## 13.1 Traveling salesman problem

Suppose that you are given $n$ cities and the distances $d_{ij}$ between them. The traveling salesman problem (TSP) is to find the shortest tour that takes you from your home city to all the other cities and back again. As there are $(n-1)!$ possible paths, this can clearly be done in $O(n!)$ time by trying all possible paths. Of course this is not very efficient.

Since the TSP is NP-complete, we cannot really hope to find a polynomial-time algorithm. But dynamic programming gives us a much better algorithm than trying all the paths.

The key is to define the appropriate subproblem. Suppose that we label our home city by the symbol 1, and other cities are labeled $2,\ldots,n$. In this case, we use the following: for a subset $S$ of vertices including 1 and at least one other city, let $C(S,j)$ be the shortest path that start at 1, visits all other nodes in $S$, and ends at $j$. Note that our subproblems here look slightly different: instead of finding tours, we are simply finding paths. The important point is that the shortest path from $i$ to $j$ through all the vertices in $S$ consists of some shortest path from $i$ to a vertex $x$, where $x \in S - \{j\}$, and the additional edge from $x$ to $j$.

for all $j$ do $C(\{i,j\},j) := d_{1j}$
  for $s$ = 3 to n do % $s$ is the size of the subset
  for all subsets $S$ of $\{1,\ldots,n\}$ of size $s$ 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 := $\min_{j \neq i} C(\{1,\ldots,n\},j) + d_{j1}$

The idea is to build up paths one node at a time, not worrying (at least temporarily) where they will end up. Once we have paths that go through all the vertices, it is easy to check the tours, since they consists of a shortest path through all the vertices plus an additional edge. The algorithm takes time $O(n^2 2^n)$, as there are $O(n2^n)$ entries in the table (one for each pair of set and city), and each takes $O(n)$ time to fill. Of course we can add in structures so that we can actually find the tour as well. **Exercise: Consider how memory-efficient you can make this algorithm.**

## 13.2 DP on trees: Dominating Set

Given a graph $G$, a vertex $u$ dominates a vertex $v$ if either $u = v$ or $(u,v) \in E(G)$. A*dominating set* $X \subseteq V(G)$ is a subset of the vertices such that every vertex of $G$ is dominated by some vertex in $X$. A minimum dominating set is a dominating set of as few vertices as possible or, if $G$ has vertex weights, a dominating set for which the sum of the weights of the vertices is as small as possible.

The problem of finding a smallest dominating set is NP-hard, so we don't expect there to exist a polynomial-time algorithm. But if we restrict the problem to graphs $G$ that are trees (or, more generally, forests), the problem becomes solvable in polynomial time.

In fact, if $G$ is a forest and there are no vertex weights, the problem is solvable by a greedy algorithm: repeatedly pick a leaf $v$ which is not yet dominated, add its neighbor $u$ to the dominating set, and delete $u$ and its now-dominated neighbors. Every dominating set of $G$ dominates $v$ and therefore contains either $v$ or $u$, and $u$ dominates a superset of the vertices that $v$ dominates, so replacing $v$ with $u$ in a dominating set gives another dominating set of at most the same size, so the greedy algorithm can't fail.

If $G$ is a forest (or, for simplicity, a tree, with some vertex $r$ that we call the root of $G$) but has vertex weights, then the greedy algorithm may fail if the weight of $v$ is less than the weight of $u$ at any point in that algorithm. However, there's a solution by dynamic programming.

For each vertex $v \in V(G)$, we'll have three subproblems related to the *subtree rooted at* $v$: the set $T(v)$ of vertices $u$ such that the unique path from $u$ to $r$ in the tree contains $v$.

1. $A(v)$ is the weight of the minimum-weight subset of $T(v)$ that dominates $T(v) \setminus \{v\}$: that is, that dominates the subtree rooted at $v$, except possibly $v$ itself.
2. $B(v)$ is the weight of the minimum-weight subset of $T(v)$ that dominates $T(v)$.
3. $C(v)$ is the weight of the minimum-weight subset of $T(v)$ containing $v$ that dominates $T(v)$.

We'll solve these problems starting with single-vertex subtrees (that is, subtrees rooted at leaves, other than the subtree rooted at $r$ if $r$ is a leaf), and work toward bigger subtrees.

If $u_0, u_1, \ldots, u_{k-1}$ are a vertex $v$'s children (that is, the vertices such that the next vertex on the path to $r$ is $v$), then:

1. $A(v) = \min(w(v) + \sum A(u_i), \sum B(u_i))$.

2. $B(v) = \min(w(v) + \sum A(u_i), C(u_0) + \sum_{i \neq 0} B(u_i), C(u_1) + \sum_{i \neq 1} B(u_i), \ldots, C(u_{k-1}) + \sum_{i \neq k-1} B(u_i))$
3. $C(v) = w(v) + \sum A(u_i)$.

We can prove by induction that these recursive formulas correctly calculate the subproblems defined above:

1. A subset of $T(v)$ that dominates $T(v) \setminus \{v\}$ either contains $v$ or doesn't. If it contains $v$, then $v$ dominates $u_0$, $u_1$, …, $u_k$, but any other vertices in their subtrees must be dominated by vertices in those subtrees, and the cheapest way to do so is to pick the minimum-weight subset of $T(u_i)$ that dominates $T(u_i) \setminus \{u_i\}$ for each $i$, whose weight is exactly $A(u_i)$, for a total of $w(v) + \sum A(u_i)$. If it doesn't contain $v$, then all vertices in the subtrees must be dominated by vertices in those subtrees, and the cheapest way to do so is exactly, by definition, $B(u_i)$ for each of them.
2. A subset of $T(v)$ that dominates $T(v)$ either contains $v$ or doesn't. If it contains $v$, then the cheapest weight is $w(v) + \sum A(u_i)$ by the same argument as above. If not, we must dominate $v$ by picking at least one of $u_0$, $u_1$, …, $u_{k-1}$ to be in the dominating set. If we pick (at least) $u_j$, then we need to dominate all of $T(u_j)$ by a set that includes $u_j$, and the cheapest way to do so is exactly, by induction, $C(u_j)$, and we need to dominate all of $T(u_i)$ for all other $i$, and the cheapest way to do so is exactly, by induction, $B(u_j)$.
3. $C(v) = w(v) + \sum A(u_i)$ by the same argument as in the first case of the proof that we correctly calculate $A(v)$ above: the definition of $C(v)$ forces us to pick $v$ as a vertex in the dominating set, so only that first case applies.

Finally, the answer we care about at the end is the cheapest way to dominate the whole tree $G = T(r)$, whose cost is $B(r)$.

The runtime of this algorithm is $O(|V(G)|)$: there are $O(|V(G)|)$ subproblems. An upper bound on the time to solve each of the subproblems is $O(|V(G)|)$, since we need to take a minimum over all children of $v$, and there may be up to $|V(G)| - 1$ of them. (For the subproblems $B(v)$, we also need to calculate up to $|V(G)| - 1$ sums of the form $\sum_{i \neq k} B(u_i)$, but we can calculate all of those sums in total time at most $O(|V(G)|)$ (or $O$(number of children of v) by first calculating $\sum_i B(u_i)$ once and then subtracting $B(u_k)$ as necessary.) However, each vertex appears only once *as a child of another vertex*, so the total size of the mins over all subproblems is still only $|V(G)|$.

## 13.3 The Knapsack Problem

The knapsack problem: given $n$ objects of (positive integer) sizes $s_0$, $s_1$, …, $s_{n-1}$ and (positive integer) values $v_0$, $v_1$, …, $v_{n-1}$ (respectively) and a bag size $t$, find a subset of the objects whose total size is at most $t$ (so they'll fit in the

knapsack) that has the highest total (sum) value among such subsets.

The knapsack problem is NP-hard, so we don't expect there to exist an algorithm that runs in time polynomial in the number of bits needed to represent the input. We can represent numbers up to $k$ with only $\log k$ bits, so an algorithm for the knapsack problem that needs $O(k)$ time to handle numbers up to $k$ is not a polynomial-time algorithm: it's exponential in the length $\log k$ of the input. On the other hand, an algorithm whose running time is polynomial in $n$ and exponential in the number of digits of $t$ (but only polynomial in the *value* of $t$) is in some sense better than an algorithm whose running time is also exponential in $n$. Algorithms with such running times (polynomial in the values of, not the lengths of the representations of, the inputs) are called *pseudopolynomial*. We can give a pseudopolynomial-time algorithm for the knapsack problem by dynamic programming.

Subproblems: for each $i \in [-1, n)$ and each $j \in (0, t]$, let $D[i, j]$ be the highest value we can get from a subset of the objects up to object $i$ whose total size is at most $j$.

We claim that

1. $D[i, j] = \max(D[i-1, j], D[i-1, j-s_i] + v_i)$ if $i \geq 0$ and $j \geq s_i$,
2. $D[i, j] = D[i-1, j]$ if $i \geq 0$ and $j < s_i$,
3. $D[i, j] = 0$ if $i = -1$

We can prove inductively that these correctly calculate the highest value we can get from a subset of the first $i$ objects whose total size is at most $j$:

1. If $i \geq 0$ and $j \geq s_i$, a subset of the first $i$ objects can either contain object $i$ or not. If not, then it's a subset of the first $i-1$ objects, of which the highest value is $D[i-1, j]$. If so, the total size of the other objects in the subset (which are a subset of the first $i-1$ objects) is at most $j - s_i$. The highest value of such a subset is, by induction, $D[i-1, j-s_i]$, and we also took object $i$ of value $v_i$. Conversely, every subset of the first $i-1$ elements of total size at most $j$ is also a subset of the first $i$ elements of total size at most $j$, and every subset of the first $i-1$ elements of total size at most $j - s_i$ can be extended to a subset of the first $i$ elements of total size at most $j$ by adding element $i$, so the maximum value above is attained.
2. The case when if $i \geq 0$ and $j < s_i$ is the same, but we can't take element $i$.
3. If $i = -1$, the only subset of the objects up to object $-1$ is the empty set, since there aren't any such objects. The empty set has value 0.

The final answer is $D[n-1,t]$, the highest value we can get from a subset of the objects up to object $n-1$ (that is, all the objects) whose total size is at most $t$.

Each subproblem can be solved in $O(1)$ time, and there are $O(nt)$ subproblems, so we can compute answers to all the subproblems in time $O(tn)$, which is pseudopolynomial (as defined above). Naively, we'd need $O(nt)$ space as well, but we only need to store the answers to subproblems $D[i-1,*]$ to calculate the answers to the subproblems $D[i,*]$, so we only need $O(t)$ space.