Dynamic Programming

Christopher R. Genovese

Department of Statistics & Data Science

31 Oct 2024
Session #18

Plan

Graph Traversals: Finishing

Plan

Graph Traversals: Finishing

Dynamic Programming

Plan

Graph Traversals: Finishing

Dynamic Programming

Appendix

Announcements

- Survey link `https://docs.google.com/forms/d/e/1FAIpQLSd5OfhtlpNVznFKUKFxEPQNM6ZOXh2y8tOseyTc2xs8eANxIA/viewform` Please complete asap
- `fpc` code, python 3.12
- **Reading**:
  - Dynamic Programming
  -
- Democracy Day
- **Homework**: `migit-2` due Tuesday 05 Nov, `kd-tree` Exercises #1 and #2

Plan

Graph Traversals: Finishing

Dynamic Programming

Appendix

Graphs

A **graph** is a collection of nodes and edges that describe pairwise relationships (edges) between entities (nodes).

There are many, many flavors of graphs. Particularly important features:

- Directed versus Undirected – do the edges have a direction?
- Simple versus multi – can there be more than one edge between two nodes?
- Loops versus No Loops – can there be an edge from a node to itself?
- Labeled (edge or node) versus unlabeled – is there extra information associated with nodes and/or edges?

Function and Functorial representations.

- Graph representations (adjacency list and matrix)
- Traversals: depth-first, breadth-first, priority-first, ...
- Getting more out of the traversal.

Depth-First, Breadth First, Priority First Traversal

- **Breadth-First Search** (BFS) :: visit all neighbors of the current node before visiting any of *their* neighbors.
- **Depth-First Search** (DFS) :: visit all neighbors of the next visited node before visiting the other neighbors of the *current* node.
- **Priority Search** :: visit nodes in priority order, adding neighbors at each stage

Aside:

- Stack: push, pop, isEmpty, peek
- Queue: enqueue, dequeue, peek, isEmpty
- Priority Queue: enqueue(obj, priority), dequeue, peek, isEmpty
- Deque: pushFront, popFront, pushBack, popBack, peekFront, peekBack, isEmpty

Traversal Powerup

See code in `documents/Src/graphs`.

- A traversal template
- Tracking traversal state
- Configuring the traversal: actions and stores
- Applications

Breadth First: Traversing a tree by level

> *Given a tree with n nodes, create a tree of the same shape but with the leaves numbered* 1..*n level by level from the root and left to right.*

Last time, we realized that we needed a ***queue*** to arrange the visitation. Let's build this together briefly.

Demo

A Template for Graph Traversal

Basic Applications

1. Print log of traversal as it runs (showing nodes as they are visited and processed and edges as they are traversed. (See `print-history.py`.)
2. Use DFS to detect cycles in a graph
   Idea: In an undirected graph, if there are no back edges, we have a tree – hence, no cycles. But any back edge creates a cycle. So, look for back edges.
3. A **topological sort** of a Directed Acyclic Graph (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.
   Given a DAG, how do we use DFS to do a topological sort?
4. Use DFS to count the number of "descendants" of a node.

Basic Applications

1. Print log of traversal as it runs (showing nodes as they are visited and processed and edges as they are traversed. (See `print-history.py`.)
2. Use DFS to detect cycles in a graph
   Idea: In an undirected graph, if there are no back edges, we have a tree – hence, no cycles. But any back edge creates a cycle. So, look for back edges.

```
# Graph -> NodeId -> NodeId -> State -> State
def detect_back_edge(g, from_node, to_node, ts):
    if ts.visited(to_node) and ts.parent[from_node] != to_node:
        # Found back edge
        from_label = gr.node_properties(from_node, 'label')
        to_label = gr.node_properties(to_node, 'label')

        print("Found cycle with nodes {from_n}"
              "and {to_n}".format(from_n=from_label, to_n=to_label))

        ts.finished = True
    return ts
```

3. A **topological sort** of a Directed Acyclic Graph (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.
   Given a DAG, how do we use DFS to do a topological sort?
4. Use DFS to count the number of "descendants" of a node.

Basic Applications

1. Print log of traversal as it runs (showing nodes as they are visited and processed and edges as they are traversed. (See `print-history.py`.)
2. Use DFS to detect cycles in a graph
   Idea: In an undirected graph, if there are no back edges, we have a tree – hence, no cycles. But any back edge creates a cycle. So, look for back edges.
3. A **topological sort** of a Directed Acyclic Graph (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.
   Given a DAG, how do we use DFS to do a topological sort?

```
acc = []  # pass this to traverse

# Graph -> NodeId -> State -> State
def record_processed(g, node, ts):
    if ts.processed(node):
        ts.acc.push(node)
    return ts
```

   Final acc now has a topo sort of node ids
4. Use DFS to count the number of "descendants" of a node.

Basic Applications

5. Use DFS to count the compute a path between two nodes.
6. Use BFS to find the shortest path from starting node to any node

Basic Applications

5. Use DFS to count the compute a path between two nodes.
6. Use BFS to find the shortest path from starting node to any node

```
def find_path(from_node, to_node, parents) -> Optional[list[NodeId]]
    path = []
    end = to_node

    while from_node != end and end is not None:
        path.append(end)
        end = parents[end]

    if end is not None:
        path.append(from_node)
        path.reverse()
        return path
    else:
        return None

# ts = bfs(...)
# find_path(0, 3, ts.parent)
```

Basic Applications

7. Use BFS to find the connected components of a graph.
8. Use BFS to determine if a graph is 2-colorable, or bipartite, (i.e., we can assign one of two colors to every node so that no two nodes of the same color are adjacent).

Basic Applications

7. Use BFS to find the connected components of a graph.

```
def collect_visited(graph, node, state):
    "Accumulates list of nodes as they are visited."
    state.accumulator.append(node)

def grab_component(graph, components, start, state=None):
    "Collect one connected component and reset state accumulator."
    state = graph.bfs(start, [], before_node=collect_visited, ts=state)
    components.append(state.accumulator)
    state.accumulator = []  # reset
    return state

def connected_components(g: Graph) -> list[list[NodeId]]:
    "Returns a list of connected components for a graph g"

    components = []
    ts = None

    for node in g.nodes:
        if ts.fresh(node):
            ts = grab_component(g, components, node, state=ts)
    return components
```

8. Use BFS to determine if a graph is 2-colorable, or bipartite, (i.e., we can assign one of two colors to every node so that no two nodes of the same color are adjacent).

Basic Applications

7. Use BFS to find the connected components of a graph.
8. Use BFS to determine if a graph is 2-colorable, or bipartite, (i.e., we can assign one of two colors to every node so that no two nodes of the same color are adjacent).

```
def complementary_color(color):
    return 1 - color

def check_edge(graph, node, neighbor, state):
    node_color = graph.node_properties(node, "color")
    nghb_color = graph.node_properties(neighbor, "color")

    if node_color == nghb_color:
        ts.accumulator = False  # Bipartite indicator
        ts.finished = True
    graph.update_node_properties(neighbor, color=complementary_color(node_color))

def two_coloring(g: Graph) -> List[tuple[NodeId, int]] | Literal[False]:
    "Returns a two-coloring of a graph g if bipartite, else False."
    ts = None

    for node in g.nodes:
        if ts is None or ts.fresh(node):
            g.update_node_properties(node, color=0)
            ts = bfs(g, node, True, on_edge=check_edge, ts=ts)
        if ts.finished:
            break

    if ts.accumulator:
        return [(node, g.node_properties(node, "color")) for node in g.nodes]
    return False
```

Plan

Graph Traversals: Finishing

Dynamic Programming

Appendix

The Limitations of Divide-and-Conquer

Consider this recursive calculation of Fibonacci numbers:

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

How does this perform?

The Limitations of Divide-and-Conquer

Consider this recursive calculation of Fibonacci numbers:

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

How does this perform?

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

This can be *exponential* because we end up recomputing the same value multiple times. We are essentially visiting every node of a complete tree when we don't have to.

The Limitations of Divide-and-Conquer

Consider this recursive calculation of Fibonacci numbers:

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

How does this perform?

Compare:

```
type Phi = Phi Integer Integer

instance Monoid Phi where
  munit = Phi 1 0
  mcombine (Phi a b) (Phi m n) = Phi (a*m + b*n) (a*n + b*(m + n))

fib n = extract (fastMonoidPow n (Phi 0 1))
  where extract (Phi _ b) = b
```

The Limitations of Divide-and-Conquer

Consider this recursive calculation of Fibonacci numbers:

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

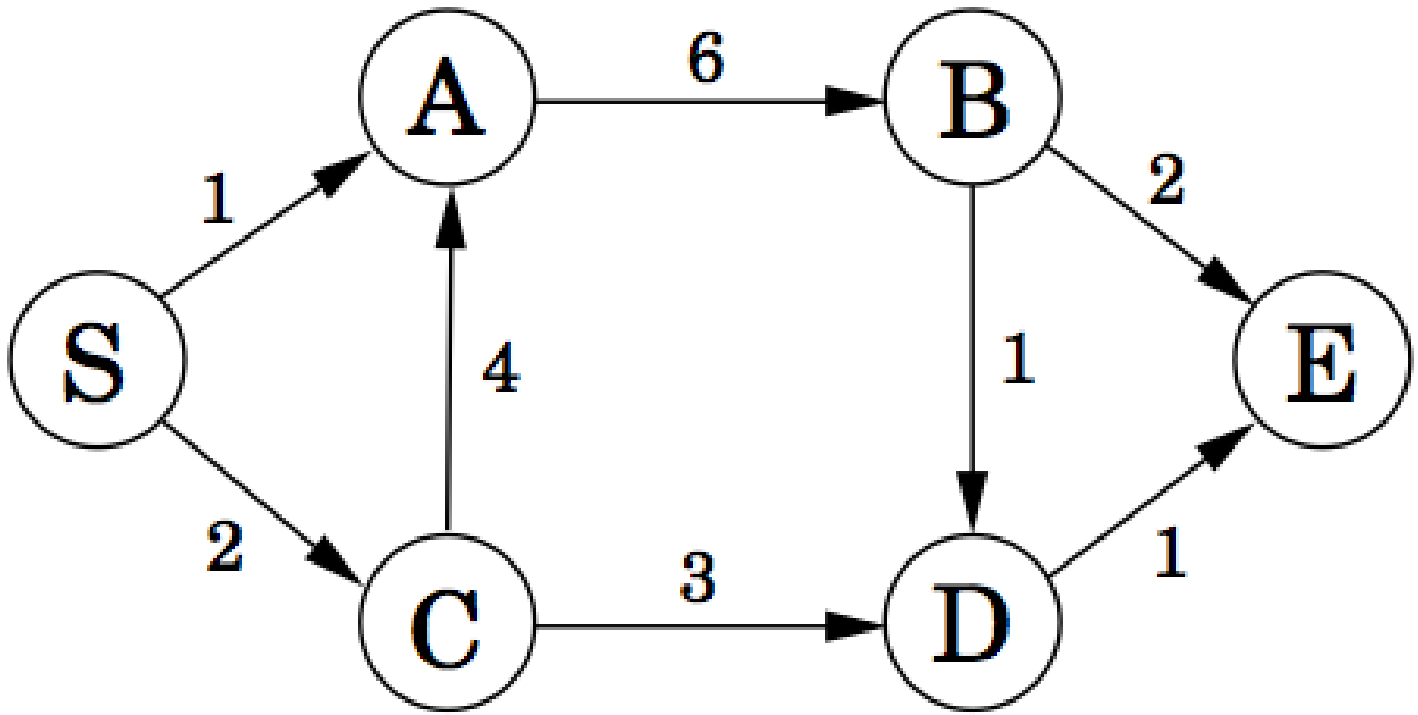
How does this perform?

Two key strategies:

- Solve the problem in a particular order (from smaller n to larger n).
- Store the solutions of problems we have already solved (memoization).

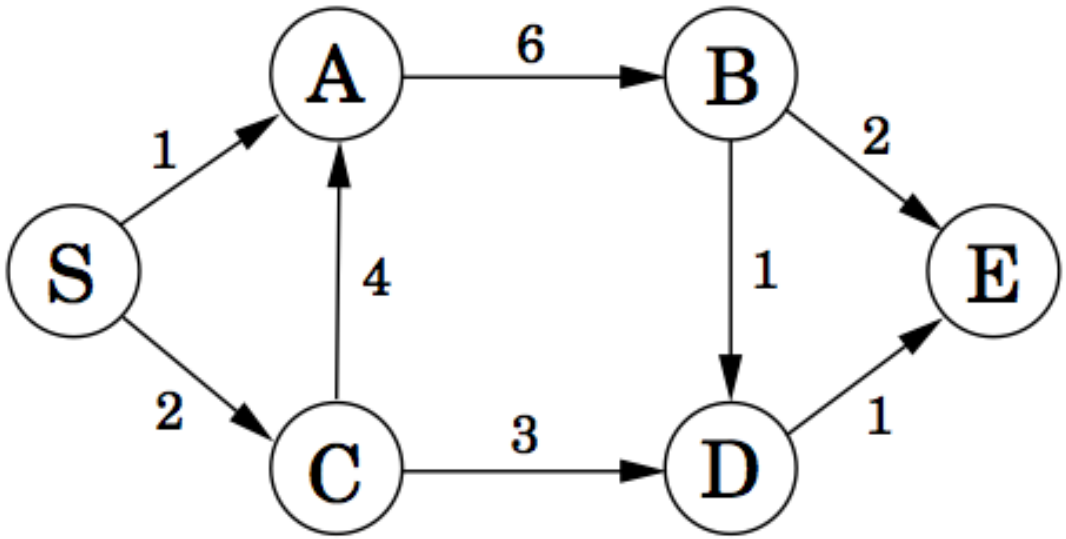
A Prototypical Problem

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



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

A Prototypical Problem



Start from E and work backward.

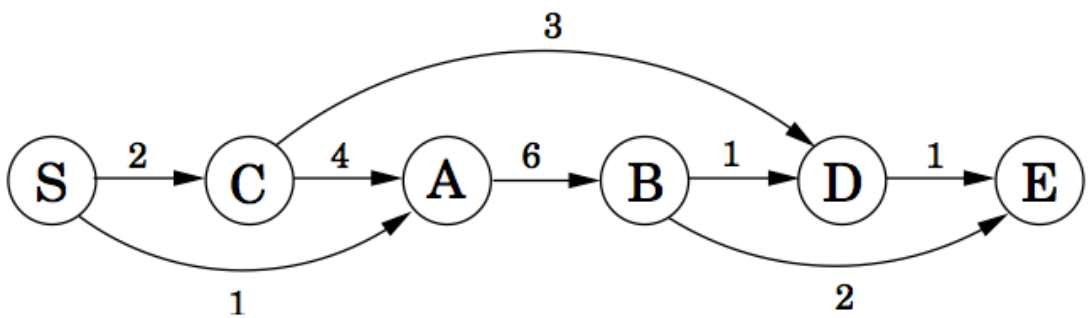
- The best route from E to E costs 0.
- The best route from D to E costs 1.
- The best route from B to E costs 2.
- The best route from A to E costs 8.
- The best route from C to E costs 4.
- The best route from S to E costs 6. [S $\rightarrow$ C $\rightarrow$ D $\rightarrow$ E]

A Prototypical Problem (cont'd)

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?



This is just a *topological sort* of the DAG. It presents the subproblems in *decreasing order*: `S`, `C`, `A`, `B`, `D`, `E`, though in practice we often handle the subproblems in *increasing order*.

Dynamic Programming

Basic Strategy:

1. Decompose a problem into (possibly many) smaller subproblems.
2. Arrange those subproblems in **the order they will be needed**.
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 a specific way.

Dynamic Programming

Detailed Strategy:

1. Decompose a problem into smaller subproblems.

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

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

   We will call the subproblem order *decreasing* if edges points from superproblems to subproblems and *increasing* if the reverse.

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**.

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.

Quick Exercise: Memoizing a Function

For the Fibonacci example, what are the sub-problems? What is the DAG? What does memoizing look like?

Quick Exercise: Memoizing a Function

```
function fib(n):
    r = array(n)

    r[1] = 0
    r[2] = 1

    for i in 3:n:
        r[i] = r[i - 1] + r[i - 2]

    return r[n]
```

Quick Exercise: Memoizing a Function

You have code for a function a function `f`. How would you memoize it?

```
function memoize(f):
    memoizing_table = hash_table()

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

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

    setattr(f_prime, 'original', f)
    return f_prime

fib = memoize(fib)
fib(50)
```

Formalizing the Shortest Path

For nodes u in our graph, let $\mathrm{dist}(u)$ be the minimal cost of a path from u to E (the end node). We want $\mathrm{dist}(S)$. Finding $\mathrm{dist}(u)$ 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:

- Initialize $\mathrm{dist}(u) = \infty$ for all u.
- Set $\mathrm{dist}(E) = 0$.
- Topologically sort the graph, giving us a sequence of nodes from S to E.
- Work through the sorted nodes in reverse. For each node v, set

$$\mathrm{dist}(v) = \min_{u \to v} (\mathrm{dist}(u) + c(u, v))$$

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

Let's try it. The decreasing order is S, C, A, B, D, E, 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 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, costs)
```

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

Note: You can use the above example as a test case.

Exercise

```
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_dist < Inf ) {
      dists[[node_index]] <- min_dist
      paths[[node_index]] <- c(dag_nodes_flow[node_index],
                               paths[[target_index + best_step - 1]])
    }
  }
```

Example: Edit Distance

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
```

Three changes transformed one into the other.

An equivalent but easier way to look at this is to think of it as an alignment problem. We use a _ marker to indicate inserts and deletions. Here are two possible alignments of Snowy and Sunny:

```
S _ n o w y
S u n n _ y

0 1 0 1 1 0   Total cost: 3
```

```
_ S n o w _ y
S u n _ _ n y

1 1 0 1 1 1 0   Total cost: 5
```

We can convince ourselves that the minimum cost here is 3, and that is the edit distance `edit(Snowy, Sunny)`.

Example: Edit Distance

But there are many possible alignments of two strings, and searching for the best among them would be costly.

Instead, we think about how to decompose this problem into sub-problems.

Consider computing `edit(s, t)` for two strings `s[1..m]`, and `t[1..n]`. The sub-problems should be smaller versios of the same problem that *help* us solve the bigger versions.

We can look at *prefixes* of the strings `s[1..i]= and =t[1..j]` as the sub-problems, and express its solutions in terms of smaller problems of the same form. Let `E_ij = edit(s[1..i], t[1..j])`.

When we find the best alignment of these strings, the last column in the table will be one of three forms

```
s[i]      _       s[i]
 _      t[j]      t[j]
```

Example: Edit Distance

The first case gives cost `1 + edit(s[1..i-1], t[1..j])`. The second case gives `1 + edit(s[1..i], t[1..j-1])`. The third case gives (s[i] != t[j]) + edit(s[1..i-1], t[1..j-1]) . Hence,

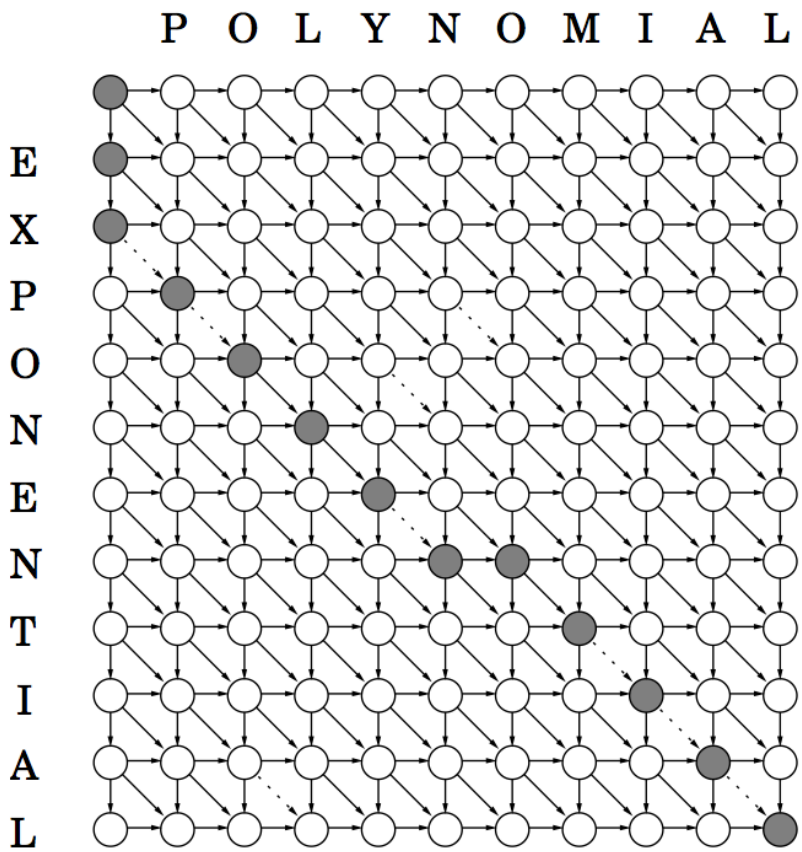
$$E_{ij} = \min(1 + E_{i-1,j}, 1 + E_{i,j-1}, d_{ij} + E_{i-1,j-1}).$$

We also have the initial conditions $E_{i0} = i$ and $E_{0j} = j$.

Hence, we can make a two dimensional table of sub-problems, which we can tackle in any order as long as E_{ij} is computed after $E_{i-1,j}, E_{i,j-1}, E_{i-1,j-1}$.

What is the DAG here? If we weight the edges, we can get generalized costs.

Example: Edit Distance



Write a function to compute the edit distance between two strings *and* a sequence of transformations of one string into the other.

What information do you need to keep track of? How do you organize the data? What types do you need?

Plan

Graph Traversals: Finishing

Dynamic Programming

Appendix

Tree Flavors

```
type BinaryTree a = Tip | Branch (BinaryTree a) a (BinaryTree a)

type RoseTree a = Node a (List (RoseTree a))

-- Heterogenous version of Rose Tree, different types on Leaf and Branch
type HTree b l = Leaf l | Branch b (NonEmptyList HTree b l)

type Trie k m a = Trie { data : Maybe a
                       , children : Map k (Trie k m a)
                       , annotation : m     -- m will be a Monoid
                       }

-- One way to think about unrooted trees; there are others.
-- Fin n is the type of integers 1..n. OrderedPairs a is like Pair a a
-- but represents pairs (x, y) where x < y.

type UnrootedTree a = forall n.
         Pair(Injection Fin(n - 1) (OrderedPairs (Fin n)), Fin n -> a)
```

Aside: A Beautiful, Lazy Solution

From Jones and Gibbon (Osaki, 2000), translated to TL1:

```
-- Given the list of next available index at each level
-- Produce a tree numbered with these at each level
-- and return the updated list

bfn : (List Int, BinaryTree a) -> (List Int, BinaryTree Int)
bfn (inds, Tip) = (inds, Tip)
bfn (Cons ind inds, BinaryTree left _ right) =
  (Cons (ind + 1) inds'', BinaryTree left' ind right')
  where (inds', left') = bfn (inds, left)
        (inds'', right') = bfn (inds', right)



-- When done, the next available index at one level is the first available
-- for the next level.  And poof, like magic

bfnum : BinaryTree a -> BinaryTree Int
bfnum t = t'
  where (ks, t') = bfn (Cons 1 ks, t)
```

Class Work: Searching with Tries

Consider a simple form of the "Trie," a special type of rose tree that can be used for efficiently associating values with a set of strings.

We consider a simplified form here

```
type Trie a = Trie { data : Maybe a
                   , children : Map Char (Trie a)
                   }
```

where `Map k v` is an associative map (e.g., dictionary) from key type `k` to value type `v`.

When a string like "foobar" (and associated value) are inserted into the trie, we start at the root node (associated with the empty string) and add a child for the first character (f), then moving to that child, add a child of that for the second character (o), and so on until the string is done. We store the value in the final node. (Picture)

We have an interface

```
empty  : Trie a
insert : String -> Trie a -> Trie a
lookup : String -> Trie a -> Maybe a
```

Let's make a simple trie implementation together.

The End