Dynamic Programming

1 Dynamic Programming

Sometimes, problems can be easily solved with recursion, but that results in an excess of repeated work. In order to eliminate wasted computation, one can utilize **memoization**: storing sub-solutions during recursion problems and accessing them as necessary. If one computes a memoization backwards (i.e., using an iterative bottom-up approach rather than a recursive top-down approach) it is called **dynamic programming**, and is an elegant and simple solution to many complex problems.

Dynamic programming can only solve problems in which the optimum is recursive – that is, the optimum of layer i is dependent upon the optimum of layer $i+1$. A common example is:

$$OPT(i,j) = max(OPT(i+1,j), OPT(i+1,j+1))$$

2 Weighted Interval Scheduling

Imagine you have to choose between going to several parties on the same night, where each one has a variable amount of fun involved. In other words, given a set of intervals I with values $v_i \geq 0$, find a subset of pairwise disjoint intervals of maximum total value. This problem is trivial if all $v_i = 1$, because then greedy algorithms would apply; however, we must be more clever to solve the general case.

For any interval i, OPT either includes it or excludes it. Either way, OPT includes the optimum subset of remaining intervals (excluding those that overlap with i if it was included). However, this resultant set of intervals can be very complex, so we must instead branch over a carefully chosen interval – the one that starts last.

Assume the intervals $i = (l, r)$ are sorted by r. Let j be the last interval in this order. We know that if j is excluded, $OPT(j) = OPT(j-1)$. If j is included, then let $p(j)$ be the longest interval such that $r_{p(j)} < l_j$. Then $OPT(j) = v_j + OPT(p(j))$. In other words:

$$OPT(j) = max(OPT(j-1), v_j + OPT(p(j)) \text{ with } OPT(0) = 0$$

3 Knapsack Problem

Imagine you are robbing a jewelry store, and each item of jewelry has a different value, but some are heavier than others. How much total value of jewelry can you steal such that you can still carry them all during your getaway? In other words: given n items with integer weights $w_i \geq 0$ and values $v_i \geq 0$, and a total weight bound W, select a set S of items with $\sum_{i \in S} w_i \leq W$ that maximizes $\sum_{i \in S} v_i$.

As a first thought, we might say that $OPT(i)$ is the optimal solution for items $1 \rightarrow i-1$, then we decide whether to add i to the knapsack. However, this doesn't address our variable weight bound b. In dynamic programming problems, it often helps to introduce new parameters insofar as it simplifies the final solution. Thus we say that if $w_i \leq b$:

$$OPT(i, b) = max(OPT(i-1, b), v_i + OPT(i-1, b-w_i))$$

And if $w_i > b$:

$$OPT(i, b) = OPT(i-1, b)$$

In algorithm form, we have:

Algorithm 1 Knapsack Problem

```
for b ≤ W do
    a[0][b] = 0
end for
for i ≤ n do
    for b ≤ W do
        if w_i > b then
            a[i][b] = a[i-1][b]
        else
            a[i][b] = max(a[i-1][b], v_i + a[i-1][b-w_i])
        end if
    end for
end for
return a[n][W]
```

The runtime of this algorithm is evidently $\Theta(nW)$, which looks polynomial at first, but is actually **pseudo-polynomial** (i.e., it would be polynomial if the input was written down in unary). The runtime depends on the amount of bits needed to store the input data. However, this can be approximated in true polynomial time by reducing the w-space by rounding the weights to restrict them to a more manageable range.

4 Related Problems

4.1 Subset Sum

Given numbers $a_1, \ldots, a_n$ and a maximum weight W, is there a subset S such that $\sum_{i \in S} a_i = W$?

Reduction to knapsack: set $w_i = v_i = a_i$.

4.2 Partition

Given numbers $a_1, \ldots, a_n$, is there a subset S with $\sum_{i \in S} a_i = \sum_{i \notin S} a_i$?

Reduction to subset sum: $W = \frac{1}{2} \sum_i a_i$

4.3 Independent Set

Given an undirected graph $\mathcal{G}$, find the maximum size subset $S \in V$ such that no two $v \in S$ share an edge. This problem is often used to solve pairwise constraint problems, and is NP-hard but solvable in linear time for trees. Greedy algorithms don't work here, so we must use dynamic programming.

Let Y signify that v is included in OPT, and let N signify that v is not included in OPT. Then, $OPT(v, Y/N)$ is the maximum number of nodes in an independent set rooted at v. We solve the problem with the follwing recurrance relation:

$$OPT(\emptyset, Y/N) = 0$$

$$OPT(v, Y) = 1 + \sum_{children} OPT(v_i, N)$$

$$OPT(v, N) = \sum_{children} max(OPT(v_i, Y), OPT(v_i, N))$$

In other words, if we include v, the maximum number of nodes is increased by one, but we cannot include v's children. If v is excluded, the maximum does not change, and we can choose whether to include each of its children.