Dynamic Programming Part 2

Atli FF

Árangursrík forritun og lausn verkefna

School of Computer Science

Reykjavík University

DP over bitmasks

- Remember the bitmask representation of subsets?
- Each subset of n elements is mapped to an integer in the range $0, \ldots, 2^n - 1$
- This makes it easy to do dynamic programming over subsets

Traveling salesman problem

- We have a graph of n vertices, and a cost $c_{i,j}$ between each pair of vertices i, j. We want to find a cycle through all vertices in the graph so that the sum of the edge costs in the cycle is minimal.
- This problem is NP-Hard, so there is no known deterministic polynomial time algorithm that solves it
- Simple to do in $O(n!)$ by going through all permutations of the vertices, but that's too slow if $n > 11$
- Can we go higher if we use dynamic programming?

Traveling salesman problem

- Without loss of generality, assume we start and end the cycle at vertex 0
- Let $\mathrm{tsp}(i, S)$ represent the cheapest way to go through all vertices in the graph and back to vertex 0, if we're currently at vertex i and we've already visited the vertices in the set S
- Base case: $\mathrm{tsp}(i, \text{all vertices}) = c_{i,0}$
- Otherwise $\mathrm{tsp}(i, S) = \min_{\ j \notin S} \ \{ \ c_{i,j} + \mathrm{tsp}(j, S \cup \{j\}) \ \}$

Traveling salesman problem

```
const int N = 20;
const int INF = 100000000;
int c[N][N];
int mem[N][1<<N];
memset(mem, -1, sizeof(mem));
int tsp(int i, int S) {
    if (S == ((1 << N) - 1)) {
        return c[i][0];
    }
    if (mem[i][S] != -1) {
        return mem[i][S];
    }
    int res = INF;
    for (int j = 0; j < N; j++) {
        if (S & (1 << j))
            continue;
        res = min(res, c[i][j] + tsp(j, S | (1 << j)));
    }

    mem[i][S] = res;
    return res;
}
```

Traveling salesman problem

- Then the optimal solution can be found as follows:

```
printf("%d\n", tsp(0, 1<<0));
```

Traveling salesman problem

- Time complexity?

- There are $n \times 2^n$ possible inputs
- Each input is computed in $O(n)$ assuming recursive calls are $O(1)$
- Total time complexity is $O(n^2 2^n)$

- Now n can go up to about 20

Subset sum problem

- Another common dynamic programming task is known as the subset sum problem.
- Given n positive integers $a_1, \ldots, a_n$ find if there is a subset with sum c. Variants also include finding the sum closest to c, greatest sum not exceeding c and so on.
- The naïve solution here would involve checking every subset, which if done efficiently (for example with gray codes) takes $O(2^n)$, which is quite slow.

Subset sum problem

- Let $f(i, s)$ be a boolean function answering whether there exists a subset of $a_1, \ldots, a_i$ with sum s.
- Then

$$f(i,s) = \begin{cases} \texttt{true} \text{ if } i = s = 0 \\ \texttt{false} \text{ if } i = 0, s \neq 0 \\ \texttt{false} \text{ if } s < 0 \\ f(i-1, j) \;\; \texttt{or} \;\; f(i-1, j-a_i) \text{ otherwise} \end{cases}$$

- The different variants can then be read from the values of f. Each state takes $O(1)$ to compute, and there are $n(a_1 + \cdots + a_n)$ states. Denoting the sum by Σ this makes our time complexity $O(n\Sigma)$, which isn't great, but is often better than $O(2^n)$.

Subset sum problem

```cpp
const int N = 20;
const int SIGMA = 10000;
int a[N];
int dp[N][SIGMA];
// use int so -1 is unmemoized
// 0 and 1 are the bools as usual
bool subsetsum(int i, int s) {
    if(i < 0) return s == 0;
    if(s < 0) return false;
    if(dp[i][s] != -1) return dp[i][s];
    return dp[i][s] = subsetsum(i - 1, s) || subsetsum(i - 1, s - a[i]);
}
```

Subset sum problem - variant

- Say we want to find the most even way to split the numbers into two groups, that is to say in a way that minimizes the difference of the sums of the two groups.
- Furthermore we want to actually output these numbers rather than just the difference in sum.
- We can use the subset sum solution to do this, simply adding a table that keeps track of what choices we made at what point.

Subset sum problem

```cpp
#include <bits/stdc++.h>
using namespace std;
vector<int> a;
vector<vector<int>> dp, mv;
bool subsetsum(int i, int s) {
        if(i < 0) return s == 0;
        if(s < 0) return false;
    if(dp[i][s] != -1) return dp[i][s];
    dp[i][s] = 0;
    if(subsetsum(i - 1, s)) {
        dp[i][s] = 1;
        mv[i][s] = 1;
    } else if(subsetsum(i - 1, s - a[i])) {
        dp[i][s] = 1;
        mv[i][s] = 0;
    }
        return dp[i][s];
}
```

Subset sum problem

```
int main() {
        int n, sm = 0; cin >> n;
        a = vector<int>(n);
        for(int i = 0; i < n; ++i)
                cin >> a[i], sm += a[i];
        dp = mv = vector<vector<int>>(n, vector<int>(sm, -1));
        int bst = sm / 2;
        while(!subsetsum(n - 1, bst)) bst--;
        vector<bool> group1(n, false);
        for(int i = n - 1; i >= 0; --i) {
                if(!mv[i][bst]) {
                        group1[i] = true;
                        bst -= a[i];
                }
        }
        cout << "Difference: " << abs(bst - (sm - bst)) << '\n';
        cout << "Group 1: ";
        for(int i = 0; i < n; ++i) if(group1[i]) cout << a[i] << ' ';
        cout << "\nGroup 2: ";
        for(int i = 0; i < n; ++i) if(!group1[i]) cout << a[i] << ' ';
        cout << '\n';
}
```

Multidimensional DP comment

- The order in which you put the dimensions in a multidimensional dp can affect the running time.
- This is due to cache locality, so if you are fetching sequentially from one dimension and not the other, this can make one order faster.
- Usually it doesn't matter, but in a few cases it might.

Knapsack problem

- The subset sum problem time complexity is exponential, as the sums of the numbers is exponential in the size of the actual input.
- Among similar "hard" problems (in terms of time complexity) is the knapsack problem.
- We have n items, each with some value and some weight. We also have a knapsack with a maximum weight capacity and want to maximize the value with respect to this condition.

Knapsack problem

- We can once more use dynamic programming to solve this.
- We let $f(i, j)$ be the maximum value we can get from the first i items if our maximum weight is j.
- Let $v_1, \ldots, v_n$ be the values and $w_1, \ldots, w_n$ the weights. Then

$$f(i, j) = \begin{cases} -\infty \text{ if } j < 0 \\ 0 \text{ otherwise if } i = 0 \\ \max(f(i-1, j), f(i-1, j-w_i) + v_i) \text{ otherwise} \end{cases}$$

- The time complexity is then $O(nS)$ where S is the sum of the weights.
- We'll leave translating this into code as an exercise.

Egg dropping problem

- The previous problems are all very well known and classic in computer science. Let us also take a slightly less common example called the egg dropping problem.
- We have a building with k floors and we wish to figure out from which floor we have to drop an egg so it breaks. I.e. for some x dropping it from the x-th floor the egg will break, but dropping it from the $(x-1)$-st floor the egg won't break it.
- If we have n eggs, how few trials can we get away with in the worst case?

Egg dropping problem

- Say we drop the egg from a floor y.
- If the egg breaks, we only need to check floors $< y$ and have one less egg. This is essentially the same problem again but with $y - 1$ floors and $n - 1$ eggs.
- If the egg doesn't break we only need to check floors $> y$, so the problem is again the same with $k - y$ floors and n eggs.
- Since we are looking at the worst case, our result is the worse of these two.
- Since we can choose any y, we take the best result among all y.

Egg dropping problem

- Let $f(n, k)$ be the minimum number of trials for n eggs and k floors.
- We note that if we have one egg, we must always go through the floors in order since we can't afford to break an egg.
- All together this gives us

$$f(i,j) = \begin{cases} 1 \text{ if } k = 1 \\ 0 \text{ if } k = 0 \\ k \text{ if } i = 1 \\ \min_{1 \le x \le k} 1 + \max(f(i-1, x-1), f(i, j-x)) \end{cases}$$

- We see that this takes $O(k)$ per state and we have $O(nk)$ states, so the time complexity is $O(nk^2)$.

Egg dropping problem

- As a side note, this can also be solved in $O(nk)$ with a different dynamic programming approach. Try considering calculating the maximum number of floors you can check with n eggs and k trials using dynamic programming.
- Using some more clever ideas this can even be brought down to $O(n \log(k))$, but we won't need this here.