Student name: Quan Hong Doan

Student ID: 986956

Assignment: Lab 15

Week: 15

Due: Jan 20, 2020

**Lab 15**

**Problem 1**:

Suppose Prob1, Prob2, and Prob3 are decision problems and Prob1 is polynomial reducible to Prob2, and Prob2 is polynomial reducible to Prob3. Explain why Prob1 must be polynomial reducible to Prob3.

**Answer**:

Assumption:

* Prob1 is polynomial reducible to Prob2 with function *f*: p1 in Prob1 ⬄ f(p1) in Prob2
* Prob2 is polynomial reducible to Prob3 with function *g*: p2 in Prob2 ⬄ g(p2) in Prob3

Now we consider a computation *g o f*: Prob1 is polynomial reducible to Prob3. Based on the above, we have:

p1 ∈ Prob1 ⬄ f(p1) ∈ Prob2 ⬄ g(f(p1)) ∈ Prob3

In other words,

p1 ∈ Prob1 ⬄ (g o f)(p1) ∈ Prob3

Because *f()* and *g()* run in polynomial time, so *(g o f)()* also runs in polynomial time.

Therefore Prob1 is polynomial reducible to Prob3, and this is a transitive property of reducibility.

**Problem 2**:

Show that TSP is NP-complete. (Hint: use the relationship between TSP and HamiltonianCycle. You may assume that the HamiltonianCycle problem is NP-complete.)

**Answer**:

Hamiltonian Cycle is a Hamiltonian path, starts and ends at the same point, and NP-complete.

For the TSP, we have to visit all the customers and wants to finish in the shortest time, with shortest distance and come back at the starting point. Meaning we need to find the shortest route that visits all clients and comes back at the beginning point. Therefore:

* TSP is a Hamiltonian Cycle problem.
* TSP is to find the path that contains permutation of every node in the graph.

Therefore TSP is NP-complete problem and the shortest path is known polynomial time.

**Problem 3**:

The SubsetSum problem is the following: Given a set S of n integers together with an integer k, is there a subset of S whose sum is exactly k? (This problem was stated in Lab 3.) Show that the SubsetSum is polynomial reducible Knapsack. Assuming that you know SubsetSum is NP-complete (this is indeed true), explain the steps of logic that verify that Knapsack must also be NP-complete.

**Answer**:

The Knapsack problem is to find the maximum value from a set of items with the maximum weight limitation. To solve this, we can go into 2 phases.

* We need to compute value of all subset of the items, and then compare the total weight of the subset with the limitation of allowed weight. If the total weight is less than the limitation, we will keep this value in a set of result.
* From the set of result, we will find the maximum value and that is the final optimal subset for Knapsack problem.

SubsetSum is exactly the first phase, when we compute the sum of value of all items in the subsets.

SubsetSum runs in polynomial time with a dynamic programming solution or approximate algorithm (Reference: <https://en.wikipedia.org/wiki/Subset_sum_problem>), and NP-complete.

Therefore, SubsetSum is polynomial reducible to Knapsack problem, and by transitivity property, Knapsack is also NP-complete.

**Problem 4**:

Show that the worst case for VertexCoverApprox can happen by giving an example of a graph G which has these properties:

1. G has a smallest vertex cover of size s
2. VertexCoverApprox outputs size 2\*s as its approximation to optimal size.

**Answer**:

Let assume:

* c is a sets output by VertexCoverApprox
* c\* is the set with optimal size.
* A is the set of edges during the iteration in the VertexCoverApprox when we select the edge.
* Size of smallest vertex cover = s.

We have added both vertices of the checking edge, then c = 2|A|, but in the c\* we would have added 1 of 2. Therefore, we need to prove: c ≤ 2 x c\*

Since, no two edge in A are covered by the same vertex from c\*, because we already used the edge in A, other edges incident to its vertices are removed, and the lower bound will be:

|c\*| ≥ A

On the size of an optimal vertex cover, we picked both vertices yielding an upper bound.

|c| ≤ 2|A|

Which means the exact case for optimal case will be |c|=2|A|

Or |c|/2 = |A|

So from the lower bound:

|c\*| ≥ A ⬄ |c\*| ≥ |c|/2 ⬄ |c\*|/|c| ≥ ½ ⬄ |c|/|c\*| ≤ 2 ⬄ |c| ≤ 2 x |c\*|

Therefore, in the worst case, c = 2 x c\*.

**Problem 5**:

*Extra Credit +5*. Read through the supplementary lecture on Dynamic Programming and the Knapsack Problem. (this is the file dyn-knapsack.pdf in Resources > Lecture Slides > Supplementary Lectures). Then use the Version 2 approach given in that document to solve the Knapsack optimization problem where S = {s1, s2, s3, s4, s5}, w[] = {1, 3, 2, 5, 4}, v[] = {4, 2, 3, 1, 1}, W = 7. You should show the full 2-D array that is used for memoization, as in the examples in the document. You will be carrying out the algorithm manually in this problem.

**Answer**:

The version 2 actually uses a 2D matrix for the calculation and memorization. The matrix has [r=w.length + 1] rows and [c=W] columns. The first row will have all 0 occurrences.

The algorithm indeed will have 2 nested iterations. It first will find the first occurrence the uppermost entry in column W with the value matrix[r][c]. If this occurs in the row j then the item j is the item selected for the optimal solution to the problem.

To find the next item, we move over w[j] columns and up one row and repeat the process.

To illustrate, we will see the phases below.

Phase 1 of iteration 1:

The value 9 is first occurs in row S3, item S3 is selected. The rest **blued** entries will be skipped.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| S1 | 4 | 4 | 4 | 4 | 4 | 4 | 4 |
| S2 | 4 | 4 | 4 | 6 | 6 | 6 | 6 |
| S3 | 4 | 4 | 7 | 7 | 7 | **9** | 9 |
| S4 | 4 | 4 | 7 | 7 | 7 | **9** | 9 |
| S5 | 4 | 4 | 7 | 7 | 7 | **9** | 9 |

Phase 2 of iteration 1:

The algorithm now reflects the weights contained in the knapsack by moving over the weight of item S3, w[2] = 2, columns and then go up 1 row, ending the first iteration with r=3 and c=2, which is in **green**.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| S1 | 4 | 4 | 4 | 4 | 4 | 4 | 4 |
| S2 | 4 | 4 | 4 | 6 | 6 | 6 | 6 |
| S3 | 4 | 4 | **7** | 7 | 7 | **9** | 9 |
| S4 | 4 | 4 | 7 | 7 | 7 | **9** | 9 |
| S5 | 4 | 4 | 7 | 7 | 7 | **9** | 9 |

Repeat the process, phase 1 of iteration 2:

The value 6 first occurs at row item S2, which will be selected in **red**. The rest will be skipped.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| S1 | 4 | 4 | 4 | 4 | 4 | 4 | 4 |
| S2 | 4 | 4 | 4 | **6** | 6 | 6 | 6 |
| S3 | 4 | 4 | **7** | 7 | 7 | **9** | 9 |
| S4 | 4 | 4 | 7 | 7 | 7 | **9** | 9 |
| S5 | 4 | 4 | 7 | 7 | 7 | **9** | 9 |

Phase 2 of iteration 2:

Move up and found the value at row item S1 with column c=1, which is in **red**.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| S1 | **4** | 4 | 4 | 4 | 4 | 4 | 4 |
| S2 | 4 | 4 | 4 | **6** | 6 | 6 | 6 |
| S3 | 4 | 4 | **7** | 7 | 7 | **9** | 9 |
| S4 | 4 | 4 | 7 | 7 | 7 | **9** | 9 |
| S5 | 4 | 4 | 7 | 7 | 7 | **9** | 9 |

Therefore the solution will be [1, 1, 1, 0, 0] will give us the maximum value = 9.