CS2010 – Data Structures and Algorithms II

Lecture 11 – Traveling Salesman

stevenhalim@gmail.com
Outline

DP Algorithms on (Implicit) DAG using the classic TSP (Traveling Salesman Problem) as the underlying theme

The key point of this lecture:
• Conversion of a general graph into a DAG with an addition of one (or more) extra parameter(s)
  – One simpler example: The simpler TSP variant
  – One harder example: The classic TSP with DP solution

Reference: CP3 Section 2.2, 3.5, and 4.7.1
What is the |LIS| of $X = \{8, 3, 6, 4, 5, 7, 7\}$?

LIS = Longest Increasing Subsequence

1. 1
2. 2
3. 3
4. 4
5. 5
6. 6
7. 7
What is the $|\text{LND}_S|$ of $X = \{8, 3, 6, 4, 5, 7, 7\}$?

$\text{ND} = \text{Non Decreasing}$

1. 1
2. 2
3. 3
4. 4
5. 5
6. 6
7. 7
How many distinct paths to go from \((0, 0)\) to \((3, 3)\) if you can only go down or right at every cell?

1. 6
2. 10
3. 20
4. 40
5. \(\infty\)
TRAVELING SALESMAN PROBLEM
(THE SIMPLER EXAMPLE)
Motivating Problem
(UVa 10702 – Traveling Salesman)

• There are \( C \) cities
• A salesman starts his sales tour from city \( S \) and can end his sales tour at any city labeled with \( E \)
• He wants to visit many cities to sell his goods
• Every time he goes from city \( U \) to city \( V \), he gets \( \text{profit}[U][V] \)
  – \( \text{profit}[U][U] \) (i.e. staying at the same city) is always 0...
• What is the maximum profit that he can get?

profit\([U][V]\) is shown as the weight of edge\((U, V)\)
What is the maximum profit that he can get?

1. 3
2. 5
3. 7
4. 17
5. $\infty$
Reducing to SS Longest (non simple) Path Problem but on General Graph

This is a problem of finding the longest (non simple) path on general graph :O

- General graph has something that does not exist in DAG discussed in previous lecture: Cycle(s)
- There are several positive weight cycles in this graph, e.g. \(0 \rightarrow 1 \rightarrow 2 \rightarrow 0\) (weight 13); \(0 \rightarrow 1 \rightarrow 0\) (weight 8), etc
  - The salesman can keep re-visiting these cycles to get more $$ :O
A Note About Longest Path Problem on General Graph

The longest (non simple) path from city $S = \text{city 0}$ to any city with label $E$ will have $\infty$ weight

- One can go through any positive weight cycle to obtain $\infty$

The longest (simple) path from city $S = \text{city 0}$ to any city with label $E$ is: $0 \rightarrow 2 \rightarrow 1$ with weight $5+2 = 7$

- But this is hard and requires “backtracking”, as shown later
Conversion to a DAG

The actual problem (UVa 10702) has an extra parameter that converts the general graph into a DAG

- The salesman can only make $T$ inter-city travels :O
  - In a valid tour, he must arrive at an ending vertex after $T$ step
- **KEY STEP**: Now each vertex has an additional parameter:
  - **num of steps left**
- The graph now looks like this, a DAG:

Suppose he start at S with $T=3$
Suppose he start at S with T=3

Notice overlapping vertices here
A Note About Longest Path Problem on Directed Acyclic Graph

There is no longest *non simple* path on DAG

- This is because every paths on DAG are simple, including the longest path 😊

So we can use the term SSLP on DAG, but we usually have to use the term SSL(simple)P on general graph
What is the solution for the SSLP on DAG problem?

- We are already familiar with this from the previous lecture
  - SS Longest paths on DAG can be solved with either:
    A. Find topological order and “stretch” edges according to this order/bottom-up DP (or sometimes called as ‘graph way’)
    B. Or write a recursive function (top-down DP) with memoization

But this problem is *harder* to be solved with graph way

- The vertices now contain *pair* of information: (vertex_number, steps_left)
  - The number of vertices is not C, but now C*T
    - Pure graph implementation is slightly more difficult...
Let’s solve this problem with top-down DP (recursion + memo)

• We will not actually build the (implicit) DAG

Let \text{get\_profit}(u, t) be the maximum profit that the salesman can get when he is at city \textbf{u} with \textbf{t} number of steps to go:

• if \textbf{t} = 0
  – If the salesman can end his tour at city \textbf{u}, i.e. city \textbf{u} has label \textit{E};
    • Then \text{get\_profit}(u, t) = 0
  – else if the salesman cannot end his tour at city \textbf{u};
    • Then \text{get\_profit}(u, t) = -\text{INF} (to say that this is a bad choice)

• else (if \textbf{t} > 0),
  – \text{get\_profit}(u, t) = \max(\text{profit}[u][v] + \text{get\_profit}(v, t-1))
    for all \textbf{v} \in [0 . C-1] and we skip the case where \textbf{v} = \textbf{u} (as we cannot stay)
In Java code (see UVa10702.java):

```java
private static int get_profit(int u, int t) {
    if (t == 0) // last inter-city travel?
        return canEnd[u] ? 0 : -INF;
    if (memo[u][t] != -1) // computed before?
        return memo[u][t]; // use a 2D array as memo
    memo[u][t] = -INF;
    for (int v = 0; v < C; v++) {
        if (v == u) continue; // we cannot stay
        memo[u][t] = Math.max(memo[u][t],
            profit[u][v] + get_profit(v, t-1));
    }
    return memo[u][t]; // to avoid re-computation
}
```
DP Feature @ VisuAlgo

Try [visualgo.net/recursion.html](https://visualgo.net/recursion.html) and load this example: “UVa 10702” to get the same DAG, and then you can modify the recursion and/or the test case to dynamically generate new recursion DAG.
DP Analysis

What is the num of distinct states/space complexity?
• That is, the #vertices in the DAG
  – Answer: $O(C \times T)$

What is the time to compute one distinct state?
• That is, the out-degree of a vertex
  – Answer: $O(C)$, scan all $C$ vertices but exclude staying at current vertex

What is the overall time complexity?
• That is, #edges in the DAG = #vertices*outdegree of each vertex
  – Or number of distinct states * time to compute one distinct state
  – Answer: $O((C \times T) \times C) = O(C^2 \times T)$
5 minutes break
Then, another example of DP on General Graph

TRAVELING SALESMAN PROBLEM
(THE CLASSIC VERSION)
Traveling Salesman Problem (TSP)

The classic TSP is actually “simple” to describe:

• Given a list of \( V \) cities and their \( \sqrt{C_2} \) pairwise distances
  – That is, a complete weighted graph, which is a general graph

• Find a shortest tour that visits each city exactly once
  – Thus the tour will have exactly \( V \) edges, a simple tour
  – The tour must start and end at the same city

Note that this problem is different from UVa 10702 shown earlier in this lecture

• Take some time to examine the differences
The shortest tour for this TSP instance is ...
(you will need some time to compute this)

1. Tour A-B-C-D-A with cost
   ____________________________

2. Tour A-C-B-D-A with cost
   ____________________________
How many possible tours are there in a TSP instance of $V$ cities?

1. $V$ valid tours
2. $V \log V$ valid tours
3. $V^2$ valid tours
4. $V!$ valid tours
What is the value of “10!”?

1. 10
2. 100
3. 3628800
4. 10000000
5. 9.332621544394415e+157
What is the value of “100!”?

1. 10
2. 100
3. 3628800
4. 10000000
5. 9.332621544394415e+157
Brute Force (Naïve) Solution

An $O(V! \times V)$ solution in sketch:

1. Try all $V!$ tour permutations
2. Compute the cost for each tour – doable in $O(V)$
3. Pick the one with the minimum cost...

But this sketch is too coarse for proper implementation

- ‘Simple’ question:
  How to generate $V!$ permutations of $V$ vertices?
Demo: Generating All Permutations

Note: This is usually hard for first timers

The demonstration steps (see TSPslow.java):

1. I start from DFSrec (Lecture 06)
   - DFS is $O(V+E)$, but a complete graph has $V = N$ and $E = N^2$
   - Thus DFS runs in $O(N+N^2) = O(N^2)$

2. I will show how to change DFSrec into a backtracking routine that tries all permutations, while still using the \texttt{visited} flag
   - This is an $O(N!)$ algorithm, as there are $(N-1)!$ possible tours
Conversion to a DAG (1)

To do backtracking in this complete general graph, we have to use the *visited* flag that is turned on when entering the recursion and turned off when exiting the recursion (that is the main difference with DFSrec).

- Backtracking is one example of Complete Search technique.
private void DFSrec(int u) {
    visited[u] = true; // to avoid cycle
    for (int j = 0; j < AdjList.get(u).size(); j++) {
        IntegerPair v = AdjList.get(u).get(j);
        if (!visited[v.first()])
            DFSrec(v.first());
    }
    visited[u] = false; // let vertex u be reusable later
}
Conversion to a DAG (2)

To do backtracking in this complete general graph, we have to use the visited flag that is turned on when entering the recursion and turned off when exiting the recursion (different from DFSrec).

- Backtracking is one example of Complete Search technique.

This essentially converts the complete general graph into a DAG, where each vertex is now has one more parameter: The set of vertices already visited up to the current one, see the next slide for a sample figure.
TSP DAG (1)

(goes back from D to A)

(goes back from C to A)
TSP DAG (2)

The path in red (it is a tour actually) has the minimum total weight of 97.

There are lots of overlapping sub problems on larger instance of TSP DAG.
The path in red (it is a tour actually) has the minimum total weight of 97.

There are lots of overlapping sub problems on larger instance of TSP DAG.
The path in red (it is a tour actually) has the minimum total weight of 97.

There are lots of overlapping sub problems on larger instance of TSP DAG.
Now, how many vertices are there in this DAG?

• $N \times 2^N$
  – $N$ is the number of vertices in the input (complete weighted) graph
  – $2^N$ is the size of the set of visited vertices

Then, how to store this “set of Boolean” effectively?

• We have discussed **bitmask** in Lecture 05
  – A “data structure” for lightweight set of Boolean

http://visualgo.net/bitmask.html
DP Solution (2) – in Pseudo Code

// memo is a 2D table of size N * 2^N

int DP_TSP(u, m) // m = visited mask (we will use bitmask)
    if m is all 1 (in binary) // all vertices have been visited
        return weight(u, 0) // no choice, return to vertex 0
    if memo[u][m] does not equals to -1 // computed before?
        return memo[u][m]
    memo[u][m] = INF // general case
    for each v in V // Q: do we need to check if (u, v) exists?
        if v != u and v is not turned on in bitmask m
            memo[u][m] = min(memo[u][m],
                weight(u, v) + DP_TSP(v, m with v-th bit turned on);
            return memo[u][m]

// note: You will implement this (and something else) in PS6/R
DP Feature @ VisuAlgo

Try [visualgo.net/recursion.html](http://visualgo.net/recursion.html) and load this example: “Traveling Salesman” to get the same DAG, and then you can modify the recursion and/or the test case to *dynamically* generate *new recursion DAG*.
DP Analysis

What is the num of distinct states/space complexity?
• Answer: \( O(N \times 2^N) \)

What is the time to compute one distinct state?
• Answer: \( O(N) \), must check all neighbors of a vertex as the input is a complete graph, each vertex has out-degree \( N-1 \)

What is the overall time complexity?
• Answer: \( O((N \times 2^N) \times N) = O(N^2 \times 2^N) \)

PS: The name of this DP algorithm is Held-Karp’s algorithm
# Larger TSP Instances

<table>
<thead>
<tr>
<th>N</th>
<th>N!</th>
<th>$N^2 \times 2^N$</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>362,880</td>
<td>81 * 512 = 41,472</td>
<td>8.75 x</td>
</tr>
<tr>
<td>10</td>
<td>3,628,800</td>
<td>100 * 1024 = 102,400</td>
<td>35.44 x (PS6 A)</td>
</tr>
<tr>
<td>11</td>
<td>39,916,800</td>
<td>121 * 2048 = 247,808</td>
<td>161.07 x</td>
</tr>
<tr>
<td>12</td>
<td>479,001,600</td>
<td>144 * 4096 = 589,824</td>
<td>812.11 x</td>
</tr>
<tr>
<td>13</td>
<td>6,227,020,800</td>
<td>169 * 8192 = 1,384,448</td>
<td>4,497.84 x</td>
</tr>
<tr>
<td>14</td>
<td>87,178,291,200</td>
<td>196 * 16384 = 3,211,264</td>
<td>27,147.66 x</td>
</tr>
<tr>
<td>15</td>
<td>1,307,674,368,000</td>
<td>225 * 32,768 = 7,372,800</td>
<td>177,364.69 x</td>
</tr>
<tr>
<td>16</td>
<td>20,922,789,888,000</td>
<td>256 * 65,536 = 16,777,216</td>
<td>1,247,095.46 x (PS6 B)</td>
</tr>
<tr>
<td>17</td>
<td>355,687,428,096,000</td>
<td>289 * 131,072 = 37,879,808</td>
<td>9,389,895.22 x</td>
</tr>
<tr>
<td>20 :O</td>
<td>~2.4 * $10^{18}$</td>
<td>400 * 1,048,576 = 419,430,400</td>
<td>PS6R v3 (2013)</td>
</tr>
<tr>
<td>30 :O</td>
<td>~2.6 * $10^{32}$</td>
<td>900 * 1,073,741,824 = 966,367,641,600</td>
<td>PS6R v4 (2014)</td>
</tr>
<tr>
<td>51</td>
<td>Hm...</td>
<td>5,856,931,315,395,330,048 ⌠</td>
<td>PS6R v5 (2015)</td>
</tr>
<tr>
<td>105 ?</td>
<td>My PhD thesis 😊</td>
<td>4.47227131760519332848036e+35 ⌠</td>
<td>TSP is an NP-hard problem</td>
</tr>
</tbody>
</table>

TSP is an NP-hard problem.
Your Guess on the State-of-the-Art TSP Solver

1. No one has solved TSP instances beyond ~20 vertices
2. I think someone has solved TSP instances up to... 100 vertices?
3. Maybe up to 1000?
4. Hm... 10000?
How to solve this?

http://www.tsp.gatech.edu/world/images/anim1a.html
Summary (1)

By definition, a general graph has cycle(s)

- We cannot write a recursive formula in a cyclic structure...
  - Therefore we cannot use DP technique on general graph :O...
Summary (2)

We have seen how to convert *some* general graphs into DAGs by introducing one extra parameter

- Now we can write recursive formulas and use DP technique
- Analysis of the space and time complexity of *basic* DP is *easy*
- Note: Some of these problems are better solved as top-down DP (written as recursive functions), i.e. we do not explicitly build the implicit DAG...
  - Note that harder DP problems may require us to introduce *more than one* extra parameters...