

Dynamic Programming

Largest Path in DAG

Input: $G(V, E)$. DAG Output: $l = \text{longest path in } G$.
 Subproblem: $L(v) = \text{length of longest path ending in } v$
 Recurrence Relation: $L(v) = \begin{cases} l+1 & \text{if } v \in V \\ \max_{u \in N^-(v)} L(u) & \text{if } v \text{ has no incoming edges} \end{cases}$
 Algo. (Runtime $O(|V| + |E|)$)

- Sort in topological order - i is the i th vertex in s.t. $\forall (i, j) \in E, i > j$ topological ordering
- For all i , set $L(i) = 0$
- For all $i=1 \dots n$, set $L(i) = \begin{cases} l+1 & \max_{u \in N^-(i)} L(u) \\ 0 & \text{if } i \text{ has no incoming edges} \end{cases}$

Longest Increasing Sequences

Input: a_1, a_2, \dots, a_n Output: $l = \text{length of longest increasing sequence}$
 \Rightarrow Reduces to finding longest path in a graph.

$$G = (V, E) \quad V = \{1, 2, \dots, n\}, E = \{(i, j) \mid i < j, a_i < a_j\}$$

Approach: 1) Define appropriate subproblems
 2) Write a recurrence relation
 3) Determine order of computation
 \sqsubseteq induces a DAG structure.

Edit Distance

Input: $x[1, 2, \dots, n], y[1, 2, \dots, n]$ (add, remove, substitute)
 Goal: find minimum # of keystroke needed to edit x to y
 1) Define subproblems: $E(i, j) = E(x[1 \dots i], y[1 \dots j])$
 $= \text{Edit distance to edit } x[1 \dots i] \text{ to } y[1 \dots j]$
 2) Recurrence relation: $E(i, j) = \min \begin{cases} 1 + E(i-1, j) \\ 1 + E(i, j-1) \\ \text{diff}(x[i], y[j]) + E(i-1, j-1) \end{cases}$
 Base Case: $E(0, 0) = 0$
 $y_i, E(0, j) = E(j, 0) = j$

Algo.
 Init for all $i=1 \dots n, E(i, 0) = i$ Runtime $O(nm)$
 For all $j=1 \dots m, E(0, j) = j$
 For all $i=1 \dots n$
 For all $j=1 \dots m, E(i, j) = \min \begin{cases} 1 + E(i-1, j) \\ 1 + E(i, j-1) \\ \text{diff}(x[i], y[j]) + E(i-1, j-1) \end{cases}$

Knapsack Problem

Input: total weight capacity of a knapsack W
 list of n items with weights w_1, w_2, \dots, w_n
 values v_1, v_2, \dots, v_n
 Goal: Find set of items that will maximize total value with total weight $\leq W$
 $k(C) = \text{max value that can be packed in a bag of capacity } C$
 $k(C) = \max \{v_i + k(C-w_i)\}$. Base case: $k(0) = 0$ if $C=0$

Algo.
 Input: $W, v[1, \dots, n], w[1, \dots, n]$ Runtime $O(nW)$
 $k(0) = 0$
 For $i=1 \dots n$ exponential in $\log n$
 $k(W) = \max \{v_i + k(C-w_i)\}$,
 $i \leq n$
 Output: $k(W)$

Knapsack without repetition

$k(C, j)$: Maximum value that can be carried in a bag of capacity C using items $1 \dots j$
 $k(C, j) = \begin{cases} +k(C, j-1) & \text{if } C < w_j \\ \max \{k(C, j-1), v_j + k(C-w_j, j-1)\} & \text{if } C \geq w_j \end{cases}$
 Base Case: $k(0, j) = 0, v_j$ runtime $O(nW)$

Algo.
 Input: $W, v[1 \dots n], w[1 \dots n]$
 For $C=0 \dots W$
 $k(C, 0) = 0$
 For $i=1 \dots n$
 For $C=1 \dots W$
 $k(C, i) = \max \{k(C, i-1), v_i + k(C-w_i, i-1)\}$
 Output: $k(W, n)$

Chain Matrix Multiplication

Input: $A_1, \dots, A_m, A_1 \times \dots \times A_m, A_1 \times \dots \times A_m \dots A_m$
 Output: Minimum number of multiplications needed.

$$(A \times B)_{ij} = \sum_{k=1}^m A_{ik} B_{kj} \quad (A_1 \times \dots \times A_m) \times (B_1 \times \dots \times B_m)$$

Subproblem: $M(i, j) = \text{the minimum number of multiplications needed to multiply } A_i \times A_{i+1} \dots A_j$
 $M(i, n)$ is what we want

Recurrence Relation:

$$M(i, j) = \min_{1 \leq k \leq j-1} \{ M(i, k) + M(k+1, j) + m_{i,k} m_k m_j \}$$

Base Case: $M(i, i) = 0$ for all $i \in \{1, 2, \dots, n\}$.

Order of Computation

Runtime $O(n^3 \cdot n) = O(n^3)$

Algo.
 For $i=1 \dots n, M(i, i) = 0$ Runtime $O(n^3) = O(n^3)$
 For $s=1 \dots n-1$
 For $i=1 \dots n-s$
 $j=i+s$
 $M(i, j) = \min_{k=i \dots j-1} \{ M(i, k) + M(k+1, j) + m_{i,k} m_k m_j \}$
 Return $M(1, n)$

Subproblem Structure

(i) input $x_1 \dots x_n$ and a subproblem is $x_1 \dots x_i$

(ii) input $x_1 \dots x_n \& y_1 \dots y_m$ and a subprob is $x_1 \dots x_i$
 $y_1 \dots y_j$

(iii) input $x_1 \dots x_n$ and a subproblem is $x_1 \dots x_j$
 $x_1 \dots x_i \dots x_n$

SSSP (Single Source Shortest Path)

Input: Weighted graph $G = (U, E)$. We for edge e (WEIR) source s

Output: For every $v, \text{dist}(v) = \text{shortest path from } s \text{ to } v$
 ① $\text{dist}(v)$ for $v \in V$
 ② $\text{dist}(v) = \min \{ \text{dist}(s, u) + \text{w}_{uv} \}$ $(u, v) \in E$
 ③ Need to redefine the subproblems.
 ④ $\text{dist}(v, k) = \text{distance of shortest path from } s \text{ to } v \text{ involving up to } k \text{ edges}$
 Base case:
 ⑤ $\text{dist}(v, k) \leftarrow \text{dist}(s, k-1)$
 ⑥ $\text{dist}(s, 0) = 0$
 ⑦ $\text{dist}(v, 0) = \infty \forall v \in V$
 ⑧ $\text{dist}(v, k) = \min \{ \text{dist}(v, k-1), \min_{u \in N^-(v)} \{ \text{dist}(u, k-1) + \text{w}_{uv} \} \}$
 ⑨ Nice order to compute $k=1, k=2, \dots$

Algo.

For $v \in V, \text{dist}(v, 0) = \infty$
 $\text{dist}(s, 0) = 0$
 For all $k=1 \dots n-1$ $n(n+m) \rightarrow O(nm)$
 For all $v \in V$
 $\text{dist}(v, k) = \text{dist}(v, k-1)$
 For all $(u, v) \in E$
 $\text{if } \text{dist}(v, k) > \text{dist}(u, k-1) + \text{w}_{uv}$
 $\text{then } \text{dist}(v, k) = \text{dist}(u, k-1) + \text{w}_{uv}$
 Output $\forall v \in V, \text{dist}(v, n-1)$

All Pairs Shortest Path

Input: Weighted graph $G = (U, E)$. We for edge e (WEIR)

Output: For every $u, v, \text{dist}(u, v) = \text{shortest path from } u \text{ to } v$
 ① Run BF n times to get u to v paths for all u $O(nm)$
 ② $\text{dist}(u, v, k) \rightarrow$ shortest path from u to v with atmost k edges $O(nm)$
 ③ $\text{dist}(u, v, k) \rightarrow$ shortest path from u to v that takes vertex in $\{1, \dots, k\}$ only.

claim: On the shortest path from u to v , not vertex w happens twice.

$$\text{dist}(u, v, k) = \min \{ \text{dist}(u, v, k-1), \text{dist}(u, k, k-1) + \text{dist}(k, v, k-1) \}$$

Algo.

For $i, j=1 \dots n, \text{dist}(i, j, 0) = \infty$
 For $(i, j) \in E, \text{dist}(i, j, 0) = w_{ij}$ Runtime $O(n^3)$
 For $i=1 \dots n, \text{dist}(i, i, 0) = 0$
 For $k=1 \dots n$
 For $i=1 \dots n$
 $\text{for } j=1 \dots n$
 $\text{dist}(i, v, k) = \min \{ \text{dist}(i, v, k-1), \text{dist}(i, k, k-1) + \text{dist}(k, v, k-1) \}$
 Output $\text{dist}(i, j, k)$ for all $i, j \in V$

Travelling Salesman Problem

Input: n cities & distances $d(i, j)$

Goal: Find minimum distance path from city 1 back to city 1 visiting each city exactly once.

$C(S, j) \rightarrow$ the least cost path that ① starts at 1
 ② visits all nodes in set S ③ ends in node j

base case: $|S|=2, v \neq 1, C(S, j) = d_{1,j}$ $(S=\{1, j\})$

For $|S| > 2, C(S, j) = \min_{i \in S \setminus \{1, j\}} \{ C(S \setminus \{i\}, i) + d_{i,j} \}$

Algo.

$C(\{1\}, 1) = 0$
 for $s=2 \dots n$
 for all subsets $S \subseteq \{1 \dots n\}$ of size s and containing 1:
 $C(S, 1) = \infty$
 for all $j \in S, j \neq 1$
 $C(S, j) = \min_{i \in S \setminus \{j\}} \{ C(S \setminus \{i\}, i) + d_{i,j} \}$. Runtime
 return $\min_j \{ C(\{1 \dots n\}, j) + d_{1,j} \}$. $2^n \times n \rightarrow O(2^n n^2)$

Max Flow

Definition (s-t flow): An s-t flow is an assignment $f: E \rightarrow \mathbb{R}^+$ such that

1) Capacity Constraint: For each edge e , flow on $e \in$ capacity of $e: f_e \leq c_e$

2) Conservation Constraint: For each vertex $v \neq s/t$. Flow coming into v = Flow leaving v : $\sum_{u \rightarrow v} f_{u \rightarrow v} = \sum_{v \rightarrow w} f_{v \rightarrow w}$

Algorithm for Max Flow

REPEAT:

- * FIND A PATH P from s to t with non-zero capacity in Residual graph
- [Terminate if NO PATH P exists]
- * Add flow along P to the current flow.

Residual Graph



Definition (Capacity of an s-t cut):

Capacity $(L, R) =$ Total capacity of edges from L to $R = \sum_{u \in L, v \in R} c_{u \rightarrow v}$
 $\cup L, \cup R$

CLAIM: For every $s-t$ cut $L|R$ and for every $s-t$ flow f .

$$\text{Size}(f) \leq \text{CAPACITY}(L|R)$$

\Rightarrow Maximum Flow \leq Minimum Cut

Theorem: In any graph G ,

Maximum flow $s-t = \text{Minimum } s-t \text{ cut}$

NP

Satisfiability or SAT

Input: a formula ϕ over variables x_1, x_2, \dots, x_n

Solution: find a satisfying assignment or output no
satisfying assignment exists.

Example: $\phi = (x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \bar{x}_2) \wedge (\bar{x}_2 \vee x_3) \wedge (\bar{x}_2 \vee \bar{x}_3)$
try all possible assignments $\rightarrow 2^n$

VERIFY (Instance ϕ , Solution $F: \{x_1, \dots, x_n\} \rightarrow \{T, F\}$) \rightarrow $\exists x \in \{T, F\}^n$ such that $\phi(F(x)) = T$

P & NP

P: Class of search problems where we can find a solution in polynomial in $|I|$ time.

HornSAT $\in P$. $\geq 2\text{SAT} \in P$

NP: Class of all search problems, i.e. the class of all problems where we can verify a solution in polynomial time. SHENP. 2SATNP. PSNP

12th SS:

- ① zero-sum game
- ② bipartite matching. (removing $\Theta(n^3)$)
- ③ Dynamic Programming
- ④ Horn Formula.

Common shifts

