Project 3

Lela Bones

May 7, 2018

Contents

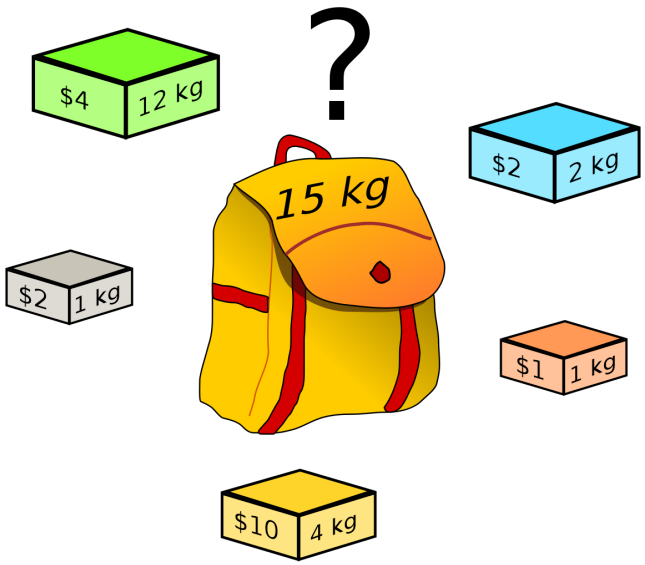
Abstract

My research project includes the study of time complexity of the dynamically programmed solution to the knapsack problem. The knapsack problem is a combinatorial optimization problem that, given a set of items that have a weight and value, it is possible to determine the most valuable combination of the items given a limit (max weight the knapsack can hold). Some every day applications of knapsacks includes loading cargo ships, packing for trips, choosing fantasy sport teams, encrypting messages, and so on.

1 Introduction

The earliest documented study of the knapsack problem was in 1897 by Tobias Dantzig. The basic idea of the problem was that there exists a solution that allows for one to fix numerous decision variables for a given instance of the problem[3]. There exists many different forms of the knapsack problem and many different reduction algorithm solutions to each problem.

The most fundamental knapsack problem is the Knapsack 0-1 problem. This version contains a solution where only whole items can be placed in the knapsack and no repetition is allowed[2]. A similar but slightly different problem is the Bounded Knapsack Problem (BKP). This problem also only uses whole items, but it allows for repetition. This is the problem that I am studying the solution of.



The knapsack problem can be solved in many different ways. Some of these solutions involve brute force, dynamic memoization, greedy algorithms, genetic algorithms, parallel programming, and dominance relations. The brute force solution involves testing every possible combination and finding the max value. This solution is very expensive and runs at NP-hard. Meaning that it takes greater than polynomial time, in this case exponential, to find the solution, but only polynomial time to check if a solution is correct. Other solutions run at pseudo-polynomial time, full polynomial time, and non-polynomial time. The goal of my research is to study the BKP (with repetition) and the dynamic solution for it.

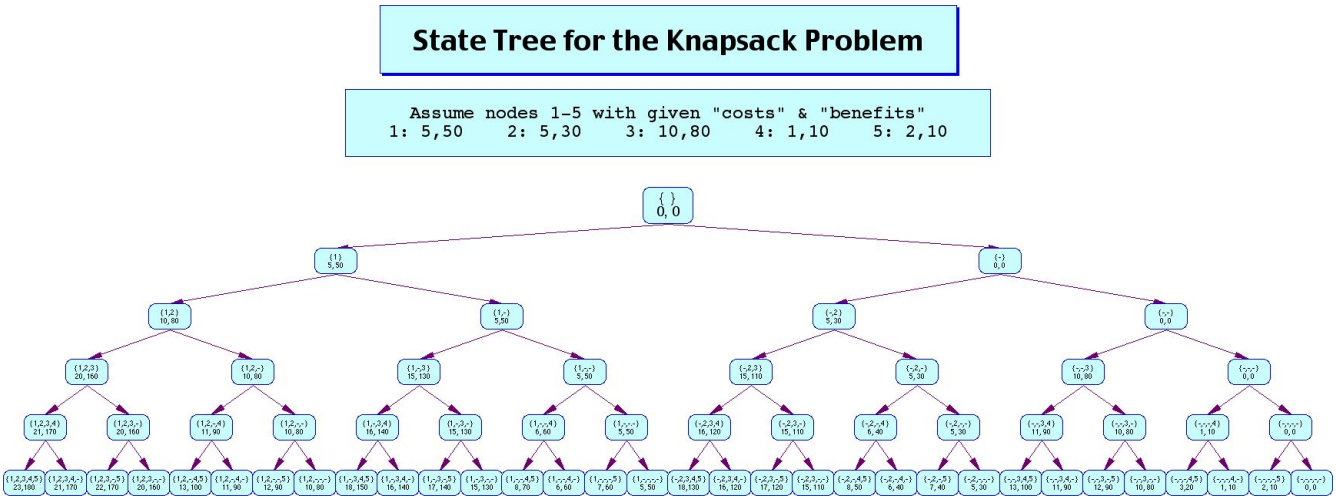
2 Topic Details

There is a thief that is robbing a mansion. He wasn't expecting there to be so many loot-able items in the house, and only brought a bag (knapsack) K that can hold a set weight W in pounds. The thief must choose amongst n items, which ones to keep. Each item has a weight $w_1, w_2, ..., w_n$ and a monetary value $v_1, v_2, ..., v_n$, and we need a solution to fit the bag with the most optimal value given $K(W)$. Each item x_1 is allowed to be repeated a non-negative integer value of c. Another way to represent this problem is by the following [3]:

$$maximize \sum_{i=1}^{n} v_i x_i$$

$$subject\ to \sum_{i=1}^{n} w_i x_i \leq W and\ x_i \in 0 \leq W x_1 \leq c$$

We can solve this solution using dynamic programming and more specifically, memoization. What we do is we use a recursive formula that breaks down the knapsack into smaller knapsacks and stores the solutions to the smaller knapsacks in a table. This is solved in a bottom-up manner where we start at the smallest knapsack and use those solutions to work our way up to larger ones and eventually the original knapsack [1]. The following is an example tree of a small knapsack problem:



For the BKP we use hash tables to store the values of $K(W)$ that has already been computed. Each time the algorithm is called it checks to see if $K(W)$ has already been computed for and placed in the table for that specific W. If yes then it continues to call the recursion on the next largest W. If no then it proceeds to calculate the $K(W)$ for that call. The algorithm continues to repeat until we reach the limit of the Knapsack. The following is the pseudo-code for the memoization algorithm:

Algorithm 1 Memoization

```
1: procedure KNAPSACK(W)
2:     if w is in hash table then
3:         return K(w)
4:     K(w) = max[knapsack(w − w_i) + v_i : w_i ≤ w]
5:     insertK(w)into hash table, with key w
6:     return K(w)
```

The resulting table from this algorithm is a one-dimensional table of length $W + 1$, in order of left to right [1]. Each entry in the table takes up to $O(n)$ time to compute, because it doesn't repeat the solutions of already found sizes like a naive recursion would. Therefore, the solution is solvable in pseudo-polynomial time or $O(nW)$. Pseudo-polynomial time means the runtime is bound not only on the size of the input (e.g. n items), but also on the magnitude of the inputs of the problem [3].

3 Theory

Theorem 1 *Knapsack is **NP**-complete.*

Proof: First, we must prove that the Knapsack Problem is **NP**. The proof is given a set S of items that are chosen to go in a knapsack and it takes $\sum_{i \in S} s_i$ and $\sum_{i \in S} v_i$, which is takes polynomial time for the size of the input.

Second, we prove that the Subset Problem, which we already know to be NP-complete is reduceable to the Knapsack Problem in polynomial time. The Subset Problem is that given non-negative integer numbers $s_1, s_2, ...s_n$ and t, we can make a subset of these numbers with a total sum of t. We can easily reduce the Subset Sum problem to and instance of the Knapsack problem by creating a Knapsack problem that $\left\{ \begin{array}{c} a_i = c_i = s_i \\ b = k = t \end{array} \right\}$ [4]. The *Yes/No* answer to this new problem corresponds to the same answer to the original problem. We now prove that the following deduction implies the new problem is equivalent to the original problem $\left\{ \begin{array}{c} \sum_{i \in S} a_i \leq b \Leftrightarrow \sum_{i \in S} s_i \leq t \\ \sum_{i \in S} c_i \leq k \Leftrightarrow \sum_{i \in S} s_i \geq t \end{array} \right\}$ $\Leftrightarrow \sum_{i \in S} s_i = t$. Suppose that we have a *Yes* answer to the new problem. This means that we can fund a subset $S \subset [1, 2, ..., n]$ that satisfies the left part of the deduction. This subset S also satisfies the right side. Which means we get a *Yes* to the original problem. Conversely the same works for no.

Therefore we can say that because the Knapsack Problem is NP and we can reduce the Subset Sum problem to the Knapsack problem, we can say that the Knapsack Problem is NP-complete. ■

4 Conclusion

In conclusion, the dynamic memoization problem is merely one solution to the *BKP* and the *BKP* is merely one of the forms of the knapsack problem. There contains many other version with variations from fractions or whole numbers, bounds or no bounds, repetition or no repetition, etc. The memoization algorithm for this problem is not the fastest one that there is, but it does contain some fun things to note.

Because memoization uses a bottom-up approach it only solves the sub-knapsacks that it uses. This only shaves off a constant from the runtime, but it makes it slightly faster than other dynamic approaches. Another cool thing to note is that every table makes an underlying *Directed Acyclic Graph (DAG)*. This particular variant of knapsack boils down to finding the longest path in each *DAG* [1].

References

[1] S. Dasgupta, C. H. Papadimitriou, and U. V. Vazirani *Algorithms*.

[2] Bingler, A.H.E., Bulkan, S., Aaolu, M., (2016). *A Heuristic Approach for Shelf Space Allocation Problem*, Journal of Military and Information Science, Vol4(1),38-44.

[3] Hans Kellerer, Ulrich Pferschy, David Pisinger. *Knapsack Problems*. Springer-Verlag Berlin Heidelberg 2004

[4] Thomas Feng. *Solution to Assignment 4*. McGill, 2003

5 Appendix

The following is the code for my implementation of the *BKP*.

```
1  //Lela Bones
2  //Cosc 320 Project 3
3  //knapsacks.h
4  //This file will hold the prototypes for the knapsack class
5
6  #ifndef KNAPSACKS_H
7  #define KNAPSACKS_H
8  #include <iostream>
9  #include <vector>
10
11 // struct Items{
12 //    int index;
13 //    int weight;
14 //    std::vector<int> appearances;
15 // };
16 // structure to hold the items contained in the file
17 struct ItemStats
18 {
19     std::string name;
20     int value;
21     int size;
```

```
};

struct Item
{
    int index;
    int weight;
    std::vector<int> appearances;
};

//knapsack class declaration goes here
class Knapsacks{
  public:
    Knapsacks(){}
    ~Knapsacks(){}
    std::vector<int> DynamicKnapsack(int[], int[], int, int); //dynamic
    implentation
    bool IsSolution(std::vector<Item>&, int, int); //checks solution
    std::vector<ItemStats> ReturnBest(std::vector<ItemStats>&, std::vector<int>&);
    //returns solution
    void Display(std::vector<ItemStats>& packages, unsigned size);
    int TotalSize(std::vector<ItemStats>&);
    int Totalvalue(std::vector<ItemStats>&);
};

#endif
```

knapsack.h file

```
//Lela Bones
//Cosc 320 Project 3
//knapsacks.cpp
//This file will hold the functions of the knapsack class
#include <iostream>
#include "knapsacks.h"
#include <vector>
#include <algorithm>

std::vector<int> Knapsacks::DynamicKnapsack(int itemWeight[], int itemvalue[],
  int numItems, int maxWeight)
{
  // declare variables
  std::vector<int> best;
  std::vector<int> temp;
  std::vector<Item> pset;
  int** T = new int*[numItems+1];
  int deleted = 0;
  int currWeight = maxWeight;
  bool dontCheck = false;
  for(int i=0; i <= numItems; i++)
  {
    T[i] = new int[maxWeight+1];
    Item access;
    dontCheck = false;
    temp.clear();

    for(int y=0; y <= maxWeight; ++y)
    {
```

```
30        if(i==0 || y==0)
31        {
32          T[i][y] = 0;
33        }
34        else if(itemWeight[i-1] <= y)
35        {
36          T[i][y] = std::max(T[i-1][y-itemWeight[i-1]]
37              + itemvalue[i-1], T[i-1][y]);
38          if(T[i][y] == (T[i-1][y-itemWeight[i-1]]
39              + itemvalue[i-1]))
40          {
41            // if we find a valid packet, place it into
42            // the temp vector for storage
43            if(!dontCheck &&
44              !(std::find(temp.begin(), temp.end(),
45              itemWeight[i-1]) != temp.end()))
46            {
47              temp.push_back(itemWeight[i-1]);
48            }
49            // find all of the weight instances where
50            // this packet is "valid"
51            access.appearances.push_back(y);
52            dontCheck = true;
53          }
54        }
55        else
56        {
57          T[i][y] = T[i-1][y];
58        }
59      }// end for loop
60
61      // gather info about the packet we just found
62      if((std::find(temp.begin(), temp.end(),
63          itemWeight[i-1]) != temp.end()))
64      {
65        access.index = i-1;
66        access.weight = itemWeight[i-1];
67        pset.push_back(access);
68      }
69
70      // memory management, used to delete
71      // array indexes thats no longer in use
72      if(i > 1)
73      {
74        delete[] T[deleted++];
75      }
76    }// end for loop
77
78    delete[] T;
79
80    // obtain the best possible knapsack solutuion, and save
81    // their array indexes into a vector, starting from the end
82    // NOTE: this places the knapsack solution in opposite (reverse) order
83    for(int i = pset.size()-1; i >= 0; i--)
84    {
85      if(IsSolution(pset, i, currWeight))
```

```
86      {
87        best.push_back(pset.at(i).index);
88        currWeight -= pset.at(i).weight;
89      }
90      pset.pop_back();
91    }
92
93    // reverse the vector back into ascending order
94    std::reverse(best.begin(), best.end());
95
96    return best;
97  }// end of DynamicKnapsack
98
99  bool Knapsacks::IsSolution(std::vector<Item>& pset, int index, int currWeight)
100 {
101   return std::find(pset.at(index).appearances.begin(),
102     pset.at(index).appearances.end(), currWeight) !=
103     pset.at(index).appearances.end();
104 }// end of IsSolution
105
106 std::vector<ItemStats> Knapsacks::ReturnBest(std::vector<ItemStats>& items,
107   std::vector<int>& knapResult)
108 {
109   std::vector<ItemStats> best;
110   for(unsigned x=0; x < knapResult.size(); ++x)
111   {
112     best.push_back(items.at(knapResult.at(x)));
113   }
114   return best;
115 }// end of ReturnBest
116
117 void Knapsacks::Display(std::vector<ItemStats>& items, unsigned size)
118 {
119 for(unsigned x=0; x < size && x < items.size(); ++x)
120 {
121 std::cout<<items.at(x).name<<" "
122      <<items.at(x).size<<" "<<items.at(x).value<<std::endl;
123 }
124 }// end of Display
125
126 int Knapsacks::TotalSize(std::vector<ItemStats>& items)
127 {
128   std::vector<int> s = std::vector<int>();
129   int total = 0;
130
131   // if theres 2 parameters
132   if(!s.empty())
133   {
134     for(unsigned x=0; x < s.size(); ++x)
135     {
136       total += items.at(s.at(x)).size;
137     }
138   }
139   else // if theres only 1
140   {
141     for(unsigned x=0; x < items.size(); ++x)
```

```
    {
      total += items.at(x).size;
    }
  }
  return total;
}// end of TotalSize

int Knapsacks::Totalvalue(std::vector<ItemStats>& items)
{
  std::vector<int> s = std::vector<int>();
  int total = 0;

  // if theres 2 parameters
  if(!s.empty())
  {
    for(unsigned x=0; x < s.size(); ++x)
    {
      total += items.at(s.at(x)).value;
    }
  }
  else // if theres only 1
  {
    for(unsigned x=0; x < items.size(); ++x)
    {
      total += items.at(x).value;
    }
  }
  return total;
}// end of Totalvalue
```

knapsack.cpp file

```
//Lela Bones
//Cosc 320 Project 3
//main.cpp
//This file is the driver for the knapsack class

#include <iostream>
#include "knapsacks.h"
#include <fstream>
#include <cstdlib>
#include <cassert>
#include <fstream>

int main()
{
  // declare variables;
  Knapsacks proj5;
  ItemStats access;
  std::ifstream infile;
  std::vector<ItemStats> packages;
  std::vector<ItemStats> bestCombo;
  std::vector<int> knapResult;
  int maxWeight;
  int totPackages = 0;

  // open file & make sure it exists
```

```
  infile.open("knapsack_packages.txt");
  if(infile.fail())
  {
    std::cout<<"Cant find file!";
    exit(1);
  }

  // get the total number of packages from the file
  infile >> maxWeight;
  infile >> totPackages;
  int* packageWeight = new int[totPackages];
  int* packagevalue = new int[totPackages];
  // get the remaining info from the file
  //                      std::string        int             int
  for(int x=0;(infile >> access.name >> access.value >> access.size)
    && x < totPackages; ++x)
  {
    packages.push_back(access);
    packageWeight[x] = access.size;
    packagevalue[x] = access.value;
  }
  infile.close();

  // display stats
  std::cerr<<"There are "<<totPackages<<" items and the total weight is "<<
    maxWeight
    << std::endl << "Dynamic Search Solution" << std::endl;

  // return a vector containing the array indexes
  // of the best knapsack package solution
  knapResult = proj5.DynamicKnapsack(packageWeight, packagevalue, totPackages,
    maxWeight);

  // using the data found from above, return
  // // the packages that reside in those array indexes
  bestCombo = proj5.ReturnBest(packages, knapResult);

  // display info to the screen
    std::cout<<"Number of packages generated in this set: "<<bestCombo.size() <<
    std::endl
      <<"First 20 items.." << std::endl;

    // display the best solution packages
    proj5.Display(bestCombo, 20);

    // display the size and total value
    std::cout<<"Total Size = "<<proj5.TotalSize(bestCombo)<<" -- Total value = "
      <<proj5.Totalvalue(bestCombo)<<std::endl;

    // display the elapsed time
  delete[] packageWeight;
  delete[] packagevalue;
  return 0;
}
```

main.cpp file
This code is available on https://github.com/lazyyybonez/Knapsacks