1 Introduction and Objective

This is a short tutorial which will be a prelude to the third part of CS2010: Dynamic Programming, abbreviated as DP.

DP is usually viewed as a problem solving paradigm but it can also be viewed as an algorithm on a special class of graph: Directed Acyclic Graph (DAG). The problems discussed in this tutorial involves DAGs. And since we will work with DAGs, we will also re-discuss topological sort on DAG which is an essential concept behind DP technique. We will see how topological ordering of the vertices of our DAG allows us to perform efficient (DP) algorithms on it.

We will heavily use http://visualgo.net/recursion.html and also the topological sort feature in http://visualgo.net/dfsbfs.html in this tutorial.

The tutor will try to end the tutorial earlier so that you can ask about past Written Quiz 2 papers, if any.
2 Tutorial 09 Questions

Review Topological Sort

Q1. Given the DAG in Figure 1, show how to use One-Pass Bellman Ford’s (a.k.a. DP) to find both the Shortest Paths and the Longest Paths from source vertex 2 to every other (reachable) vertices. The tutor is empowered to actually change the DAG in the picture during the actual tutorial. Steven has not actually implement this feature in http://visualgo.net/sssp.html so this has to be semi-manual for now.

Figure 1: A Sample DAG

Q1.a). Find the topological order of the DAG in Figure 1 using the modified DFS routine as taught in Lecture 06!

The topological order of this DAG found by the modified DFS algorithm is \{6, 8, 7, 2, 0, 5, 3, 1, 4\}. Recall that the modified DFS as taught in Lecture 06 prefers vertex with lower vertex number if it has to choose one of several valid vertices.

Later when discussing Question 2, the tutor will ask students to link bank to this example.

Q1.b). Now perform One-Pass Bellman Ford’s to get the Shortest Paths from source vertex 2 to every other (reachable) vertices.

First, initialize the shortest path estimate as usual (\(\text{dist}[s] = \text{dist}[2] = 0\), \(\text{dist}[\text{the rest}] = \infty\)).

Now relax the outgoing edges of all vertices in the given topological order. Since vertex 6, 8, and 7 are not reachable from the source vertex 2, their shortest path estimates are still \(\infty\) and relaxing their edges will not result in any changes (this fact can be used to reduce the number of edge relaxations).
When we reach vertex 2 onwards in the topological order, all in-coming edges to a vertex will have been relaxed before we reach it, so it’s shortest path value will have been found by the time we reach it. In one pass, we can compute the shortest paths from vertex 2 to all other vertices in the graph.

The final shortest paths spanning tree from source vertex 2 can be seen in Figure 2.

Q1.c). Now perform One-Pass Bellman Ford’s to get the Longest Paths from source vertex 2 to every other (reachable) vertices.

First, initialize the longest path estimate as usual (dist[s] = dist[2] = 0, dist[the rest] = −∞, notice the negative).

Now stretch the outgoing edges of all vertices in the given topological order. Since vertex 6, 8, and 7 are not reachable from the source vertex 2, their longest path estimates are still −∞ and stretching their edges will not result in any changes. When we reach vertex 2 onwards in the topological order, all in-coming edges to a vertex will have been stretched before we reach it, so it’s longest path value will have been found by the time we reach it. In one pass, we can compute the longest paths from vertex 2 to all other vertices in the graph.

The longest paths values for vertex {0,1,2,3,4,5} are {2,6,0,43,21,6}, respectively. The final longest paths spanning tree from source vertex 2 is very similar to Figure 2, just that we have edge 2 → 4 instead of edge 1 → 4.

Q2. The modified DFS topological sort algorithm given in the Lecture 06 only gives only one valid topological ordering. How can we find all possible valid topological orderings for a given DAG? For
example, there are \textbf{1008} possible valid topological orderings of the DAG in Figure 1 on top of one that you find in Q1.a).

The best case is if we are given \( V \) vertices as a linked list of \( E = V - 1 \) edges, so there is only one valid topological ordering. The worst case is if we are given \( V \) vertices without any edge, as that means we will have all \( V! \) possible topological orderings.

Knowing that the worst case is \( O(V!) \), we might as well try all those \( V! \) permutation of \( V \) vertices and run an \( O(E) \) check to see if that permutation is a valid topological ordering. This algorithm runs in \( O(V! \times E) \). Very slow, but it will produce correct answer.

Another way is to modify DFS that behaves more like backtracking by setting \textit{visited}[u] to false again before exiting the recursion for a vertex \( u \) (we will see this technique in a Lecture 11). This modification will explore all possible valid permutations in a more efficient manner than the other strategy above, but the worst case behavior is still \( O(V!) \) and there is no way we can optimize this further.

Oh, and if you don’t believe there are 1008 valid topological orderings in Figure 1, you can run this C++ code.

```cpp
#include <bits/stdc++.h>
using namespace std;

typedef pair<int, int> ii;
int i, V, p[9], u, v, pos_u, pos_v, ans;
vector<ii> EL;

int main() {
    V = 9; // 9 vertices
    // 9 edges
    EL.push_back(ii(0,1)); EL.push_back(ii(0,5));
    EL.push_back(ii(1,4));
    EL.push_back(ii(2,0)); EL.push_back(ii(2,1)); EL.push_back(ii(2,4));
    EL.push_back(ii(5,3));
    EL.push_back(ii(6,7)); EL.push_back(ii(6,8));

    for (i = 0; i < V; i++) p[i] = i;

    ans = 0;
    do {
        bool valid = true;
        for (i = 0; i < (int)EL.size(); i++) {
```

```cpp
    ```
u = EL[i].first;
v = EL[i].second;
for (pos_u = 0; pos_u < V; pos_u++) if (p[pos_u] == u) break;
for (pos_v = 0; pos_v < V; pos_v++) if (p[pos_v] == v) break;
if (pos_u > pos_v) valid = false;
}
if (valid) ans++;
}
while (next_permutation(p, p+9));
printf("There are %d valid topological orderings\n", ans);
return 0;

Dynamic Programming - Basic Ideas

Q3. What are the two important properties of a problem that allows for a DP solution?

This is just a quick review question about the most basic idea of DP. Keep this in mind and see if the two very simple DP problems discussed in this tut09 have these two properties.

1. Optimal Substructure.
(refer to shortest/longest/counting path on a DAG examples in Lecture 09)

If you can write a recurrence that describe the problem, this part is satisfied. You will strengthen your recursive skills by solving these DP problems.

2. Overlapping Subproblems
(refer to shortest/longest/counting path on a DAG examples in Lecture 09)

Certain subproblems are revisited more than once, usually many times. This is something that we want to avoid with DP. This is the key part of the last topic in CS2010 syllabus: Space-Time trade-off.

Q4. You are given the following implementation for finding the $n$-th Fibonacci number. Is it a DP solution? If yes, explain! If no, how to write a DP solution (either Top-Down or Bottom-Up) for finding the $n$-th Fibonacci number?

Algorithm 1 fib($n$)

\[
\text{if } n == 0 \text{ or } n == 1 \text{ then}
\text{return } n
\]
\text{end if}
\text{return } \text{fib($n$-1) + fib($n$-2)}
The first criteria of optimal substructure is satisfied as \( fib(n) \) is a recursive function. However, the second criteria is not satisfied. Even though there are overlapping subproblems, the solutions to such subproblems are re-computed again and again, e.g. see the recursion tree of \( fib(5) \) in Figure 3. The tutor can use VisuAlgo to show the recursion tree of larger \( n \). Notice that computations for \( fib(0), fib(1), fib(2), \) and \( fib(3) \) are repeated many times. Only \( fib(4) \) and \( fib(5) \) that are only computed once in Figure 3.

![Recursion Tree of fib(5)](http://visualgo.net/recursion.html)

There are two ways to write DP solution to avoid re-computations of overlapping sub problems.

The first solution is called: Top-Down DP where we use memo table to help the complete search recursion to check if the current computation is the first one or if it is a repeated one. If it is the first one, the recursion will proceed as per normal and store the result in memo table. If it is a repeated one, the recursion will not continue and simply report the result already stored in memo table previously.

```java
// initially, the memo table is set to all -1
```

The Recursion DAG (no longer Recursion Tree) of \( fib_{dp}(5) \) is as shown in Figure 4. There is no redundant computation as we memoize the solutions of subproblems. Furthermore, do you notice that \( fib_{dp}(n) \) actually computes the number of paths from vertex \( n \) to vertex 1 in this Fibonacci DAG?
Algorithm 2 \texttt{fib\_dp}(n)
\begin{verbatim}
if \( n == 0 \) or \( n == 1 \) then \triangleright \text{The base cases (the easy cases) remain as they are}
return \( n \)
end if
if \( \text{memo}[n] \neq -1 \) then \triangleright \text{If this subproblem has been computed before (the value is no longer -1)}
return \( \text{memo}[n] \) \triangleright \text{Do not compute again, simply report this value}
end if
return \( \text{memo}[n] = \text{fib\_dp}(n-1) + \text{fib\_dp}(n-2) \) \triangleright \text{Compute and store the result in memo table.}
\end{verbatim}

Figure 4: Recursion DAG of \texttt{fib\_dp}(5) as visualized in \url{http://visualgo.net/recursion.html}

We will also use this opportunity to show the second solution: Bottom-Up DP. This version does not use recursion, but just a table (we do not call it memo table, but usually just a DP table) and usually some loops. We start from known base cases and then use the information that we already know to compute the optimal value for other subproblems according to a certain topological order of the (computation) DAG.

Algorithm 3 \texttt{bottomup\_fib\_dp}(n)
\begin{verbatim}
fib[0] = 0, fib[1] = 1 \triangleright \text{These are the two known base cases}
for int \( i = 2; i <= n; i++ \) do
fib[\( i \)] = fib[\( i - 1 \)] + fib[\( i - 2 \)] \triangleright \text{with the previous two values, we can compute the current value}
end for
\end{verbatim}

Now, do you realize that in bottom-up DP, the computation DAG is the same as in Figure 4 but with ALL arrow direction reversed, i.e. you start from \( \text{fib}(0) \) and \( \text{fib}(1) \), then point two arrows to \( \text{fib}(2) \). Then, \( \text{fib}(1) \) and \( \text{fib}(2) \) point arrows to \( \text{fib}(3) \), and so on.

Both DP runs in \( O(n) \). Top-down analysis: There are \( O(n) \) states, each state is computed two times.
So overall time complexity: $O(2n) = O(n)$. Bottom-up analysis: It is clear that the for loop runs from 2 to $n$ (inclusive), or $O(n - 1) = O(n)$.

Q5. Given a target value $n$ in cents, and a list $C$ of $k$ different coin denominations (each coin denomination is also given in terms of cents), what is the minimum number of coins that we must use in order to obtain the amount $n$? You can assume we have an unlimited supply of coins of any type, $1 \leq n, k \leq 1000$, and the smallest coin denomination is a 1 cent coin.

Example: Given $n = 10$, $k = 2$ and $C = \{1, 5\}$, we can use:

1. Ten 1 cent coins = $10 \times 1 = 10$. Total coins used = 10
2. One 5 cents coin + five 1 cent coins = $1 \times 5 + 5 \times 1 = 10$. Total coins used = 6
3. Two 5 cents coins = $2 \times 5 = 10$. Total coins used = 2 → Optimal

Now your tasks:

1. Model the coin change problem above as a DAG (the key point).
2. Once you have that DAG, solve the problem with a standard graph algorithm that you have actually learned in Lecture 06, 08, and 09. Alternatively, write either a Top-Down or Bottom-Up DP solution for this problem.

We can model this coin change problem as a DAG (see Figure 5, 6, and 7). We represent each value from 0 to $n$ as a node and label them as such. We link node $u$ to $v$ if $\text{value}(u) - \{\text{some coin denomination} i\} = \text{value}(v)$.

We can use a simple BFS to solve this.

Figure 5: DAG of Coin Change problem
Figure 6: Recursion Tree of cc(10) as visualized in [http://visualgo.net/recursion.html](http://visualgo.net/recursion.html).

Figure 7: Recursion DAG of cc(10) as visualized in [http://visualgo.net/recursion.html](http://visualgo.net/recursion.html), the layout is not as good as manual placement in Figure 5 as we utilize generic graph drawing/layout algorithm.
Solving Coin Change problem with DP (tutor may want to show test case where greedy algorithm fails on general case):

The major drawback of the graph solution above is that you have to build the graph (the DAG) explicitly. We do not have to do that. We can let recursion (Top-Down DP) or carefully constructed loops (Bottom-Up DP) to achieve the same feat.

Top-down DP: Let $v$ be the remaining cents we want to change and $\text{change}(v)$ be the minimum number of coins needed to change $v$.

// memo is an array of size $N+1$ (because we need index 0 to $N$, inclusive)

// we initialize this memo table with all -1, because this value -1 is not used in the problem (there is nothing as -1 number of coin)


Algorithm 4 change($v$)

if $v == 0$ then
  return 0  \text{ ▷ Base case}
end if

if $v < 0$ then
  return $\infty$  \text{ ▷ Invalid value}
end if

if memo[$v$] != -1 then
  return memo[$v$]  \text{ ▷ Already computed before}
end if

memo[$v$] = $\infty$

for $i = 0$ to $k - 1$ do
  memo[$v$] = min(memo[$v$], 1 + $\text{change}(v-C[i])$)  \text{ ▷ The recursive case}
end for

return memo[$v$]

Call change($n$) in main program. If change($n$) reports $\infty$, then there is no way to make change for $N$ cents; otherwise, report the value of change($n$). This top-down approach runs in $O(nk)$, because there are $O(n)$ states and each state needs a loop that runs $k$ iterations, so the overall time complexity is $O(nk)$. 
The bottom-up DP again has similar structure as the top-down version, just that this time we use DP table and loops. This one is clearly $O(nk)$ but maybe not intuitive for students who are more inclined towards recursion.

**Algorithm 5 Bottom-Up DP for coin change**

```plaintext
for i = 1 to n do                        ▷ Initialization
    change[i] = ∞
end for

change[0] = 0                              ▷ Base case

for i = 1 to n do                         ▷ For the next coin values, from 1 to n, because that is the topological order
    for j = 0 to k − 1 do
        if i − C[j] >= 0 then
            change[i] = min(change[i], 1 + change[i − C[j]])  ▷ To prevent negative index
        end if
    end for
end for

the answer is at change[n]
```

**Discuss Past Written Quiz 2**

Before dismissing the class, the tutor will give opportunity for student to ask any one random question from past WQ2 papers and it will be quickly discussed in class. For subsequent questions, students can meet the tutor outside class.