南方科技大学
SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY

Algorithm Design and Analysis (H)

CS216

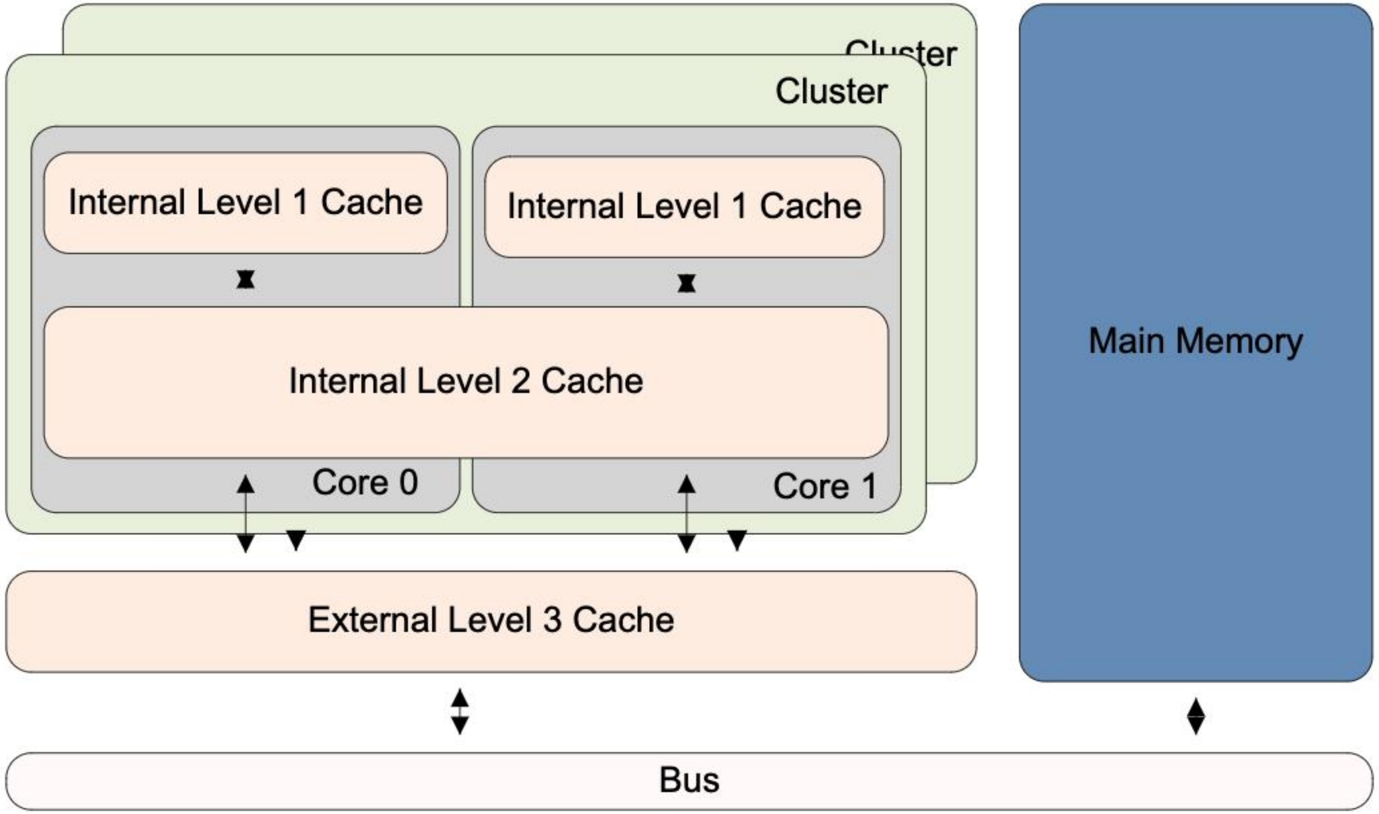
Prof. Shiqi Yu (于仕琪)
yusq@sustech.edu.cn
http://faculty.sustech.edu.cn/yusq/

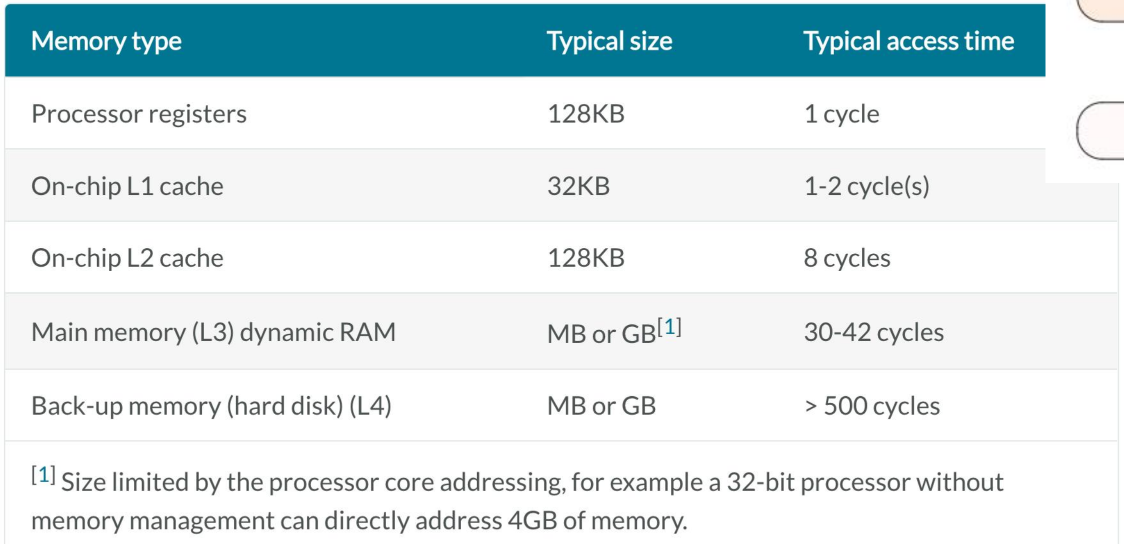
Greedy Algorithms

4. Optimal Caching

Caching

- HD->memory->cache





Memory type	Typical size	Typical access time
Processor registers	128KB	1 cycle
On-chip L1 cache	32KB	1-2 cycle(s)
On-chip L2 cache	128KB	8 cycles
Main memory (L3) dynamic RAM	MB or GB[1]	30-42 cycles
Back-up memory (hard disk) (L4)	MB or GB	> 500 cycles

[1] Size limited by the processor core addressing, for example a 32-bit processor without memory management can directly address 4GB of memory.

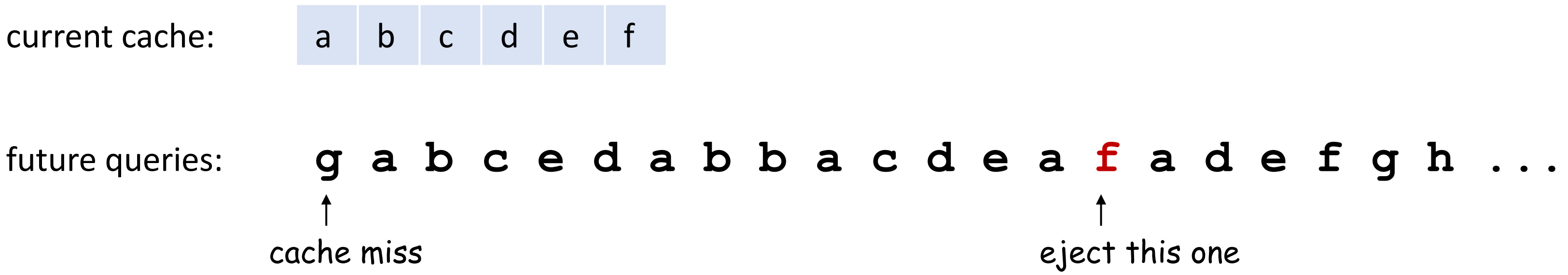
https://developer.arm.com/documentation/den0024/a/Caches

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.

Optimal Offline Caching: Farthest-In-Future

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



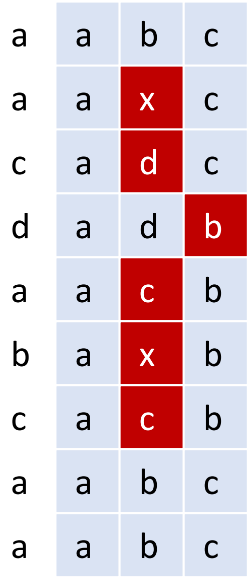
current cache: a b c d e f

future queries: `g a b c e d a b b a c d e a f a d e f g h ...`

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

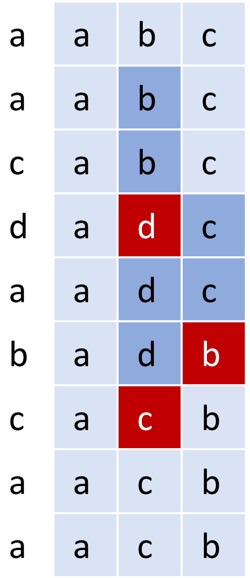
Reduced Eviction Schedules

- 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.



a	a	b	c
a	a	x	c
c	a	d	c
d	a	d	b
a	a	c	b
b	a	x	b
c	a	c	b
a	a	b	c
a	a	b	c

an unreduced schedule



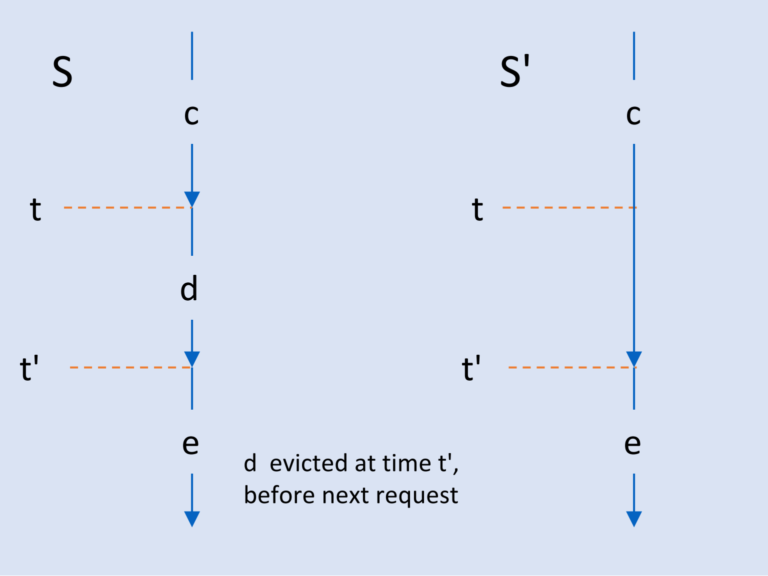
a	a	b	c
a	a	b	c
c	a	b	c
d	a	d	c
a	a	d	c
b	a	d	b
c	a	c	b
a	a	c	b
a	a	c	b

a reduced schedule

Reduced Eviction Schedules

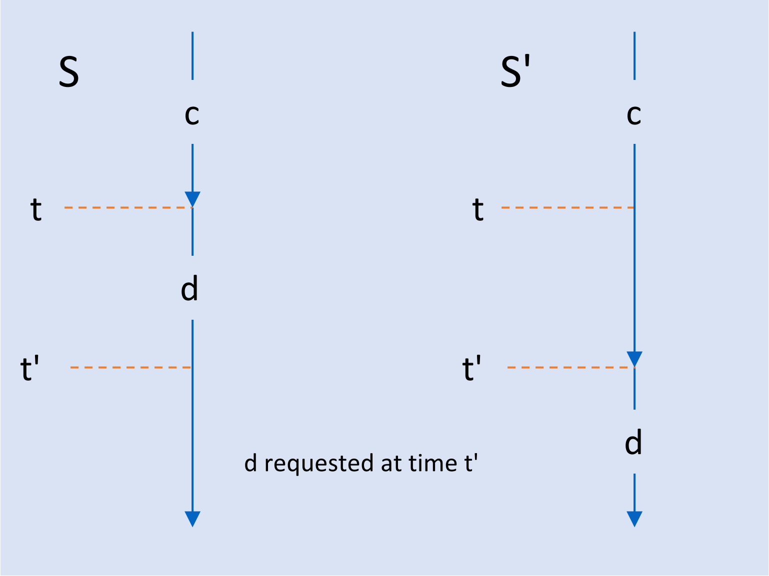
- Claim. Given any unreduced schedule S, can transform it into a reduced schedule S' with no more cache misses.
- Pf. (by induction on number of unreduced items)
  - Suppose S brings d into the cache at time t, without a request.
  - Let c be the item S evicts when it brings d into the cache.
  - Case 1: d evicted at time t', before next request for d.
  - Case 2: d requested at time t' before d is evicted. ■

doesn't enter cache at requested time



d evicted at time t', before next request

Case 1



d requested at time t'

Case 2

Farthest-In-Future: Analysis

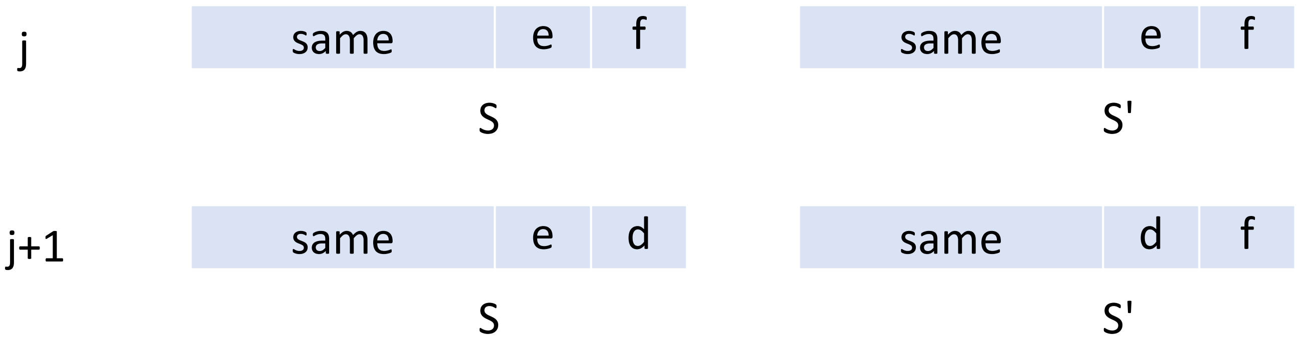
- Theorem. FF is optimal eviction algorithm.
- Pf. (by induction on number or requests j)

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.

Farthest-In-Future: Analysis

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



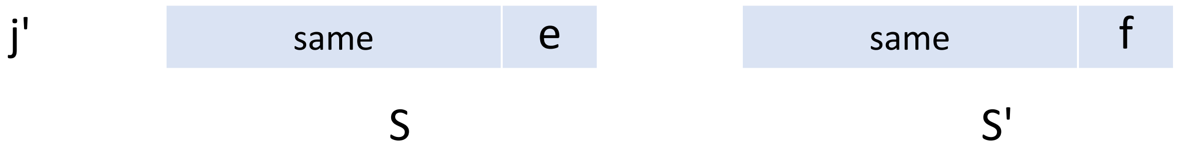
	S			S'		
j	same	e	f	same	e	f
j+1	same	e	d	same	d	f

- ✓ 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

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.

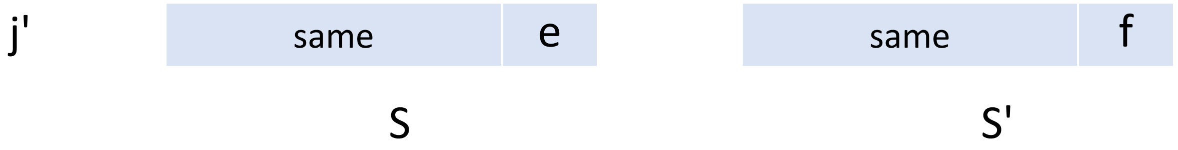
- ➢ Case 3b: g = f. Element f can't be in cache of S, so let e' be the element that S evicts.
  - ✓ if e' = e, S' accesses f from cache; now S and S' have same cache
  - ✓ if e' $\neq$ e, S' evicts e' and brings e into the cache; now S and S' have the same cache

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

Farthest-In-Future: Analysis

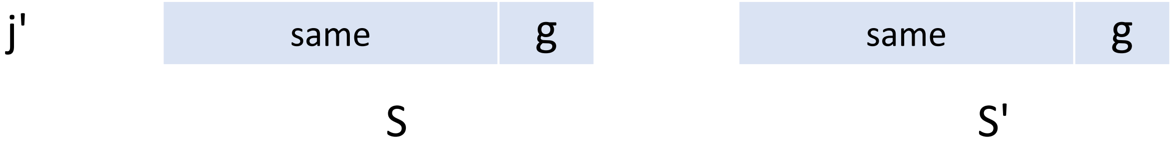
- Let j' be the first time after j+1 that S and S' take a different action, and let g be item requested at time j'.

↑ must involve e or f (or both)



otherwise S' would take the same action ↓

- ➢ Case 3c: $g \neq e, f$. S must evict e.
  Make S' evict f; now S and S' have the same cache. ▪



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.

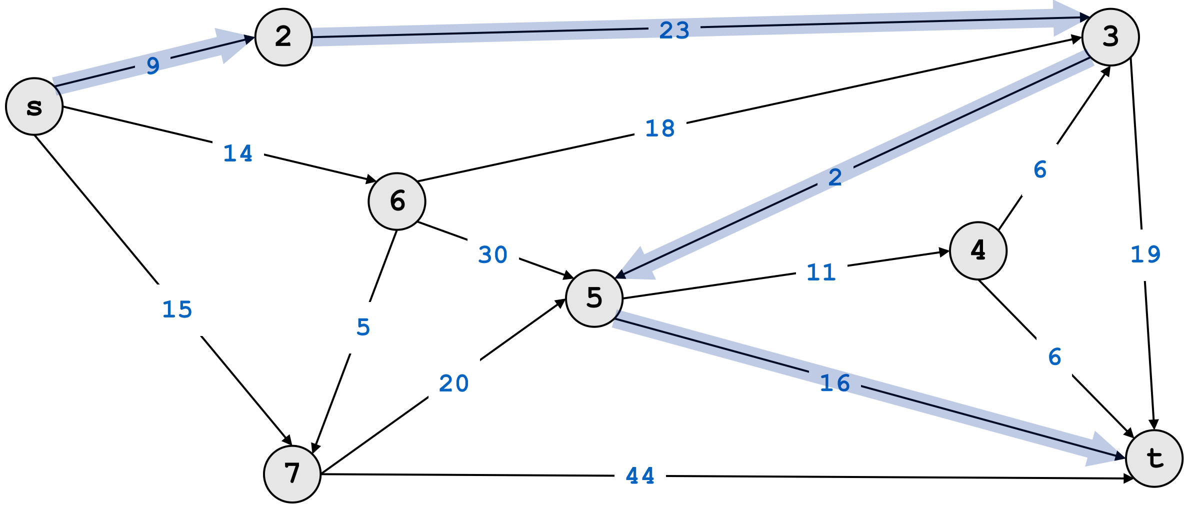
南方科技大学
SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY

5. Shortest Paths in a Graph

Shortest Path Problem

- Shortest path network.
  - Directed graph G = (V, E).
  - Source s, destination t.
  - Length ℓ_e = length of edge e.
- Shortest path problem: find shortest directed path from s to t.

cost of path = sum of edge costs in path



Cost of path s-2-3-5-t
= 9 + 23 + 2 + 16
= 50.

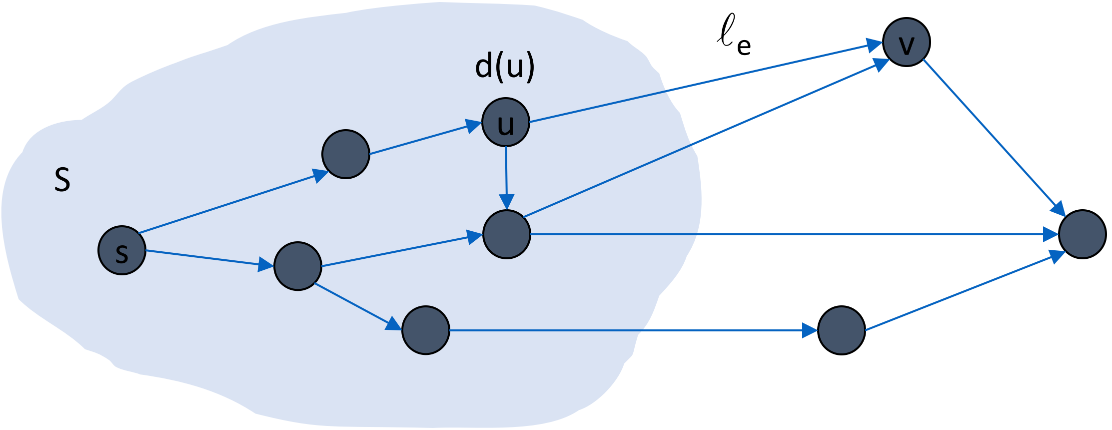
Dijkstra's Algorithm

- 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.
  - Repeatedly choose unexplored node v which minimizes

$$\pi(v) = \min_{e = (u,v) : u \in S} d(u) + \ell_e$$

shortest path to some u in explored part, followed by a single edge (u, v)

add v to S, and set d(v) = π(v).



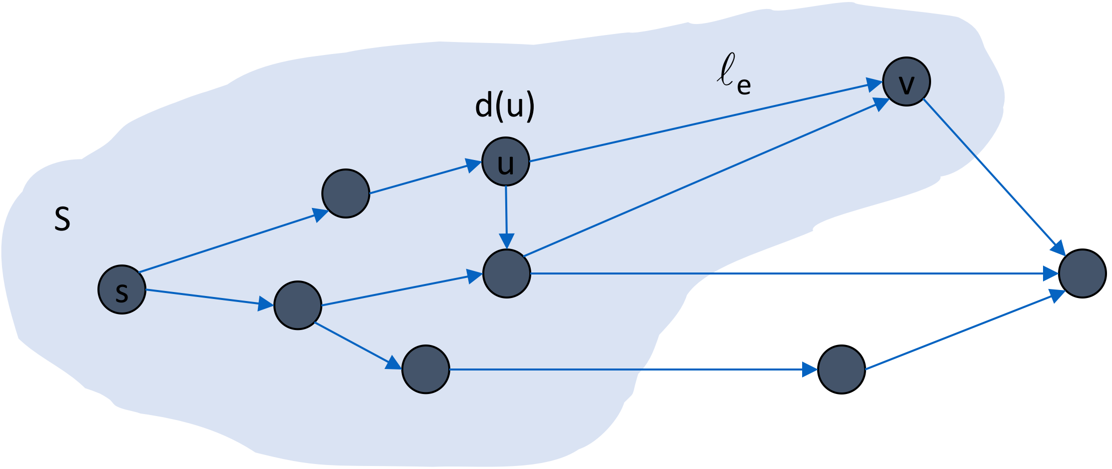
Dijkstra's Algorithm

- 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.
  - Repeatedly choose unexplored node v which minimizes

$$\pi(v) = \min_{e = (u,v) : u \in S} d(u) + \ell_e$$

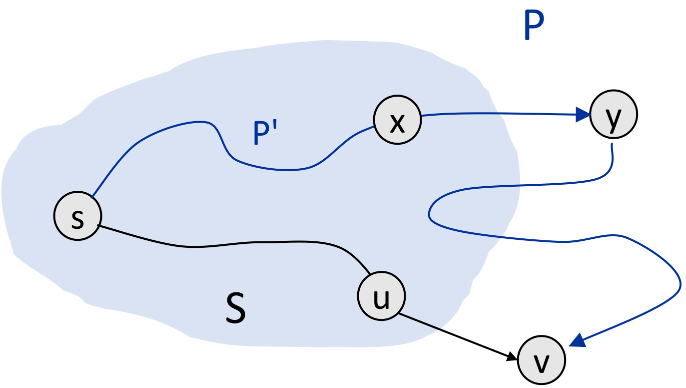
shortest path to some u in explored part, followed by a single edge (u, v)

add v to S, and set d(v) = π(v).



Dijkstra's Algorithm: Proof of Correctness

- Invariant. For each node $u \in S$, $d(u)$ is the length of the shortest s-u path.
- Pf. (by induction on |S|)
- Base case: $|S| = 1$ is trivial.
- 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 $\pi(v)$.
  - Consider any s-v path P. We'll see that it's no shorter than $\pi(v)$.
  - 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.



$$\underset{\substack{\uparrow \\ \text{nonnegative} \\ \text{weights}}}{\ell(P) \geq} \ell(P') + \ell(x,y) \underset{\substack{\uparrow \\ \text{inductive} \\ \text{hypothesis}}}{\geq} d(x) + \ell(x,y) \underset{\substack{\uparrow \\ \text{defn of } \pi(y)}}{\geq} \pi(y) \underset{\substack{\uparrow \\ \text{Dijkstra chose v} \\ \text{instead of y}}}{\geq} \pi(v)$$

Dijkstra's Algorithm: Implementation

- For each unexplored node, explicitly maintain $\pi(v) = \min_{e = (u,v) : u \in S} d(u) + \ell_e$.
  - Next node to explore = node with minimum $\pi(v)$.
  - When exploring v, for each incident edge e = (v, w), update

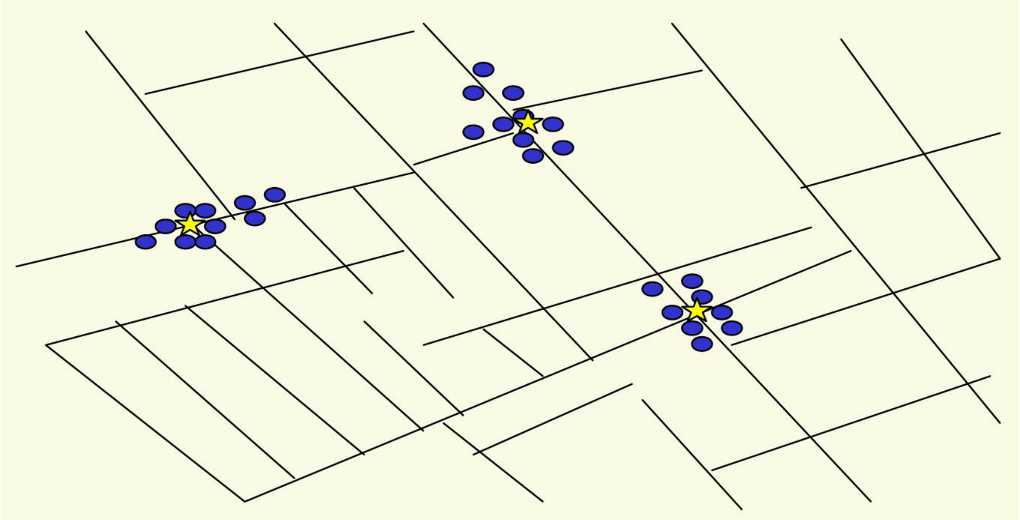
$$\pi(w) = \min \{ \pi(w), \pi(v) + \ell_e \}.$$

- Efficient implementation. Maintain a priority queue of unexplored nodes, prioritized by $\pi(v)$.

PQ Operation	Dijkstra	Array	Binary heap	d-way Heap	Fib heap †
Insert	n	n	log n	$d \log_d n$	1
ExtractMin	n	n	log n	$d \log_d n$	log n
ChangeKey	m	1	log n	$\log_d n$	1
IsEmpty	n	1	1	1	1
Total		n^2	m log n	$m \log_{m/n} n$	m + n log n

† Individual ops are amortized bounds

6. Clustering



Outbreak of cholera deaths in London in 1850s.
Reference: Nina Mishra, HP Labs

Clustering

- Clustering. Given a set U of n objects labeled $p_1, \ldots, p_n$, classify into coherent groups.

↑ photos, documents. micro-organisms

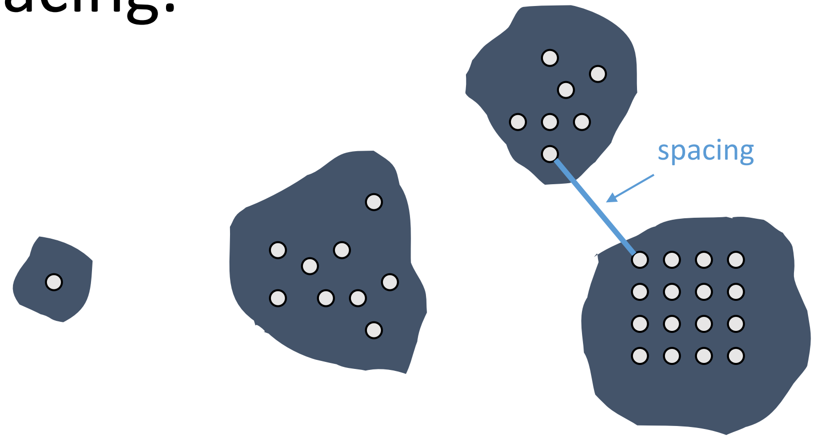
- Distance function. Numeric value specifying "closeness" of two objects.

↑ number of corresponding pixels whose intensities differ by some threshold

- Fundamental problem. Divide into clusters so that points in different clusters are far apart.
  - ➢ Routing in mobile ad hoc networks.
  - ➢ Identify patterns in gene expression.
  - ➢ Document categorization for web search.
  - ➢ Similarity searching in medical image databases
  - ➢ Skycat: cluster 10^9 sky objects into stars, quasars, galaxies.

Clustering of Maximum Spacing

- k-clustering. Divide objects into k non-empty groups.
- Distance function. Assume it satisfies several natural properties.
  - $d(p_i, p_j) = 0$ iff $p_i = p_j$ (identity of indiscernibles)
  - $d(p_i, p_j) \geq 0$ (nonnegativity)
  - $d(p_i, p_j) = d(p_j, p_i)$ (symmetry)
- Spacing. Min distance between any pair of points in different clusters.
- Clustering of maximum spacing. Given an integer k, find a k-clustering of maximum spacing.



k = 4

Greedy Clustering Algorithm

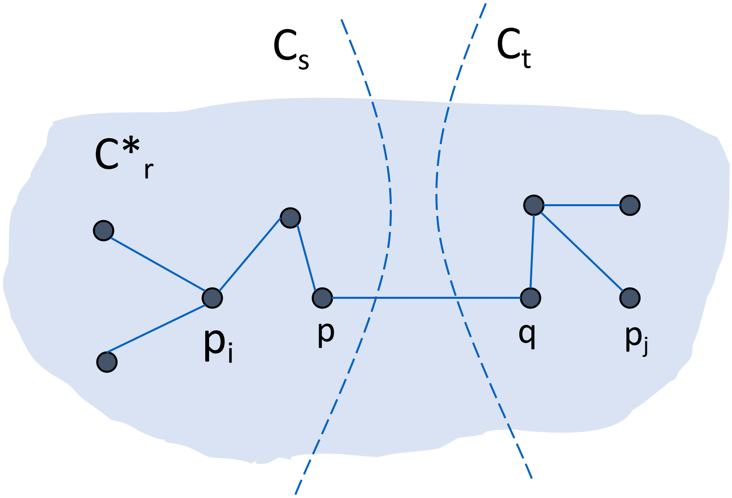
- Single-link k-clustering algorithm.
    - Form a graph on the vertex set U, corresponding to n clusters.
    - Find the closest pair of objects such that each object is in a different cluster, and add an edge between them.
    - Repeat n-k times until there are exactly k clusters.

- Key observation. This procedure is precisely Kruskal's algorithm (except we stop when there are k connected components).

- Remark. Equivalent to finding an MST and deleting the k-1 most expensive edges.

Greedy Clustering Algorithm: Analysis

- Theorem. Let C* denote the clustering $C^*_1, \ldots, C^*_k$ formed by deleting the k-1 most expensive edges of a MST. C* is a k-clustering of max spacing.



- Pf. Let C denote some other clustering $C_1, \ldots, C_k$.
  - The spacing of C* is the length d* of the (k-1)st most expensive edge.
  - Let p_i, p_j be in the same cluster in C*, say C^*_r, but different clusters in C, say C_s and C_t.
  - Some edge (p, q) on p_i-p_j path in C^*_r spans two different clusters in C.
  - All edges on p_i-p_j path have length $\leq$ d* since Kruskal chose them.
  - Spacing of C is $\leq$ d* since p and q are in different clusters. ▪

Extra Slides

Coin Changing

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.

Coin-Changing: Greedy Algorithm

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

```
Sort coins denominations by value: $c_1 < c_2 < \ldots < c_n$.

  coins selected
S ← $\phi$
while (x ≠ 0) {
   let k be largest integer such that $c_k \le x$
   if (k = 0)
      return "no solution found"
   x ← x - $c_k$
   S ← S ∪ {k}
}
return S
```

- Q. Is cashier's algorithm optimal?

Coin-Changing: Analysis of Greedy Algorithm

- Theorem. Greed is optimal for U.S. coinage: 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 greedy algorithm. ▪

k	c_k	All optimal solutions must satisfy	Max value of coins 1, 2, …, 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

Coin-Changing: Analysis of Greedy Algorithm

- Observation. Greedy algorithm is sub-optimal for US postal denominations: 1, 10, 21, 34, 70, 100, 350, 1225, 1500.

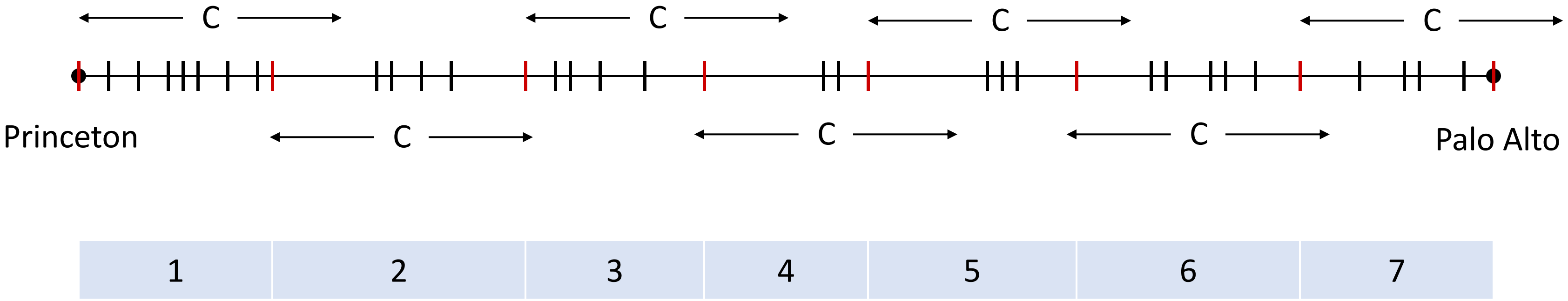
- Counterexample. 140¢.
  - Greedy: 100, 34, 1, 1, 1, 1, 1, 1.
  - Optimal: 70, 70.

Selecting Breakpoints

Selecting Breakpoints

- Selecting breakpoints.
  - Road trip from Princeton to Palo Alto along fixed route.
  - Refueling stations at certain points along the way.
  - Fuel capacity = C.
  - Goal: makes as few refueling stops as possible.

- Greedy algorithm. Go as far as you can before refueling.



Selecting Breakpoints: Greedy Algorithm

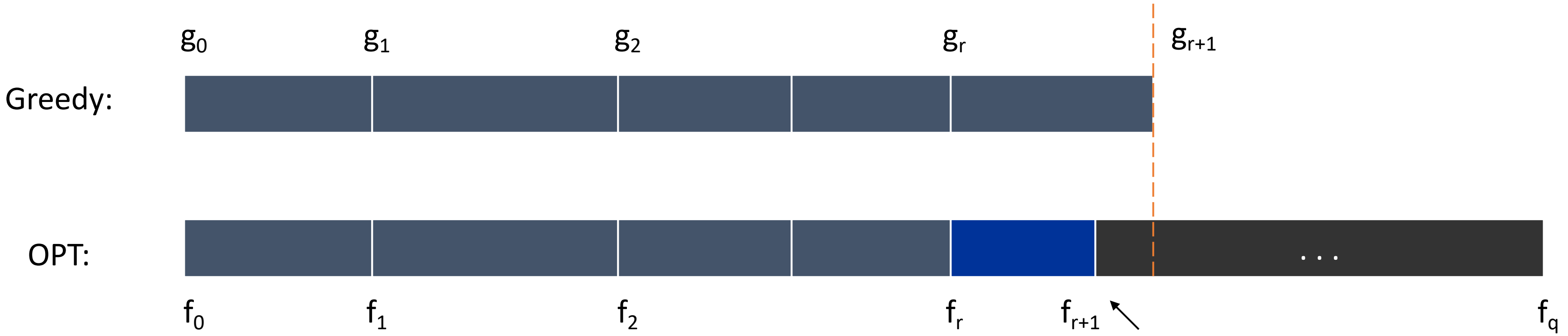
- Truck driver's algorithm.

```
Sort breakpoints so that: $0 = b_0 < b_1 < b_2 < ... < b_n = L$

$S \leftarrow \{0\}$ ← breakpoints selected
$x \leftarrow 0$     ← current location

while ($x \neq b_n$)
   let p be largest integer such that $b_p \leq x + C$
   if ($b_p = x$)
      return "no solution"
   $x \leftarrow b_p$
   $S \leftarrow S \cup \{p\}$
return S
```

- Implementation. O(n log n)
  - Use binary search to select each breakpoint p.

Selecting Breakpoints: Correctness

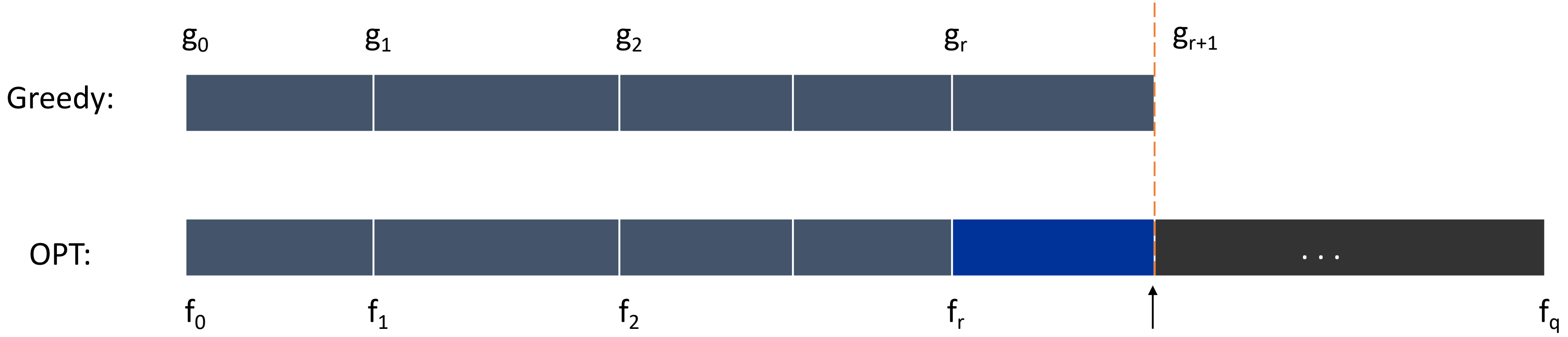
- Theorem. Greedy algorithm is optimal.
- Pf. (by contradiction)
  - ➢ Assume greedy is not optimal, and let's see what happens.
  - ➢ Let $0 = g_0 < g_1 < \ldots < g_p = L$ denote set of breakpoints chosen by greedy.
  - ➢ Let $0 = f_0 < f_1 < \ldots < f_q = L$ denote set of breakpoints in an optimal solution with $f_0 = g_0, f_1 = g_1, \ldots, f_r = g_r$ for largest possible value of r.
  - ➢ Note: $g_{r+1} > f_{r+1}$ by greedy choice of algorithm.



why doesn't optimal solution drive a little further?

Selecting Breakpoints: Correctness

- Theorem. Greedy algorithm is optimal.
- Pf. (by contradiction)
  - Assume greedy is not optimal, and let's see what happens.
  - Let $0 = g_0 < g_1 < \ldots < g_p = L$ denote set of breakpoints chosen by greedy.
  - Let $0 = f_0 < f_1 < \ldots < f_q = L$ denote set of breakpoints in an optimal solution with $f_0 = g_0, f_1 = g_1, \ldots, f_r = g_r$ for largest possible value of r.
  - Note: $g_{r+1} > f_{r+1}$ by greedy choice of algorithm.



another optimal solution has
one more breakpoint in common
⇒ contradiction

Edsger W. Dijkstra

The question of whether computers can think is like the question of whether submarines can swim.

Do only what only you can do.

In their capacity as a tool, computers will be but a ripple on the surface of our culture. In their capacity as intellectual challenge, they are without precedent in the cultural history of mankind.

The use of COBOL cripples the mind; its teaching should, therefore, be regarded as a criminal offence.

APL is a mistake, carried through to perfection. It is the language of the future for the programming techniques of the past: it creates a new generation of coding bums.