22	28	28	28	31	**40**

Table 2: Knapsack Tabular form $M[i, w]$

Answer Knapsack1

Here I have run the algorithm using the Knapsack1 algorithm with dominating set elements. The upper set which has a strike through is because of weight exceeding, so its not included in the bottom line. The strike through in the bottom set is because of dominating set pairs.

{1} v=3, w=3
A(1)={(0,0),(3,3)}

{1,2} v=6, w=4
A={(0,0),(3,3), (6,4),(9,7)}
A(2)={(0,0),(3,3),(6,4),(9,7)}

{1,2,3} v=18, w=5
A={(0,0),(3,3),(6,4),(9,7), (18,5),(21,8),(24,9),(27,12)}
A(3)={(0,0),(3,3),(6,4),(18,5),(21,8),(24,9)}

{1,2,3,4} v=22, w=6
A={(0,0),(3,3),(6,4),(18,5),(21,8),(24,9), (22,6),(25,9),(28,10),(46,11),~~(43,14),(46,15)~~}
A(4)={(0,0),(3,3),(6,4),(18,5),(21,8),(24,9),~~(22,6)~~,(25,9),(28,10),(46,11)} //dominating

{1,2,3,4,5} v=28, w=7
A={(0,0),(3,3),(6,4),(18,5),(21,8),(24,9),(40,11), (28,7),(31,10),(34,11),~~(46,12),(49,15)~~,
~~(52,16),(68,18)~~}
A(5)={(0,0),(3,3),(6,4),(18,5),(21,8),(24,9),(22,6),(25,9),(28,10),(46,11),~~(28,7)~~,~~(31,10)~~,(34,11)}

Answer c

For Knapsack fraction problem we can go for greedy approach and this way worst case scenario the time complexity will not go more than $O(nlogn)$ while for Knapsack1 for getting the optimal answer we have to use the dynamic programming approach and for this the space complexity will go $O(n)^2$.

Knapsack algorithm solves the problem in $O(nW)$ time and $O(nW)$ space for very large Ws we can even get $O(2^n)$. It is the same for the other Knapsack1 algorithm as well.

---

5. (Chapter 7 Problems 1 and 2 of the text book) 20 points

   (a) List all minimum $s-t$ cuts in the flow network pictured in Figure 1. The capacity of each edge appears as a label next to the edge.

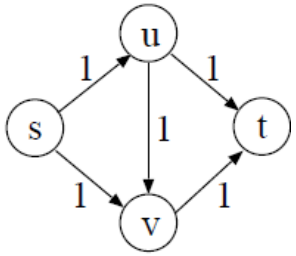


Figure 1: Figure for question (a)

Answer

There are a total of 4 possible cuts in this image. The cuts are as follows:

- {s}, {u, v, t}
- {s, u}, {v, t}
- {s, v}, {u, t}
- {s, u, v}, {t}

The capacities are determined by cutting the graph in two parts and looking at the outward arrows from the first to the second graph. So, given the four cuts above the capacities are 2, 3, 2, 2 (in order). So, the cuts with minimum capacities are three, which are as follows:

- {s}, {u, v, t}
- {s, v}, {u, t}
- {s, u, v}, {t}

(b) What is the minimum capacity of an $s - t$ cut in the flow network in Figure 2? Again, the capacity of each edge appears as a label next to the edge.

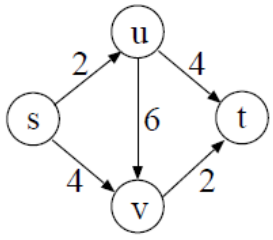


Figure 2: Figure for question (b)

Answer

There are 4 possible cuts for this question as well, which are:

- {s}, {u, v, t}
- {s, u}, {v, t}
- {s, v}, {u, t}
- {s, u, v}, {t}

The capacities are determined by cutting the graph in two parts and looking at the outward arrows from the first to the second graph. So, given the four cuts above the capacities are 6, 14, 4, 6 (in order). So, the cuts with minimum capacities is just one, which is: {s, v}, {u, t}

Figure 3 shows a flow network on which an $s-t$ flow has been computed. The capacity of each edge appears as a label next to the edge, and the numbers in boxes give the amount of flow sent on each edge.

(c) What is the value of this flow? Is this a maximum (s,t) flow in this graph?

(d) Find a minimum $s-t$ cut in the flow network pictured in Figure 3, and also say what its capacity is.

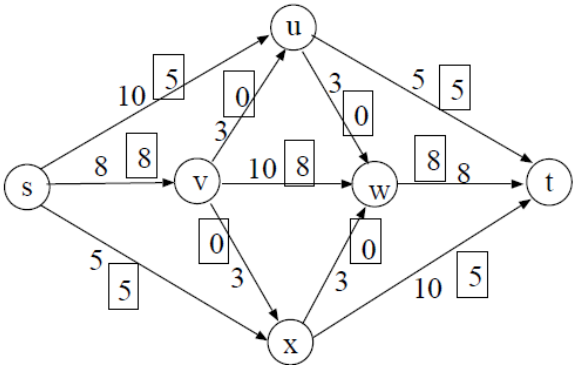


Figure 3: Figures for questions (c) and (d).

Answer (c)

The value of this flow will be 18. This is not a maximum flow.

Answer (d)

The minimum cut in this image is ({s, x}, {u, v, w, t}) and its capacity is 31.