Homework 2 Solutions  
CSE 101 Summer 2017

1 Assignment Problem

A construction company has four large bulldozers located at four different garages. The bulldozers are to be moved to four different construction sites. The distances in miles between the bulldozers and the construction sites are given below.

<table>
<thead>
<tr>
<th>Bulldozers/Sites</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>110</td>
<td>35</td>
<td>90</td>
<td>45</td>
</tr>
<tr>
<td>2</td>
<td>105</td>
<td>55</td>
<td>85</td>
<td>60</td>
</tr>
<tr>
<td>3</td>
<td>115</td>
<td>75</td>
<td>100</td>
<td>50</td>
</tr>
<tr>
<td>4</td>
<td>125</td>
<td>65</td>
<td>95</td>
<td>70</td>
</tr>
</tbody>
</table>

Each site must receive exactly one bulldozer. How should the bulldozers be moved to the construction sites in order to minimize the total distance traveled?

1.1 Complete Enumeration

1. The following graph shows a portion of the tree constructed by Complete Enumeration. Complete the $(1 \rightarrow B)$ branch using Complete Enumeration.

```
root
   /|\   /|\   /|\   /|\
  1->A 1->B 1->C 1->D
```

2. Write pseudo code that solves this problem by Complete Enumeration.

INPUT:
Distance[1,...,N][1,...N]: the distances in the table

OUTPUT:
X[1,...,N]: X[i]=j means bulldozer i goes to site j.

AssignmentCE(Distance[1,...,N][1,...,N])

Sol: 1.
Using BFS

2.
You are NOT required to write pseudo code in either midterm or final exam. When you are asked to write your own algorithm to solve a problem in the final, you may just explain your idea in word. You may still use pseudo code to help, but it is not required.

You may ask for the real working code in C++ in Ping’s OH.

INPUT:
Distance[1,...,N][1,...N]: the distances in the table

OUTPUT:
X[1,...,N]: X[i]=j means bulldozer i goes to site j.

initialize:
int X[4] = { -1,-1,-1,-1 };  // initial permutation
int X_temp[4] = { 0,1,2,3 };  // temp permutation
int MIN = 100000;
int Min_temp = 0;

void findMinAssignment(int row, int left, int size)
{
    if (left == size)
    {
        // one permutation reached
        if (Min_temp < MIN)
        {
            MIN = Min_temp;
            // copy X_temp[] to X[]
            for (int i = 0; i < size; i++)
            {
                X[i] = X_temp[i];
            }
        }
        return;
    }
    for (int i = left; i < size; i++)
    {
        swap(left, i);
        Min_temp += Distance[row][X_temp[left]];
        findMinAssignment(row, i + 1, size);
    }
}
findMinAssignment(row + 1, left + 1, size);
Min_temp -= Distance[row][X_temp[left]];
swap(left, i);
}
}

1.2 Greedy Heuristic

Use Greedy Heuristic that was discussed in the class to find an upper bound.

Sol:
First, find the smallest number 35, cross out row 1 and column B. Then find the second number 50, and
cross out row 3 and column D. Then find the third number 85, cross out row 2 and column C, and this leaves
125.
So the upper bound is 35 + 50 + 85 + 125 = 295.

1.3 Branch-And-Bound

1. What is the Row-Min lower bound, that is, if each bulldozer can move to whatever site that is closest
to it?
2. What is the Column-Min lower bound, that is, if each site can receive whatever bulldozer that is
closest to it?
3. Complete the (1 → B) branch of the following tree which solves the problem by Branch-And-Bound.
Use Row-Min to determine the lower bound.

4. Complete all branches of the following tree which solves the problem by Branch-And-Bound. This
time, use Column-Min to determine the lower bound. Use Breadth-First-Search.

5. Write pseudo code that solves this problem by Branch-And-Bound using Column-Min.

INPUT:
UpperBound: an upper bound computed by Greedy Heuristic
Distance[1,...,N][1,...,N]: the distances in the table

OUTPUT:
X[1,...,N]: X[i]=j means bulldozer i goes to site j.
AssignmentBranchBound(Distance[1,...,N][1,...N], UpperBound)

Sol:
1. Row-Min: 35 + 55 + 50 + 65 = 205.
3.

4.

5. You are NOT required to write pseudo code in either midterm or final exam. When you are asked to write your own algorithm to solve a problem in the final, you may just explain your idea in word. You may still use pseudo code to help, but it is not required.

You may ask for the real working code in C++ in Ping’s OH.

INPUT:
UpperBound: an upper bound computed by Greedy Heuristic
Distance[1,...,N][1,...N]: the distances in the table

OUTPUT:
X[1,...,N]: X[i]=j means bulldozer i goes to site j.
initialize:
int X[4] = { -1,-1,-1,-1 };
int X_temp[4] = { 0,1,2,3 };
int MIN = 1000000;
int Min_temp = 0;
int UPPERBOUND = 295;

int colMin[4][4] =
{{105,35,85,45},
{105,55,85,50},
{115,65,95,50},
{125, 65,95,70}};

void findMinAssignment(int row, int left, int size)
{
if (left == size)
{
//one permutation reached
if (Min_temp < MIN)
{
MIN = Min_temp;
//copy X_temp[] to X[]
for (int i = 0; i < size; i++)
{
X[i] = X_temp[i];
}
return;
}
}
for (int i = left; i < size; i++)
{
swap(left, i);
Min_temp += Distance[row][X_temp[left]];

int lowerbound = Min_temp;
if (lowerbound < UPPERBOUND) {
//find the lower bound by colMin
int j = left + 1;
while (j < size) {
lowerbound += colMin[row + 1][X_temp[j]];
j++;
}
}
if (lowerbound >= UPPERBOUND) {//cross out
Min_temp -= Distance[row][X_temp[left]];
swap(left, i);
continue;
}
findMinAssignment(row + 1, left + 1, size);
Min_temp -= Distance[row][X_temp[left]];
swap(left, i);
}
2 Knapsack Problem

During a robbery, a burglar finds much more loot than he had expected and has to decide what to take. His bag (or “knapsack”) will hold a total weight of at most 11 pounds. There are 4 items to pick from, whose weights and values are shown below.

<table>
<thead>
<tr>
<th>Item</th>
<th>Weight</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>6</td>
<td>36</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>15</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>16</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>6</td>
</tr>
</tbody>
</table>

What’s the most valuable combination of items he can fit into his bag?

2.1 Complete Enumeration

1. Complete the tree below to show the Complete Enumeration.

2. Write pseudo code that solves this problem by Complete Enumeration.

INPUT:
CAPACITY: the capacity of the knapsack
N: there are N items
W[1,...,N]: the weights of N items
V[1,...,N]: the values of N items

OUTPUT:
X[1,...,N]: X[i]=1 means item i is in the knapsack, X[i]=0 means it is not in the knapsack.

KnapsackCE(CAPACITY, N, W[1,...,N], V[1,...,N])

Sol: 1.
You are NOT required to write pseudo code in either midterm or final exam. When you are asked to write your own algorithm to solve a problem in the final, you may just explain your idea in word. You may still use pseudo code to help, but it is not required.

You may ask for the real working code in C++ in Ping’s OH.

**INPUT:**
CAPACITY: the capacity of the knapsack
N: there are N items
W[1,...,N]: the weights of N items
V[1,...,N]: the values of N items

**OUTPUT:**
X[1,...,N]: X[i]=1 means item i is in the knapsack, X[i]=0 means it is not in the knapsack.

initialize:
int Weights[4] = { 6,3,4,2 };
int Values[4] = { 36,15,16,6 };
int X[4] = { -1,-1,-1,-1 };
int X_temp[4] = { -1,-1,-1,-1 };

int CAPACITY = 11;
int MAX_Value = 0;

void findMaxValue(int itemID, int totalItemCount)
{
    if (itemID == totalItemCount)//no items
    {
    }
    else if (total_weight_in_knapsack <= CAPACITY)
    {
        if (current_value_in_knapsack > MAX_Value)
        {
            MAX_Value = current_value_in_knapsack;
        //pocy X_temp to X
    for (int i = 0; i < totalItemCount; i++)
    

    int current_value_in_knapsack = 0;
    int total_weight_in_knapsack = 0;
2.2 Greedy Heuristic

Use Greedy Heuristic that was discussed in the class to find a lower bound. The greedy algorithm always picks the item with the highest value in the remaining items.

**Sol:** Pick item 1 and 3, and the lower bound is 36 + 16 = 52.

2.3 Branch-And-Bound

1. What are the relative value of these four items? Please complete the following table. Hint: relative value is value/weight.

<table>
<thead>
<tr>
<th>Item</th>
<th>Weight</th>
<th>Value</th>
<th>Unit Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>6</td>
<td>36</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>6</td>
<td></td>
</tr>
</tbody>
</table>

2. What is the upper bound of the total value that may be put in this bag? Use the technique learned in class to compute this value.

3. Complete the tree below which solves the problem by Branch-And-Bound. Selection rule: take the item with largest weight first. Use BFS (Breadth-First-Search). You must add the reason when crossing out a vertex.

4. Write pseudo code that solves this problem by Branch-And-Bound. Selection rule: take the item with largest weight first. Use BFS (Breadth-First-Search).
INPUT:
CAPACITY: the capacity of the knapsack
N: there are N items
W[1,...,N]: the weights of N items
V[1,...,N]: the values of N items
LowerBound: the lower bound computed by Greedy Heuristic.

OUTPUT:
X[1,...,N]: X[i]=1 means item i is in the knapsack, X[i]=0 means it is not in the knapsack.

KnapsackBranchBound(CAPACITY, N, W[1,...,N], V[1,...,N], LowerBound)

Sol:
1.

<table>
<thead>
<tr>
<th>Item</th>
<th>Weight</th>
<th>Value</th>
<th>Relative Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>6</td>
<td>36</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>15</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>16</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>6</td>
<td>3</td>
</tr>
</tbody>
</table>

2. $6 \times 6 + 3 \times 5 + 2 \times 4 = 59$.
3. 

4. You are NOT required to write pseudo code in either midterm or final exam. When you are asked to write your own algorithm to solve a problem in the final, you may just explain your idea in word. You may still use pseudo code to help, but it is not required.

INPUT:
CAPACITY: the capacity of the knapsack
N: there are N items
W[1,...,N]: the weights of N items
V[1,...,N]: the values of N items
LowerBound: the lower bound computed by Greedy Heuristic.

OUTPUT:
X[1,...,N]: X[i]=1 means item i is in the knapsack, X[i]=0 means it is not in the knapsack.
int Weights[4] = { 6,3,4,2 };  
int Values[4] = { 36,15,16,6 };  
int X[4] = { -1,-1,-1,-1 };  
int X_temp[4] = { -1,-1,-1,-1 };  
int relativeValue[4] = { 6,5,4,3 };  

int LOWERBOUND = 52;  
int CAPACITY = 11;  
int MAX_Value = 0;  

int current_value_in_knapsack = 0;  
int total_weight_in_knapsack = 0;  

void findMaxValue(int itemID, int totalItemCount)  
{
    if (itemID == totalItemCount)//no items
    {
        if (total_weight_in_knapsack <= CAPACITY)
        {
            if (current_value_in_knapsack > MAX_Value)
            {
                MAX_Value = current_value_in_knapsack;
                LOWERBOUND = MAX_Value;  
            }  
        }  
        //pocy X_temp to X
        for (int i = 0; i < totalItemCount; i++)
        {
            X[i] = X_temp[i];  
        }  
        return;  
    }
    //do not put itemID in the knapsack
    X_temp[itemID] = 0;  
    findMaxValue(itemID + 1, totalItemCount);  
    //put item ID in the knapsack
    X_temp[itemID] = 1;  
    current_value_in_knapsack += Values[itemID];  
    total_weight_in_knapsack += Weights[itemID];  
    if (total_weight_in_knapsack > CAPACITY)
    {
        //cross out
        return;  
    }
    //compute remaining_upper_bound
    
    if (remaining_upper_bound <= LOWERBOUND)
    {
        return;  
    }
findMaxValue(itemID + 1, totalItemCount);
current_value_in_knapsack = Values[itemID];
total_weight_in_knapsack = Weights[itemID];}

3 Proving NP-completeness

3.1 Bin Packing Problem

Problem: In the decision version of the Subset Sum problem we are given a sequence of non-negative integers \( a_1, \ldots, a_n \) and a target value \( t \). The question is whether there is a subset \( S \subseteq \{1, \ldots, n\} \) such that \( \sum_{i \in S} a_i = t \).

In the decision version of the Bin Packing problem we are given volumes \( v_1, \ldots, v_n \), a volume bound \( B \), and a target \( k \). The question is whether we can partition the integers \( v_1, \ldots, v_n \) into \( k \) subsets such that the integers in each subset sum to at most \( B \).

Given that the Subset Sum problem is NP-complete, prove that the Bin Packing problem is also NP-complete.

Solution:

Step 1: The Bin Packing problem is in NP

If a subset of volumes is given as the solution to the Bin Packing problem, it can be verified that the number of subsets is \( k \) and that the sum of all volumes in each subset is at most \( B \), in polynomial time. Hence, the Bin Packing problem is in NP.

Step 2: The NP-complete Subset Sum problem can be reduced to the Bin Packing problem i.e., Subset Sum problem \( \propto \) Bin Packing problem

Let \((a_1, \ldots, a_n, t)\) be an instance of the Subset Sum problem, and \( A = \sum_{i=1}^{n} a_i \) be the total sum of the integers. Let \((v_1, \ldots, v_n, v_{n+1}, v_{n+2})\) be an instance of the Bin Packing problem where \( v_i = a_i \), for all \( i \leq n \) and \( v_{n+1} = 2A - t \), \( v_{n+2} = A + t \). Now, \( \sum_{i=1}^{n+2} v_i = A + 2A - t + A + t = 4A \). Let us set the number of bins \( k = 2 \), which leads to bin capacity being at least \( B = 2A \). The Bin Packing problem has a solution for the given instance if and only if there is a subset of integers that sums to \( 2A \).

(i) If \( S \subseteq \{1, \ldots, n\} \) is the solution of the Subset Sum problem such that \( \sum_{i \in S} a_i = t \). Then \( S \cup \{a_{n+1}\} \) and \( S \cup \{a_{n+2}\} \) are the two subsets such that the integers in each subset sums to at most \( 2A \), which is the solution to the Bin Packing problem.

(ii) If \( (S \subseteq \{1, \ldots, n+2\}, S) \) is the solution to the Bin Packing problem such that \( \sum_{i \in S} a_i = \sum_{i \in S} a_i = 2A \). Then, \( n+1 \) and \( n+2 \) must be present in separate bins. If \( (n+1) \in S \), then \( S - \{n+1\} \) is a solution to the Subset Sum problem. And, if \( (n+2) \in S \), then \( S - \{n+2\} \) is a solution to the Subset Sum problem.

Thus, we can change any instance of the Subset Sum problem into an instance of the Bin Packing problem. Given an instance for the Subset Sum problem, compute the sum ‘\( A \)’ of all the members of the instance and add two more members ‘\( 2A - t \)’ and ‘\( A + t \)’ to the instance. We need to determine whether the members of this new set can be divided into \( k = 2 \) subsets such that the integers in each subset sum to at most \( B = 2A \). If yes, then there is a solution to the Subset Sum problem for the original instance, else there is no solution to it.

Step 3: Proof that the reduction can be done in polynomial time

We need to sum all the members in the given instance of the Subset sum problem, and two more members to the instance using this sum. Of course this can be done in polynomial time.

Since the Bin Packing problem is in NP, and we can reduce an NP-complete problem to it in polynomial time, the Bin Packing problem must be NP-complete.
3.2 Clique Problem

**Problem:** Vertex Set of an undirected graph $G = (V, E)$ covers all edges of the given graph $G$. It is a subset $V'$ of its vertices $V$ such that for every edge $(u, v)$ in $E$, either '$u'$ or '$v'$ is in the vertex cover. Given an undirected graph $G$, the decision version of the Vertex Cover problem is to determine whether $G$ has a vertex cover of a given size $k$.

A Clique, in an undirected graph $G = (V, E)$ is a subset $C$ of its vertices $V$ such that every two distinct vertices in $C$ are connected by some edge in $E$. Given an undirected graph $G$, the decision version of the Clique problem is to determine whether a clique of size $k$ exists in $G$.

Given that the Vertex Cover problem is NP-complete, prove that the Clique problem is also NP-complete.

**Solution:**

**Step 1:** The clique problem is in NP

If a subset of vertices is given as the solution to the clique problem, it can be verified that the number of vertices is $k$ and that they are all pairwise connected, in polynomial time. Hence, the clique problem is in NP.

**Step 2:** The NP-complete vertex cover problem can be reduced to the clique problem i.e., vertex cover problem $\propto$ clique problem

Let $\overline{G}$ consist of all possible edges between vertices in $G$ that are not present in $G$ i.e., $\overline{G} = (V, \overline{E})$, where $\overline{E} = (u,v) : u, v \in V, u \neq v, (u,v) \notin E$.

If there exists a clique $C$ in $G$, then $V - C$ is a vertex cover in $G$. To prove this statement:

(i) Assume that there is a clique $C$ of size $k$ in $G$. Then if $\overline{C} = V - C$, then $\overline{C}$ has size $|V| - k$. However, we know that $C$ forms a clique in $\overline{G}$, so there are no edges between vertices of $C$ in $\overline{G}$. Thus, every edge of $\overline{G}$ is adjacent to some vertex of $\overline{C}$, which means that it is a feasible vertex cover. So we have a vertex cover of size $|V| - k$.

(ii) Assume that we have a vertex cover $V'$ of size $k$ in $G$. Let $\overline{V'} = V - V'$, which has size $|V'| - k$. Every edge in $G$ is connected to some vertex of $V'$ since it is a vertex cover. This means if $u, v \in V'$, there can be no edge $(u,v) \in E$. But if there are no such edges in $E$, then $(u,v) \in \overline{E}$ for any such $u, v \in V'$. Thus, $\overline{V'}$ forms a clique in $\overline{G}$, and we have a clique of size $|V'| - k$.

Thus, we can change any instance of vertex cover problem into an instance of the clique problem. Given a graph $G = (V, E)$ and integer $k$, we just need to determine whether a clique of size $|V| - k$ exists in $\overline{G}$. If it exists then $G$ has a vertex cover of size $k$, else it does not.

**Step 3:** Proof that the reduction can be done in polynomial time

We need to scan over all pairs of vertices in the graph $G$, and generate an edge if there isn’t an edge between the pair, to generate the complement graph $\overline{G}$. This operation can be done in polynomial time.

Since the clique problem is in NP, and we can reduce an NP-complete problem to it in polynomial time, the clique problem must be NP-complete.
Approximation Algorithm for Bin Packing

Problem: Consider the following problem. Given $n$ items of weights $w_1, w_2, ..., w_n \leq 1$. You are tasked with the problem of putting these items into bins so that the total weight of all items in any one bin is at most 1. So for example, if you had items of weight 0.6, 0.5 and 0.3, you could put them all in different bins, or put the first and third in a bin together and the second in a different bin, but you could not put the first and second in the same bin because $0.6 + 0.5 > 1$. Your objective is to do this while minimizing the total number of bins used. It turns out that this problem is NP-Hard (See Q3.1). Give a polynomial time algorithm that provides a 2-approximation for this problem, and prove correctness.

Hint: As long as your bins are full enough that you cannot combine any pair of them, the average weight in each bin is at least $1/2$.

Solution:
We use the following greedy algorithm:

1. Put each item in a separate bin
2. While there exist two bins whose total weight is less than 1
3. Combine the items in those bins into a single bin

This algorithm clearly takes $O(n^3)$ time (in fact clever implementations can be made to run in $O(n \log(n))$), since we can combine bins at most $O(n)$ times, and we can check whether or not there are two combinable bins in $O(n^2)$ time. The greater difficulty is in showing that this algorithm gives a 2-approximation.

Suppose that this algorithm provides a solution with $k$ final bins. It must be the case that for any two bins the total weight of all the items in either bin is more than 1, since otherwise we would have combined those bins. Let $x_1, x_2, ..., x_k$ be the total weights of each bin. If $k = 1$, then we use only one bin, which clearly cannot be improved upon. Otherwise, $x_i + x_j > 1$ for all $i \neq j$. Therefore, we have that

$$2(k - 1) \sum_i x_i = \sum_{i \neq j} x_i + x_j > \sum_{i \neq j} 1 = k(k - 1).$$

Therefore, we have that the total weight of all items is $\sum_i x_i$, which is at least $k/2$. Since no packing is allowed to put more than one unit of weight in any bin, any packing must use at least $k/2$ bins. Therefore, the ratio between what our algorithm achieves and the optimal is at most $k/(k/2) = 2$. Therefore, we have a 2-approximation.

Alternate Solution: For each item of weight $w_i \leq 0.5$ put it in its own bucket. If there are items remaining, take a new bucket and put items in it one at a time until it is filled at to at least half capacity. You will always be able to add another item because all remaining items have weight less than 1/2. Repeat this until there are no items remaining.

This algorithm is clearly linear time, the difficulty comes in proving that it is a 2-approximation. However, it is clear that all buckets that this algorithm produces other than the last one are at least half full. Therefore, if this algorithm uses $k + 1$ buckets, the total weights of the items in the first $k$ buckets is at least $k/2$, and therefore, the total weight of all items is strictly more than $k/2$. Therefore, the optimal solution must use more than $k/2$ buckets, and hence must use at least $(k + 1)/2$. Therefore, our algorithm uses at most twice as many buckets as the optimal solution, and therefore is a 2-approximation.