IE 511: Integer Programming, Spring 2021 **30 Mar, 2021**

Lecture 19: Solving technique 3: Dynamic Programming

Lecturer: Karthik Chandrasekaran *Scribe: Karthik*

Disclaimer: *These notes have not been subjected to the usual scrutiny reserved for formal publications.*

We will see another useful solving technique for integer programs/discrete optimization problems. This is the dynamic programming (DP) technique. The technique was introduced by Bellman in 1957. Once again, we will illustrate the technique through applications. We will also formulate IPs for these applications before using the DP technique to design algorithms.

19.1 Application 1: Shortest Path Problem

The shortest path problem arises frequently in travel-related contexts (e.g., travel websites and GPS maps).

Given: A directed graph $G = (V, A)$ (where $n := |V|$, $m := |A|$), arc lengths $l : A \to \mathbb{R}_+$, origin vertex s, destination vertex t

Goal: $\min\{\sum_{e \in P} \ell_e : P \text{ is a } s \to t \text{ path in } G\}$

Example: See Figure 19.1.

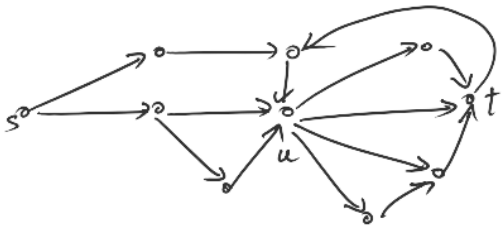


Figure 19.1: A shortest path problem instance with $l(e) = 1\ \forall e \in A$

We will assume that s has no incoming edges and t has no outgoing edges in the given instance (a shortest path will not be using such edges, so we can delete such edges from the given instance).

BIP Formulation. The following BIP formulates the shortest path problem (x_e indicates if arc e is chosen or not and the constraints ensure that exactly one outgoing arc from the origin vertex s is chosen, exactly one incoming arc to the destination vertex t is chosen, and for every other node, we choose the same number of incoming and outgoing arcs—since the arc lengths are non-negative, the chosen arcs in an optimal solution will necessarily form a path).

$$
\begin{aligned}
&\min \sum_{e \in A} l_e x_e \\
&\sum_{e \in \delta^{\text{out}}(s)} x_e - \sum_{e \in \delta^{\text{in}}(s)} x_e = 1 \\
&\sum_{e \in \delta^{\text{in}}(t)} x_e - \sum_{e \in \delta^{\text{out}}(t)} x_e = 1 \\
&\sum_{e \in \delta^{\text{out}}(u)} x_e - \sum_{e \in \delta^{\text{in}}(u)} x_e = 0 \quad \forall u \in V \setminus \{s, t\} \\
&x_e \in \{0, 1\} \quad \forall e \in A
\end{aligned}
\tag{19.1}
$$

Exercise. Show that the LP-relaxation of the BIP (19.1) has an integral optimal solution.

We will see an algorithm to solve this BIP.

Outline. Consider the Example in Figure 19.1. A shortest path P from s to t passes through u. The sub-path of P from s to u should be a shortest s to u path (otherwise, we can replace it by a shorter s to t path to obtain a shorter s to t path). Similarly, the sub-path of P from u to t should be a shortest u to t path. This leads to the following observation:

Observation. If the shortest path P from s to t passes thorough a vertex u, then the sub-path of P from s to u and the sub-path of P from u to t should be a shortest $s \to u$ and a shortest $u \to t$ paths respectively.

This means that any sub-path of a shortest path must be a shortest path itself. This is known as the principle of optimality. If we apply the principle to all predecessors u of t, then we get a recurrence for the shortest path distance.

Let $d(u) :=$ length of a shortest s to u path. Then we need to find $d(t)$.

A relationship based on observation: $d(u) = \min_{v:\overrightarrow{vu} \in A} d(v) + l(\overrightarrow{vu})$.

If we know the shortest path distance $d(v)$ from s to all predecessors v of u, then we can easily compute the shortest path distance from s to u. This approach to compute the shortest path distances has the following issue: In order to compute $d(v)$, we may need $d(u)$ if u is a predecessor of u.

Let us first consider special directed graphs where this issue does not arise.

Definition 1. A directed graph $G = (V, A)$ is a *directed acyclic graph (DAG)* if the vertices can be ordered as $1, \ldots, n$ such that $i < j$ for every arc $\overrightarrow{ij} \in A$.

Example: See Figure 19.2.

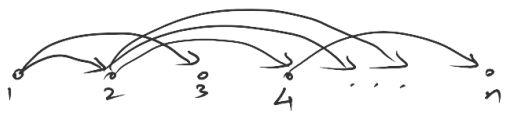


Figure 19.2: A DAG

Given such an ordering with $s = 1$, we can compute shortest path distances to all vertices by processing the vertices in increasing order—in order to compute $d(j)$, we only need the values $d(i)$ for $i < j$ and these values would have already been computed. Thus, we have the following theorem:

Theorem 2. *Shortest path problem in DAGs can be solved in $O(m)$ time.*

The runtime is because each edge is examined at most once.

Note that we built the solution value in DAGs recursively using the optimal values of slightly different subproblems. Let us now see how to extend this technique to solve the shortest path problem in general digraphs. The relationship for $d(u)$ derived above is not directly useful since we do not have an ordering of the vertices in general digraphs; to overcome this issue, we will modify the sub-problems which will allow us to enforce an ordering.

Let $d_k(v)$ be the length of a shortest path from s to v using *at most* k arcs. We need to find $d_n(t)$. Let P be a shortest s to t path using at most k arcs. We have exactly one of two possibilities for P:

1. Either P uses at most $k-1$ arcs, in which case, P most be an optimum solution for $d_{k-1}(v)$,
2. Or P uses k arcs, in which case, it should pass thorough a vertex u within $k-1$ arcs and use the edge $\overrightarrow{uv}$ to reach v. It means that the sub-path of P from s to t should be an optimum solution for $d_{k-1}(u)$ and $\overrightarrow{uv}$ must be such that $d_{k-1}(u) + l(\overrightarrow{uv}) = \sum_{e \in P} l_e$.

Therefore, we have the following recursion:

Recursion: $d_k(v) = \min\{d_{k-1}(v), \min_{u:\overrightarrow{uv} \in A}\{d_{k-1}(u) + l(\overrightarrow{uv})\}$.

This recursion can be used to compute all shortest path distances.

Algorithm 1: Algorithm to find a shortest path

Input: A graph $G = (V, A)$ ($n := |V|$, $m := |A|$), arc length $l : A \to \mathbb{R}_+$, origin vertex s, destination vertex t

Output: A shortest path from s to t

Initialize $d_1(v) := \begin{cases} l(\overrightarrow{sv}) & \text{if } \overrightarrow{sv} \in A \\ \infty & \text{otherwise} \end{cases}$ $\forall v \in V$; `// takes` $O(m)$ `time`

for $k = 2, 3, \ldots, n-1$ **do**

  use $d_{k-1}(\cdot)$ values to compute $d_k(\cdot)$ values via the recursion relationship mentioned above ;

  `// takes` $O(m)$ `time because each arc is examined exactly once.`

We have already argued the correctness of Algorithm 1 by proving the recursion. The run-time follows since each for-loop takes $O(m)$ time. Thus, we have shown the following result:

Theorem 3. *Shortest path problem can be solved in $O(nm)$ time.*

Note that Algorithm 1 finds shortest path distance from s to every vertex in the graph.

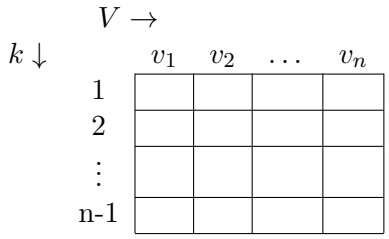
19.2 DP: General Ideas.

The significant concepts underlying the DP technique are:

1. *States*: Subproblems are known as states.
2. *Principle of optimality*: Compute an optimal solution value from solution values of subproblems.
3. *Stages*: Ordering in which we solve the subproblems is known as stages.

The non-trivial aspect of using the dynamic programming technique is the definition of states. One needs to come up with careful definition of states so that the principle of optimality holds; the principle of optimality allows us to derive a recursion; the recursion is useful to solve the subproblems/states by proceeding in stages.

It is helpful to visualize a DP algorithm as filling the entries of a table which is indexed by the states. In each stage, we compute one table entry from the other table entries which have been computed earlier (see figure below for an illustration of the table associated with solving the shortest path problem).



19.3 Application 2: Knapsack Problem

The knapsack problem is a quintessential IP that involves only one constraint. It frequently arises in investment applications.

Given: Knapsack capacity $b \in \mathbb{Z}_+$, n items, weights $a_1, \ldots, a_n \in \mathbb{Z}_+$, profits $c_1, \ldots, c_n$
Goal: $\max\{\sum_{i \in S} c_i : S \subseteq [n], \sum_{i \in S} a_i \leq b\}$

BIP. We have already seen a BIP formulation:

$$\max \left\{ \sum_{i=1}^{n} c_i x_i : \sum_{i=1}^{n} a_i x_i \leq b, x \in \{0,1\}^n \right\}.$$

We will see a DP-based algorithm to solve this BIP that runs in time $O(nb)$.

States (subproblems):

Let $P_k(\lambda) :=$ max profit achievable using the first k items when the capacity of the knapsack is λ, i.e., $P_k(\lambda) = \max\left\{\sum_{i=1}^{k} c_i x_i : \sum_{i=1}^{k} a_i x_i \leq \lambda, x \in \{0,1\}^k\right\}$. We need to find $P_n(b)$.

Deriving the Recursion for $P_k(b)$:

We first consider the initialization step. Suppose $k = 1 : P_1(\lambda) = \begin{cases} c_1 & \text{if } a_1 \leq \lambda \\ 0 & \text{otherwise} \end{cases}$

We now derive the recursion. Suppose $k \geq 2$: Let x^* be an optimum solution for $P_k(\lambda)$. We have exactly one of two possibilities:

1. Either x^* does not use item k, in which case, x^* must be an optimum solution for $P_{k-1}(\lambda)$,
2. Or x^* uses item k, in which case, x^* uses weight at most $\lambda - a_k$ for items $1, \ldots, k-1$ and hence $(x_1^*, \ldots, x_{k-1}^*)$ should be an optimal solution for $P_{k-1}(\lambda - a_k)$.

This leads to the following recurrence:

Recursion: $P_k(\lambda) = \max\{P_{k-1}(\lambda), P_{k-1}(\lambda - a_k) + c_k\}$.

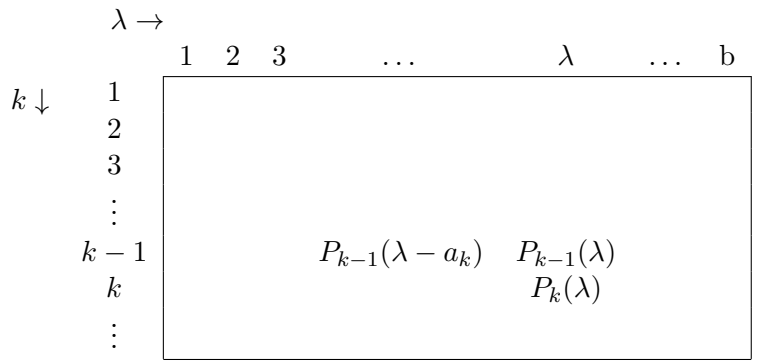
If we know the values of $P_{k-1}(r)$ for every $r \in \{1, \ldots, \lambda\}$, then we can compute $P_k(\lambda)$ by inspecting just two values. The following algorithm exploits this:

Algorithm 2: Algorithm to solve the knapsack problem

Input: Knapsack capacity $b \in \mathbb{Z}_+$, n items, weights $a_1, \ldots, a_n \in \mathbb{Z}_+$, profits $c_1, \ldots, c_n$
Output: $\max\{\sum_{i \in S} c_i : S \subseteq [n], \sum_{i \in S} a_i \leq b\}$

Initialize $P_1(\lambda) = \begin{cases} c_1 & \text{if } \lambda \geq a_1 \\ 0 & \text{otherwise} \end{cases}$;
for $k = 2, \ldots, n$ **do**
  for $\lambda = 1, \ldots, b$ **do**
    Compute $P_k(\lambda)$ using the recursion.

Note that we can visualize the algorithm as filling the entries of the following table.



We have already argued the correctness of Algorithm 2 by proving the recursion. The run-time follows since we are filling nb entries in the table and each cell can be filled in constant time. Thus, we have shown the following result:

Theorem 4. *The knapsack problem can be solved in time $O(nb)$ time.*

Note that Algorithm 2 is not a polynomial-time algorithm since the input size is only

$$O(n \log a_{\max} + n \log c_{\max} + \log b).$$

19.4 Application 3: Maximum Weight Subtree Problem

Given: A tree $T = (V, E)$ rooted at $r \in V$, vertex weights $p : V \to \mathbb{R}$ (could be negative)
Goal: Find a subtree of T rooted at r with maximum weight.

Example: See Figure 19.3.

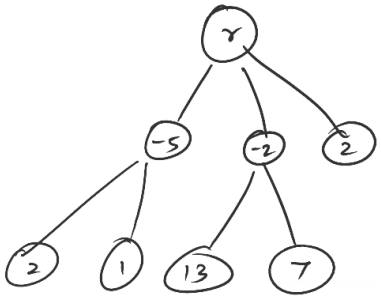


Figure 19.3: An instance of the optimal subtree problem with vertex weights specified at vertices.

Motivation. The problem is motivated from the perspective of service providers who would like to decide which customers should be connected: The root is the service provider while the vertices correspond to customers and the value $p(v)$ is the immediate profit from connecting a customer to the service provider.

Exercise. Formulate an IP for this problem.

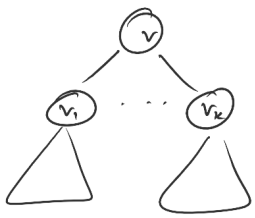
We will see an algorithm to solve this problem that is based on DP. The DP algorithm that we will see is a standard technique when encountering discrete optimization problems over trees.

States: Let $g(u) :=$ maximum weight of a subtree rooted at u. We need to find $g(r)$.

Deriving the recursion for $g(u)$: The recursion is typical for DP algorithm on trees. Compute $g(u)$ bottom up starting from the leaves.

If u is a leaf, then $g(u) = \max\{0, p(u)\}$. Suppose u has children $v_1, \ldots, v_k$. Let T' be an optimal subtree rooted at u. We have exactly one of two possibilities for T':

1. Either T' is empty,
2. Or T' is composed of subtrees rooted at v_is in which case any subtree rooted at v_i included in T' should be a max-weight subtree rooted at v_i (principle of optimality).



Therefore, we have the following recurrence:

Recursion: $g(u) = \max\left\{0, p(v) + \sum_{i=1}^{k} g(v_i)\right\}$.

If we know the values of $g(v)$ for every child of u, then we can compute $g(u)$using this recursion. This leads to the following theorem:

Theorem 5. *Optimal subtree problem can be solved in $O(|V|)$ time.*

Proof. Use the recursion to compute $g(u)$ bottom up (from leaves to the root). Time to compute $g(u)$ is $O(|\text{children}(u)|)$. Total time is $O(\sum_{v \in V} |\text{children}(u)|) = O(|V|)$. □