Dynamic Programming
Statistics 650/750
Week 5 Thursday

Christopher Genovese and Alex Reinhart

28 Sep 2017

Announcements

- Next time: challenge 1 overview and selection
  - Project selection and start: 5 October
  - Peer Review: 19 October
  - Submission: 26 October
- Mid-semester grades
- New problems appearing include dynamic-programming

A Few More Thoughts on Graphs

Last Thursday's Notes

Illustrative Challenge: Making Change

Dr. Ecco, whom we will call a "consulting statistician," receives a visit from the director of a mint from a small country. The president of the country (who is a mathematician herself) has asked him to design the country's currency, but the president is obsessed with minimizing the number of coins that her citizens need (on average) to make a purchase.

The mint has studied the situation and has a good estimate of the distribution of item cost remainders (e.g., costs mod 100) across the country and of the costs to the mint of producing any given size of the coin set. The president believes she knows the proper trade off between number of these latter costs with the efficiency of making change.

The general question [1] is: given this inforation, can we find coin denominatios of each size that minimize the expected number of coins required for a purchase, and in turn balance the costs of each coin set size against the benefits from efficiency. Because costs can be 1 cent (assume the country uses currency units with 100 "cents" per unit), we will consider cases where "pennies" are always included, but the more general case is an interesting extension.

We distinguish two types of counting:

intuitive the coins for a given purchase cost can always be obtained by choosing the largest denomination that works at each step. (Ex: 1, 5, 10, 25 is intuitive.)

[1] Based on "Small Change for Mujikhistan" from *Doctor Ecco's Cyberpuzzles* by Dennis E. Shasha]

non-intuitive the set is not intuitive. (Ex: 1, 10, 25.)

A related question is whether non-intuitive sets are ever required given a constrained number of coins.

Exercise: For a *given candidate set* of possible coins, how do you find the optimal change to make, in the intuitive and non-intuitive cases, for every purchase cost from 1 to 99. You want to *minimize* the number of coins used to make change.

Example: Set: 1,10,25 Purchase 30: 10,10,10 better than 25,1,1,1,1,1

The more general question seems to require searching over all subsets of the allowed denominations, which would be of exponential complexity in general, though manageable for reasonably small sizes. See `ClassFiles/week5/change` for some code to do this. Can we do better?

Related question: What are some boundary test cases to consider with this code?

Overview: Dynamic Programming (DP)

Today, we consider a useful algorithmic strategy called **dynamic programming** that is based on decomposing problems into sub-problems in a particular way.

Note: The term "programming" here is used in the old sense: referring to planning, scheduling, routing, assignment – and the optimization thereof. As with "linear programming," "quadratic programming," "mathematical programming" you can think of it as a synonym for *optimization*.

And indeed, dynamic programming is useful for a wide variety combinatorial optimization problems, as we will see.

The Key Ideas Behind Dynamic Programming

1. Decompose a problem into (possibly many) smaller subproblems.
2. Arrange those subproblems in a **special ordering**.
3. Compute solutions to the subproblems in order, *storing* the solution to each subproblem for later use.
4. The solution to a subproblem *combines* the solutions to earlier subproblems in an essential way.

A simple but illuminating example

Sketch out a *recursive* algorithm for computing the Fibonacci numbers. How does it perform? Why?

```
fib(n):
   if n == 0: return 0
   if n == 1: return 1
   return fib(n-1) + fib(n-2)

fib(7) -> fib(6) -> fib(5), fib(4) -> fib(4), fib(3), fib(3), ...
          fib(5) -> fib(4), fib(3)
```

Memoization: record the value of function calls in a way that is easily indexed by the arguments

Reminder: Topological Sorting DAGS

A **topological sort** of a DAG is a linear ordering of the DAG's nodes such that if (u, v) is a directed edge in the graph, node u comes before node v in the ordering.

Example: A directed graph

network1.png

and a rearrangment showing a topological sort

network2.png
The sorted nodes are S C A B D E.
For a general DAG, how do we use DFS to do a topological sort?

```
Algorithm topological-sort:

Input: A DAG G
Output: A list of nodes representing a topological sort

Steps: Run DFS on G, configured with after_node so that
after each node is processed, we push it onto the front
of a linked list (or equivalently onto a stack).

Return the list of nodes.
```

The Key Ideas Revisited

1. Decompose a problem into smaller subproblems.

   Implicitly, each subproblem is a node in a directed graph, and there is a directed edge (u, v) in that graph when the result of one subproblem is required in order to solve the other.

   There are two equivalent choices for edge orientation in this graph:

   Flow orientation (u, v) is an edge when the result of subproblem u is required in order to solve subproblem v.

   Dependent orientation (v, u) is an edge when the result of solving subproblem v requires the result of subproblem u.

   As the names suggest, **flow** orientation describes how information flows through the graph during dynamic programming, whereas **dependent** orientation illustrates the dependence of each subproblem on others. Both are used. I tend to prefer the former, but the latter is more common.

   We will write $u \succ v$ or, equivalently, $v \prec u$ to denote the actual dependence relation regardless of which edge orientation we use pictorially.

   (To be specific, $u \succ v$ means that the result subproblem u is required to solve subproblem v. So both $u \succ v$ and $v \prec u$ imply that there is an edge between the two subproblems in the underlying DAG.)

2. Arrange those subproblems in the **topologically sorted** order of the graph.

   A topological sort of the underlying DAG yields an ordering of the subproblems. We will call this a *subproblem order*.

   If the DAG was defined with *flow orientation*, we will call this *subproblem flow order*, or **flow order** for short.

   If the DAG was defined with *dependent orientation*, we will call this *subproblem dependent order*, or **dependent order** for short.

3. Compute solutions to the subproblems in order, storing the result of each subproblem for later use if needed. This storing approach is called **memoization** or **caching**.

   One common scenario is when the subroblems are computed by a single function, and we store our previous solution by **memoizing** the function. That is, when we call the function, we check if we have called it with these particular arguments before. If so, return the previously computed value. Otherwise, compute the value and store it, marking these arguments as being previously computed.

4. The solution to a subproblem *combines* the solutions to earlier subproblems through a specific mathematical relation.

   The mathematical relationship between a subproblem solution and the solution of previous subproblems is often embodied in an equation, or set of equations, called the **Bellman equations**. We will see examples below.

Question

For the Fibonacci example we just saw, what are the subproblems? What is the DAG? What does memoizing look like?

```
Fibonacci:
   Subproblems: computing fib for smaller, particular values
   The DAG relates the fib(n) = fib(n-1) + fib(n-2)

memoizing_table = hash_table()
memoizing_original = hash_table()

function  memoize(f):
   function f_prime(...):
       arglist = list(...)
       entry = memoizing_table.lookup(arglist)

       if entry:
           return entry
       else:
           value = f(...)
           memoizing_table.insert(arglist, value)
           return value

   memoizing_original.insert(f_prime, f)
   return f_prime

fib = memoize(fib)
```

Examples

Example #1: Shortest Path in a Graph

We just saw Dijkstra's single-source shortest path algorithm; I claim that this is a dynamic programming problem.

Consider a (one-way) road network connecting sites in a town, where each path from a site to a connected site has a cost.

road network

What is the lowest-cost path from S to E? How do we find it?

Solution

Start from E and work backward.

- The best route from E to E costs 0.

- The best route from D to E costs 1.
- The best route from B to E costs 2.
- The best route from A to E costs 8.
- The best route from C to E costs 4.
- The best route from S to E costs 6. [S -> C -> D -> E]

Why does this work?

The lowest-cost path from a given node to E is a *subproblem* of the original.

We can arrange these subproblems in an ordering so that we can solve the easiest subproblems first and then solve the harder subproblems in terms of the easier ones.

What ordering?

sorted road network

This is just the *topological sort* of the DAG that we saw above. This gives us a subproblem *dependent order*

```
S, C, A, B, D, E;
```

or alternatively, a *flow order*

```
E, D, B, A, C, S.
```

Formalizing this

For nodes u in our graph, let dist(u) be the minimal cost of a path from u to E (the end node). We want dist(S). Finding dist(u) is a subproblem.

For subproblem nodes u, v with an edge $u \to v$ connecting them, let $c(u, v) \equiv c(v, u)$ be the cost of that edge.

Here is our algorithm:

1. Initialize dist(u) $= \infty$ for all u.
2. Set dist(E) $= 0$.
3. Topologically sort the graph, giving us a sequence of nodes from E to S. Call this "subproblem flow order".
4. For nodes v in subproblem *flow* order, set

$$\text{dist}(v) = \min_{u \succ v} (\text{dist}(u) + c(u, v))$$

These last equations are called the **Bellman equations**.

Let's try it.

```
Subproblem flow order is E,D,B,A,C,S, yielding:
   dist(E) =                                  0
   dist(D) = dist(E) + 1                    = 1
   dist(B) = min(dist(E) + 2, dist(D) + 1) = 2
   dist(A) = dist(B) + 6                    = 8
   dist(C) = min(dist(A) + 4, dist(D) + 3) = 4
   dist(S) = min(dist(A) + 1, dist(C) + 2) = 6
```

Exercise

Write a function `min_cost_path` that returns the minimal cost path to a target node from every other node in a weighted, directed graph, along with the minimal cost. If there is no directed path from a node to the target node, the path should be empty and the cost should be infinite.

Your function should take a representation of the graph and a list of nodes in subproblem *flow* order. You can represent the graph anyway you prefer; however, one convenient interface, especially for R users, would be:

`min_cost_path(target_node, dag_nodes_flow, costs)`

where `target_node` names the target node, `dag_nodes_flow` lists all the nodes in flow order, and costs is a *symmetric* matrix of edge weights with rows and columns arranged in flow order. Assume: costs[u,v] = Infinity if no edge btwn u,v.

Note: You can use the above as a test case. Also, be aware of the `tsort` command on the Mac or Linux command line.

```
echo "S A\nS C\nA B\nC A\nC D\nB D\nB E\nD E\n" | tsort
```

```r
constantly <- function(x) {
    return( function(z){ return(x) } )
}

min_cost_path <- function(target_node, dag_nodes_flow, costs) {
    node_count   <- length(dag_nodes_flow)
    paths        <- setNames(vector("list", node_count), dag_nodes_flow)
    dists        <- lapply(paths, constantly(Inf))
    target_index <- match(target_node, dag_nodes_flow)

    if ( !is.na(target_index) ) stop("Target node not found")

    dists[[target_node]] <- 0
    paths[[target_node]] <- c(target_node)

    for ( node_index in (target_index+1):node_count ) {
        flows_from <- target_index:(node_index-1) # indices in *flow* order

        step_cost <- unlist(dists[flows_from]) + costs[flows_from, node_index]
        best_step <- which.min(step_cost)
        min_dist  <- step_cost[best_step]

        if ( min_cost < Inf ) {
            dists[[node_index]] <- min_dist
            paths[[node_index]] <- c(dag_nodes_flow[node_index],
                                     paths[[target_index + best_step - 1]])
        }
    }
    # Note: Previous loop would be more efficient w/better graph representation
    return( list(costs=dists, paths=paths,
                 target=target_node, nodes=dag_nodes_flow, weights=costs) )
}
```

Example #2: Longest Increasing Subsequence

Given a sequence s of length n ordinals, find the longest subsequence whose elements are strictly increasing.

```
5, 2, 8, 6, 3, 6, 9, 7  ->   2, 3, 6, 9
```

Let's sketch a dynamic-programming solution for this problem. Work with a partner to answer these questions.

- What are the subproblems?
- Are they arranged in a DAG? If so, what are the relations?
- What are the Bellman equations for these subproblems?
- Sketch the DP algorithm here.
- We can find the longest length, how do we get the path?
- How would a straightforward recursion implementation perform? What goes wrong?

A Solution

- Make a graph with one node per element and a link $s_i \to s_j$ iff $i < j$ and $s_i < s_j$.
- Let L_j be length of the longest path ending in node j (plus 1 since we are counting nodes not edges).
- The sub-problems are arranged in a DAG because transitivity of $<$ implies that no path can return to a predecessor.
- Any path to node j must pass through one of j's predecessors (if it has any). Hence, $L_j = 1 + \max\{L_i : i \to j\}$.
- Initialize all the L_j's to 0, topologically sort the DAG, for every node j in subproblem order set $L_j = 1 + \max\{L_i : i \to j\}$, and return $\max(L)$
- Recursion would solve the subproblems over and over again, with many calls – exponential time in general.

  Consider the recursive approach when the sequence is sorted; then

  ```
  L_j = 1 + max(L_1, L_2, ..., L_{j-1})
  ```

  What does the tree of recursive calls look like here?

Example #3: Edit Distance between Strings

When you make a spelling mistake, you have usually produced a "word" that is *close* in some sense to your target word. What does close mean here?

The *edit distance* between two strings is the minimum number of edits – insertions, deletions, and character substitutions – that converts one string into another.

Example: Snowy vs. Sunny What is the edit distance?

```
Snowy
Snnwy
Snny
Sunny
```

How can we find the edit distance for any two strings edit(s,t)?

Another example: EXPONENTIAL vs. POLYNOMIAL

```
EXPONENTIAL   EXPONENTIA  EXPONENTIAL  EXPONENTIA
POLYNOMIAL    POLYNOMIA   POLYNOMIA    POLYNOMIAL

                          EXPONENTIA
EXPONENT                  POLYNOMI
POLYNOM
                          EXPONENTIA
                          POLYNOMIA

                          EXPONENTIAL
                          POLYNOMI

                          EXPONEN
                          P
```

```
edit(s, t):  s[1..i], t[1..j]
```

Questions

- What are the subproblems?
- Are they arranged in a DAG?
- How do we combine subproblems? (The Bellman Equations)

Answers

We will use a common strategy: prefixes to find subproblems.

Specifically, to find edit(s,t), we can create a subproblem by finding E_{ij} = edit(s[1..i],t[1..j]).

We can express these subproblem solutions in terms of smaller subproblems. Consider the last entry in each substring.

Either s_i is matched up with an extra character, or t_j is, or both characters are matched up with each other, in which case they can be the same or not. When there is a mismatch (insertion or deletion) the cost is one plus the cost of the smaller string; if the two are both present but there is a difference (substitution), the cost is 1 plus the cost with both smaller lists.

$$E_{ij} = \min(1 + E_{i-1,j}, 1 + E_{i,j-1}, (s_i \neq t_j) + E_{i-1,j-1}),$$

Notice that we have a boundary case: $E_{0j} = j$ and $E_{i0} = i$. Why? This gives us the DAG.

The elements of the DAG:

- Each pair s_i and t_j represents one node in the graph.
- Each node is linked to the three nodes corresponding to
  1. s_{i+1} and t_{j+1},
  2. s_i and t_{j+1}, and
  3. s_{i+1} and t_j.

See the Figure below for the DAG that results from comparing two specific words.

lexico.png

Application: Fast file differences

Programs diff, git-diff, rsync use such algorithms (along with related dynamic programming problem Longest Common Subsequence) to quickly find meaningful ways to describe differences between arbitrary text files.

Application: Genetic Alignment

Use edit distance logic to find the best alignment between two sequences of genetic bases (A, T, C, G). We allow our alignment to include gaps ('_') in either or both sequences.

Given two sequences, we can score our alignment by summing a score at each position based on whether the bases match, mismatch, or include a gap.

```
C G A A T G C C A A A
C A G T A A G G C C T T A A

C _ G _ A A T G C C _ A A A
C A G T A A G G C C T T A A
m g m g m m x m m m g x m m

Score = 3*gap + 2*mismatch + 9*match
```

With (sub-)sequences, S and T, let S' and T' respectively, be the sequences without the last base. There are then three subproblems to solve to align(S,T):

- align(S,T')
- align(S',T)
- align(S',T')

The score for S and T is the biggest score of:

- score(align(S,T')) + gap
- score(align(S',T)) + gap
- score(align(S',T')) + match if last characters of S,T match
- score(align(S',T')) + mismatch if last characters do not match

The boundary cases (e.g., zero or one character sequences) are easy to compute directly.

Question: Longest Common Subsequence

If we want to find the longest common subsequence (LCS) between two strings, how can we adapt the logic underlying this edit distance example to find a dynamic programming solution?

```
Again look at the last element of substring pairs.
Either:
  + They both contribute to the LCS: $D_{ij} = D_{i-1,j-1} + 1$.
  + Or at least one does not: $D_{ij} = \max(D_{i-1,j}, D_{i,j-1})$.
```