Lecture 22: Dynamic Programming

COSC242: Algorithms and Data Structures

Brendan McCane

Department of Computer Science, University of Otago

Dynamic programming

The iterative and memoised algorithms for solving the 0-1 knapsack problem are examples of dynamic programming solutions to problems. Dynamic programming:

- is used for optimisation problems, where we want to find the "best way" of doing something;
- is a recursive approach that involves breaking a global problem down into more local subproblems;
- requires subproblem optimality, i.e. a simple way to combine optimal solutions of smaller problems to get optimal solutions of larger problems;
- avoids the inefficiency that subproblem overlap causes for straightforward recursion (the same subproblems occurring often and thus being solved many times);
- does this by memo-ising (i.e. storing the solutions of subproblems in a table and then looking them up).

Dynamic programming

The main difference between greedy algorithms and dynamic programming algorithms is that greedy algorithms solve a problem by running forward and greedily choosing the locally best item as the next one. Only one path or solution is ever considered.

Whereas dynamic programming algorithms solve a problem by considering multiple paths at each decision point. Dynamic programming solutions often (but not always) work backwards and define the optimal solution to a problem as a choice over optimal solutions to sub-problems.

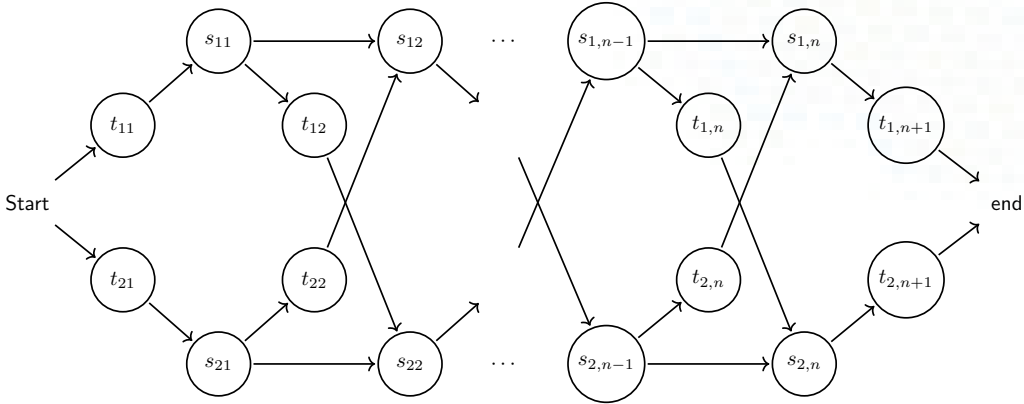
Example problems

There are many problems that can be solved efficiently using dynamic programming including:

- 0-1 knapsack problem
- assembly-line scheduling
- matrix chain multiplication (in what order to multiply matrices)
- longest common subsequence of two strings (useful for finding commonalities in genomes)
- constructing optimal binary search trees given a known distribution of search keys.

We'll have a look at assembly-line scheduling as another example.

Assembly line scheduling



Explanation

- there are n "stations" upon which some assembly is done
- there are two "lines" (think conveyor belts)
- each station takes time s_{ij} (i indicates line number, j indicates station number).
- stations with the same j number do the same job (product is the same), but may take differing time to do it
- after completing a station i, j, there is a choice. Either:
  1. stay on the same line and progress to the next station (this has zero cost); or
  2. move to the other line and progress to the next station (this has cost $t_{i,j+1}$).

Fastest time

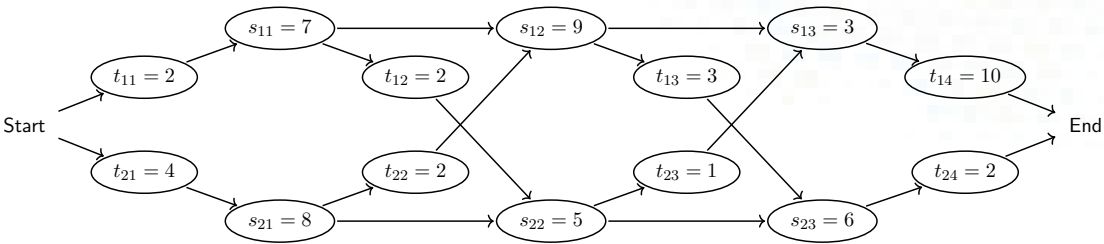
- normally, both lines are fully utilised and there is no swapping between lines, but ...
- sometimes a rush order comes in and the goal is to find the fastest way through the factory for a single item.

What's the fastest time through the factory?

We could use a brute force approach. If there are two lines, then we are essentially choosing which of the stations in line 1 to use (and therefore which of the line 2 stations to use). We can think of the chosen line 1 stations as being "on" and the not chosen ones as being "off". That looks like a binary number with n digits, and there are 2^n such numbers. Therefore, brute force is $O(2^n)$.

Simulation example

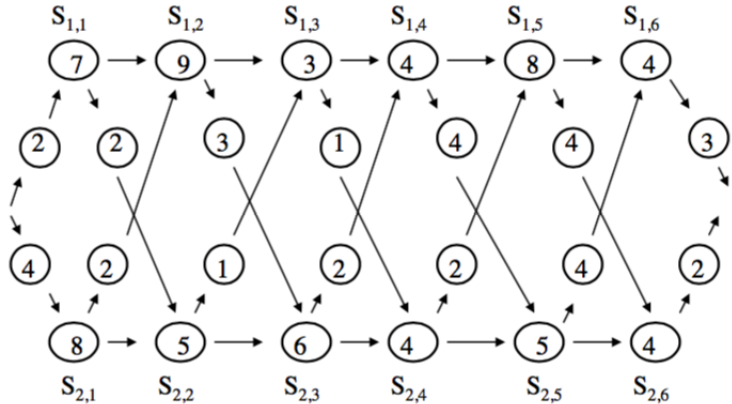
Consider the following small example:



Let's try a greedy approach (in class)

Now a dynamic programming approach (in class)

Another example



Let's try a greedy approach (in class)

Now a dynamic programming approach (in class)

Optimal substructure

Imagine our partially built widget has arrived at station s_{ij} (presumably by some optimal route). What is the optimal cost to finish the widget? Either we can stay on the same line, or we can switch lines. So the optimal must satisfy:

$$f_{ij} = \min(s_{ij} + f_{i,j+1}, s_{ij} + t_{i,j+1} + f_{3-i,j+1})$$

Once we have a recursive formula, it's trivial to write a recursive solution.

In the algorithm following, s and t are two-dimensional arrays, and i and j are indexes into the array, and n is the number of stations. i indexes which assembly line, and j indexes which station or transfer time. To be consistent with the textbook, we use 1-based indexing into the arrays.

Recursive algorithm

```
1: function RecAssembly(s, t, i, j, n)
2:     if j=n then
3:         return s[i, j] + t[i, j + 1]
4:     start_cost ← 0
5:     if j=1 then
6:         start_cost ← t[i, 1]
7:     path1_min ← start_cost + s[i, j]+ RecAssembly(s, t, i, j + 1, n)
8:     path2_min ← start_cost + s[i, j] + t[i, j + 1]+ RecAssembly(s, t, 3 − i, j + 1, n)
9:     return min(path1_min, path2_min)
```

This naive recursive algorithm has the same cost as brute force because we do two recursive calls for each station. We need to memoize.

Memoized recursive version

i and j are the only parameters that change in the naive recursive version, so our memoize array (f) needs to be big enough to handle all values of i and j.

```
1:  function MEMO(s, t, i, j, n, f)
2:      if f[i,j] ≠ −1 then return f[i,j]
3:      if j=n then
4:          f[i,j] ← s[i,j] + t[i,j+1]
5:          return f[i,j]
6:      start_cost ← 0
7:      if j=1 then start_cost ← t[i,1]
8:      path1_min ← start_cost + s[i,j]+ Memo(s,t,i,j+1,n,f)
9:      path2_min ← start_cost + s[i,j] + t[i,j+1]+ Memo(s,t,3−i,j+1,n,f)
10:     f[i,j] ← min(path1_min, path2_min)
11:     return f[i,j]
```


Iterative version

The iterative version again uses an explicit loop - looping over all possible indices in the function evaluation array.

In the version on the following page, I use a backwards approach (computing the cost for the last station first). This is the order the recursive version does it. The textbook uses a forward approach which requires thinking about the problem a bit differently. I think the backwards approach is simpler.

We also keep track of the optimal path using the variable $path$. The trick here is simply to store for each line whether the minimum stays on this line, or swaps to the other line. When we eventually get to the start on the line, we choose the minimum and trace forward the optimum solution in a separate step.

The iterative version also makes it obvious what the computational cost is.

Iterative algorithm

```
1:  function IterativeScheduler(s, t, n)
2:      initiallise f[2, n]; initiallise path[2, n]
3:      f[1, n] ← s[1, n] + t[1, n + 1]; f[2, n] ← s[2, n] + t[2, n + 1]
4:      for j from n − 1 downto 1 do
5:          for i in [1, 2] do
6:              path1_min ← s[i, j] + f[i, j + 1]
7:              path2_min ← s[i, j] + t[i, j + 1] + f[3 − i, j + 1]
8:              f[i, j] ← min(path1_min, path2_min)
9:              if path1_min < path2_min then path[i, j] ← i
10:             else   path[i, j] ← 3 − i
11:     return min(t[1, 1] + f[1, 1], t[2, 1] + f[2, 1]), BestPath(t, f, path)
```


BestPath

```
1: function BestPath(t, f, path)
2:     initiallise best_path[n]
3:     if t[1,1] + f[1,1] < t[2,1] + f[2,1] then
4:         best_path[1] ← 1
5:     else
6:         best_path[1] ← 2
7:     for j ← 2 to n do
8:         best_path[j] ← path[best_path[j − 1], j]
9:     return best_path
```


Relevant parts of textbook

Dynamic programming is discussed in general in chapter 15 which for reasons that escape me is introduced prior to simpler greedy algorithms in chapter 16.

Assembly line scheduling in particular is discussed in Section 15.1, but the approach I take here is somewhat different to the textbook. I find the way it is explained here more natural than the explanation in the textbook. But if you are struggling to understand the lecture notes, then you might find the textbook easier to follow.