Mark Watson
CS 4720
Homework #4
12/05/2011

Problem 1 Edit Distance Problem

The dynamic programming method I used was the Levenshtein distance algorithm. The way the algorithm works is by building a graph with the one string on the columns and one on the rows. The pseudo code for this algorithm is as follows:

```
s = string 1, m = length
t = string 2, n = length
d = matrix of size m by n
for i = 0 to n: d[m, i] = i
for i = 0 to m: d[i, n] = i
for i = 0 to m:
    for j = 0 to n:
        if char at s[i] = char at t[j]:
            // No change needed
            d[i, j] = d[i - 1, j - 1]
        else:
            d[i, j] = min(
              d[i-1, j] + 1,
              d[i, j-1] + 1,
              d[i-1, j-1] + 1
            )
```

The time complexity for the algorithm can be found by adding up the loops, assuming all of the code in the loops are run at some constant time:

$$\begin{aligned} T = & \sum_{i=0}^{n} C_1 + \sum_{i=0}^{m} C_2 + \sum_{i=0}^{m} \sum_{j=0}^{n} C_3 \\ = & \quad nC_1 + mC_2 + mnC_3 \\ \in & \quad \theta(mn) \end{aligned}$$

The implemented algorithm is attached. The output looks like this:

```
> run sport sort
1
> run computer commuter
1
> run word word
0
> run something nothing
4
```

The first run matches the example in the homework problem; adding one letter to "sort" results in "sport" and giving an edit distance of 1. In "computer" and "commuter" we can change one letter. The third example has the same word and so the edit distance is 0. Finally, four edits can be applied to "something" to get the word "nothing". ■

Problem 2 Making Change

a) The greedy algorithm for the change problem is simple. It starts at the largest coin and goes to the smaller ones, dividing the amount by the value of the coin each time. For example to find the change for 12 cents it tries dividing by 25 and fails. Then it divides by 10 and succeeds, subtracting 10 from the running total. It then divides the resulting 2 by 5 and fails. Finally, it divides thee 2 by 1 and subtracts 2 x 1 from 2 which gives 0. This completes the algorithm run and we get "$12 = 1(10) + 2(1)$". The Pseudo code is shown below. It needs amount to be set to the amount we're getting change for.

```
workingAmount = amount
for each coin:
    if workingAmount / coin value > 0:

        // We got the number and value of the coin to
        // use for this iteration
        output workingAmount / coin + " * " + coin
    endif
    workingAmount = workingAmount % coin
```

The algorithm is simple, and it's time complexity is clearly linearly proportional to the number of coins in the currency.

b) A good example of where the greedy algorithm fails is the English currency system before 1971[1]. The currency consists of the values: $1, 3, 6, 12, 24, 30, 60$, and 240. The greedy algorithm fails in the case of finding change for 48 pence, which is represented as $30 + 12 + 6$, while the optimal choice is $24 + 24$. This problem would also happen for the US currency if we were to add a 7 cent piece. The currency would be $25, 10, 7, 5, 1$, and if we wanted to make change for 14 we could use the greedy algorithm and get $10 + 4(1)$, while the optimal solution would be $2(7)$. It's a good thing there are no 7 cent coins.

c) The dynamic programming solution to the problem consists of two phases. The first phase calculates the coins to use to have the minimum number of coins. We then loop over the selected coins and find out how many of each one we need to use. For my implementation I also have a third step that prints out the coins. I don't count that in the following pseudo code analysis, since it's not part of the core algorithm.

```
for j = 1 until amount + 1:

    D[j] = Infinity
    for each coin:
        if j > coin and 1 + D[j-coin] < D[j]:
            D[j] = 1 + D[j - 1]

i = amount
while i > 0:

    print i
    n = n - D[i]
```

The time complexity of the code for the first part is easy to find. It's simply proportional to the amount times the number of coins in the currency. The while loop depends on the makeup of the final solution, since the number of times that need to be iterated depend on the configuration. If we let "n" be the number of coins, and "a" be the amount, we can calculate the worst case time complexity as follows:

$$\begin{aligned} T = & \quad n * a * C + a * D \\ \in & \quad O(na) \end{aligned}$$

[1]D. Pearson, "A polynomial-time algorithm for the change-making problem," Operations Research Letters, vol. 33, no. 3, pp. 231-234, May 2005. [Online]. Available: http://dx.doi.org/10.1016/j.orl.2004.06.001

The last term comes from the fact that we will not loop more than the amount, assuming all coins are of value "1" in the worst case.

The implementation for the algorithm is in the same source file as for the greedy algorithm. Here are several examples where the new algorithm works, and the old one does not:

1. Currency: $1, 3, 6, 12, 24, 30, 60$, and 240, Amount: 48
2. Currency: 25,10,7,5,1, Amount: 14
3. Currency: 4,3,1, Amount: 6
4. Currency: 8,7,1, Amount: 21
5. Currency: 6,4,1, Amount: 8

All of these examples won't work with greedy algorithms, while the dynamic programming model will work. ∎

Problem 3 Exponential Time Problems

See attached sheet. ∎

Problem 4 Extra Credit - Branch-and-Bound

I implemented a branch and bound knapsack algorithm based on the Levitin book. I tested it on various input.

```
--> Size: 4, Items: {(weight=2, value=40),(weight=1, value=1)}
> run
Take item: w=2, v=40, v/w=20
Take item: w=1, v=1, v/w=1
--> Size: 10, Items: {
    (weight=4, value=40),
    (weight=7, value=42),
    (weight=5, value=25),
    (weight=3, value=12),
    }
```

```
> run
Take item: w=7, v=42, v/w=6
Take item: w=3, v=12, v/w=4
```

The algorithm finds the optimal items to take, assuming there is only one of each item. ∎

Exponential Algorithms

Mark Watson

***Abstract*—This document describes various NP-Complete algorithms.**

***Index Terms*—algorithms, np-complete**

I. Introduction

THIS document covers five different problems that are NP-complete.

II. Problems

I'll cover the following problems that have been shown to be classified as NP-complete problems.

1) Knapsack Problem
2) Traveling Salesman Problem
3) Subset Sum Problem
4) Subgraph Isomorphism Problem
5) Graph Coloring

A. Knapsack Problem

The knapsack problem is a simple optimization problem trying to maximize to value of items included in a storage container with limited space. Given a finite list of items, each item has a different value and size. The goal is to find the combination of items that has the highest value while still fitting in the given space allocation. For example, the storage space could be a bugler's bag, and the items could be the belongings of some person being robbed. The knapsack problem is related to the subset sum problem, which will be discussed shortly.

Given this overview, we can describe the problem in more concrete terms; given a bag of size W, a number of item values, v_i, and a number of item weights, w_i, we need to find a set of items, $\{p_1, p_2, ..., p_j\}$ that maximizes v_i while keeping sum of the items less than or equal to W. In the algorithm I'm discussing here there is only one of each item. The problem can also be generalizing to have an infinite amount of each item, which is known as the unbounded knapsack problem. The bounded knapsack problem can be solved using dynamic programming. The unbounded problem is more difficult. There are several approximation algorithms that can be used to solve the unbounded problem. One possible solution is a greedy algorithm proposed by [1]. This algorithm is far from optimal at solving the bounded problem.

Another way of optimizing the problem is to use dominance relations to throw away items we don't need. This applies only to the unbounded version of the problem.

B. Traveling Salesman Problem

The traveling salesman problem is a graph traversal problem. Given a non-directed, weighted graph the goal is to find the shortest Hamiltonian tour visiting each node once. The graph is fully connected with each node visiting every other node. The problem can be thought of as a set of cities with a traveling salesman trying to find the optimal tour between them. The problem has applications in several areas like planning and logistics[1].

The problem is NP-complete, so finding a solution using brute force doesn't work. Approximate solutions involve using iterative heuristics that start with a basic solution and then get better and better results. A starting algorithm could be a greedy nearest neighbor. Optimization involves looking at nodes that are close to each other and then swapping them. There are other approximation algorithms such as genetic algorithms or ant farms to try to obtain the optimal solution, but the Lin–Kernighan–Johnson or V-opt heuristics are state of the art.

C. Subset Sum Problem

The subset sum problem is simply given a set of numbers is there a subset that adds up to zero. The problem is NP-complete and requires finding all subsets to solve in a brute force manor. Using a divide and conquer method to solve for each half at once to change the algorithm into $O(2^{N/2})$, although it is still exponential.

There are two other ways to solve the problem as well. The first is a non-approximate solution using dynamic programming. It's order O(P - N), where P is the sum of the positive values and N is the sum of the negative values. This means the algorithm isn't really polynomial time [2]. The other algorithm is a polynomial time approximate algorithm. The algorithm doesn't always give the exact result, but it doesn't take forever to run since it's polynomial time.

D. Tetris

In the game of Tetris there are a few NP-complete problems [3]. One of which is maximizing the number of pieces played before a loss, given a preset sequence of pieces. To find the optimal solution every possibility must be enumerated and then tested. Since we're trying to fin a maximum we can use other methods to find approximate solutions such as branch and bound, etc. There aren't any dedicated algorithms for this problem as it isn't particularly important.

[1] http://en.wikipedia.org/wiki/Travelling_salesman_problem
[2] http://en.wikipedia.org/wiki/Subset_sum_problem
[3] http://en.wikipedia.org/wiki/Tetris

E. Graph Coloring

Given a graph G with n vertices and some integer k we need to find out if G admits a proper vertex coloring with k colors. A proper vertex coloring happens when no two adjacent nodes are colored the same thing. The act of finding a coloring for the graph takes k^n using brute force search. Using dynamic programming we can get a faster algorithm. The best algorithm is Yates algorithm, which runs in $O(2^n n)$.

REFERENCES

[1] G. B. Dantzig, "Discrete-variable extremum problems," *Operations Research*, vol. 5, no. 2, pp. pp. 266–277, 1957. [Online]. Available: http://www.jstor.org/stable/167356

```scala
object EditDistance {
  def levenshteinDistance(s: String, t: String): Int = {
    val m = s.length
    val n = t.length
    var d = Array.ofDim[Int](m+1,n+1)

    for (i <- 0 until m+1) {
      d(i)(0) = i
    }

    for (j <- 0 until n+1) {
      d(0)(j) = j
    }

    for (i <- 0 until m) {
      for (j <- 0 until n) {
        if (s(i) == t(j)) {
          d(i+1)(j+1) = d(i)(j)
        } else {
          d(i+1)(j+1) = List(
            d(i)(j+1) + 1, // deletion
            d(i+1)(j) + 1, // insertion
            d(i)(j) + 1    // substitution
          ).min
        }
      }
    }

    d(m)(n)
  }

  def main(args: Array[String]) = {
    if (args.length == 0) {
      println("USAGE: scala edit_distance.scala STRING1 STRING2")
    }
    println(levenshteinDistance(args(0), args(1)))
  }
}
```

```scala
import collection.mutable.HashMap

object Change {
  //val currency = List(25, 10, 5, 1)
  val currency = List(240,60,30,24,12,6,3,1)

  def getDynamicChange(amt: Int) = {
    // Get the change and put it in an array using a dynamic
    // algorithm.
    val data = new Array[Float](amt + 1)
    val denom = new Array[Float](amt + 1)
    data(0) = 0
    for (j <- 1 until amt + 1) {
      data(j) = Float.PositiveInfinity
      for (i <- currency) {
        if (j >= i && 1 + data(j - i) < data(j)) {
          data(j) = 1 + data(j - i)
          denom(j) = i
        }
      }
    }

    // Loop through and get the letters.
    val finalLetters = new HashMap[Int, Int]
    var j = amt
    while (j > 0) {
      val x = denom(j).toInt
      if (finalLetters.contains(x)) {
        finalLetters(x) += 1
      } else {
        finalLetters(x) = 1
      }
      j -= x
    }

    // Get a string of the results
    var out: List[String] = Nil
    for ((denom, cnt) <- finalLetters) {
      out ::= "%d(%d)".format(cnt, denom)
    }

    out.reverse.reduceLeft(_ + " + " + _)
  }

  def getChange(amt: Int) = {
    var out: List[String] = Nil
    var workingAmt = amt
    for (x <- currency) {
      if (workingAmt / x > 0) {
        out ::= "%d(%d)".format(workingAmt / x, x)
      }
      workingAmt = workingAmt % x
    }

    out.reverse.reduceLeft(_ + " + " + _)
  }

  def usage = {
    println("USAGE: change AMOUNT")
    System.exit(1)
  }

  def main(args: Array[String]) = {
    if (args.length != 1) usage
```

```
    // Call the algorithm that will make change for a regular currency.
    val amt = args(0).toInt
    println("Greedy algorithm: " + getChange(amt))
    println("Dynamic Programming algorithm: " + getDynamicChange(amt))
  }
}
```

```scala
// Branch and bound knapsack problem
import collection.mutable.PriorityQueue

/** This class represents a thing we can put in the napsack.
  */
class Thing(wIn:Int, vIn:Int) extends Ordered[Thing] {
  val w = wIn
  val v = vIn
  def v_w = v / w
  override def toString = "w=%d, v=%d, v/w=%d".format(w, v, v_w)
  override def compare(that:Thing) = {
    v_w compare that.v_w
  }
}

/** This is a node in the state-space tree
  */
class Node(wIn:Int, vIn:Int, ubIn:Int, levelIn:Int) extends Ordered[Node] {
  val w = wIn
  val v = vIn
  val ub = ubIn
  val level = levelIn
  var left:Node = null
  var right:Node = null
  var use = false

  override def compare(that:Node) = {
    ub compare that.ub
  }

  override def toString = {
    "---\nw=%s\nv=%s\nub=%s\nlevel=%s\nuse=%s\n---".format(
      w, v, ub, level,
      use match {
        case true => "true"
        case false => "false"

      }
    )
  }
}

/** This class finds the optimal solution to the given napsack
  * problem using branch and bound.
  *
  * items: the array of things that we can put in the napsack.
  * size: the size of the knapsack.
  */
class Knapsack(itemsIn:Array[Thing], size:Int) {
  // Sort all the lists by v/w and put everything into a new object.
  val items = itemsIn.sorted.reverse

  /** This function gives the upper bound on an item, i
    */
  def upperBound(w:Int, v:Int, v_w:Int) = {
    //println("%d + (%d - %d)(%d)".format(v, size, w, items(i).v_w))
    v + ((size - w) * v_w)
  }

  /** Finds the optimal items to put in the knapsack. Returns
    * a list of the indeces of the items to use in the list.
    */
  def findOptimal:List[Int] = {
    var result:List[Int] = Nil
```

```
    // the main part - build the state-space tree
    var foundOptimal = false;
    // A priority queue to keep track of the leaves so we know which node
    // to work on next
    var leaves = new PriorityQueue[Node]
    val root = new Node(0, 0, upperBound(0, 0, 0), 0)
    leaves += root

    while (!foundOptimal) {
      // Find out which parent node to work from
      val parent = leaves.dequeue
      val level = parent.level

      // Find the v_w value
      val v_w = if (level + 1 < items.length) {
        items(level + 1).v_w
      } else 0

      // If we're using this node then add it to the result
      if (parent.use) {
        result ::= level - 1
      }

      // Terminating case
      if (level + 1 > items.length) {
        foundOptimal = true
      // Otherwise find the children nodes.
      } else {
        // Add the left node
        val l_w = parent.w + items(level).w
        // If this node is feasable and we haven't yet reached the
        // end of the nodes
        if (l_w <= size && !foundOptimal) {
          val l_v = parent.v + items(level).v
          val l_ub = upperBound(l_w, l_v, v_w)
          parent.left = new Node(l_w, l_v, l_ub, level + 1)
          parent.left.use = true
          leaves += parent.left
        }

        // Add the right node
        val r_w = parent.w
        // If this node is feasable
        if (r_w <= size) {
          val r_v = parent.v
          val r_ub = upperBound(r_w, r_v, v_w)
          parent.right = new Node(r_w, r_v, r_ub, level + 1)
          leaves += parent.right
        }
      }
    }

    result.reverse
  }
}

object Main {
  def main(args: Array[String]) = {
    // This is the sample problem defined in the Levitin book.

    val things = Array(
      new Thing(7, 42),
      new Thing(5, 25),
```

```
      new Thing(3, 12)
      //new Thing(4, 40)
    )
    val k = new Knapsack(things, 10)
    val optimal = k.findOptimal

    // val things = Array(
    //   new Thing(2, 40),
    //   new Thing(1, 1)
    // )
    // val k = new Knapsack(things, 4)
    // val optimal = k.findOptimal

    optimal.foreach(x =>
      println("Take item: " + things(x)).toString
    )
  }
}
```