1 Shortest paths (continued)

1.1 Applications of Shortest/Longest Path (continued)

**Project Management - Critical Path Method (CPM)**

See the handout titled *Project Formulation for Designer Gene’s Relocation*.

This is a typical project management problem. The objective of the problem is to find the earliest time by which the project can be completed.

This problem can be formulated as a longest path problem. The reason we formulate the problem as a *longest* path problem, and not as a *shortest* path problem (even though we want to find the *earliest* finish time) is that all paths in the network have to be traversed in order for the project to be completed. So the longest one will be the “bottleneck” determining the completion time of the project.

We formulate the problem as a longest path problem as follows. We create a node for each task to be performed. We have an arc from node $i$ to node $j$ if activity $i$ must precede activity $j$. We set the length of arc $(i, j)$ equal to the duration of activity $i$. Finally we add two special nodes *start* and *finish* to represent the *start* and the *finish* activities (these activities with duration 0). Note that in this graph we cannot have cycles, thus we can solve the problem as a longest path problem in an acyclic network.

Now we give some notation that we will use when solving this type of problems. Each node is labeled as a triplet (node name, time in node, earliest start time). Earliest start time of a node is the earliest time that work can be started on that node. Hence, the objective can also be written as the earliest start time of the finish node.

Then we can solve the problem with our dynamic programming algorithm. That is, we traverse the graph in topological order and assign the label of each node according to our dynamic programming equation:

$$t_j = \max_{(i,j)} \{t_i + c_{ij}\},$$

where $t_i$ is the earliest start time of activity $i$; and $c_{ij}$ is the duration of activity $i$.

To recover the longest path, we start from node $t_{\text{finish}}$, and work backwards keeping track of the preceding activity $i$ such that $t_j = t_i + c_{ij}$. Notice that in general more than one activity might satisfy such equation, and thus we may have several longest paths.

Alternatively, we can formulate the longest path as follows:

$$\max t_{\text{finish}}$$

$$t_j \geq t_i + c_{ij} \quad \forall(i,j) \in A$$

$$t_{\text{start}} = 0$$

From the solution of the linear program above we can identify the longest path as follows. The constraints corresponding to arcs on the longest path will be satisfied with equality.
Furthermore, for each unit increase in the length of these arcs our objective value will increase also one unit. Therefore, the dual variables of these constraints will be equal to one (while all other dual variables will be equal to zero—by complementary slackness).

The longest path (also known as critical path) is shown in Figure 1 with thick lines. This path is called critical because any delay in a task along this path delays the whole project path.

Figure 1: Project management example

**Binary knapsack problem**

**Problem:** Given a set of items, each with a weight and value (cost), determine the subset of items with maximum total weight and total cost less than a given budget.

The binary knapsack problem’s mathematical programming formulation is as follows.

\[
\text{max } \sum_{j=1}^{n} w_j x_j \\
\sum_{j=1}^{n} v_j x_j \leq B \\
x_j \in \{0, 1\}
\]

The binary knapsack problem is a hard problem. However it can be solved as a longest path problem on a DAG as follows.

Let \( f_i(q) \) be the maximum weight achievable when considering the first \( i \) items and a budget of \( q \), that is,

\[
f_i(q) = \text{max } \sum_{j=1}^{i} w_j x_j \\
\sum_{j=1}^{i} v_j x_j \leq q \\
x_j \in \{0, 1\}
\]
Note that we are interested in finding $f_n(B)$. Additionally, note that the $f_i(q)$’s are related with the following dynamic programming equation:

$$f_{i+1}(q) = \max\{f_i(q), f_i(q - v_{i+1}) + w_{i+1}\}.$$ 

The above equation can be interpreted as follows. By the principle of optimality, the maximum weight I can obtain, when considering the first $i + 1$ items will be achieved by either:

1. Not including the $i + 1$ item in my selection. In which case my total weight will be equal to $f_i(q)$, i.e. equal to the maximum weight achievable with a budget of $q$ when only considering only the first $i$ items; or

2. Including the $i + 1$ item in my selection. In which case my total weight will be equal to $f_i(q - v_{i+1}) + w_{i+1}$, i.e. equal to the maximum weight achievable when considering the first $i$ items with a budget of $q - v_{i+1}$ plus the weight of this $i + 1$ object.

We also have the the boundary conditions:

$$f_1(q) = \begin{cases} 
0 & \text{if } 0 \leq q < v_1 \\
 w_1 & \text{if } B \geq q \geq v_1.
\end{cases}$$

The graph associated with the knapsack problem with 4 items, $w = (6, 8, 4, 5)$, $v = (2, 3, 4, 4)$, and $B=12$ is given in Figure 2. (The figure was adapted from: Trick M., A dynamic programming approach for consistency and propagation of knapsack constraints.)

![Figure 2: Graph formulation of a binary knapsack problem.](image)

Note that in general a knapsack graph will have $O(nB)$ nodes and $O(nB)$ edges (each node has at most two incoming arcs), it follows that we can solve the binary knapsack problem in time $O(nB)$.

**Remark** The above running time is not polynomial in the length of the input. This is true because the number $B$ is given using only $\log B$ bits. Therefore the size of the input for the knapsack problem is $O(n \log B + n \log W)$ (where $W$ is the maximum $w_i$). Finally, since, $B = 2^{\log B}$, then a running time of $O(nB)$ is really exponential in the size of the input.
1.2 Dijkstra’s Algorithm for the Single Source Shortest Path Problem for Graphs with Nonnegative Edge Weights

Dijkstra’s algorithm is a special purpose algorithm for finding shortest paths from a single source, \( s \), to all other nodes in a graph with non-negative edge weights.

In words, Dijkstra’s algorithm assigns distance labels (from node \( s \)) to all other nodes in the graph. Node labels are either *temporary* or *permanent*. Initially all nodes have temporary labels. At any iteration, a node with the least distance label is marked as permanent, and the distances to its successors are updated. This is continued until no temporary nodes are left in the graph.

We give the pseudocode for Dijkstra’s algorithm below. Where \( P \) is the set of nodes with permanent labels, and \( T \) of temporarily labeled nodes.

```plaintext
begin
    \( N^+(i) := \{j|(i,j) \in A\} \);
    P := \{1\}; T := V \setminus \{1\};
    d(1) := 0 and pred(1) := 0;
    d(j) := c_{1j} and pred(j) := 1 for all \( j \in A(1) \),
    and d(\( j \)) := \infty otherwise;
    while \( P \neq V \) do
        begin
            (Node selection, also called FINDMIN)
            let \( i \in T \) be a node for which \( d(i) = \min\{d(j) : j \in T\} \);
            P := P \cup \{i\}; T := T \setminus \{i\};
            (Distance update)
            for each \( j \in N^+(i) \) do
                if \( d(j) > d(i) + c_{ij} \) then
                    d(\( j \)) := d(i) + c_{ij} and pred(\( j \)) := i;
                end
            end
        end
```