Recap: Greedy Algorithms

Interval Scheduling
- Goal: Maximize number of meeting requests scheduled in single conference room
- Greedy Algorithm: Sort by earliest finish time
- Running Time: $O(n \log n)$

Interval Partitioning
- Goal: Minimize number of classrooms needed to assign all lectures
- Greedy Algorithm: Sort by earliest start time
- Running Time: $O(n \log n)$

Dijkstra’s Shortest Path Algorithm
- Invariants: minimum distance $d(u)$ to all nodes in explored set $S$
- Greedy Choice: Add node $v$ to $S$ with minimum value $d(u)$
- Running Time: $O(m + n \log n)$ with Fibonacci Heap

$\pi(v) = \min_{u \in S} d(v) + \ell_v$

Remarks about Dijkstra’s Algorithm
- Yields shortest path tree from origin $s$
- Shortest path from $s$ to every other node $v$

Maximum Capacity Path Problem
- Each edge $e$ has capacity $c_e$ (e.g., maximum height)
- Capacity of a path is minimum capacity of any Edge in path
- Goal: Find path from $s$ to $t$ with maximum capacity
- Solution: Use Dijkstra
- Small Modification

$\pi(v) = \max_{e : (u,v) \in E} \min \{ c_e, \pi(u) \}$

4.2 Scheduling to Minimize Lateness
Minimizing Lateness: Greedy Algorithms

Greedy template. Consider jobs in some order.

- [Shortest processing time first] Consider jobs in ascending order of processing time $t_j$.
- [Earliest deadline first] Consider jobs in ascending order of deadline $d_j$.
- [Smallest slack] Consider jobs in ascending order of slack $d_j - t_j$.

Minimizing Lateness: Inversions

Def. Given a schedule $S$, an inversion is a pair of jobs $i$ and $j$ such that:
- $i < j$ (i.e., $d_i < d_j$),
- $j$ scheduled before $i$.

Observation. Greedy schedule has no inversions.

Claim. If a schedule (with no idle time) has an inversion, it has one with a pair of inverted jobs scheduled consecutively.
Minimizing Lateness: Inversions

**Def.** Given a schedule $S$, an **inversion** is a pair of jobs $i$ and $j$ such that: $i < j$ (i.e., $d_i < d_j$) but $j$ scheduled before $i$.

**Claim.** If a schedule (with no idle time) has an inversion, it has one with a pair of inverted jobs scheduled consecutively.

**Proof:** Let $(i, j)$ be inversion with smallest number of jobs scheduled between $j$ and $i$. Suppose for contradiction that some job $k$ is scheduled between $j$ and $i$.

- **Case 1:** $d_k < d_j$ implies $(k, j)$ is inversion (Contradiction)
- **Case 2:** $d_k > d_j > d_i$, $(i, k)$ is inversion (Contradiction)

---

Minimizing Lateness: Analysis of Greedy Algorithm

**Theorem.** Greedy schedule $S$ is optimal.

**Pf.** Define $S^*$ to be an optimal schedule that has the fewest number of inversions, and let’s see what happens.

- Can assume $S^*$ has no idle time.
- If $S^*$ has no inversions, then $S = S^*$.
- If $S^*$ has an inversion, let $i-j$ be an adjacent inversion.
  - swapping $i$ and $j$ does not increase the maximum lateness and
    strictly decreases the number of inversions
  - this contradicts definition of $S^*$

---

Greedy Analysis Strategies

**Greedy algorithm stops ahead.** Show that after each step of the greedy algorithm, its solution is at least as good as any other algorithm’s.

**Structural.** Discover a simple "structural" bound asserting that every possible solution must have a certain value. Then show that your algorithm always achieves this bound.

**Exchange argument.** Gradually transform any solution to the one found by the greedy algorithm without hurting its quality.

**Other greedy algorithms.** Kruskal, Prim, Huffman, …

---

4.3 Optimal Caching

**Caching.**
- Cache with capacity to store $k$ items.
- Sequence of $m$ item requests $d_1, d_2 \ldots, d_m$.
- Cache hit: item already in cache when requested.
- Cache miss: item not already in cache when requested: must bring requested item into cache, and evict some existing item, if full.

**Goal.** Eviction schedule that minimizes number of cache misses.

**Ex:** $k = 2$, initial cache = $ab$, requests: $a, b, c, a, c, a, b$.

**Optimal eviction schedule:** 2 cache misses.
Optimal Offline Caching: Farthest-In-Future

Farthest-in-future. Evict item in the cache that is not requested until farthest in the future.

Theorem. [Bellady, 1960s] FF is optimal eviction schedule.

Pf. Algorithm and theorem are intuitive; proof is subtle.

Reduced Eviction Schedules

Def. A reduced schedule is a schedule that only inserts an item into the cache in a step in which that item is requested.

Intuition. Can transform an unreduced schedule into a reduced one with no more cache misses.

Farthest-In-Future: Analysis

Pf. (continued)

Case 1: (d is not in the cache; S'0 exists; S exists f ≠ e).
- begin construction of S' from S by evicting e instead of f

Let j be the first time after j+1 that S and S' take a different action, and let g be item requested at time j+

Farthest-In-Future: Analysis

Let j' be the first time after j+1 that S and S' take a different action, and let g be item requested at time j'.

- Case 3: g ≠ e. Can’t happen with Farthest-In-Future since there must be a request for f before e.
- Case 3b: g = f. Element f can’t be in cache of S, so let e' be the element that S exists.
  - if e' ≠ e, S' accesses f from cache; now S and S' have same cache.
  - if e' = e, S' accesses e' and brings e into the cache; now S and S' have the same cache

Note: S' is no longer reduced, but can be transformed into a reduced schedule that agrees with SFF through step j+1.

Reduced Eviction Schedules

Claim. Given any unreduced schedule S, can transform it into a reduced schedule S' with no more cache misses.

Pf. (by induction on number of unreduced items)

- Suppose S brings d into the cache at time t, without a request.
- Let c be the item S evicts when it brings d into the cache.
- Case 1: d evicted at time t', before next request for d.
- Case 2: d requested at time t' before d is evicted.

- Case 3: (d is not in the cache; S'0 exists; S exists f ≠ e).
  - begin construction of S' from S by evicting e instead of f

Farthest-In-Future: Analysis

Let j be the first time after j+1 that S and S' take a different action, and let g be item requested at time j'.

- Case 3a: g = e. Can’t happen with Farthest-In-Future since there must be a request for f before e.
- Case 3b: g = f. Element f can’t be in cache of S, so let e' be the element that S exists.
  - if e' ≠ e, S' accesses f from cache; now S and S' have same cache.
  - if e' = e, S' accesses e' and brings e into the cache; now S and S' have the same cache

Note: S' is no longer reduced, but can be transformed into a reduced schedule that agrees with SFF through step j+1.
Farthest-In-Future: Analysis

Let \( j' \) be the first time after \( j+1 \) that \( S \) and \( S' \) take a different action, and let \( g \) be item requested at time \( j' \).

**Case 3:** \( g \neq e, f \). \( S \) must evict \( e \).

Make \( S' \) evict \( f \); now \( S \) and \( S' \) have the same cache.

\[ S \]
\[ s \]
\[ \text{same} \]
\[ S' \]
\[ s' \]
\[ \text{same} \]

**Case 3:** \( g = e \) or \( f \). \( S \) must evict either \( e \) or \( f \), or both.

Make \( S' \) evict \( e \) or \( f \); now \( S \) and \( S' \) have the same cache.

\[ S \]
\[ s \]
\[ \text{same} \]
\[ S' \]
\[ s' \]
\[ \text{same} \]

Caching Perspective

Online vs. offline algorithms:
- **Offline**: full sequence of requests is known a priori.
- **Online (reality)**: requests are not known in advance.
- Caching is among most fundamental online problems in CS.

**LIFO**. Evict page brought in most recently.

**LRU**. Evict page whose most recent access was earliest.

**Theorem**: FF is optimal offline eviction algorithm.

**Greedy Algorithms**

- **Kruskal’s algorithm.** Start with \( T = \emptyset \). Consider edges in ascending order of cost. Insert edge \( e \) in \( T \) unless doing so would create a cycle.
- **Reverse-Delete algorithm.** Start with \( T = E \). Consider edges in descending order of cost. Delete edge \( e \) from \( T \) unless doing so would disconnect \( T \).
- **Prim’s algorithm.** Start with some root node \( s \) and greedily grow a tree \( T \) from \( s \). At each step, add the cheapest edge \( e \) to \( T \) that has exactly one endpoint in \( T \).

**Remark**: All three algorithms produce an MST.

Minimum Spanning Tree

**Minimum Spanning Tree**

Given a connected graph \( G = (V, E) \) with real-valued edge weights \( c_e \), an MST is a subset of the edges \( T \subseteq E \) such that \( T \) is a spanning tree whose sum of edge weights is minimized.

**Cayley’s Theorem.** There are \( n^n \) spanning trees of \( K_n \).

MST is fundamental problem with diverse applications.

- Network design.
  - telephone, electrical, hydraulic, TV cable, computer, read
- Approximation algorithm for NP-hard problems.
  - traveling salesperson problem, Shaeffer tree
- Indirect applications.
  - max bisection graph
  - LRPC codes for error correction
  - learning invariant features for real-time face verification
  - reducing data storage in sequencing amino acids in a protein
  - model locality of particle interactions in turbulent fluid flows
  - autoconfig protocol for Ethernet bridging to avoid cycles in a network
- **Cluster analysis.**

Applications

Network design.
- telephone, electrical, hydraulic, TV cable, computer, read
- Approximation algorithm for NP-hard problems.
  - traveling salesperson problem, Shaeffer tree
- Indirect applications.
  - max bisection graph
  - LRPC codes for error correction
  - learning invariant features for real-time face verification
  - reducing data storage in sequencing amino acids in a protein
  - model locality of particle interactions in turbulent fluid flows
  - autoconfig protocol for Ethernet bridging to avoid cycles in a network
- Cluster analysis.

Greedy Algorithms

- **Kruskal’s algorithm.** Start with \( T = \emptyset \). Consider edges in ascending order of cost. Insert edge \( e \) in \( T \) unless doing so would create a cycle.
- **Reverse-Delete algorithm.** Start with \( T = E \). Consider edges in descending order of cost. Delete edge \( e \) from \( T \) unless doing so would disconnect \( T \).
- **Prim’s algorithm.** Start with some root node \( s \) and greedily grow a tree \( T \) from \( s \). At each step, add the cheapest edge \( e \) to \( T \) that has exactly one endpoint in \( T \).

**Remark**: All three algorithms produce an MST.

Greedy Algorithms

- **Kruskal’s algorithm.** Start with \( T = \emptyset \). Consider edges in ascending order of cost. Insert edge \( e \) in \( T \) unless doing so would create a cycle.
- **Reverse-Delete algorithm.** Start with \( T = E \). Consider edges in descending order of cost. Delete edge \( e \) from \( T \) unless doing so would disconnect \( T \).
- **Prim’s algorithm.** Start with some root node \( s \) and greedily grow a tree \( T \) from \( s \). At each step, add the cheapest edge \( e \) to \( T \) that has exactly one endpoint in \( T \).

**Remark**: All three algorithms produce an MST.
Greedy Algorithms

Simplifying assumption. All edge costs $c_e$ are distinct.

Cut property. Let $S$ be any subset of nodes, and let $e$ be the min cost edge with exactly one endpoint in $S$. Then the MST contains $e$.

Cycle property. Let $C$ be any cycle, and let $f$ be the max cost edge belonging to $C$. Then the MST does not contain $f$.

Cycles and Cuts

Cycle. Set of edges the form $a-b, b-c, c-d, \ldots, y-z, z-a$.

Cutset. A cut is a subset of nodes $S$. The corresponding cutset $D$ is the subset of edges with exactly one endpoint in $S$.

Greedy Algorithms

Simplifying assumption. All edge costs $c_e$ are distinct.

Cut property. Let $S$ be any subset of nodes, and let $e$ be the min cost edge with exactly one endpoint in $S$. Then the MST $T^*$ contains $e$.

Cycle property. Let $C$ be any cycle in $G$, and let $f$ be the max cost edge belonging to $C$. Then the MST $T^*$ does not contain $f$.

Prim's Algorithm: Proof of Correctness

Prim's algorithm. [Jarník 1930, Dijkstra 1959, Prim 1957]

1. Initialize $S = \{\text{any node}\}$.
2. Apply cut property to $S$.
3. Add min cost edge in cutset corresponding to $S$ to $T$, and add one new explored node $u$ to $S$. 

Claim. A cycle and a cutset intersect in an even number of edges.

Pf. (by picture)

Prim's algorithm.

Cycle-Cut Intersection

Claim. A cycle and a cutset intersect in an even number of edges.

Pf. (exchange argument)

Suppose $e$ does not belong to $T^*$, and let's see what happens.

Adding $e$ to $T^*$ creates a cycle $C$ in $T^*$.

Edge $e$ is both in the cycle $C$ and in the cutset $D$ corresponding to $S$.

$T' = T^* \cup \{e\} - \{f\}$ is also a spanning tree.

Since $c_e < c_f$, cost($T'$) < cost($T^*$).

This is a contradiction.

Pf. (exchange argument)

Suppose $f$ belongs to $T^*$, and let's see what happens.

Deleting $f$ from $T^*$ creates a cut $S$ in $T^*$.

Edge $f$ is both in the cycle $C$ and in the cutset $D$ corresponding to $S$.

$T' = T^* \cup \{e\} - \{f\}$ is also a spanning tree.

Since $c_e < c_f$, cost($T'$) < cost($T^*$).

This is a contradiction.
**Implementation: Prim's Algorithm**

**Implementation.** Use a priority queue as Dijkstra.

- Maintain set of explored nodes $S$.
- For each unexplored node $v$, maintain attachment cost $a[v]$ = cost of cheapest edge $v$ to a node in $S$.
- $O(m + n \log n)$ with Fibonacci Heap.

$$\text{Prim}(G, c) \{$$

$\text{foreach} \ (v \in V) \ a[v] \leftarrow \infty$

$\text{Initialize an empty priority queue } Q$

$\text{foreach} \ (v \in V) \ \text{insert } v \text{ onto } Q$

$\text{Initialize set of explored nodes } S \leftarrow \emptyset$

$\text{while} \ (Q \text{ is not empty}) \ { }$

$\text{u} \leftarrow \text{delete min element from } Q$

$S \leftarrow S \cup \{u\}$

$\text{foreach} \ (\text{edge } e = (u, v) \text{ incident to } u) \ { }$

if $(v \notin S)$ and $(c[e] < a[v])$

$\text{decrease priority } a[v] \text{ to } c[e]$

$$\}$$

**Implementation: Kruskal's Algorithm**

**Implementation.** Use the union-find data structure.

- Build $T$ of edges in the MST.
- Maintain set for each connected component.
- $O(m \log n)$ for sorting and $O(m \log \log n)$ for union-find.

**Implementation.** Prim's Algorithm

```c
Kruskal(0, n) {
    sort edges weights so that $c_1 \leq c_2 \leq ... \leq c_m$.
    $T \leftarrow \emptyset$
    foreach (u ∈ V) make a set containing singleton u
    for i = 1 to m
        (u,v) = $e_i$
        if (u and v are in different sets) {
            $T \leftarrow T \cup \{e_i\}$
            merge the sets containing u and v
        }
    return T
}
```

**Lexicographic Tiebreaking**

To remove the assumption that all edge costs are distinct: perturb all edge costs by tiny amounts to break any ties.

**Impact.** Kruskal and Prim only interact with costs via pairwise comparisons. If perturbations are sufficiently small, MST with perturbed costs is MST with original costs.

**Implementation.** Can handle arbitrarily small perturbations implicitly by breaking ties lexicographically, according to index.

```c
boolean less(i, j) {
    if (cost($e_i$) < cost($e_j$)) return true
    else if (cost($e_i$) > cost($e_j$)) return false
    else if (i < j) return true
    else return false
}
```

**MST Algorithms: Theory**

**Deterministic comparison based algorithms.**

- $O(m \log n)$ [Jarník, Prim, Dijkstra, Kruskal, Borůvka]
- $O(m \log \log n)$. [Cheriton-Tarjan 1976, Yao 1975]
- $O(m \beta(m, n))$. [Fredman-Tarjan 1987]
- $O(m \log^\gamma(m, n))$. [Gabow-Galil-Spencer-Tarjan 1986]
- $O(m \alpha(m, n))$. [Chazelle 2000]

Holy grail. $O(m)$.

**Notable.**

- $O(m)$ randomized. [Karger-Klein-Tarjan 1995]
- $O(m)$ verification. [Dixon-Rauch-Tarjan 1992]

**Euclidean.**

- 2-d: $O(m \log n)$
- 3-d: $O(n \log n)$
- dense Prim