Modified Knapsack Problem

Akshit Kumar (EE14B127)

27th November 2016

Abstract

This report contains the code and the algorithm used for the first question of the take home end-semester examination. The objective of the problem is to find a subset of N objects of positive weights $\{w_i\}$that maximizes their sum subject to a different constraint than the original knapsack constraint. Both a dynamic programming and greedy solution is given to the problem in the question

1 Introduction

The traditional 0-1 Knapsack problem has been modified to incorporate a penalty term which penalises the use of larger numbers of items. The penalty term in this case is $-lnm$.

2 Dynamic Programming Solution

The Dynamic Programming Solution to the problem is very similar to the algorithm used for 0-1 Knapsack DP problem. Due to the extra penalty term the algorithm, is to be tweaked a bit.

2.1 Algorithm for the DP Solution

Assume $w_1, w_2, ...w_n, W$ are strictly positive integers. We can define $m[i, w]$ to be the maximum value that can be obtained with weight less than or equal to $w - lnk$ where k is the number of items used so far. We can define $m[i, w]$ recursively as follows:

$m[0, w] = 0$

$m[i, w] = m[i-1, w]$ if $w_i > w$ or $m[i-1, w-w_i] + w_i < m[i-1, w]$

$m[i, w] = m[i-1, w-w_i] + w_i$ if $m[i-1, w-w_i] + w_i < W - lnk$

2.2 Comparison with Recursion

Time Complexity of the Dynamic Programming Solution is $O(nW)$ where n is the number of weights/items and W is the given maximum constraint. It has a space complexity of $O(nW)$ as the Dynamic Programming solution requires that time complexity for the purposes of memoizing the solution as we build the table. The recursive solution has a time complexity of $2^n$where n is the number of weights/items because each weight can either be included or left out so an exhaustive recursive solution will take exponential time to come up with the solution. Even though recursive solution is space optimal, the recursive solution may cause several function calls leading to stack overflow. In conclusion, DP solution offers a better time bound than recursive solution.

3 Greedy Solution

For the greedy solution we try to maximise the sum of weights by trying to take the maximum possible element satisfying the constraint in each iteration. This requires the elements to be sorted before hand. The Greedy approach fails to give an optimal solution. The algorithm for the same can be found below:

3.1 Algorithm for the Greedy Solution

Assume $w_1, w_2, ...w_n, W$ are strictly positive integers.

$sort(w_1, w_2, ...w_n)$ in the descending order of the weights
Add w_i to the Knapsack
Check for the constraint $\sum w_i \leq W_0 - lnm$
If the constraint is not satisfied, remove w_i from the knapsack.

3.2 Comparison with Recursion

Time Complexity of the Greedy Solution is $O(nlogn)$ which is majorly due to the time complexity of using a sorting algorithm. Once sorted, finding the greedy solution has a time complexity of $O(n)$. Again time complexity for the recursive solution is same as before. Since the greedy approach tries to find optimal solutions locally, it misses out on subtle solutions and hence fails to find the global optimal solution to the problem. But it approximates the solution well and is faster in comparison to DP and recursive solutions without an additional overhead of space.

4 Output for DP and Greedy Solutions

```
DP Solution : 9
Elements in the Knapsack due to DP Approach : 4 5
Greedy Solution : 8
Elements in the Knapsack due to Greedy Approach : 7 1
--------------------------------------------------------------------------
DP Solution : 1598
Elements in the Knapsack due to DP Approach : 106 287 380 393 111 175 146
Greedy Solution : 1598
Elements in the Knapsack due to Greedy Approach : 399 399 399 399 2
--------------------------------------------------------------------------
```

5 Source Code for the Problem

```c
/*
 * Name : Akshit Kumar
 * Roll Number : EE14B127
 * Solution to Question 1 of the take home endsem examination.
 */

// Including the necessary libraries
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <stdbool.h>
#include <string.h>
#include <limits.h>

#define max(a,b) (a > b ? a : b)

```

```c
// Defining a structure Cost to hold the weight and count of the of which cell in the DP table
typedef struct Cost{
  int weight;
  int count;
}Cost;


Cost dp[2000][2000];

int W; // Hold the maximum value
int weights[2000]; // Holds the weights
int knapsack_elements_dp[2000]; // holds the weights for the dp solution
int knapsack_elements_greedy[2000]; // holds the weights for the greedy solution
int num_elements = 0;

// Comparison function for the quick sort function
int compare_function(const void *a, const void *b){
  return *(int*)b - *(int*)a;
}

// Function to get the maximum value
int get_limit(char line[]){
  int W;
  char *pos;
  if ((pos=strchr(line, '\n')) != NULL)
    *pos = '\0';
  char *token;
  token = strtok(line," ");
  while(token != NULL){
    sscanf(token,"%d",&W);
    token = strtok(NULL," ");
  }
  return W;
}

// Function to get the weights
int get_weights(char line[]){
  char *pos;
  if((pos = strchr(line,'\n')) != NULL){
    *pos = '\0';
  }
  char *token;
  token = strtok(line," ");
  int num;
  int i = 1;
  weights[i++] = INT_MAX;
  while(token != NULL){
    sscanf(token,"%d",&num);
    weights[i++] = num;
    token = strtok(NULL," ");
  }
  return i-1;
}
```

```
// Function to get the DP solution
int dp_knapsack_solution(int W, int n){
  // Setting the first row of the dp table to 0 for both the weights and count
  for(int j = 0; j <= W; j++){
    dp[0][j].weight = 0;
    dp[0][j].count = 0;
  }
  // Applying the knapsack algorithm to this modified question
  for(int i = 1; i <= n ;i++){
    for(int j = 0; j <= W; j++){
      // if the weight is more than maximum then the dp gets the previous value
      if(weights[i] > j){
        dp[i][j] = dp[i-1][j];
      }
      // else if the modified constrainst is satified that is maximum sum of weights should be less than the maximum weight and a penalty term, in this case ln(m)
      else{
        if(dp[i-1][j-weights[i]].weight + weights[i] > dp[i-1][j].weight && dp[i-1][j-weights[i]].weight + weights[i] <= (float)j - log(dp[i-1][j-weights[i]].count + 1)){
          dp[i][j].weight = dp[i-1][j-weights[i]].weight + weights[i]; // increase the maximum sum of weights by ith weight
          dp[i][j].count = dp[i-1][j-weights[i]].count + 1; // increase the count by one
        }
        // if the modified constraint is not solved then it gets the previous value
        else{
          dp[i][j] = dp[i-1][j];
        }
      }
    }
  }
  return dp[n][W].weight;
}

// Backtracking function to get the elements in the knapsack which give the maximum sum for the constraint
int get_knapsack_elements(int W,int n){
  int line = W;
  int i = n;
  int num_elements = 0;
  while(i > 0){
    // if the sum of weights for ith is more than i-1th, then we backtrack
    if(dp[i][line].weight > dp[i-1][line].weight){
      // add that weight to the list
      knapsack_elements_dp[num_elements++] = weights[i];
      line = line - weights[i]; // backtracking code
      i--; // backtracking code
    }
    else{
      i--;
```

```c
    }
  }
  return num_elements;
}

// Function to implement a greedy solution for the modified knapsack problem
int greedy_knapsack_solution(int W,int n){
  // Sort the array of weights using the inbuilt qsort function
  qsort(weights,n,sizeof(int),compare_function);
  num_elements = 0;
  int i = 1;
  int sum = 0;
  // iterate over all the elements
  while(i <= n){
    // add that element to the knapsack
    sum += weights[i];
    num_elements++; // increase the number of elements
    // if that weight satisfies the constraint, add in the knapsack
    if((sum <= (float)W - log(num_elements))){
      knapsack_elements_greedy[num_elements] = weights[i];
    }
    // else, remove the element from knapsack and decrease the elements
    else{
      sum -= weights[i];
      num_elements--;
    }
    i++;
  }
  // return the sum
  return sum;
}

// Function to print the DP matrix on the terminal for debugging
void print_dp_matrix(int W, int n){
  for(int i = 0; i <= n; i++){
    for(int j = 0; j <= W; j++){
      printf("%d ", dp[i][j].weight);
    }
    printf("\n");
  }
}

// Function to write the DP matrix to the output file
void write_dp_solution_to_file(int W,int n){
  FILE *file = fopen("output1.dat","a");
  fprintf(file,"Dynamic Programming Table : \n");
  for(int i = 1; i <= n; i++){
    for(int j = 0; j <= W; j++){
      fprintf(file,"%d ", dp[i][j].weight);
    }
    fprintf(file,"\n");
  }
  fprintf(file,"\n");
  fclose(file);
}

```

```
// Function to print the elements found using the dp approach
void print_knapsack_elements_dp(int N){
  for(int i = 0; i < N ; i++){
    printf("%d ",knapsack_elements_dp[i]);
  }
  printf("\n");
}

// Function to print the elements found using the greedy approach
void print_knapsack_elements_greedy(int N){
  for(int i = 1; i <= N; i++){
    if(knapsack_elements_greedy[i] != 0){
      printf("%d ",knapsack_elements_greedy[i]);
    }
  }
  printf("\n");
}

int main(int argc,char **argv){
  if(argc != 2){
    printf("Usage ./a.out <filename>\n");
    exit(1);
  }
  FILE *file = fopen(argv[1],"r");
  if(file == NULL){
    printf("Unable to open file\n");
    exit(1);
  }
  char line[2000];
  char line_number = 0;
  int W;
  int n;
  // Reading in the values from the file
  while(fgets(line,sizeof line,file) != NULL){
    if(line[0] == '#' || line[0] == '\n'){
      continue;
    }
    else{
      line_number++;
      if(line_number % 2 != 0){
        W = get_limit(line);
      }
      else if(line_number % 2 == 0){
        n = get_weights(line);
        line_number -= 2;
      }
      // Printing out the solutions
      if(line_number == 0){
        printf("DP Solution : %d\n", dp_knapsack_solution(W,n));
        printf("Elements in the Knapsack due to DP Approach : ");
        print_knapsack_elements_dp(get_knapsack_elements(W,n));
        write_dp_solution_to_file(W,n);
        printf("Greedy Solution : %d\n",greedy_knapsack_solution(W,n));
        printf("Elements in the Knapsack due to Greedy Approach : ");
        print_knapsack_elements_greedy(num_elements);
        printf("--------------------------------------------------------------------------------\n
      }
```

```
      }
    }
  return 0;
}
```

knapsack.c