Dynamic Programming (cont'd)
Statistics 650/750
Week 6 Tuesday

Christopher Genovese and Alex Reinhart

03 Oct 2017

Announcements

A Few Thoughts on Second Languages

Why is there more than one programming language?

There is a dizzying number of programming languages, with new languages being developed regularly. Some are used by thousands, some by only a handful. Many are general purpose; others fill a very specialized niche. Why are there so many?

A few reasons:

- Evolution of ideas
- Performance trade-offs
- Specialized purposes
- Personal preferences and styles
- Different criteria for "better"
  - Expressive power
  - Ease of learning (for a novice)
  - Library support
  - Community support
  - Acceptance in the workplace
  - Standardization
  - Availability of efficient compilers
  - ...

The structure of a language and the constraints it imposes affect how you think about and approach problems. Learning a new language, especially one very different from one you are familiar with, opens up new ways of thinking. (This is a technological version of the Sapir-Whorf hypothesis.)

Let this goal, along with practical considerations, guide your choices.

Don't Fear the Syntax

A language like R, while based internally on other traditions, is syntactically related to the C-family, with its plethora of statements and braces.

Most other languages, while having different syntax and underlying concepts, are quite understandable in familiar terms with a few minor adjustments.

For example, languages like clojure, racket, common lisp, and scheme use **prefix style** for function calls: ()s are *only* used to signifify function calls, and these calls are in prefix order.

Example : Calling Functions

R	f(x, y, z, ...)
Clojure	(f x, y, z, ...)
	(f x y z ...)
Racket etc.	(f x y z ...)
Haskell	f x y z ...

So we just move one parenthesis... not that different. Haskell eschews parentheses completely and mixes prefix and infix forms.

While prefix style takes a little getting used to (especially for arithmetic operations), it has several advantages:

- Consistency
- Really easy editing and movement
- Broader range of names `count-words`, `prime?`, `my/bread&butter`
- No need for grouping or precedence
- No overloading of ()'s.
- Sugar `(+ )` `(+ 1 2 3 4 5 6 7 8)`, `(/ 2)`, `(* )` and `(* 1)` and `(* 1 2 3 4)` `(sample [1 2 3 4] :size 4 :replacement false)`
- ...
- Most importantly: Syntax allows: **Code as data**

After trying it a while, it becomes very loveable.

Another Example: Calling Functions

R:

```
add5 <- function(x) {
    return( x + 5 )
}
```

Clojure:

```
(defn add5 [x]
  (+ x 5))
```

Racket and Scheme:

```
(define (add5 x)
   (+ x 5))
```

Haskell:

```
add5 x = x + 5
```

Again, not that different overall.

Some Recommendations

C++	A powerful but complex object-oriented language, works with Rcpp	
Javascript	The heart of web-based programming	*
Clojure	An elegant functional language with immutability, concurrency, multiplatform	*
Rust	A fast and modern systems language	*
Racket	A well-designed, modern lisp with a well-supported ecosystem	*
Haskell	A pure and mind-blowing functional language	*
Java	Versatile object-oriented language	
Python	A clean language that is commonly used for data science	

Challenge Project Descriptions

Starting today, you'll be working on the challenge problem of your choice. Challenge problems are larger problems integrating several of the skills we've covered so far in the course. You can pick from the following (under the `challenges` tag in the problem bank):

word-clouds Use a randomized greedy algorithm to automatically create an attractive visualization for text data.

autocomplete-me Use divide-and-conquer algorithms to provide fast autocomplete suggestions.

dual-tree Compute approximate kernel density estimators quickly using a variant of kd-trees.

anomaly-speed Implement a tree-based anomaly-detection algorithm and use it to measure the speed of traffic in a video.

Projects will be peer-reviewed on **Thursday 19 October**, then turned in on **Thursday 26 October**.

Be sure to open a branch for your project on a clean master, or use your `new-homework` command if you've written it.

Your projects will be **peer-reviewed**: on 19 October, we'll match you with a classmate who will review and leave feedback on your pull request. You'll have to grant them access to the repository on GitHub. With their feedback, and feedback from us, you will revise your project to earn its final assessment.

Projects have an extra grading level: beyond Mastered, a project can be *Sophisticated*. Recall that you must earn Sophisticated on at least one Challenge to get an A. The Sophisticated grade implies a high level of mastery of the problem, with excellent code style, robust error handling, good choice of algorithms and data structures, thorough unit testing, and every other good programming practice we've discussed so far. More detailed rubrics are available on the individual challenges. Also, refer to the Checklists folder in the documents repository.

Dynamic Programming Continued

Brief Review

Dynamic Programming is an approach to solving combinatorial optimization problems with four key steps:

1. Decompose a problem into (possibly many) smaller **subproblems**, arranged (at least implicitly) in a DAG.
2. Arrange those subproblems in a **special ordering**, a topological ordering of the underlying DAG of subproblems.
3. Compute solutions to the subproblems in order, **storing** (aka, memoizing or caching) the solution to each subproblem for later use.
4. The solution to a subproblem **combines** the solutions to earlier subproblems in an essential way (the Bellman equations).

Activity/Example: 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
```

Three changes transformed one into the other.

Another way to look at this is how the two strings align, where we can mark insertions/deletions in this alignment. Here are two possible alignments of Snowy and Sunny

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

0 1 0 1 1 0   Total cost: 3    1 1 0 1 1 1 0   Total cost: 5
```

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

To do that, we need to identify subproblems whose solutions we can combine to solve the larger problem.

To this end, consider another example: EXPONENTIAL vs. POLYNOMIAL.

We will consider **prefixes** of each string. So consider several prefix pairs:

```
EXPONENTIAL    EXPONENTIA   EXPONENTIA
POLYNOMIA      POLYNOMIAL   POLYNOMIA

edit = x       edit = y     edit = z
```

What is the edit distance of EXPONENTIAL and POLYNOMIAL given x, y, and z?

- We can add a '_' after POLYNOMIA giving cost 1 + x.

- We can add a '_' after EXPONENTIA giving cost 1 + y.
- The L after POLYNOMIA and EXPONENTIA giving cost z.

Thus, the edit distance is: `min(1 + x, 1 + y, z)`.
These prefixes form our *subproblems*. The result is:

```
E X P O N E N _ T I A L
_ _ P O L Y N O M I A L
1 1 0 0 1 1 0 1 1 0 0 0

Edit distance: 6
```

Questions

- Formally, what are the subproblems?
- Are these subproblems 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

Activity

Get files `edit-fix.r` or `edit-fix.py` from the `documents` repository (in `ClassFiles/week6`) and rename the file to `edit.r` or `edit.py`.

Fix the missing code to compute edit distance and alignment functions. Feel free to work with someone.

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: $D_{ij} = \max(D_{i-1,j}, D_{i,j-1})$.
```

Example: Hidden Markov Models and Reinforcement Learning

Hidden Markov Models

In a Hidden Markov Model, the states of a system follow a discrete-time Markov chain with initial distribution α and transition probability matrix P, but we do not get to observe those states.

Instead we observe signals/symbols that represent noisy measurements of the state. In state s, we observe symbol y with probability Q_{sy}. These measurements are assumed conditionally independent of everything else given the state.

This model is denoted HMM$\langle \alpha, P, Q \rangle$.

Many applications:

- Natural language processing
- Bioinformatics
- Image Analysis
- Learning Sciences
- ...

If we denote the hidden Markov chain by X and the observed signals/symbols by Y, then there are fast recursive algorithms for finding the **marginal** distribution of a state given the observed signals/symbols and the parameters.

But usually, we want to reconstruct the **sequence** of hidden states. That is, we want to find:

$$\operatorname*{argmax}_{s_{0..n}} P_\theta \{X_{0..n} = s_{0..n} | Y_{0..n} = y_{0..n}\}$$

It is sufficient for our purposes to solve a related problem

$$v_k(s) = \operatorname*{argmax}_{s_{0..k-1}} P_\theta \{X_{0..k-1} = s_{0..k-1}, X_k = s, Y_{0..k} = y_{0..k}\}$$

Then we have

$$v_k(s) = Q_{s,y_k} \max_r P_{rs} v_{k-1}(r),$$

which gives us a dynamic programming solution.

$$\max_s v_n(s) = Q_{w_n,y_n} \max_r P_{r,w_n} v_{n-1}(r)$$

where $w_n = \operatorname{argmax}_s v_k(s)$.

This is a DAG of subproblems.

Reinforcement Learning

Reinforcement Learning builds on a similar representation where we have a Markov chain whose state transitions we can influence through actions. We get rewards and penalties over time depending on what state the chain is in.

We learn by interacting with our environment: making decisions, taking actions, and then (possibly much later) achieving some reward or penalty.

Reinforcement learning is a statistical framework for capturing this process.

A goal-directed learner, or **agent**, learns how to map situations (i.e., states) to actions so as to maximize some numerical reward.

- The agent must discover the value of an action by trying it.
- The agent's actions may affect not only the immediate reward but also the next state – and through that the downstream rewards as well.

These two features – *value discovery* and *delayed reward* – are important features of the reinforcement learning problem.

- Examples
  - Playing tic-tac-toe against an imperfect player.
  - The agent plays a game of chess against an opponent.
  - An adaptive controller adjusts a petroleum refinery's operation in real time to optimize some trade-off among yield, cost, and quantity.
  - A gazelle calf struggles to stand after birth and is running in less than an hour.
  - An autonomous, cleaning robot searches for trash among several rooms but must take care to return to a recharging station when power gets low.
  - Phil prepares his breakfast.

  Common themes: interaction, goals, rewards, uncertainty, planning, monitoring
- Elements of Reinforcement Learning
  - The **agent** – the entity that is learning
  - The **environment** – the context in which the agent is operating, often represented by a *state space*.
  - A **policy** – determines the agent's behavior.
    It is a function mapping from states of the environment to **actions** (or distributions over actions) to take in that state.
  - A **reward function** – defines the agent's **goal**.
    It maps each state or each state and action pair to a single number (called the *reward*) that measures the desirability of that state or state-and-action.
    The reward at any state (or state-action pair) is immediate. The agent's overall goal is to maximize the *accumulated* reward.
  - A **value function** – indicates the accumulated desirability of each state (or state-action pair).
    Roughly speaking, the value function at a state is the expected accumulated reward an agent can receive starting at that state (by choosing "good" actions). The value function captures the delayed rewards.
  - An **environment model** (Optional) – mimics the environment for planning purposes.
- Markov Decision Processes

  We model reinforcment using a *Markov Decision Process*.

  This is a discrete-time stochastic process on a state space S where the probabilities of transitions between states are governed by Markov transition probabilities for each action the agent can choose from a set A.

  The goal is to maximize the cumulative reward (possibly discounted) over time (with reward $r(X_k)$ at time k).

  Using the Markov structure of the process allows us to find good (or even optimal) decision *policies*.