Analysis of Algorithms Homework 5: Activity Selection Problem

Lowell Batacan, Joshua Steward

November 22, 2016

In many fields like mathematics, computer science, and economics, dynamic programming is a problem solving method that helps solve complex problems by breaking it down into subproblems and using the solutions of those subproblems to solve the entire problems. Usually it is a memory-based data structure. It saves computation time and is inexpensive because it takes up a modest storage space and solves the subproblems within the scope of the actual problem. An example of a dynamic programming algorithm is the Fibonacci Sequence. The goal of the algorithm is to calculate a list of integers where a pair of integers in chronological order create a sum which is the next number. This is done in multiple ways, one obvious method is through recursion.
Recursion is process where a process is done repeatedly to solve subproblems. It is mostly seen in computer science and mathemetics to solve problems through iteration. One probelm that makes use of the concepts of dynamic programming to to find a solution is the **Activity Selection Problem**.
The activity selection problem is a combinatorial optimization problem concerning the selection of non-conflicting activities to perform within a given time frame, given a set of activities each marked by a start time (si) and finish time (fi). The problem is to select the maximum number of activities that can be performed by a single person or machine, assuming that a person can only work on a single activity at a time. Given an input as a list of activities: (s_1, f_1, p_1)... (s_n, f_n, p_n) where $p_i > 0$, $s_i < f_i$, and all of them are non-negative real numbers.

Algorithm Pseudocode

```
1. A( 0 ) ← 0
2. for j: 1 ... k do
3.         max ← 0
4.        for i = 1 ... n do
5.        if f_i = u _j then
6                 p_i + A( H ( i ) ) > max then
7.                         max ← p_i + A( H( i ) )
8.                 end if
9.         end if
10.        end for
```

11. if A(J, 1) > max **then**
12. max ← A(J, 1)
13. **end if**
14. A(j) ← max
15. **end for**

Our take on Savitchs algorithm is broken up into several parts. First, using a more pythonic approach to populating an n x n matrix, we take input as n and create an n x n matrix from this. The matrix is then initialized, and populated as an adjacency matrix with two built in diagonal paths. The algorithm will search from the top left to the bottom right. As such, Savitchs algorithm is called with the matrix M as the first input, 0 as the second input, (n * n) 1 or the max distance as the third input, and the size of the matrix as the last input. Savitch begins by appending the first predicate as the current first vertex (u), the current vertex to search for (v) and k to the stack. It then checks for the two base cases of either having the beginning and final vertices being the same or the edge connecting to the last vertex being true, or being 1 as it is in an adjacency matrix. For the recursive call, we search every midpoint w in the path, and call Savitchs algorithm on both halves of the path, with k-1 vertices as the floor for the first recursive call, and k-1 vertices as the ceiling for the second recursive call. This returns true or false, and if true, it proceeds.