Dynamic Programming III + Linear Programming

2. Transitive Closure and the TSP

The following is an important problem: We are given a Boolean $n \times n$ matrix G —or, equivalently, the adjacency matrix of a directed graph $G = (V, E)$. We wish to determine *for all* $i, j \in V$ whether there is a path from i to j.

$$G = \begin{bmatrix} 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \end{bmatrix} \qquad T(G) = \begin{bmatrix} 1 & 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 & 1 \end{bmatrix}$$

The matrix —or graph— $T(G)$ that contains the answers to all of these questions is called the *(reflexive) transitive closure of* G (see above; notice that it contains all self-loops (i, i)). It is best computed via a dynamic programming technique of quite broad applicability.

The subproblem definition is here quite subtle. We seek all paths from i to j. Such a path may or may not use intermediate nodes. Those that do not are the edge (i, j), if it exists, and the self-loop (i, i) if $i = j$. All other paths must use intermediate nodes. We define a subproblem by *restricting the available pool of intermediate nodes.* In particular, let $T^k(i, j)$ be true if and only if there is a path from i to j *using intermediate nodes from among* $1, 2, \ldots, k$ *only.* That is, we arbitrarily order the nodes of the graph, and allow more and more nodes as intermediate nodes in the path. The recursive equation is simple to write:

$$T^k(i, j) := T^{k-1}(i, j) \vee [T^{k-1}(i, k) \wedge T^{k-1}(k, j)].$$

It expresses this obesrvation: Suppose that we have computed the paths that use intermediate nodes up to $k - 1$, and we wish now to also allow k. Consider paths from i to j. There are two cases: Either such a path uses k, or it does not. If it does not, then $T^{k-1}(i, j)$ must be true. Otherwise, there must be a path from i to j going through k just once. Thus, there must be a path from i to k using intermediate nodes up to $k - 1$, *and* a path from k to j using intermediate nodes up to $k - 1$.

The algorithm (known as the *Floyd-Warshall algorithm*, is now easy to write:

for $i, j := 1$ to n do $\{T(i, j) := G(i, j)$ or $i = j\}$
  for $k := 1$ to n do
    for $i, j := 1$ to n do
    $T(i, j) := T(i, j) \vee [T(i, k) \wedge T(k, j)]$

Notice that we have omitted the supserscripts k. This results in savings in storage (since we only need the previous "layer" of T's to compute the next). This may result in using, for example, in the computation of $T^3(5, 2)$ the value $T^3(3, 2)$ instead of the correct $T^2(3, 2)$ — but this does not result in mistakes, only in more "streamlined" computation of the transitive closure. The complexity is, of course $O(n^3)$.

Suppose now that in the above program you change the initialization to $\{T(i, j) := \mathrm{d}(i, j)$ and $T(i, j) := 0$ if $i = j\}$, and in the loop we change $\vee$ to min and $\wedge$ to $+$. We get an $O(n^3)$ *all-pairs shortest paths algorithm!.*

A similar manoeuvre yields an algorithm for transforming any finite automaton to the corresponding regular expression.

The traveling salesman problem. Suppose that you are given n cities and the distances d_{ij}

between any two cities; you wish to find the shortest tour that takes you from your home city to all cities and back.

Naturally, the TSP can be solved in time $O(n!)$, by enumerating all tours —but this is very impractical. Since the TSP is one of the NP-complete problems, we have little hope of developing a polynomial-time algorithm for it. Dynamic programming gives an algorithm of complexity $O(n^2 2^n)$ —exponential, but much faster than $n!$. The recursive equation is similar to the one for the transitive closure: The main difference between the two problems is that in the transitive closure intermediate nodes are optional, while in the TSP they are mandatory.

We define the following subproblem: Let S be a subset of the cities containing 1 and at least one other city, and let j be a city in S other than one. Define $C(S, j)$ to be *the shortest path that starts from* 1*, visits all nodes in* S*, and ends up in* j. The program now writes itself:

for all j do $C(\{1, j\}, j) := d_{1j}$
for $s := 3$ to n do (the size of the subsets considered this round)
  for all subsets S of $\{1, \ldots, n\}$ of size n and containing 1 do
    for all $j \in S, j \neq 1$ do
      $\{C(S, j) := \min_{i \neq j, i \in S}[C(S - \{j\}, i) + d_{ij}]\}$
opt:= $\min_{j \neq 1}[C(\{1, 2, \ldots, n\}, j) + d_{j1}$.

As always, we can also recover the optimum tour by remembering the i's that achieve the minima. The complexity is $O(n^2 2^n)$: The table has $n2^n$ entries (one per set and city), and it takes about n time to fill each entry.

Linear Programming

1. Introductory example

Suppose that a company produces three products, and wishes to decide the level of production of each so as to maximize profits. Let x_1 be the amount of Product 1 produced in a month, x_2 that of Product 2 and x_3 of Product 3. Each unit of Product 1 brings to the company a profit of 100, each unit of Product 2 a profit of 600, and each unit of Product 3 a profit of 1400. There are limitations on x_1, x_2, and x_3 (besides the obvious one, $x_1, x_2, x_3 \geq 0$). First, x_1 cannot be more than 200, and x_2 more than 300 —presumably because of supply limitations. Also, the sum of the three must be, because of labor constraints, at most 400. Finally, it turns out that Products 2 and 3 use the same piece of equipment, with Product 3 three times as much, and hence we have another constraint $x_2 + 3x_3 \leq 600$. What are the best levels of production?

We represent the situation by a *linear program*, as follows:

$$
\begin{aligned}
\max 100x_1 + 600x_2 + 1400x_3 & \\
x_1 &\leq 200 \\
x_2 &\leq 300 \\
x_1 + x_2 &\leq 400 \\
x_2 + 3x_3 &\leq 600 \\
x_1, x_2, x_3 &\geq 0
\end{aligned}
$$

The set of all *feasible* solutions of this linear program (that is, all vectors in 3-d space that satisfy all constraints) is precisely the polyhedron shown in Figure 1.

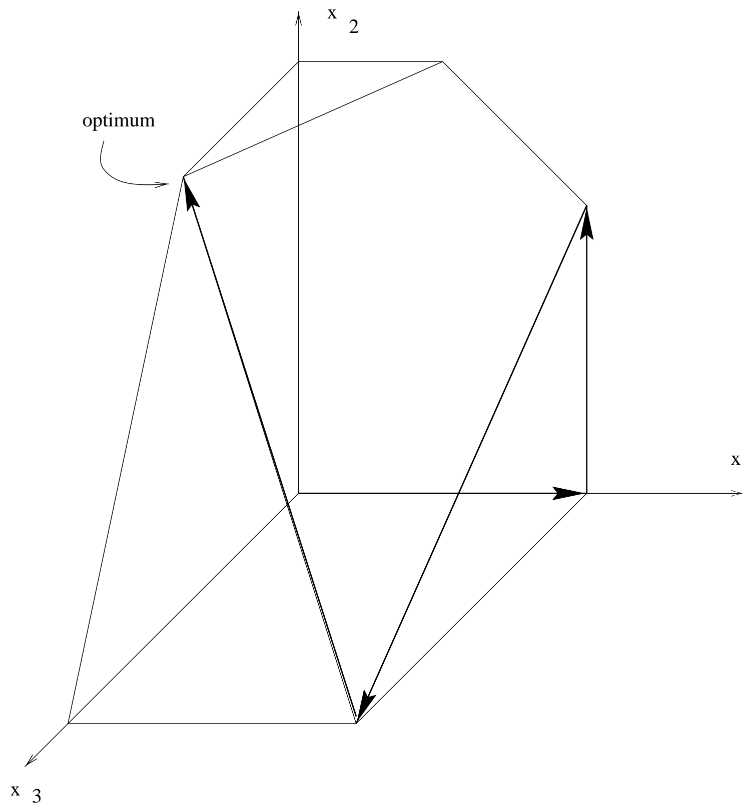

Figure 1: The feasible region

We wish to maximize the linear function $100x_1 + 600x_2 + 400x_3$ over all points of this polyhedron. This means that we want to the plane parallel to the one with equation $100x_1 + 600x_2 + 400x_3 = 0$, touches the polyhedron, and is as far towards the positive orthant as possible. Obviously, the optimum solution will be a vertex (or the optimum solution will not be unique, but a vertex will do). Of course, two other possibilities with linear programming are that (a) the optimum solution may be infinity, or (b) that there may be no feasible solution.

Such problems are solved by the *simplex method* devised by George Dantzig in 1947. The simplex method starts from a vertex (in this case the vertex $(0, 0, 0)$) and repeatedly looks for a vertex that is adjacent, and has better objective value. That is, it is a kind of *hill-climbing* in the vertices of the polytope. When a vertex is found that has no better neighbor, simplex stops and declares this vertex to be the optimum. For example, in the figure one of the possible paths followed by simplex is shown. There are now implementations of simplex that solve routinely linear programs with *many* thousands of variables and constraints.

Of course, there are two other possibilities in a linear program: It could be either that (a) the optimum solution may be infinity, or (b) that there may be no feasible solution at all. If this is the case, simplex will discover it.