Final Review

CS16: Introduction to Data Structures & Algorithms
Spring 2019
Outline

- Dynamic Programming
- PageRank
- Kruskal’s Algorithm
- Prim-Jarnik Algorithm
- Functional Programming
- Hardness
- Neural Networks
What is Dynamic Programming?

- Algorithm design paradigm/framework
  - Design efficient algorithms for optimization problems
- Optimization problems
  - “find the best solution to problem X”
  - “what is the shortest path between u and v in G”
  - “what is the minimum spanning tree in G”
- Can also be used for non-optimization problems
When is Dynamic Programming Applicable?

- **Condition #1**: sub-problems
  - The problem can be solved recursively
  - Can be solved by solving sub-problems

- **Condition #2**: overlapping sub-problems
  - Same sub-problems need to be solved many times
Sub-Problems

\[ \text{Sol}(\begin{array}{cccc}
\text{Sol}(\begin{array}{c}
\text{Sol}(\begin{array}{cc}
\text{Sol}(\begin{array}{c}
\end{array})
\end{array})
\end{array})
\end{array}) = \text{Sol}(\begin{array}{c}
\text{Sol}(\begin{array}{c}
\end{array})
\end{array}) \oplus \text{Sol}(\begin{array}{c}
\text{Sol}(\begin{array}{c}
\end{array})
\end{array}) \]

\[ \text{Sol}(\begin{array}{c}
\text{Sol}(\begin{array}{c}
\end{array})
\end{array}) = \text{Sol}(\begin{array}{c}
\text{Sol}(\begin{array}{c}
\end{array})
\end{array}) \oplus \text{Sol}(\begin{array}{c}
\text{Sol}(\begin{array}{c}
\end{array})
\end{array}) \]

\[ \text{Sol}(\begin{array}{c}
\text{Sol}(\begin{array}{c}
\end{array})
\end{array}) = \text{Sol}(\begin{array}{c}
\text{Sol}(\begin{array}{c}
\end{array})
\end{array}) \oplus \text{Sol}(\begin{array}{c}
\text{Sol}(\begin{array}{c}
\end{array})
\end{array}) \]
Overlapping Sub-Problems

\[
\text{Sol}(\begin{array}{ll}
\text{red} & \text{blue} \\
\text{blue} & \text{red}
\end{array}) = \text{Sol}(\begin{array}{l}
\text{red}
\end{array}) \oplus \text{Sol}(\begin{array}{l}
\text{blue}
\end{array})
\]

\[
\text{Sol}(\begin{array}{l}
\text{red}
\end{array}) = \text{Sol}(\begin{array}{l}
\text{blue}
\end{array}) \oplus \text{Sol}(\begin{array}{r}
\text{red}
\end{array})
\]

\[
\text{Sol}(\begin{array}{l}
\text{blue}
\end{array}) = \text{Sol}(\begin{array}{l}
\text{red}
\end{array}) \oplus \text{Sol}(\begin{array}{r}
\text{blue}
\end{array})
\]

Why solve red twice?
Why solve blue twice?
When is Dynamic Programming Applicable?

- Core idea
  - Decompose problem into its sub-problems
  - and if sub-problems are overlapping then
  - solve each sub-problem once and store the solution
  - use stored solution when you need to solve sub-problem again
Steps to Solving a Problem w/ DP

› What are the **sub-problems**?

› What is the “**magic**” step?
  
  ‣ Given solution to a sub-problem…
  
  ‣ …how do I *combine* them to get solution to the problem?

› Which **(topological) order** on sub-problems can I use?
  
  ‣ so that solutions to sub-problems available before I need them

› Design iterative **algorithm**
  
  ‣ that solves sub-problems in order and stores their solution
Shortest Path in Layered Directed Graph

- Layered
  - edge \((x, y)\) only if \(x < y\)
- Negative & positive weights
- vertex degree at most \(d\)
Outline

- Dynamic Programming
- PageRank
- Kruskal’s Algorithm
- Prim-Jarnik Algorithm
- Functional Programming
- Hardness
- Neural Networks
The Basic PageRank Algorithm

- At every round
  - each vertex splits its PR evenly among its outgoing edges
  - each vertex receives PR from all its incoming edges
  - this is done using an *update rule* which is run on every vertex
- The update rule for Basic PageRank is:

\[
PR(v) = \sum_{u \in \text{in}(v)} \frac{PR(u)}{|\text{out}(u)|}
\]
Basic PageRank: Example 2

Round 1

A → B: 0.2
B → C: 0.2
B → D: 0.2
A → C: 0.2
A → D: 0.2
C → A: 0.1
C → E: 0.1
D → B: 0.2
D → E: 0.2
E → D: 0.2

12
Basic PageRank: Example 2

Round 2
Basic PageRank: Example 2

Round 3
Basic PageRank

- Basic PageRank doesn’t work for certain graphs
  - e.g.: graphs with sinks or with cycles with no outgoing edges
  - all the pagerank gets trapped there

- How do we handle “rank traps”?

- Water flows down from high elevation to low elevation
  - why doesn’t all the water end up at the lowest points on Earth?
  - because some of the water evaporates…
  - …and rains back down on the high elevation points
Handling Rank Traps

- Let's make some of the pagerank evaporate!
  - We need a new *update rule*
- In basic update rule, nodes gave all their pagerank to neighbors
- In new update rule, a node will
  - give a \(d\) fraction of its PR to its neighbors (split evenly)
  - give a \(1-d\) fraction of its PR to all other nodes (split evenly)
- this guarantees that pagerank doesn’t accumulate anywhere
- \(d\) is usually set to \(.85\)

What happens if the node is a sink?
Disappearing PageRank

\[
\frac{1}{2} \quad (1-d)/4
\]

\[
d/2
\]

\[
(1-d)/4
\]

\[
A \quad B
\]
Disappearing PageRank

The sum of the pageranks does not sum to 1:

\[
\frac{(1 - d)}{2} + \frac{1}{2} = 1 - \frac{d}{2} < 1
\]

since \(0 < d < 1\)

We lost \(d/2\) of B's pagerank when we updated.
Handling Sinks

- There are several ways to handle sinks
- The simplest is to modify the graph as follows
  - if $\mathbf{v}$ is a sink, add an edge from $\mathbf{v}$ to every other node in the graph
- Then use the update rule we described on slide #38
The Real PageRank Algorithm

- Add edges connecting every sink to every other node
- At every round, each vertex
  - splits a $d$ fraction of its PR evenly among its outgoing edges
  - splits a $(1-d)$ fraction of its PR evenly among all nodes
- $d$ is called the damping factor & is usually set to $0.85$
The Real PageRank Algorithm

- At every round the PR of each vertex $v$ is updated using:

$$
PR(v) = \left( \sum_{u \in \text{in}(v)} \frac{d \cdot PR(u)}{|\text{out}(u)|} \right) + \sum_{u \in V} \frac{(1 - d) \cdot PR(u)}{|V|}
$$

1. nodes with edges pointing to $v$
2. $d$ fraction of $u$’s PR
3. number of edges leaving $u$
4. $(1-d)$ fraction of $u$’s PR

$$
= \left( d \cdot \sum_{u \in \text{in}(v)} \frac{PR(u)}{|\text{out}(u)|} \right) + \frac{1 - d}{|V|} \cdot \sum_{u \in V} PR(u)
$$

$$
= \frac{1 - d}{|V|} + d \cdot \sum_{u \in \text{in}(v)} \frac{PR(u)}{|\text{out}(u)|}
$$
The Real PageRank

- Runtime of a round $O(|E|)$
  - iterate over every vertex $v$ and over all incoming edges to $v$
- How many rounds should we run?
- Until the pageranks “stabilize”
  - pageranks stop changing even though we run more rounds
- We can prove that
  - if we run for large enough number of rounds then pageranks will stabilize
  - that number could be very large for some graphs…
  - …but in practice it’s usually reasonable
Alternative Sink Handling

› You can also handle sinks without modifying the graph

› but you need a slightly different update rule

\[
PR(v) = \frac{1 - d}{|V|} + d \cdot \sum_{u \in \text{in}(v)} \frac{PR(u)}{|\text{out}(u)|} + \sum_{u \in \text{sinks}(G)} \frac{d \cdot PR(u)}{|V|}
\]

\[
= \frac{1 - d}{|V|} + d \cdot \left( \sum_{u \in \text{in}(v)} \frac{PR(u)}{|\text{out}(u)|} + \sum_{u \in \text{sinks}(G)} \frac{PR(u)}{|V|} \right)
\]
Outline

- Dynamic Programming
- PageRank
- Kruskal’s Algorithm
- Prim-Jarnik Algorithm
- Functional Programming
- Hardness
- Neural Networks
Kruskal’s Algorithm

- Sort edges by weight in increasing order
- For each edge in sorted list
  - If adding edge does not create cycle…
  - …add it to MST
- Stop when you have gone through all edges
Kruskal

- How can we tell if adding edge will create cycle?
- Start by giving each vertex its own “cloud”
- If both ends of lowest-cost edge are in same cloud
  - we know that adding the edge will create a cycle!
- When edge is added to MST
  - merge clouds of the endpoints
Example

edges = [(C, E), (D, F), (B, C), (E, F), (B, D), (A, B), (A, D), (B, E), (B, F)]
Example

edges = [(D,F), (B,C), (E,F), (B,D), (A,B), (A,D), (B,E), (B,F)]
Example

edges = [(B,C),(E,F),(B,D),(A,B),(A,D),(B,E),(B,F)]
Example

edges = [(E, F), (B, D), (A, B), (A, D), (B, E), (B, F)]
Example

edges = [(B,D), (A,B), (A,D), (B,E), (B,F)]
Example

edges = [(A, B), (A, D), (B, E), (B, F)]

BD cannot be added because it would lead to a cycle
Example

edges = [(A, D), (B, E), (B, F)]
Example

AD cannot be added because it would lead to a cycle

edges = [(B,E), (B,F)]
Example

BE cannot be added because it would lead to a cycle

edges = [(B,F)]
Example

BF cannot be added because it would lead to a cycle

edges = [ ]
function \textbf{kruskal}(G):
    // Input: undirected, weighted graph G
    // Output: list of edges in MST
    for vertices \( v \) in G:
        makeCloud\( (v) \) // put every vertex into its own set
    MST = []
    Sort all edges
    for all edges \( (u,v) \) in G sorted by weight:
        if \( u \) and \( v \) are not in same cloud:
            add \( (u,v) \) to MST
            merge clouds containing \( u \) and \( v \)
    return MST
Implementing Clouds: Union-Find

- Let's rethink notion of clouds
  - think of clouds as small trees
  - cloud is represented by the root of its tree
- Every vertex in these trees has
  - a parent pointer that leads up to root of the tree
  - a rank that measures how deep the tree is
Example

edges = [(C,E), (D,F), (B,C), (E,F), (B,D), (A,B), (A,D), (B,E), (B,F)]
Example

edges = [(D, F), (B, C), (E, F), (B, D), (A, B), (A, D), (B, E), (B, F)]
Example

edges = [(B,C), (E,F), (B,D), (A,B), (A,D), (B,E), (B,F)]
Example

edges = [(E,F), (B,D), (A,B), (A,D), (B,E), (B,F)]
Example

edges = [(B,D), (A,B), (A,D), (B,E), (B,F)]
Example

edges = [(A,D), (B,E), (B,F)]
Example

edges = [(A,D), (B,E), (B,F)]
Implementing Union-Find

- At start of Kruskal
  - every node is put into own cloud

```javascript
// Decorates every vertex with its parent ptr & rank
function makeCloud(x):
    x.parent = x
    x.rank = 0
```

![Diagram showing nodes A and B with rank 0]
Merging two Clouds/Trees

- Suppose A is in cloud/tree A and B is in cloud/tree B
- To merge the two make B point to A

- Given two clouds which one should point to the other?
  - if roots have same rank then it doesn’t matter
  - if roots have different rank then it does matter
Merging two Clouds/Trees

- If roots have different rank
  - make lower-ranked root point to higher-ranked root
  - then update rank

- How do we update ranks?
  - For clouds of size 1 root always has rank 0
  - For clouds of size larger than 1 we increment rank only when merging clouds of same rank
Merging two Clouds/Trees

- Merging trees with same rank
Merging two Clouds/Trees

- Merging trees with same rank
Merging two Clouds/Trees

- Merging trees with different ranks

![Diagram showing merging trees with different ranks]
Merging two Clouds/Trees

- Merging trees with different ranks

```
1
 A
 B C D
 0 0 0
```

```
0
 B C D
 0

E
 0
```

```
1
 A
 B C D
 0 0 0
```

```
E
```

```
Outline

- Dynamic Programming
- PageRank
- Kruskal’s Algorithm
- Prim-Jarnik Algorithm
- Functional Programming
- Hardness
- Neural Networks
Prim-Jarnik Algorithm

- Traverse $G$ starting at any node
  - Maintain priority queue of nodes
  - set priority to weight of the edge that connects them to MST
- Un-added nodes start with priority $\infty$
- At each step
  - Connect the node with lowest cost
  - Update ("relax") neighbors as necessary
- Stop when all nodes added to MST
Example

\[ PQ = [ (0, A), (\infty, B), (\infty, C), (\infty, D), (\infty, E), (\infty, F) ] \]
Example

Dequeue from PQ and update neighbors

\[
\text{PQ} = [(4, B), (5, D), (\infty, C), (\infty, E), (\infty, F)]
\]
Example

PQ = [(4, C), (4, D), (6, E), (8, F)]
Example

\[ PQ = [(2,E), (4,D), (8,F)] \]
Example

$$PQ = [(4,D),(4,F)]$$

Dequeue from PQ and update neighbors
Example

Dequeue from PQ and update neighbors

\[ PQ = [(3, F)] \]
Example

PQ = []

Dequeue from PQ and update neighbors
Example
function \texttt{prim}(G):
    // Input: weighted, undirected graph G with vertices V
    // Output: list of edges in MST
    for all v in V:
        v.cost = \infty
        v.prev = null
    s = a random v in V // pick a random source s
    s = 0
    MST = []
    PQ = PriorityQueue(V) // priorities will be v.cost values
    while PQ is not empty:
        v = PQ.removeMin()
        if v.prev != null:
            MST.append((v, v.prev))
        for all incident edges \((v,u)\) of \(v\) such that \(u\) is in PQ:
            if u.cost > (v,u).weight:
                u.cost = (v,u).weight
                u.prev = v
                PQ.decreaseKey(u, u.cost)
    return MST
Runtime Analysis

- Decorating nodes with distance and previous pointers is $O(|V|)$
- Putting nodes in PQ is $O(|V|\log|V|)$ (really $O(|V|)$ since $\infty$ priorities)
- While loop runs $|V|$ times
  - removing vertex from PQ is $O(\log|V|)$
  - So $O(|V|\log|V|)$
- For loop (in while loop) runs $|E|$ times in total
  - Replacing vertex’s key in the PQ is $\log|V|$
  - So $O(|E|\log|V|)$
- Overall runtime
  $$O(|V| + |V|\log|V| + |V|\log|V| + |E|\log|V|)$$
  $$= O((|E| + |V|)\log|V|)$$
Outline

- Dynamic Programming
- PageRank
- Kruskal’s Algorithm
- Prim-Jarnik Algorithm
- Functional Programming
- Hardness
- Neural Networks
Functional Programming

- Anonymous functions
  - \texttt{lambda x: x + 1}
  - \texttt{lambda x: 100*x}
- Higher order functions
  - map: applies a function to a list of elements
  - reduce: applies a binary function to pairs of elements from a list, and accumulates the results in an accumulator
Map

\[\text{map}(\lambda x: x-2, \ [11, 9, 24, -5, 34, 4])\]
Reduce

- `reduce(lambda x, y: x+y, [1,2,3], 0)`
  - `acc = 0 + 1 = 1`
    - `acc = 1 + 2 = 3`
      - `acc = 3 + 3 = 6`
Practice Problems

- function that turns list of nouns into adverbs
  - loud $\rightarrow$ loudly

- function that sums total length of a list of strings
  - [“hi”, “cs16”] $\rightarrow$ 6

- function that counts number of times “dog” appears in a list

- function that removes numbers less than 10 from a list of ints
Practice Problem Answers

- map(lambda s: s+"ly", list)
- reduce(lambda acc, s: acc+len(s), list, 0)
- reduce(lambda acc, s: acc+1 if s == "dog" else acc, list, 0)
- reduce(lambda acc, x: acc+[x] if x > 10 else acc, list, [])
Hardness

- Hardness of a problem is defined by the runtime of the best known solution

- Hardness of comparative sorting?
  - $O(n \log n)$
  - There are $O(2^n)$ sorting algorithms, but the sorting problem is not hard

- Problems that have polynomial time solutions are tractable
  - $O(n)$, $O(n^2)$, $O(n^{500})$, ...

- Problems with super-polynomial time solutions are intractable
  - $O(n!)$, $O(2^n)$, $O(n^n)$, ...
Categories of Hardness

- **NP**: problems whose solutions can be verified in poly-time
- **P**: problems whose solutions can be found in poly-time
- **NP-Complete**: the hardest problems in NP
  - Solutions can be verified in poly time
  - if poly-time solution exists for problem, then a poly-time solution exists for all problems in NP
  - not known whether there exist any polynomial time algorithms to solve them
- **NP-Hard**: at least as hard as NP-complete
  - Don’t know if solutions can be verified in poly time
  - if poly-time solution exists for problem, then a poly-time solution exists for all problems in NP
Artificial Neuron

\[ y = \varphi \left( \sum_{i=1}^{m} x_i \cdot w_i - b \right) = \varphi (\mathbf{x}^T \cdot \mathbf{w} - b) = \begin{cases} 0, & \text{if } \mathbf{x}^T \cdot \mathbf{w} - b \leq 0 \\ 1, & \text{if } \mathbf{x}^T \cdot \mathbf{w} - b > 0 \end{cases} \]
Perceptron Network

- $-1$
- $x_1$
- $x_2$
- $x_3$
- $x_4$

$\mathbf{t}$

update weights

$\mathbf{Comp}$

$N$

$y_1$

$y_2$

$y_3$
Perceptron Network

\[ w_i = w_i + \eta (t - y_1) \cdot x_i \]

\[ \varphi \left( \sum_{i=0}^{4} w_i \cdot x_i \right) \]

\[ y_1 \]

\[ y_2 \]

\[ y_3 \]
Multi-Layer Perceptron

Inputs \( x_1, x_2, x_3, x_4 \)

Hidden Layer

Output Layer

\( y_1, y_2, y_3 \)