Class Meeting #16

*COS 226 — Spring 2018*

Mark Braverman

(based in part on slides by Kevin Wayne)
Greedy algorithms

A class of algorithms, typically for solving an optimization problem:

• Attain goal with fewest resources
• Get the most out of given fixed resources

Examples:

• Produce a given amount of change with fewest coins
• Schedule classes in fewest possible rooms
• Perform the largest possible number of jobs on a machine
Online decision making

• In classical algorithms, we are given an input and need to produce an output.
Online decision making

In online decision making, need to make irrevocable decisions based on what we’ve seen so far.
Online decision making

• In online decision making, need to make irrevocable decisions based on what we’ve seen so far.

• Examples:
  • Games
  • Control
  • Prediction tasks in Machine Learning
Online vs. offline

• Which one will perform better?
  • Offline clearly better.

• Why bother with online?
  • Often no choice...

• When we have a choice?
  • Conceptually simpler.
  • Sometimes lead to optimal performance (often with some preprocessing).
  • When we do have an optimal algorithm that operates by making local irrevocable decisions, such an algorithm is called a greedy algorithm.
Greedy algorithms

• Make local, irrevocable, decisions
• Work when we can answer the following question easily:
  • Given the current state, can I commit to part of the output right now?
Goal. Given U. S. currency denominations \{ 1, 5, 10, 25, 100 \}, devise a method to pay amount to customer using fewest coins.

Ex. 34¢.

Cashier’s algorithm. At each iteration, add coin of the largest value that does not take us past the amount to be paid.

Ex. $2.89.
Cashier's algorithm

At each iteration, add coin of the largest value that does not take us past the amount to be paid.

\textbf{CASHIERS-ALGORITHM} \((x, c_1, c_2, \ldots, c_n)\)

\textbf{SORT} n coin denominations so that \(0 < c_1 < c_2 < \ldots < c_n\).

\(S \leftarrow \emptyset\). \hspace{1cm} \text{multiset of coins selected}

\textbf{WHILE} \(x > 0\)

\(k \leftarrow \text{largest coin denomination } c_k \text{ such that } c_k \leq x\).

\textbf{IF} (no such \(k\))

\textbf{RETURN} “no solution.”

\textbf{ELSE}

\(x \leftarrow x - c_k\).

\(S \leftarrow S \cup \{k\}\).

\textbf{RETURN} \(S\).
Cashier's algorithm (for arbitrary coin denominations)

**Q.** Is cashier’s algorithm optimal for any set of denominations?

**A.** No. Consider U.S. postage: 1, 10, 21, 34, 70, 100, 350, 1225, 1500.

- Cashier’s algorithm: \( 140\,\text{¢} = 100 + 34 + 1 + 1 + 1 + 1 + 1 + 1 \).
- Optimal: \( 140\,\text{¢} = 70 + 70 \).

**A.** No. It may not even lead to a feasible solution if \( c_1 > 1 \): 7, 8, 9.

- Cashier’s algorithm: \( 15\,\text{¢} = 9 + \_ \).
- Optimal: \( 15\,\text{¢} = 7 + 8 \).

bad, irredeemable decisions
Optimality of cashier’s algorithm (for U.S. coin denominations)

**Theorem.** Cashier’s algorithm is optimal for U.S. coins \{ 1, 5, 10, 25, 100 \}.

**Pf. Succeed by not failing.**

Enough to show that we never regret adding a coin to the change. Therefore enough to show that:

1. Any amount above $1$ has a $1$ coin in optimal solution
2. Any amount above $0.25$ but below $1$ has a quarter in optimal solution
3. Any amount above $0.10$ but below $0.25$ has a dime in optimal solution
4. Any amount above $0.05$ but below $0.10$ has a nickel in optimal solution

Prove those by observing that \#pennies \leq 4, \#nickels \leq 1, \#dimes \leq 2, \#quarters \leq 3.
Interval scheduling

- Job $j$ starts at $s_j$ and finishes at $f_j$.
- Two jobs are **compatible** if they don’t overlap.
- Goal: find maximum subset of mutually compatible jobs.
Interval scheduling: greedy algorithms

**Greedy template.** Consider jobs in some natural order. **Take** each job provided it’s compatible with the ones already taken.

- **[Earliest start time]** Consider jobs in ascending order of $s_j$.
- **[Earliest finish time]** Consider jobs in ascending order of $f_j$.
- **[Shortest interval]** Consider jobs in ascending order of $f_j - s_j$.
- **[Fewest conflicts]** For each job $j$, count the number of conflicting jobs $c_j$. Schedule in ascending order of $c_j$. 
Interval scheduling: greedy algorithms

**Greedy template.** Consider jobs in some natural order. Take each job provided it's compatible with the ones already taken.

counterexample for earliest start time

counterexample for shortest interval

counterexample for fewest conflicts
Interval scheduling: earliest-finish-time-first algorithm

**Earliest-Finish-Time-First** \((n, s_1, s_2, ..., s_n, f_1, f_2, ..., f_n)\)

**Sort** jobs by finish times and renumber so that \(f_1 \leq f_2 \leq ... \leq f_n\).

\(S \leftarrow \emptyset.\) \(\text{set of jobs selected}\)

\(\text{For } j = 1 \text{ to } n\)

\(\text{If } (\text{job } j \text{ is compatible with } S)\)

\(S \leftarrow S \cup \{j\}.\)

\(\text{Return } S.\)

**Proposition.** Can implement earliest-finish-time first in \(O(n \log n)\) time.

- Keep track of job \(j^*\) that was added last to \(S\).
- Job \(j\) is compatible with \(S\) iff \(s_j \geq f_{j^*}\).
- Sorting by finish times takes \(O(n \log n)\) time.

**Note:** Can be implemented in an online fashion?

No, because need to know the future to tell whether to accept a job or not.

*Yes, if allowed to drop jobs halfway through (and lose the credit)*
Interval scheduling: analysis of earliest-finish-time-first algorithm

**Theorem.** The earliest-finish-time-first algorithm is optimal.

**Pf.** **Succeed by not failing.**

- Assume greedy made first irredeemable mistake at step $r+1$.
- Let $i_1, i_2, \ldots, i_k$ denote set of jobs selected by greedy.
- Let $j_1, j_2, \ldots, j_m$ denote set of jobs in an optimal solution with $i_1 = j_1, i_2 = j_2, \ldots, i_r = j_r$

![Diagram showing greedy and optimal schedules](image-url)
Theorem. The earliest-finish-time-first algorithm is optimal.

Pf. Succeed by not failing.

- Assume greedy made an irredeemable mistake at step r+1.
- Let $i_1, i_2, \ldots, i_k$ denote set of jobs selected by greedy.
- Let $j_1, j_2, \ldots, j_m$ denote set of jobs in an optimal solution with $i_1 = j_1, i_2 = j_2, \ldots, i_r = j_r$
Recap: greedy algorithms

• Greedy algorithms process the input in some order, and make irrevocable decisions.
• They succeed by never making decisions that are irreversibly wrong.
• Design decisions:
  • Order of processing
  • What to be greedy about
Recap: online decision making

- Online decision making algorithms operate on only part of the input, which arrives in order we can’t control.
- Typically cannot attain “optimal in hindsight” performance, the goal is to approximate its performance.
- All online decision algorithms are greedy in some sense.
- Algorithm design reduces to two goals:
  (1) Objective function design: “what to be greedy about?”
  (2) Optimization: “how to find the best next step for the objective?”
Graph algorithms

BFS vs. level-order-traversal of a binary search tree.
Level-order traversal of a binary tree.

- Process root.
- Process children of root, from left to right.
- Process grandchildren of root, from left to right.
- ...

level-order traversal: SET A R C H M
Q1. Given binary tree, how to compute level-order traversal?

```java
public Iterable<Key> levelOrder() {
    Queue<Key> keys = new Queue<Key>();
    Queue<Node> queue = new Queue<Node>();
    queue.enqueue(root);
    while (!queue.isEmpty()) {
        Node x = queue.dequeue();
        if (x == null) continue;
        keys.enqueue(x.key);
        queue.enqueue(x.left);
        queue.enqueue(x.right);
    }
    return keys;
}
```

level-order traversal: S E T A R C H M
Graph algorithms

BFS vs. level-order-traversal of a binary search tree.

```java
// BFS from single source
private void bfs(Digraph G, int s) {
    Queue<Integer> q = new Queue<Integer>();
    marked[s] = true;
    distTo[s] = 0;
    q.enqueue(s);
    while (!q.isEmpty()) {
        int v = q.dequeue();
        for (int w : G.adj(v)) {
            if (!marked[w]) {
                distTo[w] = distTo[v] + 1;
                marked[w] = true;
                q.enqueue(w);
            }
        }
    }
}
```

```java
public Iterable<Key> levelOrder() {
    Queue<Key> keys = new Queue<Key>();
    Queue<Node> queue = new Queue<Node>();
    queue.enqueue(root);
    while (!queue.isEmpty()) {
        Node x = queue.dequeue();
        if (x == null) continue;
        keys.enqueue(x.key);
        queue.enqueue(x.left);
        queue.enqueue(x.right);
    }
    return keys;
}
```
Cycle detection in digraphs

An example of timing analysis (similar to WordNet readme).

```java
public DirectedCycle(Digraph G) {
    marked = new boolean[G.V()];
    onStack = new boolean[G.V()];
    edgeTo = new int[G.V()];
    for (int v = 0; v < G.V(); v++)
        if (!marked[v] && cycle == null) dfs(G, v);
}
```

Running time?
V (always) + cost of dfs calls.
Cost of dfs(G, v) calls?

```java
private void dfs(Digraph G, int v) {
    onStack[v] = true;
    marked[v] = true;
    for (int w : G.adj(v)) {
        // short circuit if directed cycle found
        if (cycle != null) return;
        // found new vertex, so recur
        else if (!marked[w]) {
            edgeTo[w] = v;
            dfs(G, w);
        }
        // trace back directed cycle
        else if (onStack[w]) {
            cycle = new Stack<Integer>();
            for (int x = v; x != w; x = edgeTo[x]) {
                cycle.push(x);
            }
            cycle.push(w);
            cycle.push(v);
            assert check();
        }
    }
    onStack[v] = false;
}
```

Each edge visited at most once throughout the execution of DirectedCycle(..)
Cost: \( \leq E \)
Exactly once if no cycles.

Run at most once, throughout the execution of DirectedCycle(..)
Cost: \( \leq V \)
Cycle detection in digraphs

```java
public DirectedCycle(Digraph G) {
    marked = new boolean[G.V()];
    onStack = new boolean[G.V()];
    edgeTo = new int[G.V()];
    for (int v = 0; v < G.V(); v++)
        if (!marked[v] && cycle == null) dfs(G, v);
}
```

Running time?

V (always) + cost of dfs calls.

**Worst case:** $O(V+E)$;

**Best case:** $O(V)$;

Best case if there are no cycles: $O(V+E)$.
Example: exam problem from Spring’12

Given an edge-weighted digraph $G$ the bottleneck capacity of a path is the minimum weight of an edge on the path.

(a) Given an edge-weighted digraph $G$, two distinguished vertices $s$ and $t$, and a threshold value $T$, design an algorithm to find any one path from $s$ to $t$ of bottleneck capacity greater than or equal to $T$ or report that no such path exists. The order of growth of the worst case running time of your algorithm should be $E + V$.

(b) Using the subroutine from (a), design an algorithm to find a maximum bottleneck capacity path from $s$ to $t$ in an edge-weighted digraph $G$. The order of growth of the worst case running time of your algorithm should be $(E + V) \log E$.

Note: solution is on the course webpage (Final Spring’12)