# CS375 Notes: Dynamic Programming

Dynamic Programming on Trees and Graphs

---

## 1 Maximal Value Independent Set

### 1.1 Review: Independent Set on an Array

Given an array of values $x_1, x_2, \ldots, x_n$, we search for the largest value subset of elements without selecting adjacent indices.

**Subproblem Definition:**

$$\text{OPT}(i) = \text{the best value achievable over the first } i \text{ indices.}$$

**Recurrence Relation:**

$$\text{OPT}(i) = \max \begin{cases} x_i + \text{OPT}(i-2) & \text{// include } x_i \\ \text{OPT}(i-1) & \text{// do not include } x_i \end{cases}$$

### 1.2 Independent Set on a Tree

Now consider a rooted tree $T = (V, E)$ where each vertex $v \in V$ has a value $\text{val}(v)$. We want to find the maximal independent set on the tree without selecting adjacent vertices (vertices sharing an edge).

**Subproblem Definition:**

$$\text{OPT}(v) = \text{the best value achievable on the subtree } T_v \text{ rooted at vertex } v.$$

**Recurrence Relation:**

$$\text{OPT}(v) = \max \begin{cases} \text{val}(v) + \displaystyle\sum_{u \in GK(v)} \text{OPT}(u) & \text{// include } v \\ \displaystyle\sum_{u \in K(v)} \text{OPT}(u) & \text{// do not include } v \end{cases}$$

where:

- $K(v)$ is the set of all kids of $v$.
- $GK(v)$ is the set of all grandkids of $v$.

**Base Cases & Evaluation Order:** Computation begins by calculating OPT at the leaves of the tree. Because a leaf has no children or grandchildren ($K(\text{leaf}) = \emptyset, GK(\text{leaf}) = \emptyset$), the base case is:

$$\text{OPT}(\text{leaf}) = \text{val}(\text{leaf})$$

Evaluating bottom-up from leaves to the root yields the optimal solution at OPT(root).

# 2 Dynamic Programming on General Graphs

Finding the maximum value independent set on a general graph is significantly harder. The tree-based subproblem decomposition fails because arbitrary graphs contain cycles and cross-edges, preventing a clean separation into independent subproblems.

*Note: Maximum Independent Set on a general graph is NP-hard.*

## 2.1 Traveling Salesman Problem

**Input:** A weighted graph $G = (V, E)$ with distances $d(u, v)$ on the edges.
**Output:** A path that visits every vertex in the graph exactly once and minimizes the total distance.

## 2.2 Brute Force Approach

Checking all possible vertex orderings requires evaluating all permutations:

$$\text{Time Complexity: } \mathcal{O}(n \cdot n!)$$

This quickly becomes VERY computationally expensive as $n$ grows.

## 2.3 Dynamic Programming Approach

To optimize, we define subproblems based on subsets of visited vertices and the current terminating vertex.

**Subproblem Definition:**

$$\text{OPT}(S, v) = \text{shortest path length visiting every vertex in subset } S \subseteq V \text{ ending at } v \in S.$$

**Recurrence Relation:** To reach state $(S, v)$, we consider which vertex $u \in S \setminus \{v\}$ was visited immediately before $v$:

$$\text{OPT}(S, v) = \min_{u \in S \setminus \{v\}} \{d(u, v) + \text{OPT}(S \setminus \{v\}, u)\}$$

**Base Case:** For single-element sets covering 0 edge distance:

$$\text{OPT}(\{v\}, v) = 0 \quad \forall v \in V$$

## 2.4 Algorithmic Implementation & Pseudocode

```
// Data structure to store subproblem values
HashMap<(Set<Vertex>, Vertex), Double> map = new HashMap<>();

// Base cases: subsets of size 1 -- O(n)
for each vertex v in V {
    map.put(({v}, v), 0);
}

// Compute OPT for increasing subset sizes
for i = 2 ... n {
```

```
    for each subset S of V of size i {
        for each v in S {
            double minDist = Double.POSITIVE_INFINITY;
            for each u in S \ {v} {
                minDist = min(minDist, d(u, v) + map.get(S \ {v}, u));
            }
            map.put((S, v), minDist);
        }
    }
}
```

### 2.5 Complexity Analysis

**Time Complexity:** To determine the overall runtime intuitively, we analyze the nested loops over all subproblems:

- **Subsets of $V$:** The algorithm considers all $2^n$ subsets $S \subseteq V$.
- **States per Subset:** For a subset of size $k = |S|$, there are $k$ choices for the destination vertex $v$.
- **Transitions per State:** Evaluating $\text{OPT}(S, v)$ iterates over $k - 1$ potential predecessors $u \in S \setminus \{v\}$.

For any given subset $S$ of size $k$, the two innermost loops perform $k(k-1) \approx \Theta(k^2)$ total state updates.

Rather than evaluating exact summations, we observe the average case across all iterations:

- Across all $2^n$ subsets, the average subset size is $k \approx \frac{n}{2}$.
- Evaluating an average-sized subset takes $\left(\frac{n}{2}\right)^2 = \Theta(n^2)$ work.

Multiplying the total number of subsets ($2^n$) by the work required for a typical subset ($\Theta(n^2)$) yields:

$$\text{Total Time Complexity} = \Theta(n^2 2^n)$$

**Summary:** Although exponential, $\Theta(n^2 2^n)$ the dynamic programming approach is a significant optimization over the $\mathcal{O}(n \cdot n!)$ brute-force approach.