Set Partitioning

John McDonough

Language Technologies Institute,
Machine Learning for Signal Processing Group,
Carnegie Mellon University

April 9, 2012

Introduction

- In this lecture, we consider *breadth first search* (BFS) and *depth first search* (DFS).
- We will prove that BFS determines the shortest pass for unweighted graphs.
- We will also prove that DFS is useful for topologically sorting nodes.
- We also consider an algorithm for *set partitioning* that can also be used to minimize a weighted-finite state automaton.
- Finally, we will begin to consider an algorithm for *weight pushing*.

  Coverage: Cormen, Leiserson, Rivest and Stein (2009); Aho, Hopcroft, Ullman (1974), Section 4.13.

Graph Searches

- The most basic operation on a graph is to search through it to discover all vertices.
- The vertices are assigned a color during the search:
  - A node v that has not been previously discovered is *white*.
  - A node v that has been discovered, but whose adjacency list has not been fully explored is *gray*.
  - After the adjacency list of v has been fully explored, it is *black*.
  - The distance d[v] of a node v is the number of edges traversed from the start node s in order to reach v.
  - The predecessor $\pi[v]$ of a node v is the node from whose adjacency list v was discovered.

Breadth First Search

- Assume we have a directed graph $G = (V, E)$ where every $v \in V$ is initially white, and a first-in-first-out queue **Q**.
- The breadth first search (BFS) proceeds according to:

```
color[s] ← Gray
d[s] ← 0
π[s] ← NULL
push s on Q
while |Q| > 0 :
    pop u from Q
    for v ∈ adj[u] :
        if color[v] == White:
            color[v] ← Gray
            d[v] ← d[u] + 1
            π[v] ← u
            push v on Q
    u.color ← Black
```

Shortest Paths

- For a given source vertex $s \in V$, define the distance from s to some $v \in V$ as the number of arcs traversed going from s to v.
- Define the shortest-path distance $\delta(s, v)$ as the smallest possible distance of all paths from s to v.
- A path from s to v of length $\delta(s, v)$ is said to be a shortest path.
- A shortest path from s to v is not necessarily unique.

Shortest Path

- **Lemma 22.1:** Let $G = (V, E)$ be a directed graph, and let $s \in V$ be an arbitrary vertex. Then given any edge $(v, w) \in E$, it holds

$$\delta(s, w) \leq \delta(s, v) + 1.$$

- **Proof:** If v is reachable from s, then w must also be reachable from s. In this case, the shortest path from s to w cannot be longer than $\delta(s, v)$ plus one for the edge (v, w).

Distances Computed by BFS

Lemma 22.2: Let $G = (V, E)$ be a directed graph. Assume that the BFS is run beginning from the source vertex $s \in V$. Upon termination, the value d[v] computed by the BFS for every $v \in V$ satisfies d[v] $\geq \delta(s, v)$.

Proof of Lemma

- Make the inductive hypothesis $\mathrm{d}[u] \geq \delta(s, u)$.
- Each $\mathrm{d}[u]$ is set exactly once and never changed.
- Let $v \in V$ denote a node discovered while exploring $\mathrm{adj}[u]$.
  - Basis: The hypothesis clearly holds for the source vertex s given the assignment in Line 01.
  - Induction: Let $v \in V$ denote a vertex that is discovered while expanding the adjacency list of $u \in V$. The inductive hypothesis implies $\mathrm{d}[u] \geq \delta(s, u)$. Hence, $\mathrm{d}[v] = \mathrm{d}[v] + 1 \geq \delta(s, v) + 1 \geq \delta(s, v)$.

Distinct Values Maintained in the Queue

Lemma 22.3: Suppose that during the execution of BFS on a graph $G = (V, E)$, the queue Q contains the vertices $\{v_1, v_2, \ldots, v_r\}$, where v_1 is the head of Q and v_r is the tail. Then, $\mathrm{d}[v_r] \leq \mathrm{d}[v_1] + 1$ and $\mathrm{d}[v_i] \leq \mathrm{d}[v_{i+1}]$ for $i = 1, 2, \ldots, r - 1$.

Theorem: Correctness of BFS

- Let $G = (V, E)$ be a directed graph. Assume that the BFS is performed beginning from the source vertex $s \in V$. Upon termination, for every $v \in V$, $\mathrm{d}[v] = \delta(s, v)$. Moreover, one of the shortest paths from s to v is the path from s to $\pi[v]$, followed by the edge $\pi[v] \to v$.
- **Proof:** Proceeds by induction on sets of the form $V_k = \{v \in V : \delta(s, v) = k\}$.

Recursive Function visit(u)

- Assume we have a directed graph $G = (V, E)$ where every $v \in V$ is initially white, and let time denote a global time stamp.
- Define the recursive function `visit`(u) for some $u \in V$.

```
def visit(u):
  color[u] ← Gray          # u has been discovered
  discover[u] ← time ← time + 1
  for v in adj[u]:          # explore all edges of u
    if color[v] == White:
      π[v] ← u
      visit(v)
  color[u] ← Black # u done, paint it black
  finish[u] ← time ← time + 1
```

Depth First Search

Pseudocode for a complete *depth first search* (DFS) is given below.

```
def dfs(V, E):
  for u in V:
    color[u] ← White
    π[u] ← NULL
  time ← 0
  for u in V:
    if color[u] == White:
      visit(u)
```

Parenthesis Theorem

In any depth-first search of a (directed or undirected) graph $G = (V, E)$, for any two vertices u and v, exactly one of the following conditions holds:

- the intervals $[\text{discover}[u], \text{finish}[u]]$ and $[\text{discover}[v], \text{finish}[v]]$ are entirely disjoint, and neither u nor v is a descendant of the other in any depth first forest;
- the interval $[\text{discover}[u], \text{finish}[u]]$ is contained entirely within $[\text{discover}[v], \text{finish}[v]]$, and u is a descendant of v in a depth-first tree.
- the interval $[\text{discover}[v], \text{finish}[v]]$ is contained entirely within $[\text{discover}[u], \text{finish}[u]]$, and v is a descendant of u in a depth-first tree.

Topological Sort

- Let us define a *directed acyclic graph* (dag) $G = (V, E)$ as a digraph that contains no cycles.
- A *topological sort* is a linear ordering of all $v \in V$ such that if $u \to v \in E$, then u appears before v in the ordering.
- A topological sort can be performed with the following steps:
  1. Call `dfs`(G) to determine the finishing times `finish`$[v]$ for each $v \in V$.
  2. As each v is finished, insert it into the front of a linked list.
- Upon termination, the linked list contains the topologically sorted vertices.

Correctness of Topological Sort

Theorem 22.12: For a graph $G = (V, E)$, the algorithm described on the last slide provides a correct topological sort of the nodes.

Sets

- A set is a collection of distinguishable objects known as members or elements.
- That x is a member of the set S is denoted as $x \in S$ and read as “x is in S.”
- Two sets A and B are equal, which is denoted as $A = B$, iff they contain the same elements. For example, $\{1, 2, 3, 1\} = \{1, 3, 2\} = \{3, 2, 1\}$.
- Frequently encountered sets have special notations:
  - $\emptyset$ denotes the empty set.
  - **Z** denotes the set of integers, $\{..., 2, 1, 0, 1, 2, ...\}$.
  - **R** denotes the set of real numbers.
  - **N** denotes the set of natural numbers, $\{0, 1, 2, ...\}$.

Set Operations

- The *intersection* of sets A and B is the set $A \cap B = \{x : x \in A \text{ and } x \in B\}$.
- The *union* of sets A and B is the set $\{A \cup B = \{x : x \in A \text{ or } x \in B\}$.
- The *difference* between two sets A and B is the set $AB = \{x : x \in A \text{ and } x \notin B\}$.

Subsets

- If $x \in A$ implies $x \in B$, then we say A is a subset of B and write $A \subseteq B$.
- A set A is a proper subset of B when $A \subseteq B$, but $A \neq B$.
- For two sets A and B, $A = B$ if and only if $A \subseteq B$ and $B \subseteq A$.
- The number of elements in a set A is denoted as $|A|$.
- A set A has $2^{|A|}$ subsets including $\emptyset$.
- The power set of A, denoted as 2^A, is the set of all subsets of A.

Relations

- An ordered pair is denoted as (a, b). The ordered pair (a, b) is not the same as the ordered pair (b, a).
- The Cartesian product $A \times B$ of two sets is the set $\{(a, b) : a \in A \text{ and } b \in B\}$.
- A binary relation R on two sets A and B is a subset of the Cartesian product $A \times B$.
- For $(a, b) \in R$, we typically write aRb.
- That R is binary relation on A implies R is a subset of $A \times A$.

Example: “Less than” is a binary relation on the natural numbers given by $\{(a, b) : a, b \in N \text{ and } a < b\}$.

Linear Order

- A *total* or *linear order* R on a set A is a relation whereby for all $a, b \in A$ either aRb or bRa.
- In other words, every pairing of elements from A can be related by R.
- For example, is a linear order on the set of natural numbers.
- The function “is a descendant of” is not a linear order on the set of human beings, as there are pairs of individuals neither of whom is descended from the other.

Equivalence Relations

- Recall that we defined an equivalence relation xR_Ly for a language L when either xz and yz belong to L or both do *not* belong.
- The *index* is the number of equivalence classes in a language L.
- An equivalence relation R_L whereby xzR_Lyz follows from xR_Ly is known as *right invariant*.

Myhill-Nerode Theorem

The following statements are equivalent:

1. The set $L \subseteq \Sigma^*$ is accepted by a finite-state automaton.
2. L is the union of equivalence classes of a right invariant equivalence relation with finite index.
3. The equivalence relation can be defined as follows: xR_Ly holds if and only if xz is in L when yz is in L. Then L has a finite index.

Coarsest Partition

- Consider a set S and an initial partition π of S into disjoint blocks $\{B_1, B_2, \ldots, B_p\}$.
- There is also given a function f on S.
- The task is to find the coarsest partition $\pi' = \{E_1, E_2, \ldots, E_q\}$ such that
  1. π' is consistent with π (that is, each E_i is a subset of some B_j, and,
  2. a and b in E_i implies $f(a)$ and $f(b)$ are in some E_j.
- We then call π' the coarsest partition of S compatible with π and f.

Naive Solution

- Let B_i be a block.
- Examine $f(a)$ for each a in B_i.
- B_i is partitioned so that a and b are in the same block if and only if $f(a)$ and $f(b)$ are in the same block.
- This process is iterated until no further refinements are possible.

Example

- Let $S = \{1, 2, \ldots, n\}$, and let $B_1 = \{1, 2, \ldots, n-1\}$, $B_2 = \{n\}$ be the original partition.
- Define the function f on S as

$$f(i) \triangleq \begin{cases} i+1, & \text{for } 1 \leq i < n \\ n, & \text{for } i = n. \end{cases}$$

- On the first iteration, B_1 is partitioned into $\{1, 2, \ldots, n-2\}$ and $\{n-1\}$.
- This iteration requires $n-1$ steps because each element in B_1 must be examined.
- On the next iteration, we partition $\{1, 2, \ldots, n-2\}$ into $\{1, 2, \ldots, n-3\}$ and $\{n-2\}$.

Running Time of the Naive Solution

- A total of $n - 2$ such iterations are required, whereby the ith iteration requires $n - i$ steps, for a total of

$$\sum_{i=1}^{n-2} 1 = \frac{n(n-1)}{2} - 1$$

  steps.
- The problem with the naive solution is that refining each block requires $\mathcal{O}(n)$ steps, even if only a single element is removed.
- We would like to develop an algorithm whereby refining a block into two subblocks requires time proportional to the smaller subblock.
- This will result in a $\mathcal{O}(n \log n)$ algorithm.

Better Solution

- For each $B \subseteq S$, let $f^{-1}(B) = \{b | f(b) \in B\}$.
- The naive algorithm partitions a block B_i by the values of $f(a)$ for $a \in B_i$.
- Instead, let us partition with respect to B_i those blocks B_j which contain at least one element in $f^{-1}(B_i)$ and one element not in $f^{-1}(B_i)$.
- That is, each B_j is partitioned into the sets $\{b | b \in B_j \text{ and } f(b) \in B_i\}$, and $\{b | b \in B_j \text{ and } f(b) \notin B_i\}$.

Result of Partitioning

- Once we have partitioned with respect to B_i, we need not partition again with respect to B_i unless B_i is itself split.
- If initially $f(b) \in B_i$ for each element $b \in B_j$, and B_i is split into B_i' and B_i'', then we can partition B_j with respect to either B_i' or B_i''.
- That is, we partition with respect to B_i those blocks B_j which contain at least one element in $f^{-1}(B_i)$ and one element not in $f^{-1}(B_i)$.
- This follows because $\{b | b \in B_j \text{ and } f(b) \in B_i'\}$ is the same as $B_i - \{b | b \in B_j \text{ and } f(b) \in B_i''\}$.

Conventional Automaton

Let define a conventional automaton *without* weights.

Definition (finite-state machine)

A FSM is a 5-tuple $A = (\Sigma, Q, E, i, F)$ consisting of

- an *alphabet* Σ,
- a finite set of states Q,
- a finite set of *transitions* $E \subseteq Q \times (\Sigma \cup \{\epsilon\}) \times Q$,
- a *initial state* $i \in Q$,
- and a set of *end states* $F \subseteq Q$.

Conventional Automaton (cont'd.)

Definition

A transition $e = (p[e], l[e], n[e]) \in E$ consists of

- a previous state $p[e] \in Q$,
- a next state $n[e] \in Q$,
- a label $l[e] \in \Sigma$,

A final state $q \in F$ may have an associated label $a \in \Sigma$.

Problem Statement

- Consider a FSM with the set of states Q.
- We wish to partition Q into subsets $M = \{Q_i\}$ such that $\forall\ a : \exists e_1 = (p_1, a, n_1), e_2 = (p_2, a, n_2) \in E$, it holds

$$p_1, p_2 \in Q_j \Rightarrow n_1, n_2 \in Q_i \tag{1}$$

  for some i.
- We seek the *coarsest partition* $\{Q_i\}$ of Q, which is by definition the partion with fewest elements, that satisfies (1).

Problem Statement (cont'd.)

- Let ν be a partition of Q and let f be a function mapping $Q \times \Sigma$ to Q. In the present case, f is defined implicitly through the transitions $E \subseteq Q \times (\Sigma \cup \{\epsilon\}) \times Q$.
- For each $Q_i \in \nu$ define the sets

$$\text{symbol}(Q_i) = \{a \in \Sigma : \exists e = (p, a, n) \in E, n, p \in Q\}, \quad (2)$$

$$f^{-1}(Q_i, a) = \{p \in Q : \exists e = (p, a, n) \in E, n \in Q_i\}. \quad (3)$$

- So defined symbol(Q_i) is subset of symbols used as input labels on at least one edge into a node in Q_i.
- Similarly, $f^{-1}(Q_i, a)$ is the set of nodes having at least one transition labeled with a into a node in Q_i.

Pseudocode

Pseudocode for the partitioning algorithm is shown below:

```
def partition():
  Q_0 ← Q − F
  Q_1 ← F
  push Q_0 on S
  push Q_1 on S
  n ← 1
  while |S| > 0:
    pop P from S
    for a in symbol(P):
      for Q_j such that Q_j ∩ f^{-1}(P, a) ≠ ∅ and Q_j ⊈ f^{-1}(P, a):
        n += 1
        Q_n ← Q_j ∩ f^{-1}(P, a)
        Q_j ← Q_j − Q_n
        if Q_j ∈ S:
          push Q_n on S
        else:
          if |Q_n| < |Q_j|:
            push Q_n on S
          else:
            push Q_j on S
```

Discussion

- We will say the set $T \subseteq Q$ is *safe* for ν if for every $B \in \nu$, either $B \subseteq f^{-1}(T, a)$ or $B \cap f^{-1}(T, a) = \emptyset \; \forall \; a \in \Sigma$.
- The key of the algorithm is the partitioning of Q_j in Lines 11–12, which ensures that there are no transitions of the form $e_1 = (p_1, a, n_1)$ and $e_2 = (p_2, a, n_2)$, where either $p_1, p_2 \in Q_j$ or $p_1, p_2 \in Q_n$, for which (1) does not hold.
- Hence, Lines 12–13 ensure that P is safe for the resulting partition, inasmuch as if $Q_j \cap f^{-1}(P, a) \neq \emptyset$ for some Q_j, then either $Q_j \subseteq f^{-1}(P, a)$, or else Q_j is split into two blocks, the first of which is a subset of $f^{-1}(P, a)$, and the second of which is disjoint from that subset.
- For reasons of efficiency, the smaller of Q_j and Q_n is placed on **S** in Lines 16–19, unless Q_j is already on **S**, in which case Q_n is placed on **S** in Lines 13–14 regardless of whether or not $|Q_n| < |Q_j|$.

Set Partitioning Lemma

Aho *et. al* (1974) proved the following lemma.
Lemma (set partitioning): After the algorithm in the Listing terminates, every block Q_i in the resulting partition ν' is safe for the partition ν'.

Definition: Closed Semi-Ring

A *closed semiring* is a system $S \triangleq (\Sigma, \oplus, \otimes, \bar{0}, \bar{1})$ where Σ is a set of elements, $\oplus$ and $\otimes$ are binary operations on elements of Σ, satisfying the following properties:

1. $(\Sigma, \oplus, \bar{0})$ is a *monoid*, which implies it is *closed* under $\oplus$, and $\oplus$ is *associative*, and $\bar{0}$ is the *identity*. Likewise, $(\Sigma, \otimes, \bar{1})$ is a monoid. Moreover, we will assume $\bar{0}$ is an *annihilator* on $\otimes$; i.e., $a \otimes \bar{0} = \bar{0} \otimes a = \bar{0}$.
2. $\oplus$ is *commutative*; it may also be *idempotent* such that $a \oplus a = a$.
3. $\otimes$ *distributes* over $\oplus$, such that $a \otimes (b \oplus c) = a \otimes b \oplus a \otimes c$, and $(b \oplus c) \otimes a = b \otimes a \oplus c \otimes a$

Examples of Semirings: Tropical Semiring

- In ASR we typically use one of two semirings, depending on the operation.
- The *tropical semiring* $(\mathbb{R}^+, \min, +, \infty, 0)$, where $\mathbb{R}^+$ denotes the set of non-negative real numbers, is useful for finding the shortest path through a search graph.
- The set $\mathbb{R}^+$ is used in the tropical semiring because the hypothesis scores represent negative log-likelihoods.
- The two operations on weights correspond to the multiplication of two probabilities, which is equivalent to addition in the negative log-likelihood domain, and discarding all but the lowest weight, such as is done by the Viterbi algorithm.

Examples: Log-Probability Semiring

- The *log-probability semiring* $(\mathbb{R}^+, \oplus_{\text{log}}, +, \infty, 0)$ differs from the tropical semiring only inasmuch as the min operation has been replaced with the *log-add operation* $\oplus_{\text{log}}$, which is defined as

$$a \oplus_{\text{log}} b \triangleq -\log(e^{-a} + e^{-b}).$$

- The log-probability semiring is typically used for the weight pushing equivalence transformation discussed later.

Diagram of Weight Pushing

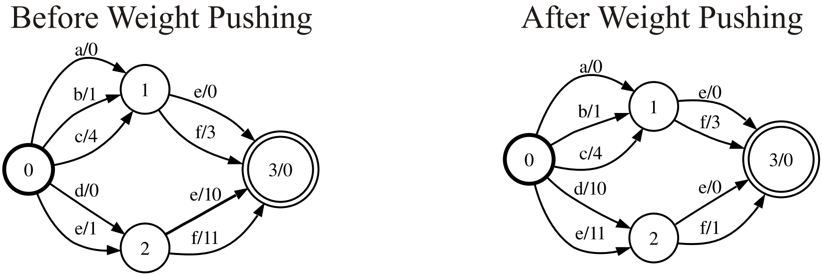

Figure: Weight pushing over the tropical semiring for a simple transducer.

Potential Function

- The weight pushing algorithm proposed begins with the definition of a *potential function* $V : Q \to \mathcal{K} - \{\bar{0}\}$.
- The weights of the transducer are then reassigned according to

$$\begin{aligned} \lambda &\leftarrow \lambda \otimes V(i), \\ \forall\, e \in E, w[e] &\leftarrow [V(p[e])]^{-1} \otimes (w[e] \otimes V(n[e])), \\ \forall\, f \in F, \rho(f) &\leftarrow [V(f)]^{-1} \otimes \rho[f]. \end{aligned}$$

- This reassignment has no effect on the weight assigned to any accepted string, as each weight from V is added and subtracted once.

Potential Function (cont'd.)

- For optimal weight pushing, we assign a potential to a state q to be equal to the weight of the shortest path from q to the set of final states F, such that

$$V(q) = \bigoplus_{\pi \in P(q)} w[\pi],$$

  where $P(q)$ denotes the set of all paths from q to F.
- The general all pairs shortest path algorithm is too inefficient for weight pushing on very large transducers.
- Instead an *approximate* shortest path algorithm is used.

Psuedocode for Calculating the Potential Function

```
def shortestDistance():
  for j in 1 to |Q|:
    d[j] ← r[j] ← 0̄
  Q ← { i }
  while |Q| > 0:
    pop q from Q
    R ← r[q]
    r[q] ← 0̄
    for e ∈ E[q]:
      if d[n[e]] ≠ d[n[e]] ⊕ (R ⊗ w[e]):
        d[n[e]] ← d[n[e]] ⊕ (R ⊗ w[e])
        r[n[e]] ← r[n[e]] ⊕ (R ⊗ w[e])
        if n[e] ∉ Q:
          push n[e] on Q
  d[i] ← 1̄
```

Psuedocode (cont'd.)

- The algorithm functions by first assigning all states q a potential of $\bar{0}$ in Lines 01–02, and placing the initial state i on a queue **Q** of states that are to be *relaxed* in Line 03.
- For each node q, the current potential $d[q]$ as well as the amount of weight $r[q]$ that has been added since the last relaxation step are maintained.
- When q is popped from **Q**, all nodes $n[e]$ that can be reached from the adjacency list $E[q]$ are tested in Line 09 to determine whether they should be relaxed.

Psuedocode (cont'd.)

- The relaxation itself occurs in Lines 10 and 11. Thereafter the relaxed node $n[e]$ is placed on **Q** if not already there in Lines 12 and 13.
- The algorithm terminates when **Q** is depleted.
- The approximation in this algorithm involves the test in Line 09, which, strictly speaking, must always be true implying, that the algorithm will never terminate.
- In practice, however, a small threshold on the deviation from equality can be set so that the algorithm terminates after a finite number of relaxations.

Psuedocode (cont'd.)

- Before calculating the potential of each node, it is necessary to first *reverse* the graph.
- This implies that for every edge $e = (p, l_i, l_o, w, n)$ in the original graph R there will be an edge $e_{\text{reverse}} = (n, l_i, l_o, w, p)$ in R_{reverse}.
- More formally, given a graph $G = (V, E)$ with weight function $w : E \to \mathbf{R}$, and a set of final states $F \subset V$, consider a directed, weigted graph $G' = (V', E')$ with initial state i, and

$$V' \triangleq V \cup \{i\},$$

$$F' \triangleq \{s\},$$

$$E' \triangleq \{v \to u : u, v \in V \text{ and } u \to v \in E\} \cup \{i \to f : f \in F\}.$$

Summary

- In this lecture, we considered *breadth first search* (BFS) and *depth first search* (DFS).
- We proved that BFS determines the shortest pass from the source node to every other node for unweighted graphs.
- We also proved that DFS is useful for topologically sorting nodes.
- We considered an algorithm for *set partitioning* that can also be used to minimize a weighted-finite state automaton.
- Finally, we began to consider an algorithm for *weight pushing*.
- Next lecture, we will see how these algorithms can be used to construct a search graph from several knowledge sources.