Single Source Shortest Paths

This problem is defined as follows: given a graph G, we want to find a shortest path from a given *source* vertex $s \in V$ to each vertex $v \in V$. We have a plethora of algorithms available to us for solving this problem, and each one runs only under a certain set of conditions. The different algorithms are described in detail in CLRS Chapter 24, but for the purposes of this course, we are only interested in DIJKSTRA's algorithm.

- DIJKSTRA: This algorithm takes as its input a weighted, directed graph where all of the edge weights are non-negative. The algorithm is very similar to MST-PRIM and also uses a priority queue. The algorithm runs in $O(E \log V)$ and depends on the implementation of the priority queue. If a Fibonacci heap is used, the running time can be improved to $O(V \log V + E)$.

```
DIJKSTRA(G,w,s)
1  INITIALIZE-SINGLE-SOURCE
2  S ← ∅
3  Q ← V
4  while Q ≠ ∅
5  do u ← EXTRACT-MIN(Q)
6       S ← S ∪ {u}
7       for  each v ∈ Adj[u]
8       do RELAX(u, v, w)
```

```
RELAX(u,v,w)
1  if d[v] > d[u] + w(u, v)
2     then d[v] = d[u] + w(u, v)
3             π[v] = u
4
```

```
INITIALIZE-SINGLE-SOURCE(G,s)
1  for ∀v ∈ V
2  do d[v] ← ∞
3       π[v] = NIL
```

The running time of this algorithm is calculated as follows. The min-priority queue Q is maintained by three priority-queue operations: INSERT (implicit in line 3, $O(\log n)$), EXTRACT-MIN (line 5, $O(\log n)$), and DECREASE-KEY (implicit in RELAX in line 8, $O(\log n)$). Thus, we see that the running time of DIJKSTRA's algorithm is $O((V + E) \log V)$ which is $O(E \log V)$ for dense graphs.

Dynamic Programming

Dynamic programming solves problems by combining the solutions to subproblems. In order to apply the dynamic programming method to a particular problem, the problem must exhibit optimal substructure and overlapping subproblems.

Longest Increasing Subsequence

Consider an unsorted array of integers:

$$A = \begin{bmatrix} 3 & 1 & 2 & 6 & 1 & 4 & 7 & 8 \end{bmatrix}$$

Our goal is to find the Longest Increasing Subsequence (hereafter abbreviated LIS) within A. In this case, increasing subequences are $\{3, 6, 7, 8\}$ or $\{1, 6\}$ or $\{1, 1, 4, 7, 8\}$ depending on whether or not the subsequences are strictly increasing. Notice that the subsequences are not contiguous, meaning that $\{1, 6\}$ is an increasing subsequence even though 1 and 6 are not adjacent in the array. In this case, a LIS is $\{1, 2, 6, 7, 8\}$ or $\{1, 1, 4, 7, 8\}$, depending on the convention.

First, consider how difficult the solution is to calculate by brute force. We need to consider not only the larger elements that come after a given element, but what order they appear in. For the first element in the array, we need to check the $n-1$ other elements and the determine all other possible increasing subsequences that start with the first element, which may or may not be the optimal starting point. Thus, the brute force method clearly takes exponential time. But dynamic programming comes to our rescue.

First, we introduce a function which will come in very handy:

$$\delta(i, j) = \begin{cases} 1 & \text{if } A[i] \leq A[j] \\ 0 & \text{otherwise} \end{cases} \tag{1}$$

Next, we will define our data structure. We will create an array L of n elements such that $L[i]$ is equal to the size of the LIS that terminates in $A[i]$. Clearly, $L[1] = 1$ because the $A[1]$ will always be a LIS of size 1. This is our base case. We fill the array according the following recursive formula:

$$L[j] = \underset{1 \leq i < j}{\text{MAX}} (\delta(i, j) * L[i] + 1) \tag{2}$$

To summarize, we simply walk the array from the beginning up to element $A[i]$, and take the maximum possible subsequence that terminates in element $A[i]$. For our example, we fill the array L in this manner:

$$L = \begin{bmatrix} 1 & 1 & 2 & 3 & 2 & 3 & 4 & 5 \end{bmatrix}$$

To find our maximum, we simply walk our array and take the maximum, which in this case is 5. To find our solution, we can simply walk backwards through our list and recreate the solution or store a pointer to the previous element in another list. In either case, this algorithm runs in $O(n^2)$.

3-Partition

Given integers $a_1, \ldots, a_n$, we want to determine whether it is possible to find a partition of elements $\{1, \ldots, n\}$ into three disjoint subsets $I, J, K \subseteq \{1, \ldots, n\}$ such that:

$$\sum_{i \in I} a_i = \sum_{j \in J} a_j = \sum_{k \in K} a_k = \frac{1}{3} \sum_{i=1}^{n} a_i \tag{3}$$

For example, $(1, 2, 3, 4, 4, 5, 8)$ is a YES-instance, because there is the partition $(1, 8), (4, 5), (2, 3, 4)$, while $(2, 2, 3, 5)$ is a NO-instance, because $2 + 2 + 3 + 5 = 12$, and there are not 3 disjoint subsets that sum to 4. Consider the following dynamic programming solution to this problem. On input $a_1, \ldots, a_n$, let $A = (1/3) \sum_{i=1}^{n} a_i$, and define a boolean matrix M of size $(A + 1) \times (A + 1) \times (n + 1)$, with the meaning that $M[x, y, k]$ is true if and only if there are two disjoint subsets $I, J \subseteq \{1, \ldots, k\}$ such that $\sum_{i \in I} a_i = x$ and $\sum_{j \in J} a_j = y$. Once we construct the matrix, the answer to the 3-PARTITION problem is in the entry $M[A, A, n]$. The recursive definition is:

$$M[x, y, i] = M[x, y, i-1] \vee M[x, y - a_i, i-1] \vee M[x - a_i, y, i-1] \tag{4}$$

with base cases $M[0, 0, 0] = 1$ and $M[x, y, 0] = 0$ for $x + y > 0$. If we index off the table, we count that as a *false* value. There are $O(A^2 n)$ entries in the matrix, each of which can be filled in $O(1)$ time. Thus, this algorithms takes $O(A^2 n)$ time. Here is a very simple example on the set of integers $\{1, 2, 3, 3\}$. Since the sum of this set of integers is 9, a 3-PARTITION would be a subset that sums to 3.

$$M[x,y,0] = \begin{array}{c|cccc} & 0 & 1 & 2 & 3 \\ \hline 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 2 & 0 & 0 & 0 & 0 \\ 3 & 0 & 0 & 0 & 0 \end{array}, \quad M[x,y,1] = \begin{array}{c|cccc} & 0 & 1 & 2 & 3 \\ \hline 0 & 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 \\ 2 & 0 & 0 & 0 & 0 \\ 3 & 0 & 0 & 0 & 0 \end{array}, \quad M[x,y,2] = \begin{array}{c|cccc} & 0 & 1 & 2 & 3 \\ \hline 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 0 \\ 2 & 1 & 1 & 0 & 0 \\ 3 & 1 & 0 & 0 & 0 \end{array},$$

$$M[x,y,3] = \begin{array}{c|cccc} & 0 & 1 & 2 & 3 \\ \hline 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 1 \\ 2 & 1 & 1 & 0 & 1 \\ 3 & 1 & 1 & 1 & 1 \end{array}, \quad M[x,y,4] = \begin{array}{c|cccc} & 0 & 1 & 2 & 3 \\ \hline 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 1 \\ 2 & 1 & 1 & 0 & 1 \\ 3 & 1 & 1 & 1 & 1 \end{array}$$

Notice that the matrixes are all symmetric around the diagonal, and also that once a cell is set to *true* it will remain set to *true* throughout the duration of the algorithm. Thus, we can see that we can easily use only $O(A^2)$ space, and that we only need to fill in the upper triangle of the matrix. However, our overall running time will still be $O(A^2 n)$.

Subset Sum

SUBSET-SUM is a very simple variation of the knapsack problem. However, it figures prominently in the NP-Completeness reduction tree, and is therefore worth looking at in its own right. The problem is defined as follows: Given an array A, is it possible to find a subset that sums exactly to a bound B? Notice that this is a *decision* problem, meaning the answer is *yes* or *no*.

$$A = \begin{bmatrix} 3 & 2 & 4 & 5 & 3 & 7 & 13 & 10 & 6 & 11 \end{bmatrix}$$

Consider the above array A. In this case, is it possible to find a subset of A that sums to exactly 17? The answer is yes, because the sum of $4 + 13 = 17$ and therefore $\{4, 13\}$ is such a subset. What about 19? The answer is again yes, $\{2, 7, 10\}$. What about 22 or 26? After we derive the dynamic programming recursive formula, run it yourself and see.

We will define an $n \times B$ matrix, where n is the number of elements in our array and B is the desired sum. We define the cell $M[i, j]$ to contain 1 if it is possible to find a subset of the integers 1 through i that sum to exactly j and 0 otherwise. Again, as in knapsack, we have a 0-index column, but this time, we initialize that column to 1, meaning that it is always possible to have a subset that sums to 0. Therefore, the following recursive formula presents itself:

$$M[i, j] = \text{MAX}(M[i-1, j], M[i-1, j-A_i]) \tag{5}$$

Exactly as in the knapsack algorithm, we justify this solution accordingly: We check if we can get the sum j by not including the element i in our subset, and we check if we can get the sum j by including i by checking if the sum $j - A_i$ exists without the i-th element. This is identical to knapsack, except that we are only storing a yes/no answer. Note that the base case of initializing the 0-column to 1 allows us to always include the element, because when $A_i = j$ then $M[i-1, j-A_i] = M[i-1, 0] = 1$. Consider the following example to determing whether or not the above array A contains a subset that sums to 5:

$$M = \begin{bmatrix} & 0 & 1 & 2 & 3 & 4 & 5 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 3 & 1 & 0 & 0 & 1 & 0 & 0 \\ 2 & 1 & 0 & 1 & 1 & 0 & 1 \\ 4 & 1 & 0 & 1 & 1 & 1 & 1 \\ 5 & 1 & 0 & 1 & 1 & 1 & 1 \\ 3 & 1 & 0 & 1 & 1 & 1 & 1 \\ 7 & 1 & 0 & 1 & 1 & 1 & 1 \\ 13 & 1 & 0 & 1 & 1 & 1 & 1 \\ 10 & 1 & 0 & 1 & 1 & 1 & 1 \\ 6 & 1 & 0 & 1 & 1 & 1 & 1 \\ 11 & 1 & 0 & 1 & 1 & 1 & 1 \end{bmatrix}$$

We find our solution by checking cell $M[n, B]$, in this case $M[11, 5]$. If $M[n, B] = 1$ then it is possible to find a subset of A that sums to B. If not, then no subset is possible. To find our optimal subset, we simply recurse backwards through the matrix. For example, suppose that $M[n, B] = 1$. We want to know if A_n is included in our set, and so we check the two cells $M[n-1, B]$ and $M[n-1, n-A_n]$. If $M[n-1, B] = 1$, then we do not include A_n in our set, and we continue to recurse. If $M[n-1, n-A_n] = 1$, then we do include A_n. If both equal 1, we can choose which subset to take.

There are $n \times B$ cells in our matrix, and therefore this algorithm runs in $O(nB)$ time. Notice that this is not polynomial because our running time depends on two variables, and one could easily be an exponential function of the other. This is consistent with the result that SUBSET-SUM is NP-Complete.