Greedy Algorithms

Activity Selection, Shortest Path, Minimal Spanning Tree, Huffman Codes, Knapsack Problem, Clustering, Set Cover,

Acknowledgement: The notes have drawn heavily from the following books: (1) Éva Tardös and Jon Kleinberg, "Algorithm Design", Addison-Wesley, (2) Michael T. Goodrich and Roberto Tamassia,"Algorithm Design and Applications", Wiley, (3) Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein, "Introduction to Algorithms", The MIT Press.

What is a Greedy Algorithm?

- In Wall Street (iconic movie of the 1980s), Michael Douglas gets up in front of a room full of stockholders and proclaims, "Greed . . . is good. Greed is right. Greed works."
- Wikipedia defines greed "As secular psychological concept, greed is an inordinate desire to acquire or possess more than one needs". A game like chess can be won only by thinking ahead: a player who is focused entirely on immediate advantage is easy to defeat. But in many other games, such as Scrabble, it is possible to do quite well by simply making whichever move seems best at the moment and not worrying too much about future consequences.
- This sort of myopic behavior is easy and convenient, making it an attractive algorithmic strategy (faster). *Greedy algorithms build up a solution piece by piece, always choosing the next piece that offers the most obvious and immediate benefit*. Although such an approach can be disastrous for some computational tasks, there are many for which it is optimal.
- Locally optimal decisions are called **greedy**
  - Short sighted strategy (e.g. thinking only one move ahead in a game), usually easy, faster to implement
  - Generally efficient, sometimes provably optimal, but …
  - Do not always lead to a globally optimal solution
    - Sometimes, close enough to globally optimal anyway (approximate or near optimal solution)

Greedy Technique

- It is hard, if not impossible, to define precisely what is meant by a greedy algorithm.
- Basic steps to finding efficient greedy algorithms:
  - Start by finding a dynamic programming style solution
  - Prove that at each step of the recursion, the min/max can be satisfied by a "greedy choice" (greedy substructure)
  - Show that only one recursive call needs to be made once the greedy choice is assumed. This is often natural when all the recursive calls are made by the min/max.
  - Find the recursive solution using the greedy choice
  - Convert to an iterative algorithm if possible
- More generally, taking the direct approach:
  - Show the problem is reduced to a sub problem via a greedy choice
  - Prove there is an optimal solution containing the greedy choice
  - Prove that combining the greedy choice with an optimal solution of the remaining sub problem yields an optimal solution.
- Our first example is Interval Scheduling, also known as Activity Selection (some activities that use a resource in a non sharable way, e.g., two classes cannot be scheduled in a single classroom when the class interval overlaps – there are many variations and extensions to this problem)

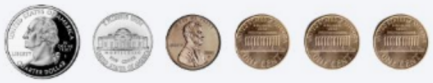
Characterization of Greedy Algorithms

- Greedy is a strategy that works well on optimization problems with the following characteristics:
  1. **Greedy-choice property**: A global optimum can be arrived at by selecting a local optimum.
  2. **Optimal substructure**: An optimal solution to the problem contains an optimal solution to subproblems.
- The second property *may* make greedy algorithms look like dynamic programming. However, the two techniques are quite different.

Coin Changing

Goal. Given currency denominations: 1, 5, 10, 25, 100, devise a method to pay amount to customer using fewest number of 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.

```
CASHIERS-ALGORITHM (x, c1, c2, ..., cn)
SORT n coin denominations so that c1 < c2 < ... < cn
S ← φ        ← set of coins selected
WHILE x > 0
   k ← largest coin denomination ck such that ck ≤ x
   IF no such k, RETURN "no solution"
   ELSE
      x ← x − ck
      S ← S ∪ { k }
RETURN S
```

Q. Is cashier's algorithm optimal?

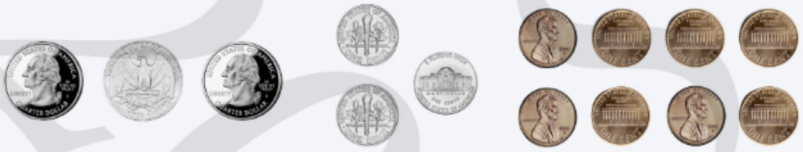
Properties of optimal solution

Property. Number of pennies $\leq$ 4.
Pf. Replace 5 pennies with 1 nickel.

Property. Number of nickels $\leq$ 1.
Property. Number of quarters $\leq$ 3.

Property. Number of nickels + number of dimes $\leq$ 2.
Pf.

- Replace 3 dimes and 0 nickels with 1 quarter and 1 nickel;
- Replace 2 dimes and 1 nickel with 1 quarter.
- Recall: at most 1 nickel.



Analysis of cashier's algorithm

Theorem. Cashier's algorithm is optimal for U.S. coins: 1, 5, 10, 25, 100.
Pf. [by induction on x]

- Consider optimal way to change $c_k \leq x < c_{k+1}$: greedy takes coin k.
- We claim that any optimal solution must also take coin k.
  - if not, it needs enough coins of type $c_1, \ldots, c_{k-1}$ to add up to x
  - table below indicates no optimal solution can do this
- Problem reduces to coin-changing $x - c_k$ cents, which, by induction, is optimally solved by cashier's algorithm. ▪

k	c_k	all optimal solutions must satisfy	max value of coins $c_1, c_2, \ldots, c_{k-1}$ in any OPT
1	1	$P \leq 4$	–
2	5	$N \leq 1$	4
3	10	$N + D \leq 2$	4 + 5 = 9
4	25	$Q \leq 3$	20 + 4 = 24
5	100	*no limit*	75 + 24 = 99

Cashier's algorithm for other denominations

Q. Is cashier's algorithm for any set of denominations?

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

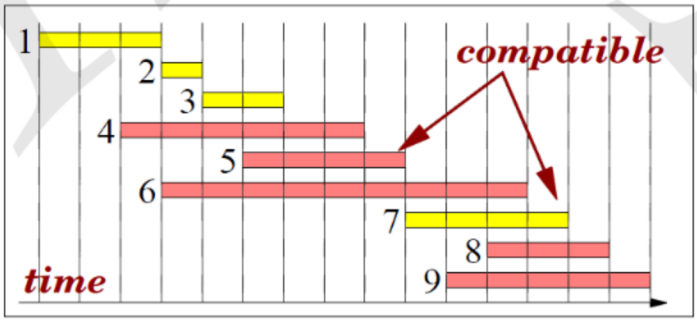
- Cashier's algorithm: 140¢ = 100 + 34 + 1 + 1 + 1 + 1 + 1 + 1.
- Optimal: 140¢ = 70 + 70.

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

- Cashier's algorithm: 15¢ = 9 + ???.
- Optimal: 15¢ = 7 + 8.

Activity Selection (Interval Scheduling)

- Problem Statement: Let S = {1, 2, … n} be the set of activities that compete for a resource. Each activity i has its starting time s_i and finish time f_i with $s_i \leq f_i$, namely, if selected, i takes place during time $[s_i; f_i)$. No two activities can share the resource at any time point. We say that activities i and j are compatible if their time periods are disjoint. The activity-selection problem is the problem of selecting the largest set of mutually compatible activities *within a given time*.
  - Easy to see that this identical to scheduling intervals such that intervals do not overlap.



Greedy Activity Selection Algorithm

- **Algorithm**: In this algorithm the activities are **first sorted according to their finishing time**, from the earliest to the latest, where a **tie can be broken arbitrarily**. Then the activities are **greedily** selected by going down the list and by picking whatever activity that is compatible with the current selection.
- **Run Time Analysis**:
  - Sorting takes O(n log n) time.
  - Second part takes O(n) time – scan a list sequentially, checking compatibility with the next job takes O(1) time, there are n jobs to scan.
- **Correctness Analysis**: Correctness means the output is a maximal set of mutually compatible activities. We use induction:
  - For the base case n = 1. Correctness trivially holds. For the induction case, let $n \geq 2$ and assume the claim holds for all values of n less than the correct one. Let p be the number of activities in each optimal solution for [1 …n – 1] and let q be the number for [1 …n]. Here $p \leq q$; why? It is because each optimal solution for [1 …n – 1] is a solution for [1 …n] [This is the greedy substructure.
  - Is it possible $p = q - 2$? NO. Why? Let W be any optimal solution for [1 …n]. Let W = W – {n} if W contains {n} and W = W otherwise. Then W does not contain {n} and is a solution for [1 …n – 1]. This contradicts the assumption that optimal solutions for [1 …n – 1] have p activities.

Optimality Proof

- We must first note that the greedy algorithm always finds some set of mutually compatible activities.
- (Case 1): Suppose p = q. Then each optimal solution for [1 … n – 1] is optimal for [1 … n]. By our induction hypothesis, when n – 1 has been examined an optimal solution for [1 … n – 1] has been constructed. So, there will be no addition after this; otherwise, there would be a solution of size > q. So, the algorithm will output a solution of size p, which is optimal.
- (Case 2): Suppose p = q – 1. Then each optimal solution for [1 … n] contains n. Let k be the largest integer i, $1 \leq i \leq n-1$, such that $f_i \leq s_n$. Since $f_1 \leq \ldots \leq f_n$, for all i, $1 \leq i \leq k+1$, i is compatible with n, and for all i, $k+1 \leq i \leq n-1$, i is incompatible with n. This means that each optimal solution for [1 …n] is the union of {n} and an optimal solution for [1 …k]. So, each optimal solution for [1 …k] has p activities. This implies that no optimal solutions for [1 …k] are compatible with any of $k+1, \ldots, n-1$.

Let W be the set of activities that the algorithm has when it has finished examining k. By our induction hypothesis, W is optimal for [1, …, k]. So, it has p activities. The algorithm will then add no activities between k +1 and n – 1 to W but will add {n} to W. The algorithm will then output $W \cup \{n\}$. This output has q = p+1 activities, and thus, is optimal for [1, …, n].

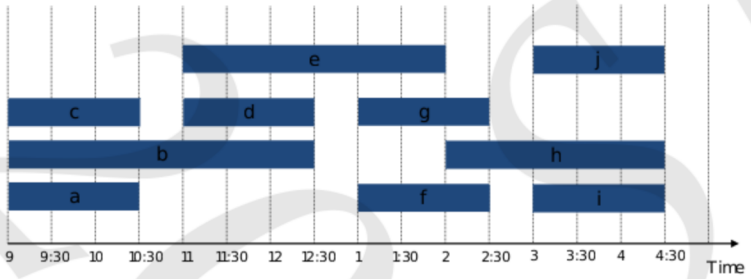
Formulation using Dynamic Programming

Greedy Substructure

- Let us be a bit formal. The problem is: Given a set of activities $S = \{a_1, \ldots, a_n\}$, where a_i starts at time $s_i \geq 0$ and finishes at time $f_i > s_i$, find a maximal subset $A \subseteq S$ such that for distinct activities either $s_i \geq f_j$ or $s_j \geq f_i$.
- Convenience for notations:
  - Let a_0 be an imaginary activity finishing at time 0.
  - Let a_{n+1} be an imaginary activity starting at time ∞
  - $S_{ij} = \{a_k \in S: f_i \leq s_k < f_k \leq s_j \}$
  - Observe that $S_{0,n+1}$ contains all activities.
- Let $f_m = \min\{f_k : a_k \in S_{ij}\}$, i.e., activity a_m has the earliest finish time in S_{ij}
  - Claim 1: am is used in some maximal solution for the activities in S_{ij} : Suppose a_k is the first activity in some maximal solution – it can be safely removed and replaced by am.
  - Claim 2: $S_{im} = \emptyset$: Nothing else starting after a_i finishes before a_m.
- Thus, it is always safe to include a_m, and solve the remaining problem for S_{mj} only.

Activity Selection (Multiple resources) or Interval Partitioning

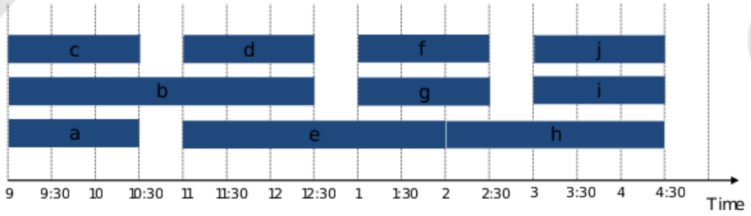
- We extend the problem to multiple (identical) resources (to accommodate all activities)
- **Interval partitioning**.
  - Lecture j starts at s_j and finishes at f_j.
  - Goal: find minimum number of classrooms to schedule all lectures so that no two occur at the same time in the same room.
- Ex: This schedule uses 4 classrooms to schedule 10 lectures.



15

Interval Partitioning

- Interval partitioning.
  - Lecture j starts at s_j and finishes at f_j.
  - Goal: find minimum number of classrooms to schedule all lectures so that no two occur at the same time in the same room.
- Ex: This schedule uses only 3.



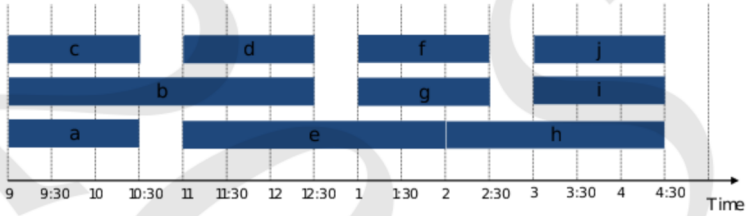
16

Interval Partitioning: Lower Bound on Optimal Solution

- Def. The depth of a set of open intervals is the maximum number that contain any given time.
- Key observation. Number of classrooms needed ≥ depth.
- Ex: Depth of schedule below = 3 ⇒ schedule below is optimal.

a, b, c all contain 9:30

- Q. Does there always exist a schedule equal to depth of intervals?



17

Interval Partitioning: Greedy Algorithm

- Greedy algorithm. Consider lectures in increasing order of start time: assign lecture to any compatible classroom.

```
Sort intervals by starting time so that s1 ≤ s2 ≤ ... ≤ sn.
d ← 0
                ← number of allocated classrooms

for j = 1 to n {
   if (lecture j is compatible with some classroom k)
      schedule lecture j in classroom k
   else
      allocate a new classroom d + 1
      schedule lecture j in classroom d + 1
      d ← d + 1
}
```

- Implementation. O(n log n).
  - For each classroom k, maintain the finish time of the last job added.
  - Keep the classrooms in a priority queue.

18

Interval Partitioning: Greedy Analysis

- **Observation**. Greedy algorithm never schedules two incompatible lectures in the same classroom.
- **Theorem**. Greedy algorithm is optimal.
- **Proof.**
  - Let d = number of classrooms that the greedy algorithm allocates.
  - Classroom d is opened because we needed to schedule a job, say j, that is incompatible with all d – 1 other classrooms.
  - Since we sorted by start time, all these incompatibilities are caused by lectures that start no later than s_j.
  - Thus, we have d lectures overlapping at time s_j +
  - Key observation all schedules use d classrooms.

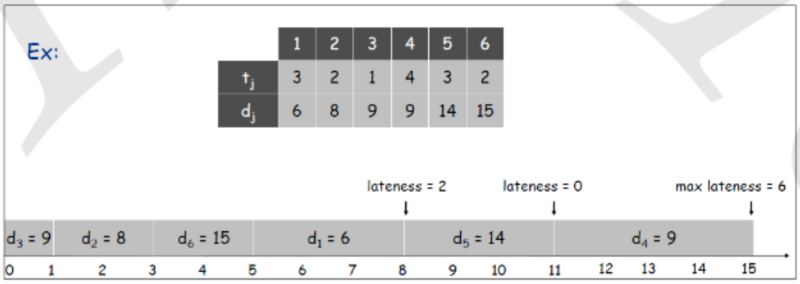
19

Scheduling to Minimizing Lateness

- Minimizing lateness problem.
  - Single resource processes one job at a time.
  - Job j requires t_j units of processing time and is due at time d_j. [No restriction on start time]
  - If j starts at time s_j, it finishes at time $f_j = s_j + t_j$.
  - Lateness: $\ell_j = \max \{ 0, f_j - d_j \}$.
  - Goal: schedule all jobs to minimize maximum lateness $L = \max_j \ell_j$.

Ex:

	1	2	3	4	5	6
t_j	3	2	1	4	3	2
d_j	6	8	9	9	14	15



20

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 $d_j - t_j$.

21

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.

	1	2
t_j	1	10
d_j	100	10

counterexample

  - [Smallest slack] Consider jobs in ascending order of slack $d_j - t_j$.

	1	2
t_j	1	10
d_j	2	10

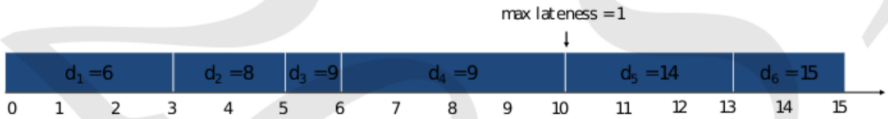
counterexample

22

Minimizing Lateness: Greedy Algorithm

- Greedy algorithm. Earliest deadline first.

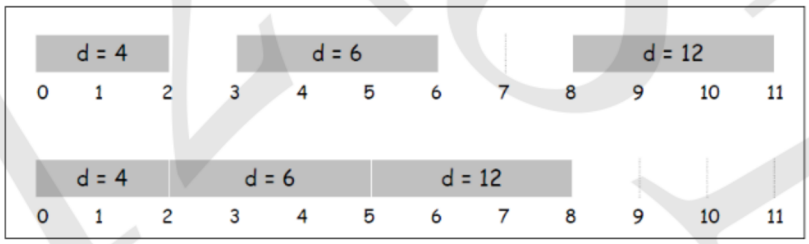
```
Sort n jobs by deadline so that d_1 ≤ d_2 ≤ ... ≤ d_n

t ← 0
for j = 1 to n
   Assign job j to interval [t, t + t_j]
   s_j ← t, f_j ← t + t_j
   t ← t + t_j
output intervals [s_j, f_j]
```



23

Scheduling to Minimize Lateness

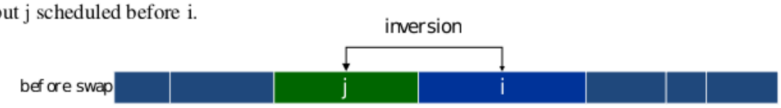
- **Observation**. There exists an optimal schedule with no idle time. [Remember, No restriction on start time]



- **Observation**. The greedy schedule has no idle time. Why? Obvious
- We want to prove that earliest-deadline-first greedy algorithm is optimal. How? We use exchange argument – Change into greedy schedule without losing optimality. Things to ponder:
  - What do we know about the greedy schedule?
  - How can we change the optimal to be more like that without losing optimality?

Minimizing Lateness: Inversions

- **Definition**. An inversion in schedule S is a pair of jobs i and j such that: i < j but j scheduled before i.

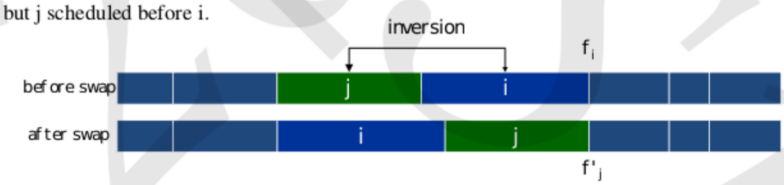


- **Observation**. Greedy schedule has no inversions. Why
- **Question**. If a schedule (with no idle time) has an inversion, it has one with a pair of inverted jobs scheduled consecutively. Why? [Recall: jobs sorted in ascending order of due dates]
  - Only difference is an "inversion" of i and j with equal deadline (d_i=d_j). Maximum lateness of i and j is only influenced by last job (f_i - d_i). Maximum lateness of i and j is the same if i and j are swapped.
- **Observation**: All schedules without inversions have same lateness.
- **Question:** If a schedule (with no idle time) has an inversion, how can we find it?

going through schedule, at some point deadline decreases

Minimizing Lateness: Inversions

- **Definition**. An inversion in schedule S is a pair of jobs i and j such that: i < j but j scheduled before i.



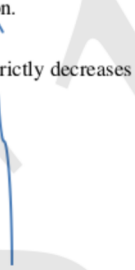
- **Claim**. Swapping two adjacent, inverted jobs reduces the number of inversions by one and does not increase the max lateness.
- **Proof.** Let ℓ be the lateness before the swap, and let ℓ' be it afterwards.
  - $\ell'_k = \ell_k$ for all k ≠ i, j
  - $\ell'_i \leq \ell_i$
  - If job j is late:

$$\begin{aligned} \ell'_j &= f'_j - d_j && \text{(definition)} \\ &= f_i - d_j && \text{(j finishes at time } f_i\text{)} \\ &\leq f_i - d_i && \text{(i < j)} \\ &\leq \ell_i && \text{(definition)} \end{aligned}$$

26

Minimizing Lateness: Analysis of Greedy Algorithm

- Theorem. Greedy schedule S is optimal.
- Proof. (by contradiction)
- Suppose S is not optimal.
- Define S* to be an optimal schedule that has the fewest number of inversions (of all optimal schedules) and has no idle time.
- Clearly S≠S*.
  - If S* has no inversions, then maxlate(S) = maxlate(S*). Contradiction.
  - 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*
- So S is an optimal schedule.



Greedy has no inversions. All schedules without inversions have same lateness.

"This proof can be found on pages 128 – 131.

27

Greedy Analysis Strategies

- **Greedy algorithm stays ahead**. Show that after each step of the greedy algorithm, its solution is at least as good as any other algorithm's.
- **Exchange argument**. Gradually transform any solution to the one found by the greedy algorithm without hurting its quality.
- **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.
  - Greed is myopic; at every iteration, the algorithm chooses the best morsel it can swallow, without worrying about the future.
  - Objective function does not explicitly appear in greedy algorithm! Hard, if not impossible, to precisely define "greedy algorithm."

28

Example Exercise

Planning a mini-triathlon:

1. swim 20 laps (one at a time)
2. bike 10 km (can be done simultaneously)
3. run 3 km (can be done simultaneously)

expected times are given for each contestant

Def. The completion time is the earliest time all contestants are finished.

Q. In what order should they start to minimize the completion time?

Q. Proof that this order is optimal (minimal).

Ex.

	1	2	3	
				← contestant
s_j	3	2	4	← time required for swimming
b_j	5	4	6	← time required for biking
r_j	3	3	3	← time required for running

Variant: Scheduling to Minimizing Total Lateness

Outside our scope

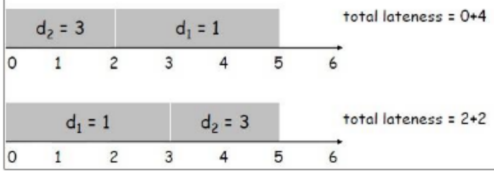
Minimizing total lateness problem.

- Single resource processes one job at a time.
- Job j requires t_j units of processing time and is due at time d_j.

- If j starts at time s_j, it finishes at time $f_j = s_j + t_j$.
- Lateness: $\ell_j = \max \{ 0, f_j - d_j \}$.
- Goal: schedule all jobs to minimize total lateness $L = \sum_j \ell_j$.

Ex:

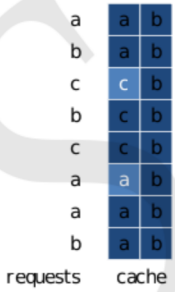
	1	2	
			← job number
t_j	3	2	← time required
d_j	1	3	← deadline



No polynomial algorithm can compute optimal schedule (unless P=NP)

Optimal Offline 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, b, c, a, a, b.
- Optimal eviction schedule: 2 cache misses.



31

Optimal Offline Caching: Farthest-In-Future

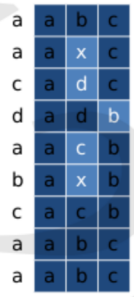
- Farthest-in-future. Evict item in the cache that is not requested until farthest in the future.

- Theorem. [Bellady, 1960s] FF is optimal eviction schedule.
- Pf. Algorithm and theorem are intuitive; proof is subtle.

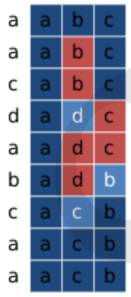
32

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.



an unreduced schedule

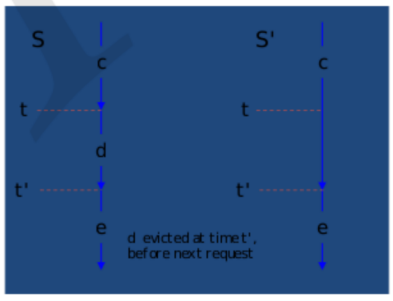


a reduced schedule

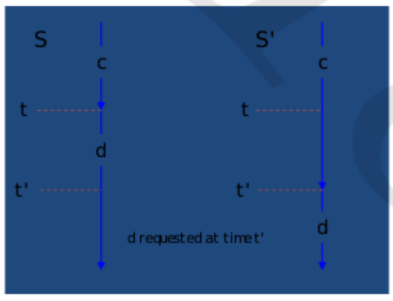
33

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) ← doesn't enter cache at requested time
  - 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 1



Case 2

34

Farthest-In-Future: Analysis

- Theorem. FF is optimal eviction algorithm.
- Pf. (by induction on number or requests j)

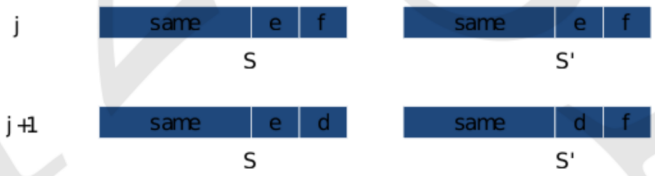
> Invariant: There exists an optimal reduced schedule S that makes the same eviction schedule as S_{FF} through the first j+1 requests.

- Let S be reduced schedule that satisfies invariant through j requests. We produce S' that satisfies invariant after j+1 requests.
  - Consider $(j+1)^{st}$ request $d = d_{j+1}$.
  - Since S and S_{FF} have agreed up until now, they have the same cache contents before request j+1.
  - Case 1: (d is already in the cache). S' = S satisfies invariant.
  - Case 2: (d is not in the cache and S and S_{FF} evict the same element). S' = S satisfies invariant.

35

Farthest-In-Future: Analysis

- Pf. (continued)
  - Case 3: (d is not in the cache; S_{FF} evicts e; S evicts f ≠ e).
    - begin construction of S' from S by evicting e instead of f



    - now S' agrees with S_{FF} on first j+1 requests; we show that having element f in cache is no worse than having element e

36

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'.

must involve e or f (or both)

  - 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 evicts.
    - if e' = e, S' accesses f from cache; now S and S' have same cache
    - if e' ≠ e, S' evicts 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 S_{FF} through step j+1

37

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'.

must involve e or f (or both)

otherwise S' would take the same action

  - Case 3c: g ≠ e, f. S must evict e.
    Make S' evict f; now S and S' have the same cache. ■

38

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.

↑ FF with direction of time reversed!

- Theorem. FF is optimal offline eviction algorithm.
  - Provides basis for understanding and analyzing online algorithms.
  - LRU is k-competitive. [Section 13.8]
  - LIFO is arbitrarily bad.

39

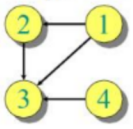
Quick Refresh of Graphs

- A Graph G = (V, E) is an ordered pair consisting of a set V of vertices (nodes) and a set of edges (links) E ⊆ V × V. An edge set E consists of pair of vertices [the pair is unordered for undirected graphs and ordered for directed graphs]. In either case, |E| $O(V^2)$; moreover if G is connected |E| ≥ |V| – 1.
- **Adjacency matrix** of G = (V, E), where V = {1, 2, …, n} is given by

$$A[i, j] = \begin{cases} 1 \text{ if } (i, j) \in E \\ 0 \text{ if } (i, j) \notin E \end{cases}$$ This takes $O(V^2)$ storage

- **Adjacency-List representation**

An ***adjacency list*** of a vertex $v \in V$ is the list $Adj[v]$ of vertices adjacent to v.



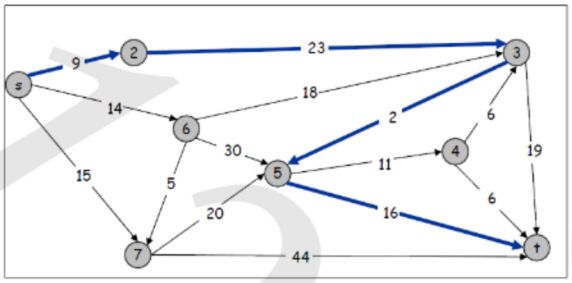
$Adj[1] = \{2, 3\}$
$Adj[2] = \{3\}$
$Adj[3] = \{\}$
$Adj[4] = \{3\}$

For undirected graphs, $|Adj[v]| = degree(v)$.
For digraphs, $|Adj[v]| = out\text{-}degree(v)$.

- Adjacency lists need O(|V| + |E|) storage – the so called sparse representation vs. dense representation using adjacency matrix.
- When to use which one – depends on application

Shortest Path(s) in a Graph

- Assume a that each edge of a graph is associated with a weight function w: E → R^+ (only non-negative weights). For simplicity, assume that all edge weights are distinct (this is not necessary).
- Shortest Path Problem: Given a source node, say s, and a destination node, say t, find the shortest (cost of the path is the sum of the costs of all edges lying on the path) path from s to t.
- Example:

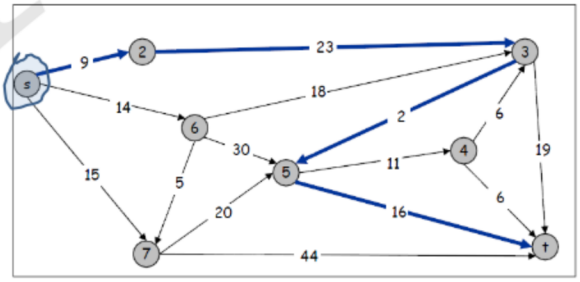


- Cost of the shortest path P from s to t = w(P) = 9 + 23 + 2 +16 = 50, where P = (s→2 →3 → 5 → t) .
- It is easy to see that there may exist more than one shortest paths.
- We are interested to find one such shortest path from s to t.

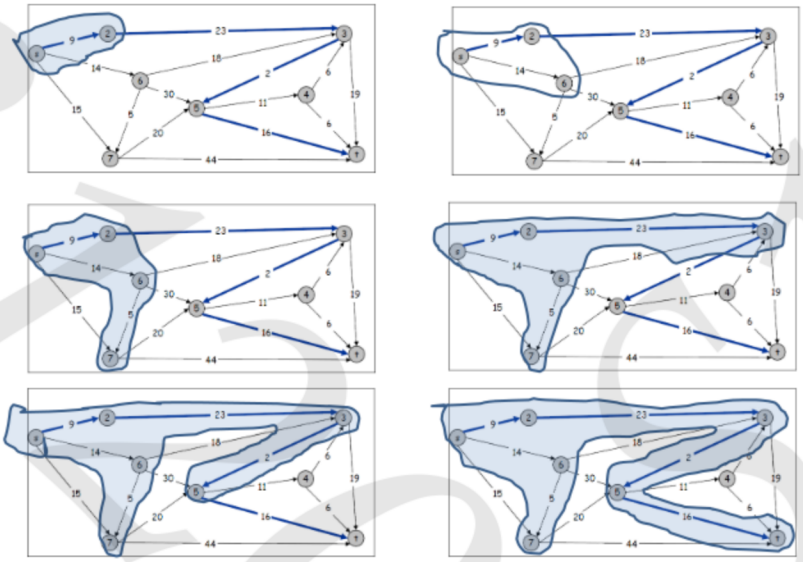
- Approach: We start from source s; we know the nodes directly reachable from s; none of them is the destination t in our example. Think of 2 things: (1) we have not seen t yet; the path from s to t must pass through one of the neighbors of s; (2) we sure need to explore the entire graph to compute the shortest path. [Note that the discussion applies equally well to both directed and undirected graphs.

Dijkstra's Algorithm

- Maintain a set of explored nodes S for which we have determined the shortest path distance d(u) from s to u.
- Initialize S = { s }, d(s) = 0.
- while S ≠ V
  - Select a node v ∉ S with at least one edge from *S* for which $d'(v) = \min_{e=(u,v):\, u \in S} d(u) + w(e)$ is as small as possible and set S = S ∪ {v} + w(e) and d(v) = d'(v).
- To produce the s-u paths corresponding to the distances found, we simply record the edge (u, v) on which it achieved the value $d'(v) = \min_{e=(u,v):\, u \in S} d(u) + w(e)$ [simple recursive backtracking will do]. Consider the previous graph; initially, the situation is as follows [the shaded area is S.

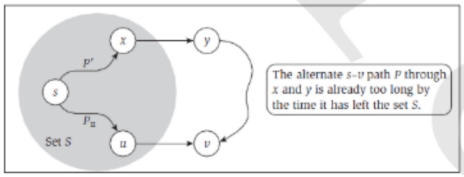


Dijkstra's Algorithm: Example Execution



Dijkstra's Algorithm: Proof of Correctness

- The Invariant is: For each node u ∈S, d(u) is the length of the shortest s-u path [s is the given source node].
- We will prove by induction on |S|.
- Base case: Initially, S = {s}, or |S| = 1; the claim is trivially true.
- Inductive hypothesis: Assume true for $|S| = k \geq 1$
  - Let v be next node added to S, and let u-v be the chosen edge.
  - The shortest s-u path plus (u, v) is an s-v path of length $d'(v)$.
  - Consider any s-v path P. We'll see that it's no shorter than $d'(v)$.
  - Let x-y be the first edge in P that leaves S, and let P' be the subpath to x; P is already too long as soon as it leaves S.



$$w(P) \geq w(P') + w(x,y) \geq d(x) + w(x,y) \geq d'(y) \geq d'(v).$$

non-negative wts. — ind. hypothesis — defn. of d' — v is chosen over y

Dijkstra's Algorithm: Implementation

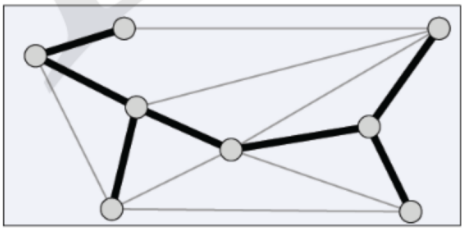
- There are n − 1 iterations of the while loop for a graph with n nodes, as each iteration adds a new node v to S.
- We explicitly maintain the values of the minima $\pi(v) = \min_{e=(u,v):\ u \in S} d(u) + w(e)$ for each node $v \in V - S$; we maintain this information in a min-binary_heap [we have seen this simple data structure before; select_min operation is done in O(log n) worst case time, n is the number of elements in the heap.
- Next node to explore = node with minimum $\pi(v)$.
- When exploring v, for each incident edge e = (v, w), update $\pi(w) = \min \{\pi(w), \pi(v)+w(e)\}$
- We get an implementation with O(m log n) worst case time.
- We can get a better amortized time of $O(m \log_{m/n} n)$ if we use a Fibonacci heap.

DIJKSTRA (V, E, s)

```
Create an empty priority queue.
FOR EACH v ≠ s : d(v) ← ∞; d(s) ← 0.
FOR EACH v ∈V : insert v with key d(v) into priority queue.
WHILE (the priority queue is not empty)
    u ← delete-min from priority queue.
    FOR EACH edge (u, v) ∈E leaving u:
        IF d(v) > d(u) + w(u, v)
        decrease-key of v to d(u) + w(u, v) in priority queue.
        d(v) ← d(u) + w(u, v).
```

Spanning Tree Properties

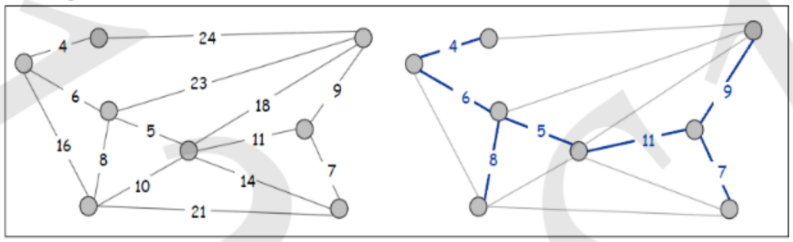
- Consider a connected, undirected graph G = (V, E); Let T = (V, F), F ⊆ E, be a subgraph of G = (V, E). All of the following statements must be true:
  - T is a spanning tree of G.
  - T is acyclic and connected.
  - T is connected and has n – 1 edges.
  - T is acyclic and has n – 1 edges.
  - T is minimally connected: removal of any edge disconnects it.
  - T is maximally acyclic: addition of any edge creates a cycle.
  - T has a unique simple path between every pair of nodes.



The black edges constitute the spanning tree. Satisfy yourself that all those statements are true [some are equivalent]

Minimal Spanning Tree

- A connected, undirected graph G = (V, E) with weight function w: E→ ℝ (set of reals); we'll assume all edge weights are distinct [Explain briefly the general case when duplicates are allowed]
- A connected acyclic subgraph T of G that spans all vertices is called a spanning tree of the graph. A minimal spanning tree (MST) is a spanning tree of minimum weight, i.e., a spanning tree whose sum of edge weights is minimized.
- Example:



- **Note:** Cayley's theorem. There are n^{n-2} spanning trees of K_n ; thus, not possible to find MST by exhaustive enumeration.

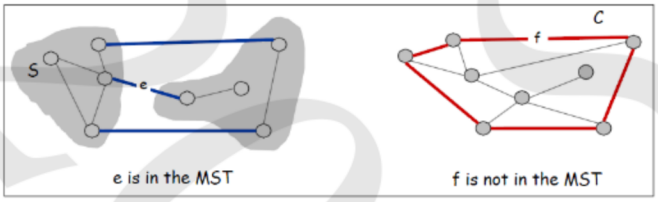
Applications

- MST is fundamental problem with diverse applications.
- Network design.
  - telephone, electrical, hydraulic, TV cable, computer, road
- Approximation algorithms for NP-hard problems.
  - traveling salesperson problem, Steiner tree
- Indirect applications.
  - max bottleneck paths
  - LDPC codes for error correction
  - image registration with Renyi entropy
  - learning salient 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 for MST

1. **Kruskal's algorithm**. Start with T = ∅. Consider edges in ascending order of cost. Insert edge e in T unless doing so would create a cycle.
2. **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.
3. **Prim's algorithm**. Start with some root node s and greedily grow a tree T from s outward. At each step, add the least weight edge e to T that has exactly one endpoint in T.

- We'll do 1 and 3. The greedy structure is the same; implementations are different, one uses the min-heaps and the other uses Union-Find data structure. Both works in O(m log n) time.
- **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.

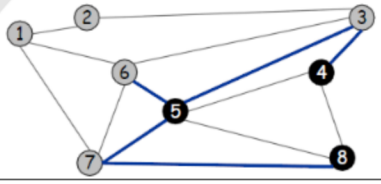


Cycles and Cuts

Cycle. Set of edges the form a-b, b-c, c-d, …, y-z, z-a.

Cycle C = 1-2, 2-3, 3-4, 4-5, 5-6, 6-1

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

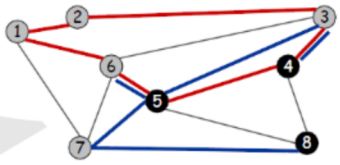


Cut S = { 4, 5, 8 }
Cutset D = 5-6, 5-7, 3-4, 3-5, 7-8

A cut is a partition of the nodes into two nonempty subsets S and V – S.

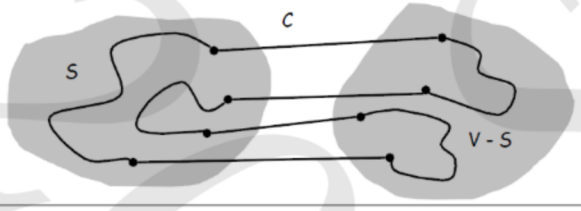
Cycle-Cut Intersection

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



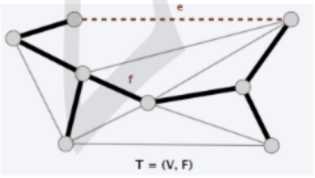
Cycle C = 1-2, 2-3, 3-4, 4-5, 5-6, 6-1
Cutset D = 3-4, 3-5, 5-6, 5-7, 7-8
Intersection = 3-4, 5-6

Pf. (by picture)



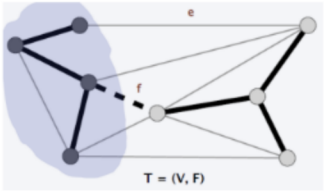
Fundamental Cycle and Fundamental Cutset

- Fundamental cycle.
  - Adding any non-tree edge e to a spanning tree T forms unique cycle C.
  - Deleting any edge f ∈C from T ∪ { e } results in new spanning tree.



Observe: If w(e) < w(f), then T is not an MST

- Fundamental cutset.
  - Deleting any tree edge f from a spanning tree T divide nodes into two connected components. Let D be cutset.
  - Adding any edge e ∈D to T – { f } results in new spanning tree.



Observe: If w(e) < w(f), then T is not an MST

A Simple Greedy Algorithm

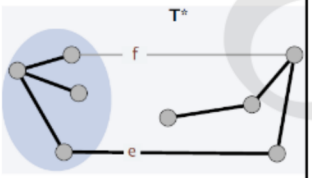
- There are two simple rules: Apply the red and blue rules (in any sequence) until n – 1 edges are colored blue. The blue edges form an MST.
  - Red Rule: Select a cycle C in G with no red edges; Select an uncolored edge of C of max weight and color it red.
  - Blue Rule: Select a cut-set with no blue edges; Select an uncolored edge in D of min weight and color it blue.
- **Color invariant.** There exists an MST T* containing all of the blue edges and none of the red edges.
- We will prove by induction on the number of iterations;
  - **Base case**. No edges colored ⇒ every MST satisfies invariant.
  - We consider the two rules separately to show that the invariant is true.

Proof of Correctness

- **Induction step (red rule).** Suppose color invariant true before red rule.
  - let C be chosen cycle, and let e be edge colored red.
  - if e ∉T*, T* still satisfies invariant.
  - Otherwise, consider fundamental cut-set D by deleting e from T*.
  - let f ∈D be another edge in C.
  - f is uncolored and w(e) ≥ w(f) since (1) f ∉T* ⇒ f not blue, (2) red rule ⇒ f not red and w(e) ≥ w(f)
  - Thus, T* ∪ { f } – { e } satisfies invariant.

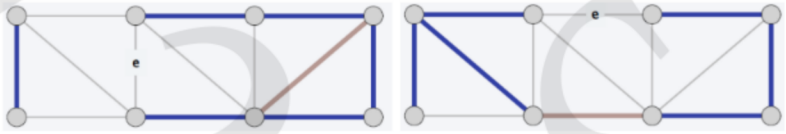
Induction step (blue rule). Suppose color invariant true before blue rule.

- let D be chosen cutset, and let f be edge colored blue.
- if f ∈T*, T* still satisfies invariant.
- Otherwise, consider fundamental cycle C by adding f to T*
- let e ∈C be another edge in D.
- e is uncolored and w(e) ≥ w(f) since (1) e ∈T* ⇒ e not red (2) blue rule ⇒ e not blue and ce ≥ cf
- Thus, T* ∪ { f } – { e } satisfies invariant.



Proof of Termination

- **Theorem**. The greedy algorithm terminates. Blue edges form an MST.
- **Proof**. We need to show that either the red or blue rule (or both) applies.
  - Suppose edge e is left uncolored.
  - Blue edges form a forest.
  - Case 1: both endpoints of e are in same blue tree.
    ⇒ apply red rule to cycle formed by adding e to blue forest.
  - Case 2: both endpoints of e are in different blue trees.
    ⇒ apply blue rule to cutset induced by either of two blue trees.



Case 1 Case 2

Prim's Algorithm

- **Algorithm**: Initialize S = any node, in O(m log n) time. Then repeat n – 1 times {Add to tree the min weight edge with one endpoint in S; Add new node to S.}
- **Theorem**. Prim's algorithm computes the MST. **Proof**: Special case of greedy algorithm (blue rule repeatedly applied to S).
- Implement: [d(v) = weight of cheapest known edge between v and S] The data structure and approach are almost identical to Dijkstra's algorithm for shortest path

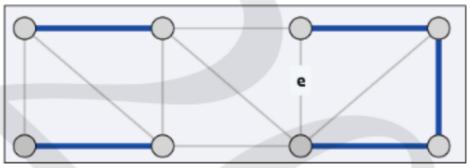
PRIM (V, E, w)
Create an empty priority queue.
s ← any node in V.
FOR EACH v ≠ s : d(v) ← ∞; d(s) ← 0.
FOR EACH v : insert v with key d(v) into priority queue.
REPEAT n – 1 times
  u ← delete-min from priority queue.
  FOR EACH edge (u, v) ∈E incident to u:
    IF d(v) > w(u, v)
      decrease-key of v to w(u, v) in priority queue.
      d(v) ← w(u, v).

Kruskal's algorithm

- Consider edges in ascending order of weight:
- **Algorithm**: Add to tree unless it would create a cycle.
- **Theorem**. Kruskal's algorithm computes the MST.
- **Proof**. Special case of greedy algorithm.
  - Case 1: both endpoints of e in same blue tree.
    ⇒ color red by applying red rule to unique cycle.
  - Case 2. If both endpoints of e are in different blue trees.
    ⇒ color blue by applying blue rule to cut-set defined by either tree.

all other edges in cycle are blue

no edge in cut-set has smaller weight
(since Kruskal chose it first)



Kruskal's Algorithm: implementation

- Theorem. Kruskal's algorithm can be implemented in O(m log m) time.
- Algorithm:
  - Sort edges by weight.
  - Use union-find data structure to dynamically maintain connected components.

KRUSKAL (V, E, w)
SORT m edges by weight so that $w(e_1) \leq w(e_2) \leq \ldots \leq w(e_m)$
$S \leftarrow \varphi$
FOREACH $v \in V$: MAKESET(v).
FOR i = 1 TO m
  (u, v) $\leftarrow e_i$
  IF FINDSET(u) $\neq$ FINDSET(v)
    $S \leftarrow S \cup \{ e_i \}$
    UNION(u, v).
RETURN S

Are u and v in the same component?

make u and v in same component

Huffman Coding