Abueg, Jezic F.
2024-02705

EEE121 - THVW2
SP2

Problem 1: Activity Selection Problem
–act_selection.h
–act_selection.cpp

Approach used: Greedy Approach

Intuitively, we can do the most number of activities when we first do the ones that end first to have more time left for other activities. From this, we can see that a greedy approach can be used since we want to always pick the activity that ends first.

As such, activities' end time were sorted in increasing order. Conflicts in schedule were avoided by comparing the start time of the activity to be chosen and the end time of last activity chosen. If the start time is less than the end time, then it is a conflict.

Problem 2: 0/1 Knapsack Problem
–knapsack.h
–01_kp.cpp

Approach used: Dynamic Approach

Since we can only take either the whole item or none of the item, we need to check every possibility. For example, for a knapsack of capacity five (5), we first put item #1 and for the remaining capacity we put an item that can fit. We repeat this process until we get the highest value. However, if we visualize this, it is almost similar to the fibonacci tree wherein there are overlapping substructures (same capacity left in bag) and optimal substructures (we put the highest value that can be fit inside the bag).

As such, dynamic programming can be used. If we know what the maximum value for each capacity of the knapsack is, then we can use that to solve for the succeeding capacities. The program used a bottom-up approach where a table was made with rows as the items and the columns as the capacity. For each row, we check if the "new item" can fit inside the capacity indicated by the column. If it can be fit, we also add the best value for the remaining capacity (column - weight of "new item"). Then, we compare this value with the value in the row above. If it is better, then this is the best value for that capacity. If not, then the value in the row above is still the best value for that capacity.

Problem 3: Fractional Knapsack Problem
–knapsack.h
–fractional_kp.cpp

Approach used: Greedy Approach

Unlike the 0/1 Knapsack, we can take a fraction of an item and put it in the knapsack. One important thing to realize here is that we will always be able to fill up the knapsack since,

again, fractions of items can be put. Intuitively, if your knapsack contains the item that has the highest value per weight, then the knapsack contains more value.

As such, we get the value per unit weight (v/w) of the items and sort them in decreasing order. We fill up the knapsack with the whole weight of the highest v/w item as much as we can, while decreasing the capacity of the knapsack to keep track. Once we are in the item where its whole weight cannot fit, we take its fraction by multiplying its v/w with the remaining capacity. At this point we know the bag is full since we have just taken a fraction of the item with its weight being the capacity. This is what allows this to be greedy.

Another way to think of this is the dice game (from LE2). As much as we can, we take the "whole" of the dice (6/6) and once there are less than 6 steps, we take the "fraction" of the dice (1/6, 2/6, 3/6, 4/6, 5/6).

Problem 4: Kruskal Algorithm
–kruskal.h
–kruskal.cpp
—-utils/Graph.h
—-utils/Graph.cpp

Approach used: Greedy Approach

Kruskal's goal is to make a minimum spanning tree (MST). In the algorithm, we know it is spanning and connected through the use of the Union and Find, and the fact that the minimum number of edges is V-1. The greedy approach takes care of the minimum (sum of weights) part of the MST. Again, intuitively, we know that picking the lowest possible edge weight as much as possible would lead to an overall lower sum of weights. This is seen in 1 + 1 < 1 + 2.

As such, we sort the edges by their weight in increasing order. In the program, we keep track of the Vertex and its root node in a set somewhere, which is initialized as itself. Then, we check each edge by comparing the root nodes of the source and destination vertices (Find). If they are the same, we do not connect them. Otherwise, we connect them and make them have the same root node (Union). We do this until we have V-1 edges.

* In the program, a very simplified and rough Union-Find was implemented.

Problem 5: Rod Cutting Problem
–rod_cutting.h
–rod_cutting.cpp

Approach used: Dynamic Approach

Just like the 0/1 Knapsack, we need to check every possible cut (including no cut) on the rod. For example, for a rod of length 8, we cut it into 2 and 6. We check if we can cut 2 and 6 into smaller pieces (eg. 2-> 1 + 1, 6 = 4 + 2) and we check every single possibility. This looks

like a recursion problem with overlapping substructures (same cuts in different situations) and optimal substructures (we take the highest value cut).

As such, a dynamic bottom-up approach was used. If we know the best price for each length of the rod, then we can use that to solve for the succeeding lengths. Focusing on the first array “table” where 0 length is initialized as 0 value, we check each cut “j” on the length “i” by having its value equal to (price of “j” length + best value “i - j” length). If this is better than the one in the table, then this is the best for that length “i”. We also have the array “sol” to store the optimal cuts.

EXAMPLE USE OF PROGRAM

Problem Selection (and compilation):

```
PS C:\Users\hp\Desktop\EEE121\SP2> g++ main.cpp rod_cutting.cpp kruskal.cpp fractional_kp.cpp act_selection.cpp 01_kp.cpp utils/Graph.cpp -o
  main
PS C:\Users\hp\Desktop\EEE121\SP2> ./main
Choose which problem to do:
(0) Activity Selection
(1) 0/1 Knapsack
(2) Fractional Knapsack
(3) Kruskal Algorithm
(4) Rod Cutting Problem
Choice:
```

Activity Selection Problem

```
Choice: 0
Enter number of activities: 4
Enter start time of each activies (space-separated): 16 4 8 0
Enter end time of each activies (space-separated): 24 12 9 6
Activities:
0, 6
8, 9
16, 24
No. of activities: 3
--DONE--
```

Activity	EEE 121	EEE 123	EEE 128	EEE 199
Start	0	8	4	16
Finish	6	9	12	24

Time is set as the 0~24 clock (military time)

EEE 121 → EEE 123 → EEE 199

0/1 Knapsack Problem

```
Choice: 1
Enter capacity of knapsack: 10
Enter number of items: 5
Enter value of each item (space-separated): 35 13 14 8 20
Enter weight of each item (space-separated): 11 3 4 2 6
Maximum value: 35
Items included are:
Item #4
Item #3
Item #2
--DONE--
```

n	x = 0	x = 1	x = 2	x = 3	x = 4	x = 5	x = 6	x = 7	x = 8	x = 9	x = 10
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	20	20	20	20	20
2	0	0	8	**8**	8	8	20	20	28	28	28
3	0	0	8	8	14	14	22	**22**	28	28	34
4	0	0	8	13	14	21	22	27	28	35	**35**
5	0	0	8	13	14	21	22	27	28	35	35

Fractional Knapsack Problem

```
Choice: 2
Enter capacity of knapsack: 50
Enter number of items: 3
Enter value of each item (space-separated): 120 100 60
Enter weight of each item (space-separated): 30 20 10
Items included are:
10 kg of Item #3
20 kg of Item #2
20 kg of Item #1
Maximum value: 240
Weight of items: 50
--DONE--
```

Item	Dirt	Cobblestone	Iron Ore
Value	60	100	120
Weight	10	20	30

Limit: 50 kg

Knapsack

Resultant Profit

240

Resultant Solution

1,1,20/30

Kruskal Algorithm

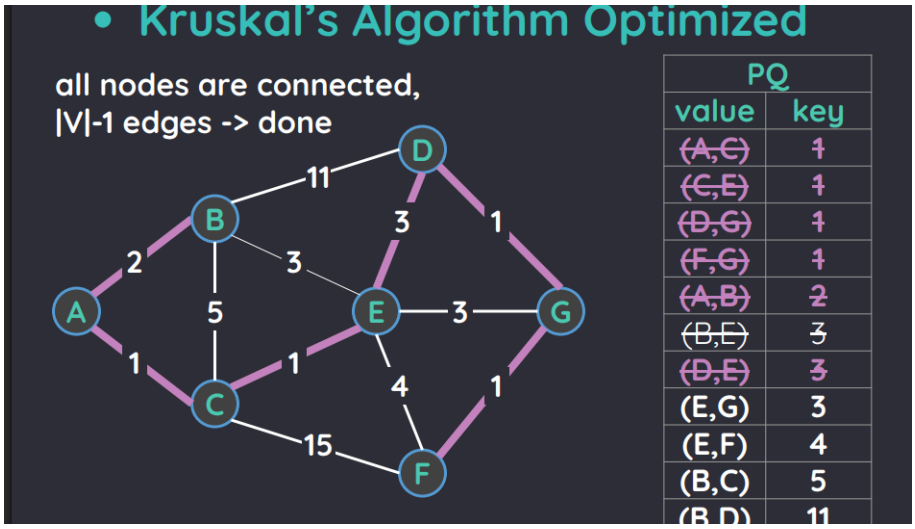
**treat each edge as a directed edge*

```
Choice: 3
Enter number of vertices: 7
!! Kindly use the form (neighbor, weight) [with parenthesis] !!
!! e.g. (2, 10)(3, 5) or (2,10)(3,5) or (2,10)   (3,   5)   [spaces do not matter] !!
!! Enter a newline if the vertex is not connected to other vertices !!
!! Moreover, make sure that any of the neighbors are within [1, n_vertices]. TY!

Enter neighbor and weight for node 1: (2,2)(3,1)
Enter neighbor and weight for node 2: (1,2)(3,5)(5,3)(4,11)
Enter neighbor and weight for node 3: (1,1)(2,5)(5,1)(6,15)
Enter neighbor and weight for node 4: (2,11)(5,3)(7,1)
Enter neighbor and weight for node 5: (2,3)(3,1)(4,3)(6,4)(7,3)
Enter neighbor and weight for node 6: (3,15)(5,4)(7,1)
Enter neighbor and weight for node 7: (4,1)(5,3)(6,1)
```

```
Printing Adjacency list
Source | Dest
1 -> {2, 2} {3, 1}
2 -> {1, 2} {3, 5} {5, 3} {4, 11}
3 -> {1, 1} {2, 5} {5, 1} {6, 15}
4 -> {2, 11} {5, 3} {7, 1}
5 -> {2, 3} {3, 1} {4, 3} {6, 4} {7, 3}
6 -> {3, 15} {5, 4} {7, 1}
7 -> {4, 1} {5, 3} {6, 1}

Edges included (weight, src, dest): (1, 4, 7) (1, 7, 6) (1, 5, 3) (1, 1, 3) (2, 2, 1) (3, 5, 4)
Total weight of MST: 9
```



(1,4,7) -> (1, D, G)
(1, 7,6)->(1, G, F)
(1,5,3)->(1, E, C)
(1,1,3)->(1, A, C)
(2,2,1)->(2, B, A)
(3, 5, 4)->(3, E, D)
vertices are flipped but its fine and more important is edges are sorted

1 + 1 + 1 + 1 + 2 + 3 = 9 !!

Rod Cutting Problem

```
Choice: 4
Enter length of rod: 8
Enter prices per (integer) length of rod: 1 5 8 9 10 17 17 20
Maximum value: 22
Rod should be cut into: 2 6
```

Input: *price[] = [1, 5, 8, 9, 10, 17, 17, 20]*
Output: *22*
Explanation: *The maximum obtainable value is 22 by cutting in two pieces of lengths 2 and 6, i.e., 5 + 17 = 22.*

from geeksforgeeks