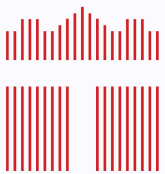
Slido: #ADA2020

CSIE 2136 Algorithm Design and Analysis, Fall 2020



National Taiwan University
國立臺灣大學

Approximation Algorithms

Hsu-Chun Hsiao

3-week Agenda

- NP-Completeness Overview
  - Warm up: four color problem
  - Decision vs. optimization
  - Complexity classes
  - Reduction
  - P-time solving vs. verification
- Proving NP-Completeness
  - Formula satisfiability problem
  - 3-CNF-SAT
  - The clique problem
  - The vertex-cover problem
  - The independent-set problem
  - Traveling salesman problem
  - Hamiltonian cycle
- Approximation algorithms
  - Vertex Cover
  - TSP
  - 3-CNF-SAT
- Randomized algorithms
  - Karger's min-cut algorithm
  - Probabilistic data structures

Coping with NP-Hard problems

- Since polynomial-time solutions are unlikely (unless P = NP), we must sacrifice either optimality, efficiency, or generality

- Approximation algorithms: guarantee to be a fixed percentage away from the optimum
- Local search: simulated annealing (hill climbing), genetic algorithms, etc.
- Heuristics: no formal guarantee of performance
- Randomized algorithms: make use of a randomizer (random number generator) for operation

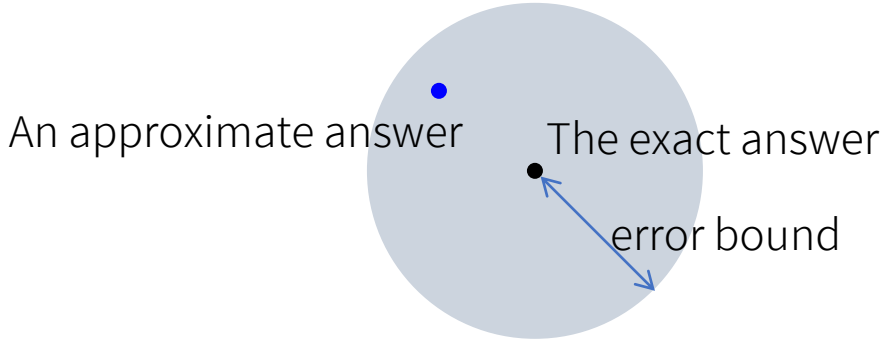
Coping with NP-Hard problems

- Since polynomial-time solutions are unlikely (unless P = NP), we must sacrifice either optimality, efficiency, or generality

- Pseudo-polynomial time algorithms: e.g., dynamic programming for the 0-1 Knapsack problem
- Exponential algorithms/Intelligent exhaustive search: feasible only when the problem size is small
- Restriction: work on some special cases of the original problem. e.g., the maximum independent set problem in circle graphs

What is approximation?

- "A value or quantity that is nearly but not exactly correct"
- Approximation algorithms for optimization problems: the approximate solution is guaranteed to be close to the exact solution (i.e., the optimal value)
  - Cf. heuristics search: no guarantee
  - Note: we cannot approximate decision problems



Why approximation algorithms?

- Most practical optimization problems are NP-hard
  - Recall that it's widely believed that P ≠ NP
  - Since polynomial-time solutions are unlikely, we must sacrifice either optimality, efficiency, or generality
- Approximation algorithms sacrifice optimality, return near-optimal answers
  - How "near" is near-optimal?

Approximation algorithms

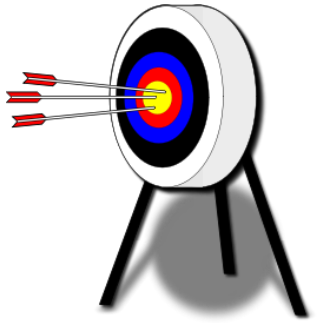
- $\rho(n)$ -approximation algorithm
  - Efficient: guaranteed to run in polynomial time
  - General: guaranteed to solve every instance of the problem
  - Near-optimal: guaranteed to find solution within a factor of $\rho(n)$ of the cost of an optimal solution
- Approximation ratio $\rho(n)$
  - n: input size
  - C^*: cost of an optimal solution
  - C: cost of the solution produced by the approximation algorithm

$$\max\left(\frac{C}{C^*}, \frac{C^*}{C}\right) \leq \rho(n)$$

Maximization problem: $\frac{C^*}{C} \leq \rho(n)$

Minimization problem: $\frac{C}{C^*} \leq \rho(n)$

Approximate ratio ρ(n)



$$\max\left(\frac{C}{C^*}, \frac{C^*}{C}\right) \leq \rho(n)$$

n: input size
C^*: cost of optimal solution
C: cost of approximate solution

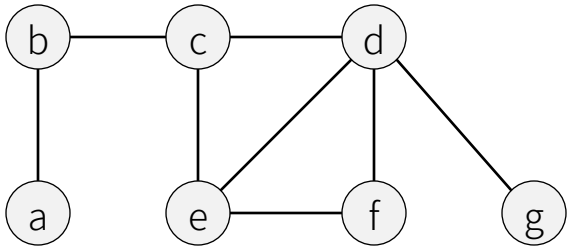
- $\rho(n) \geq 1$
- $\rho(n)$ 越小越好！
- An exact algorithm has $\rho(n) = 1$

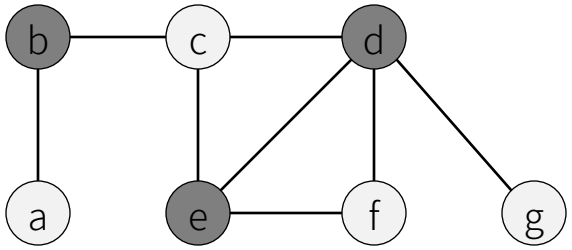
- Challenge: prove that C is close to C^* without knowing C^*!

Approximate Vertex-Cover

The vertex-cover problem

- A vertex cover of $G = (V, E)$ is a subset $V' \subseteq V$ such that if $(w, v) \in E$, then $w \in V'$ or $v \in V'$ or both
  - A vertex cover "covers" every edge in G
- Optimization problem: find a vertex cover of minimum size in G
  - The decision version is NP-complete





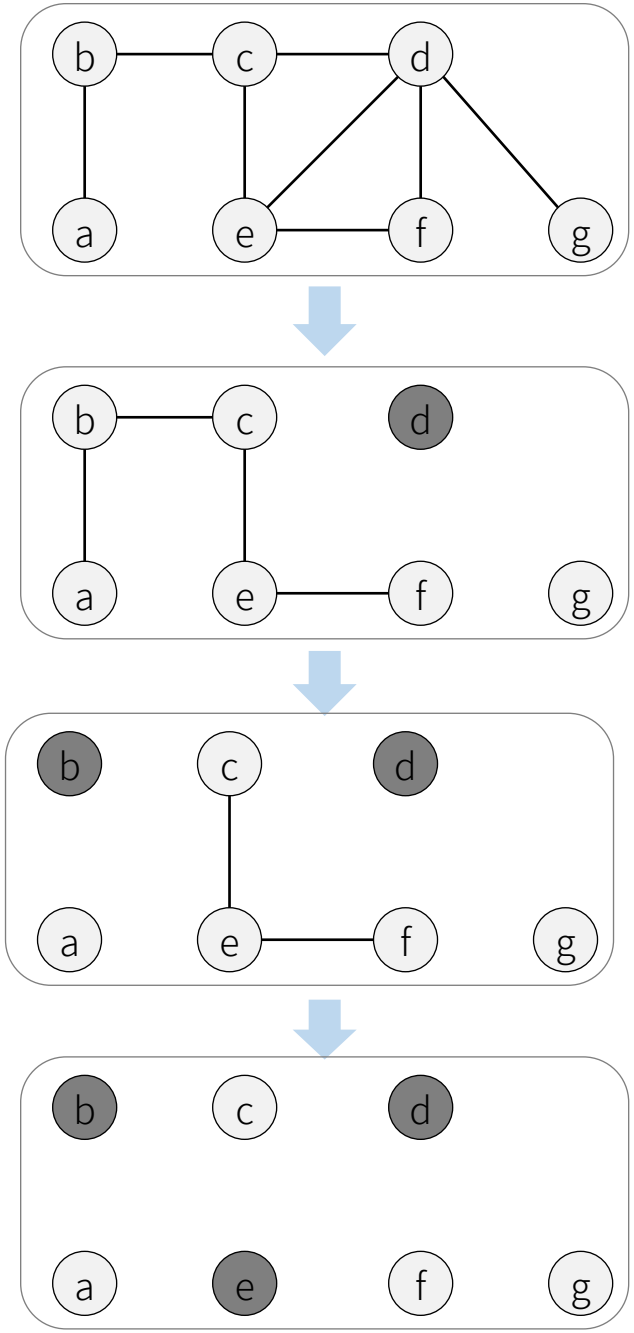
b, d, e is a minimum vertex cover (size = 3)

A greedy heuristic to Vertex-Cover

- Greedy heuristic: In each iteration, cover as many edges as possible (vertex with the maximum degree) and then delete the covered edges.

Q: can this greedy heuristic always find an optimal solution?

No



b, d, e is a vertex cover of size 3 found by the greedy algorithm (and it's optimal!)

A greedy heuristic to Vertex-Cover

- **Greedy heuristic:** cover as many edges as possible (vertex with the maximum degree) at each stage and then delete the covered edges.
- The greedy heuristic cannot always find an optimal solution (otherwise we would have proven P = NP!)
- Moreover, it seems no guarantee that C is always close to C^*.

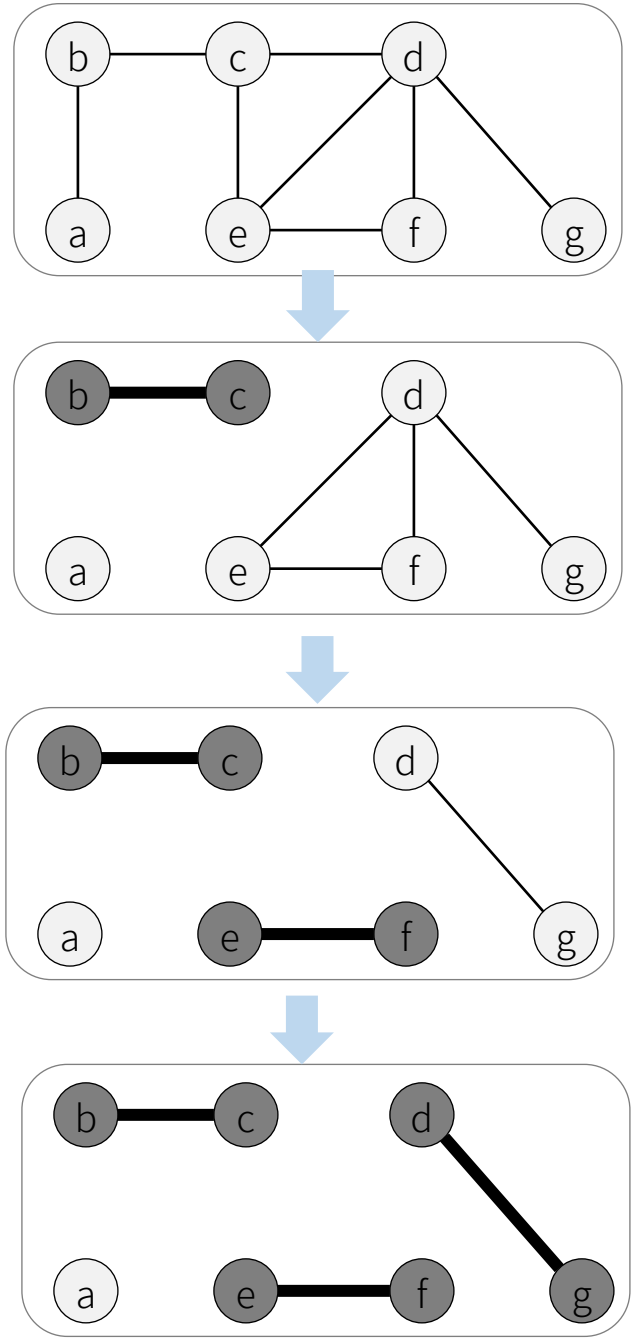
Slides provided by Prof. Yao-Wen Chang

An approximation algorithm to Vertex-Cover

```
APPROX-VERTEX-COVER(G)
1. S = ∅
2. E' = G.E
3. while E' ≠ ∅
4.     let (u, v) be an arbitrary edge of E'
5.     S = S ∪ (u, v)
6.     remove from E' every edge incident on either u or v
7. return S
```

- APPROX-VERTEX-COVER
  - Randomly select one edge at a time
  - Add both vertices to the cover
  - Remove all incident edges
- Running time = $O(V + E)$
- Claim: Approximation ratio $\rho(n) = 2$

An approximation algorithm to Vertex-Cover

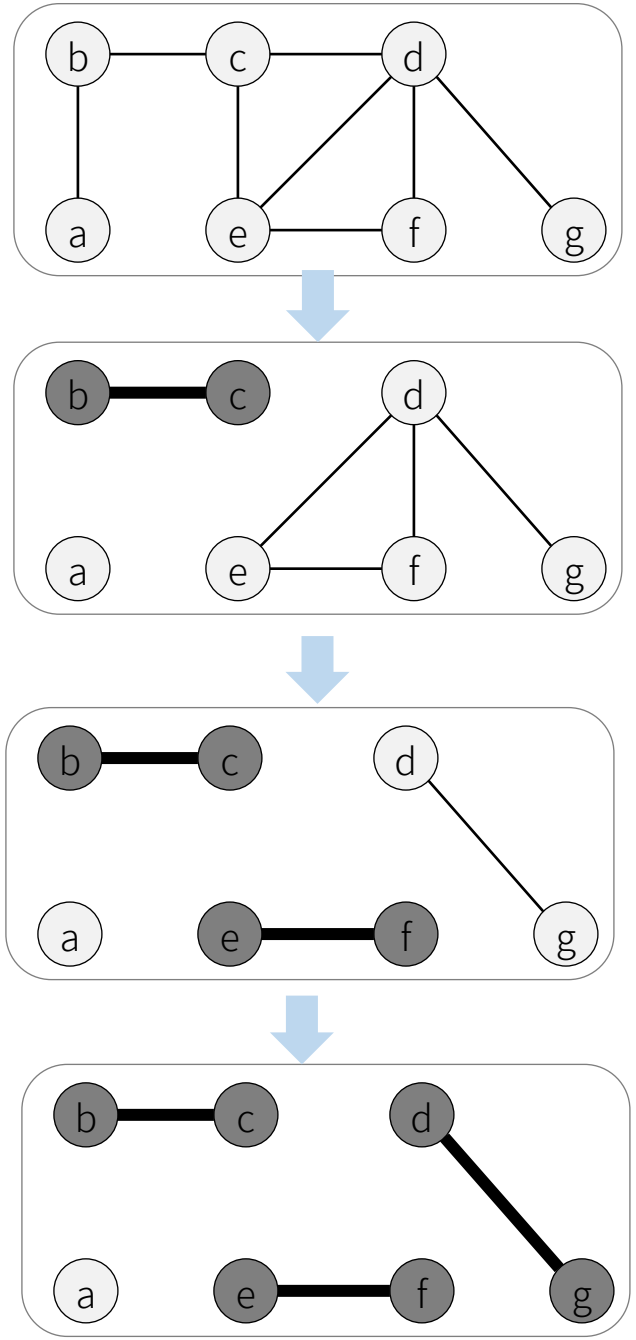


b, c, d, e, f, g is a vertex cover of size 6 found by the approximation algorithm (not optimal!)

APPROX-VERTEX-COVER is 2-approximation

- Let A denote the set of edges picked in line 4, $|S| = 2|A|$
- In *any* vertex cover S (including S^*), every vertex v in S covers at most one edge in A because no two edges in A share a vertex.
- $\Rightarrow |A| \leq |S^*|$
- $\Rightarrow \frac{1}{2}|C| = |A| \leq |S^*|$
- $\Rightarrow \rho(n) = 2$

Note: the proof doesn't require knowing the actual value of $C^* = |S^*|$

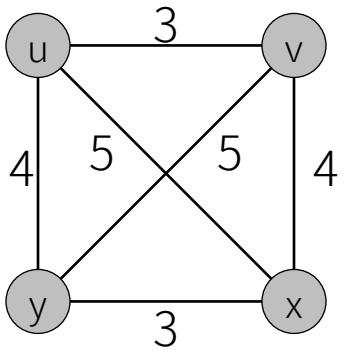


Approximate TSP

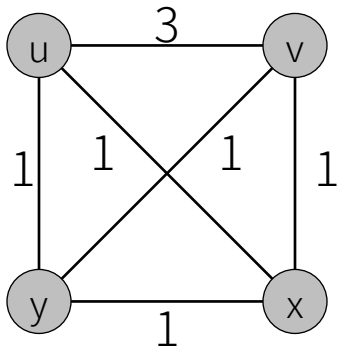
Traveling Salesman Problem (TSP)

- Optimization problem: Given a set of cities and their pairwise distances, find a tour of lowest cost that visits each city exactly once.
- We say that the Inter-city distances satisfy triangle inequality if $\forall u, v, w \in V, d(u, w) \leq d(u, v) + d(v, w)$.

Satisfy triangle inequality



Do not satisfy triangle inequality

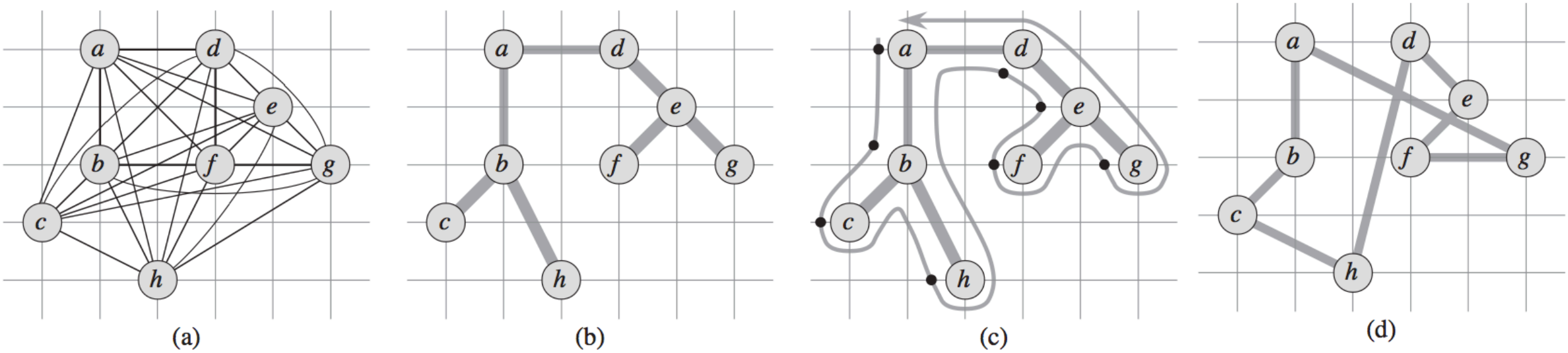


An approximation algorithm for TSP with triangle inequality

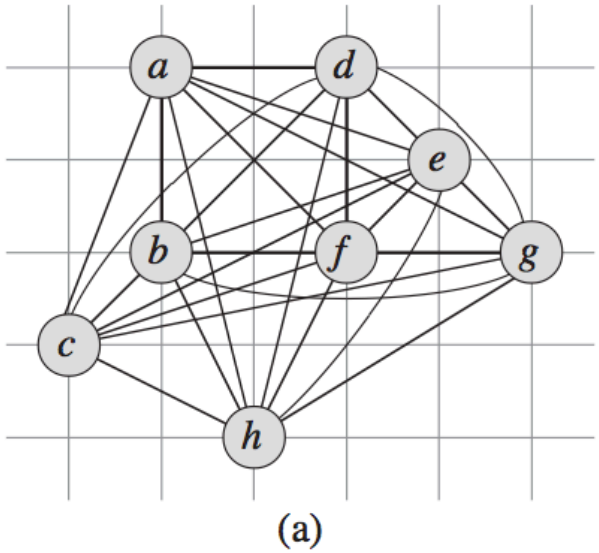
```
APPROX-TSP-TOUR (G)
1. select a vertex r ∈ G.V to be a "root" vertex
2. grow a minimum spanning tree T for G
3. let H be the list of vertices visited in a preorder tree walk of T
4. return H
```

- Running time is dominated by finding a MST
  - MST is in P: $O(V^2)$ when using adjacency matrix
- Claim: Approximation ratio $\rho(n) = 2$

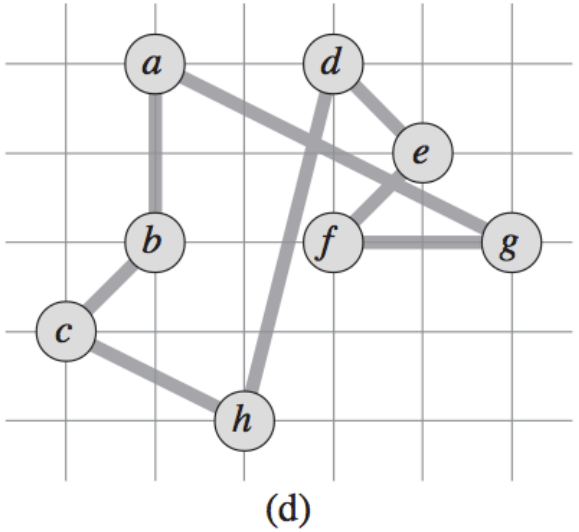
An approximation algorithm for TSP with triangle inequality



An approximation algorithm for TSP with triangle inequality

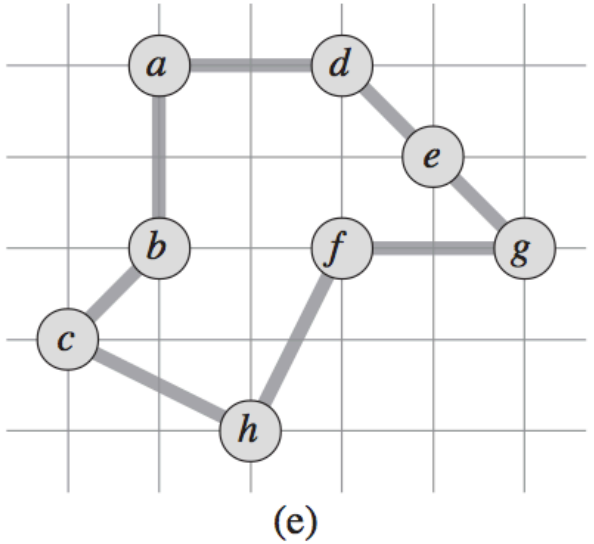


(a)



(d)

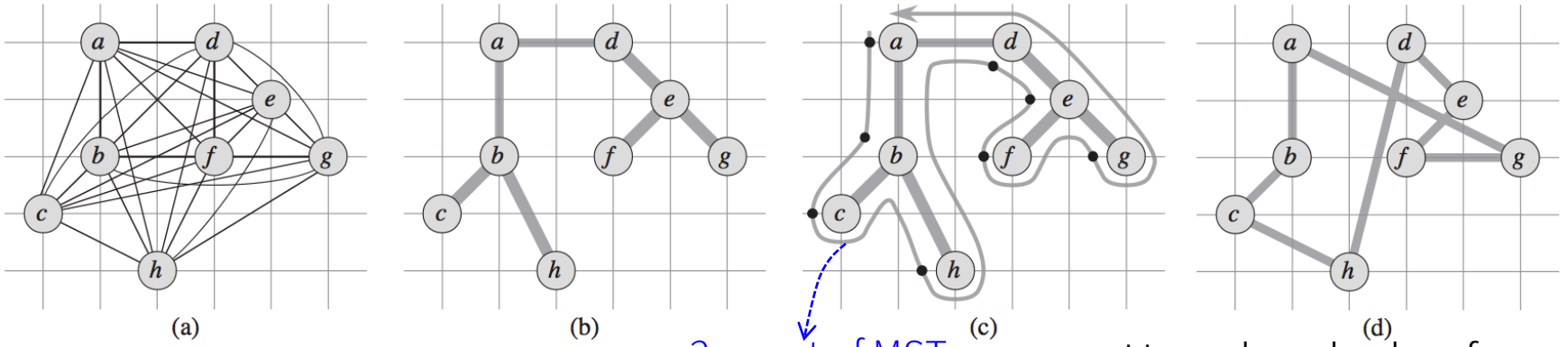
H = a, b, c, h, d, e, f, g, a



(e)

OPT H^* = a, b, c, h, f, g, e, d, a

APPROX-TSP-TOUR with triangle inequality is 2-approximation



- With triangle inequality: $cost(H) \leq 2 \times$ cost of MST
- Let H^* denote an optimal tour
- $\Rightarrow H^*$ is formed by some tree T plus some edge e, i.e., $cost(H^*) = \text{cost}(T) + \text{cost}(e)$
- $\Rightarrow$ cost of MST $\leq cost(H^*)$
- $\Rightarrow cost(H) \leq 2 \times$ cost of MST $\leq 2 \times cost(H^*)$
- $\Rightarrow \rho(n) = 2$

Theorem 35.3 General TSP (when triangle inequality may not hold)

If P ≠ NP, there is no polynomial-time approximation algorithm with a constant ratio bound ρ for the general TSP.

Proof by contradiction

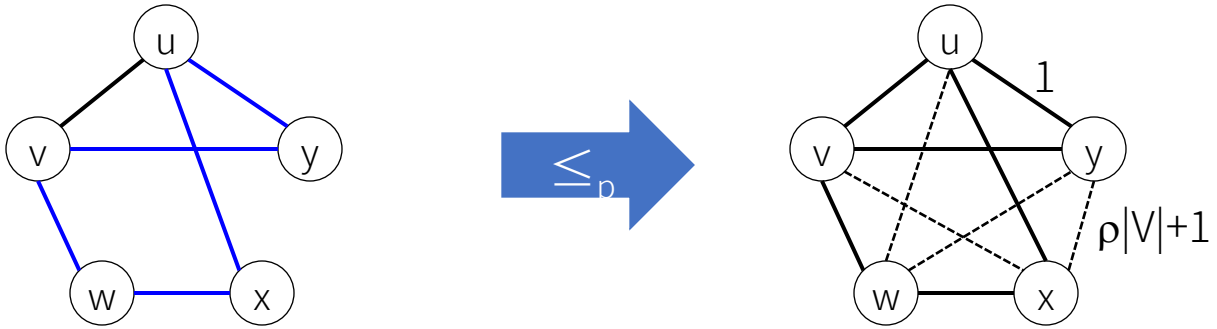
- Suppose there is such an algorithm A to approximate TSP with a constant ρ. We will use A to solve HAM-CYCLE in polynomial time.
- Consider the following algorithm for HAM-CYCLE:
  - Convert $G = (V, E)$ into an instance I of TSP with cities V (resulting in a complete graph $G' = (V, E')$): $c(u, v) = \begin{cases} 1 & \text{if } (u, v) \in E \\ \rho\,|V| + 1 & \text{otherwise .} \end{cases}$
  - Run A on I.
  - If the reported cost $\leq \rho|V|$, then return "Yes" (i.e., G contains a tour that is an Hamiltonian cycle), else return "No."

Proof by contradiction (cont'd)

- If G has an HC: G' contains a tour of cost $|V|$ by picking edges in E, each with cost of 1.
- If G does not have an HC: any tour of G' must use some edge not in E, which has a total cost of $\geq (\rho|V| + 1) + (|V| - 1) > \rho|V|$.
- Algorithm A guarantees to return a tour of cost $\leq \rho\, cost(H^*)$

$\Rightarrow$ A returns a cost $\leq \rho|V|$ if G contains an HC; A returns a cost $> \rho|V|$, otherwise.

$\Rightarrow$ HAM-CYCLE can be solved in polynomial time, contradiction!

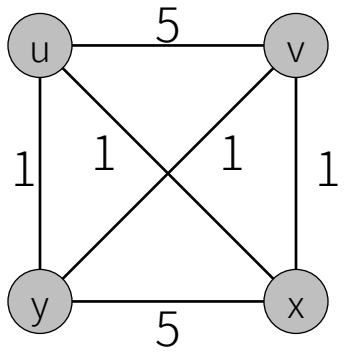


u, y, v, w, x, u is a Hamiltonian Cycle

u, y, v, w, x, u is a traveling-salesman tour with cost |V|

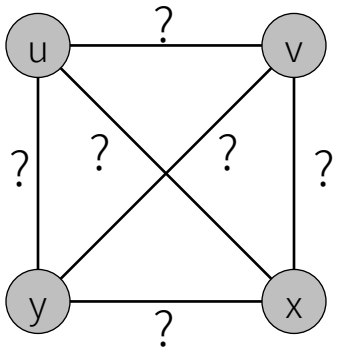
Exercise 35.2-2 Show how in polynomial time we can transform one instance of the traveling-salesman problem into another instance whose cost function satisfies the triangle inequality. The two instances must have the same set of optimal tours. Explain why such a polynomial-time transformation does not contradict **Theorem 35.3**, assuming that P≠NP.

General TSP



$\leq_p$

TSP satisfying triangle inequality

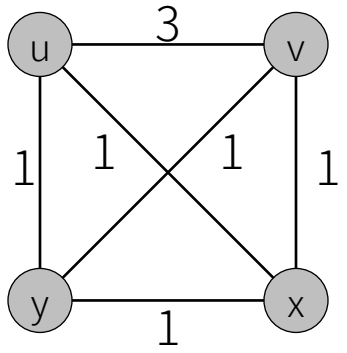


Exercise 35.2-2

- For example, we can add d_{max} (the largest cost) to each edge
- G contains a tour of minimum cost k ⇔ G' contains a tour of minimum cost $k + d_{max} * |V|$
- G' satisfies triangle inequality because $\forall u, v, w \in V$,

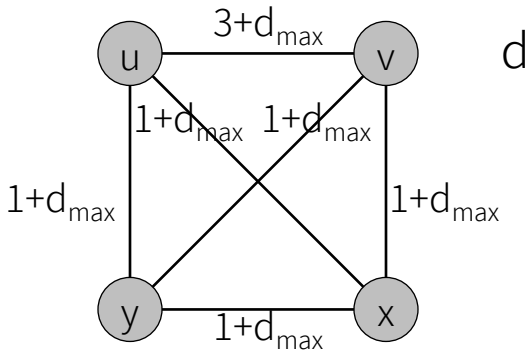
$$d'(u, w) = d(u, w) + d_{max} \leq 2 * d_{max} \leq d'(u, v) + d'(v, w)$$

General TSP



$\leq_p$

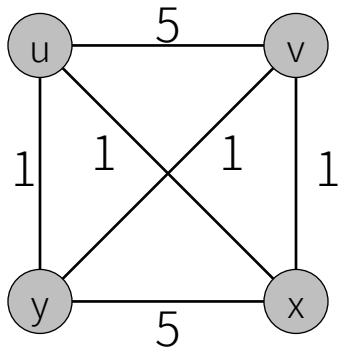
TSP satisfying triangle inequality



$d_{max} = 3$

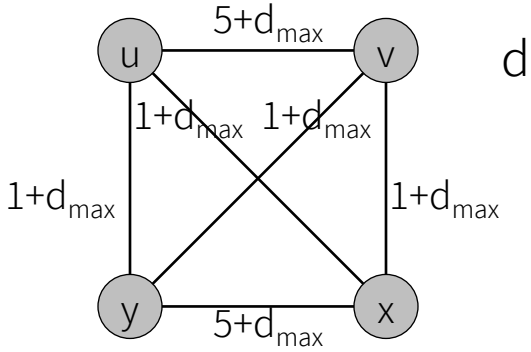
Exercise 35.2-2

General TSP



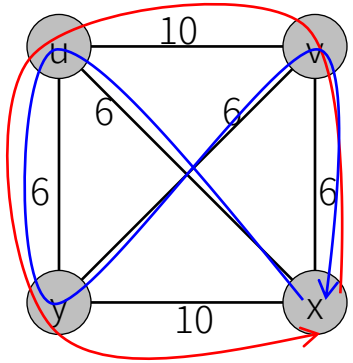
$\leq_p$

TSP satisfying triangle inequality



$d_{max} = 5$

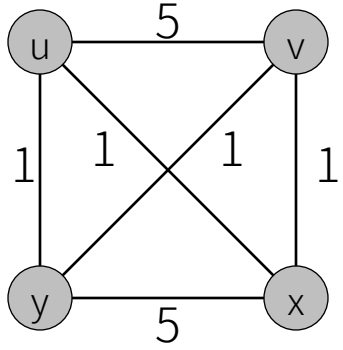
approximate



Cost(H) = 32

Cost(H*) = 24

Cost(H)/Cost(H*) ≤ 2



Cost(H) = 12

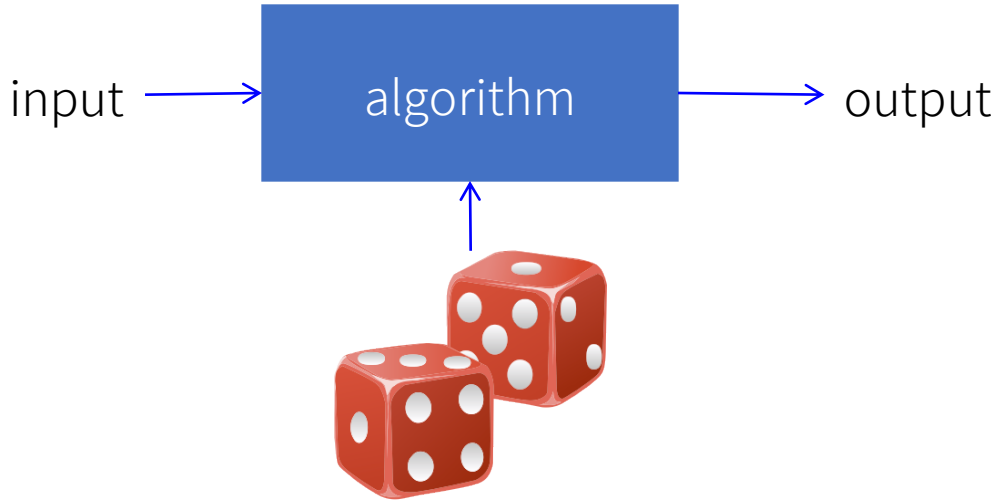
Cost(H*) = 4

Cost(H)/Cost(H*) > 2!

Randomized Approximate 3-CNF-SAT

Randomness

- A randomized algorithm is an algorithm that employs a degree of randomness as part of its logic
- A randomized data structure is a data structure that employs a degree of randomness as part of its logic



Randomized approximation algorithm

- Randomized algorithm's behavior is determined not only by its input but also by values produced by a random-number generator

	Exact	Approximate
Deterministic	MST	APPROX-TSP-TOUR
Randomized	Quick Sort	MAX-3-CNF-SAT

MAX-3-CNF-SAT

- **3-CNF-SAT**: Satisfiability of Boolean formulas in 3-conjunctive normal form (3-CNF)
  - 3-CNF = AND of clauses, each is the OR of exactly 3 distinct literals
  - A literal is an occurrence of a variable or its negation, e.g., x_1 or $\neg x_1$
- 3-CNF-SAT is a decision problem. What should be an optimization version of 3-CNF-SAT?

MAX-3-CNF-SAT

- MAX-3-CNF-SAT: find an assignment of the variables that satisfies as many clauses as possible
  - Closeness to optimum is measured by the fraction of satisfied clauses

$$(x_1 \vee \neg x_1 \vee \neg x_2) \wedge (x_3 \vee x_2 \vee x_4) \wedge (\neg x_1 \vee \neg x_3 \vee \neg x_4)$$

$\langle x_1, x_2, x_3, x_4 \rangle = \langle 0, 0, 1, 1 \rangle$ satisfies 3 clauses
$\langle x_1, x_2, x_3, x_4 \rangle = \langle 1, 0, 1, 1 \rangle$ satisfies 2 clauses

Note that this clause is always SAT since it always evaluates to 1. For simplicity, we can assume no clause containing both of a literal and its negation.

Randomized approximation algorithm for MAX-3-CNF-SAT

- A randomized 8/7-approximation algorithm:
  - 丟硬幣決定變數要設成0或是1

然後呢？

然後就沒有了！

Theorem 35.6

Given an instance of MAX-3-CNF-SAT with n variables $x_1, x_2, \ldots, x_n$ and m clauses, the randomized algorithm that independently sets each variable to 1 with probability 1/2 and to 0 with probability 1/2 is a randomized 8/7-approximation algorithm.

* Satisfying 7/8 of the clauses in expectation

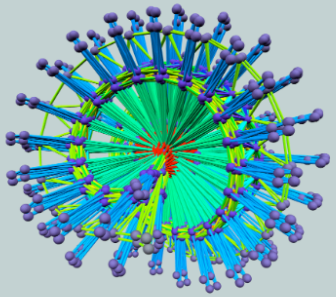
Theorem 35.6

Given an instance of MAX-3-CNF-SAT with n variables $x_1, x_2, \ldots, x_n$ and m clauses, the randomized algorithm that independently sets each variable to 1 with probability 1/2 and to 0 with probability 1/2 is a randomized 8/7-approximation algorithm.

Assume no clause contains both a variable and its negation (Exercise 35.4-1 remove this assumption)

Proof

- Each clause is the OR of exactly 3 distinct literals
- $\Pr[x_i = 0] = \Pr[x_i = 1] = 1/2$
- $\Rightarrow$ for all $x_1 \neq x_2 \neq x_3$, $\Pr[(x_1 \vee x_2 \vee x_3) = 0] = 1/8$
- $\Rightarrow$ E[# of satisfied clauses] = m * E[clause j is satisfied]
  $\geq m * (1 - \frac{1}{8}) = \frac{7}{8}m$
- $\Rightarrow$ $\rho(n)$ = max # of satisfied clauses / E[# of satisfied clauses] = 8/7



SAT Competition 2020

Affiliated with the 23rd International Conference on Theory and Applications of Satisfiability Testing taking place on the 5th - 9th of July 2020 in Alghero, Italy.

The 2020 SAT Competition is a competitive event for solvers of the Boolean Satisfiability (SAT) problem. It is organized as a satellite event to the 23rd International Conference on Theory and Applications of Satisfiability Testing and stands in the tradition of the yearly SAT Competitions and SAT-Races / Challenges.

Objective

The area of SAT Solving has seen tremendous progress over the last years. Many problems (e.g. in hardware and software verification) that seemed to be completely out of reach a decade ago can now be handled routinely. Besides new algorithms and better heuristics, refined implementation techniques turned out to be vital for this success.

To keep up the driving force in improving SAT solvers, we want to motivate implementors to present their work to a broader audience and to compare it with that of others.

Tracks

SAT Competition 2020 will consist of the following tracks*:

- Main Track
  - with Glucose-Hack Award

～以下是補充～

Randomized Min-Cut Algorithms

Min-Cut

- Max-flow min-cut theorem shows max-flow and min-cut are dual problems
- Max-flow can be found in polynomial time

Karger's Min-Cut Algorithm

- In each iteration
  1. Pick a random edge and merge the two nodes
  2. Repeat until only two nodes left
     - Keep parallel edges, but remove self-loops
  3. Output the partition as a guess of min cut
- Claim: Pr[found a min cut in one iteration] >= $2/n^2$
- Practice: show that
  - The average degree of a node is $2|E|/n$
  - The size of the minimum cut is at most $2|E|/n$
  - If an edge is picked at random, the probability that it lies across the minimum cut is at most $2/n$.
  - Use these to prove the claim is correct

https://www.cs.princeton.edu/courses/archive/fall13/cos521/lecnotes/lec2final.pdf

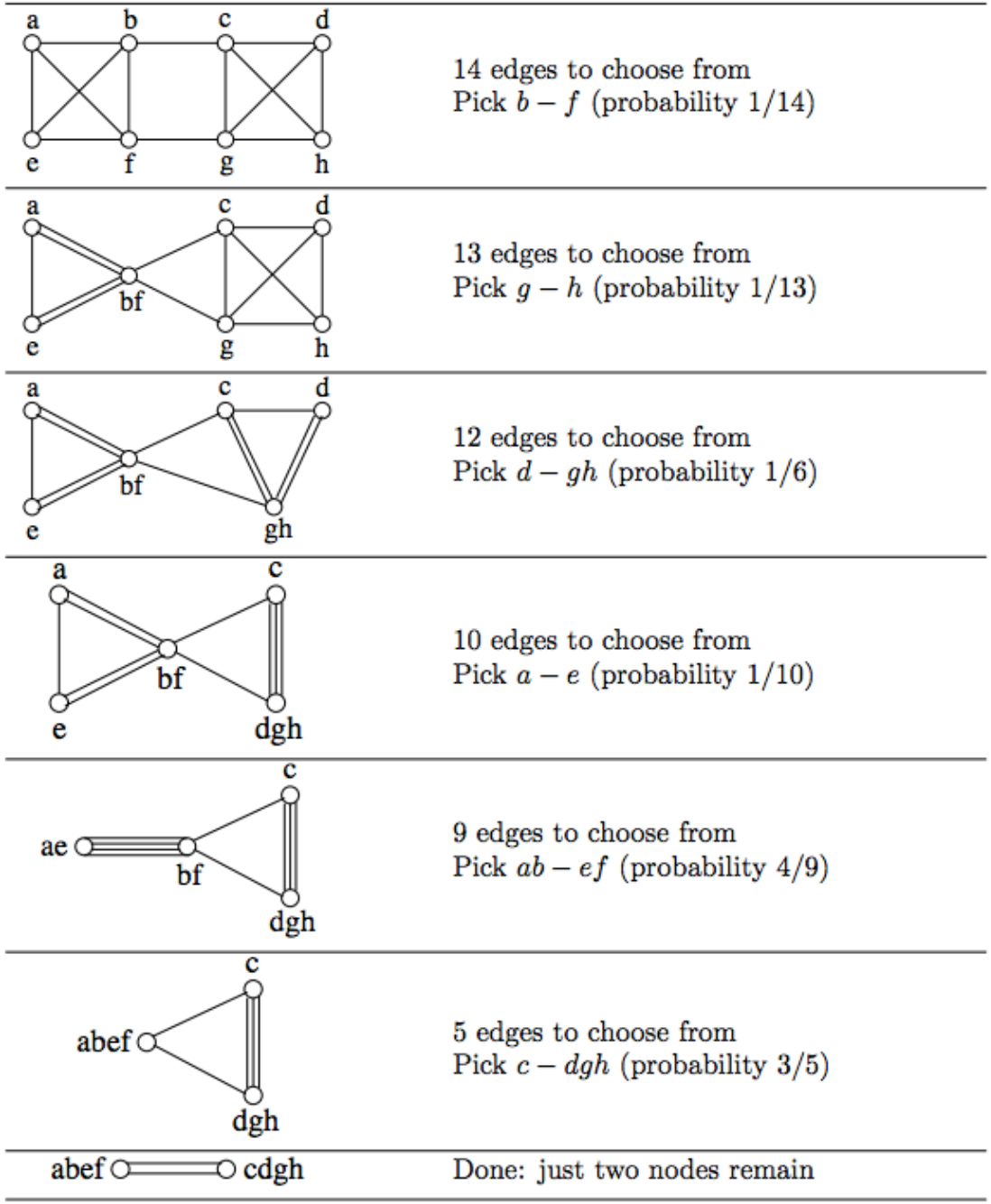


Figure 4.1. Karger's algorithm at work.

https://www.cs.princeton.edu/courses/archive/fall13/cos521/lecnotes/lec2final.pdf

Alternative Implementation of the Karger's Min-Cut Algorithm

1. Assign each edge a random weight
2. Run Kruskal's algorithm to get the minimum spanning tree
3. Break the largest edge in the tree to get the two clusters

Q: Why is this equivalent to the previous implementation?

https://www.cs.princeton.edu/courses/archive/fall13/cos521/lecnotes/lec2final.pdf

Probabilistic Data Structure: Bloom Filters

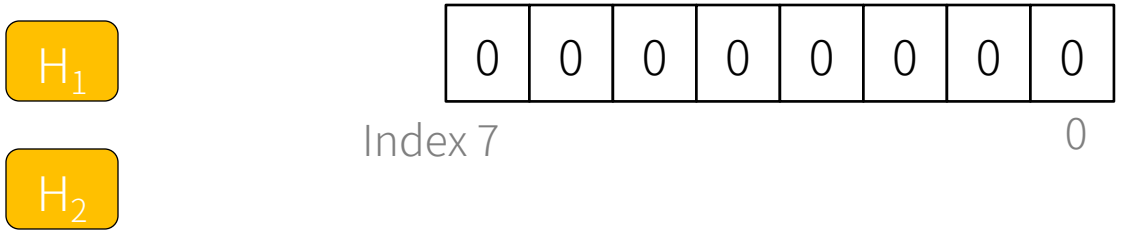
Membership testing

- Membership testing: check whether an element is in a set
- Many applications require membership testing:
  - Does an edge (u, v) exist in a graph G?
  - Is this URL on our blacklist?
  - Has a ticket been used already (duplicate detection)?
  - Does this software match the malware database?
- Required operations
  - Insert(e): add an element e to the set
  - IsMember(e): check if e is in the set
- What data structure can we use?

Bloom filter (布隆過濾器)

- A space-efficient probabilistic data structure
  - Insert: O(1)
  - IsMember: O(1)
- A Bloom filter is a bit vector of size m initialized to zeros
- There are also k hash functions H_i such that
  - H_i maps an element to a position in the vector uniformly at random
  - H_is are mutually independent
  - Note that H_i is a function: output is the same given the same input

Example: m = 8, k = 2

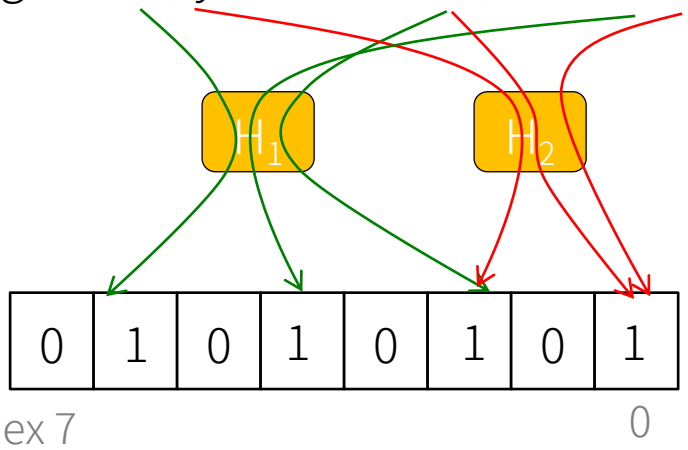


Bloom filter: insertion

- A Bloom filter is a bit vector of size m initialized to zeros
- There are also k hash functions H_i such that
  - H_i maps an element to a position in the vector uniformly at random
  - H_is are mutually independent
- **Insert(e)**: set the bit at position $H_i(e)$ to 1 for all i

Example: m = 8, k = 2, a list of URLs as a whitelist

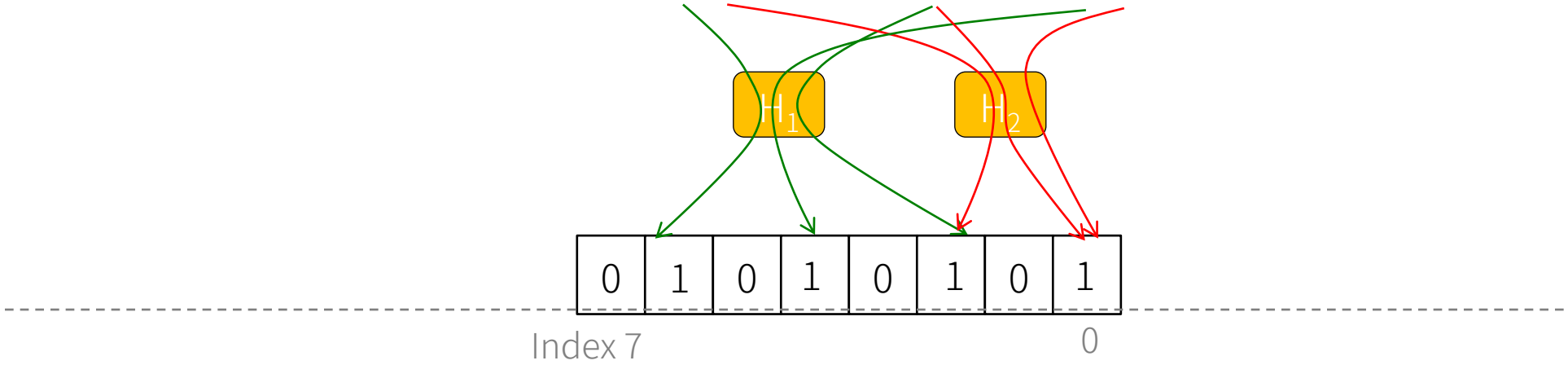
Insert {google.com, youtube.com, ntu.edu.tw}



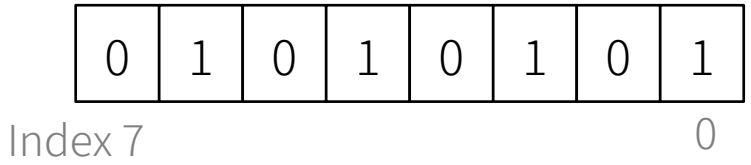
Bloom filter: membership test

Example: m = 8, k = 2, a list of URLs as a whitelist

Insert {google.com, youtube.com, ntu.edu.tw}



Membership test

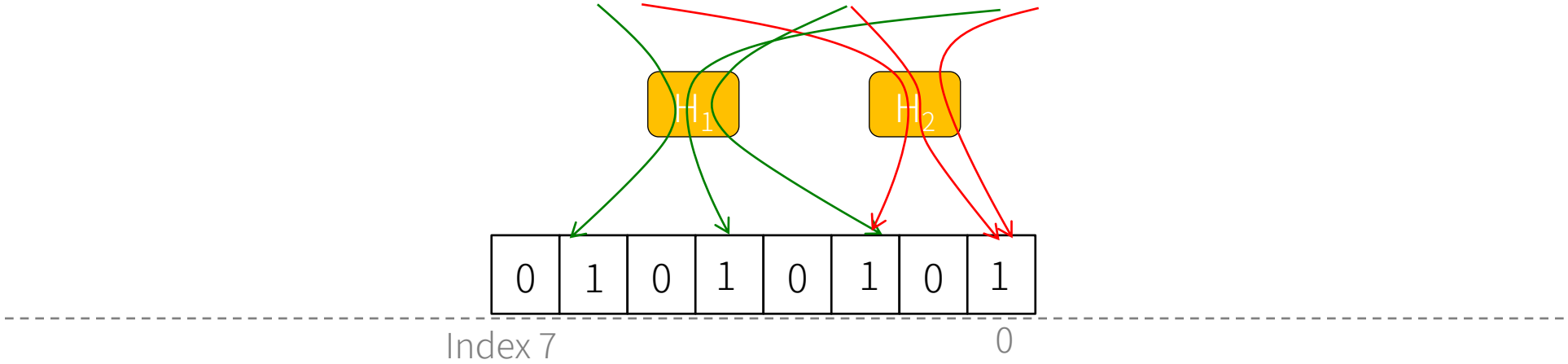


IsMember(e): return YES if all bits at position $H_i(e)$ are 1s; otherwise, return NO

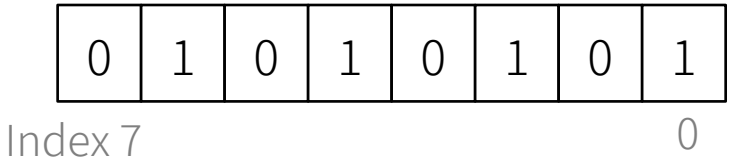
Bloom filter: membership test

Example: m = 8, k = 2, a list of URLs as a whitelist

Insert {google.com, youtube.com, ntu.edu.tw}



Membership test



Suppose H_1(a.com) = 2, H_2(a.com) = 5. Is a.com on the whitelist? NO

H_1(google.com) = 6, H_2(google.com) = 2. Is google.com on the whitelist? YES

Suppose H_1(b.edu) = 0, H_2(b.edu) = 6. Is b.edu on the whitelist? YES?

Bloom filter: membership test

- False positive (FP): Membership test returns YES when e is not a member
- False negative (FN): Membership test returns NO when e is a member

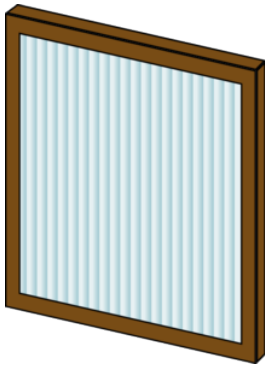
Q: Does Bloom filter have false positives?

Yes, due to hash collisions

Q: Does Bloom filter have false negatives?

No, since we never reset a bit to zero

Q: Is it worth trading accuracy for efficiency?
What's the false positive probability?



Bloom filter parameters

- m: # of bits in a Bloom filter
- k: # of hash functions
- n: # of inserted elements
- Pr(FP) = $\left(1 - \left[1 - \frac{1}{m}\right]^{kn}\right)^k \approx \left(1 - e^{-kn/m}\right)^k.$
- k to minimize P(FP): k = $\frac{m}{n} \ln 2 \approx 0.7\frac{m}{n},$
- For that choice of k, resulting Pr(FP) = p = $2^{-k} \approx 0.6185^{m/n}.$
- Given optimal k, choice of optimal $m = -\frac{n \ln p}{(\ln 2)^2}.$
- $1.44 \log_2(1/\epsilon)$ bits per element

Source: http://en.wikipedia.org/wiki/Bloom_filter

Bloom filter size examples

- m: # of bits in a Bloom filter
- k: # of hash functions
- n: # of inserted elements

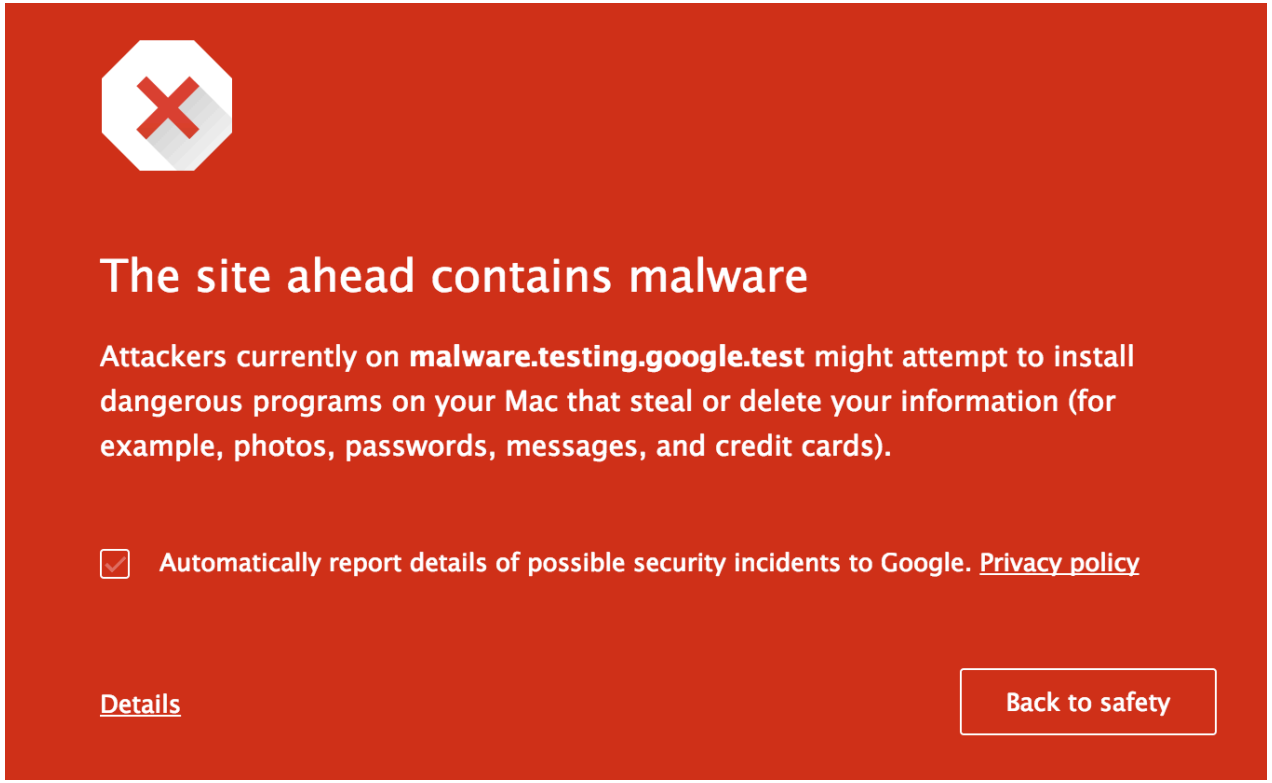
Example #1
$n = 10^6$
$p = 1\%$
$m = 9.6 * 10^6$ bits ~ 1.2 Mbyte
$k \sim 6.6 \rightarrow 7$
With $k = 7$, $p \sim 1\%$

Example #2
$n = 10^6$
$p = 0.1\%$
$m = 14.4 * 10^6$ bits ~ 1.8 Mbyte
$k = 10$

Combining multiple Bloom filters

- Bloom filters with the same of hash functions can be combined using bitwise OR
- Useful for parallelization

Real-world application: Chrome's blacklist



http://malware.testing.google.test/testing/malware/

"The Google Chrome web browser used to use a Bloom filter to identify malicious URLs. Any URL was first checked against a local Bloom filter, and only if the Bloom filter returned a positive result was a full check of the URL performed."

Extensions of Bloom filters

- Counting Bloom filter: support deletion of elements
  - Replace each bit with a counter
- Spectral Bloom filter: estimate # of occurrences of an element
- And many more!

Summary of Bloom filters

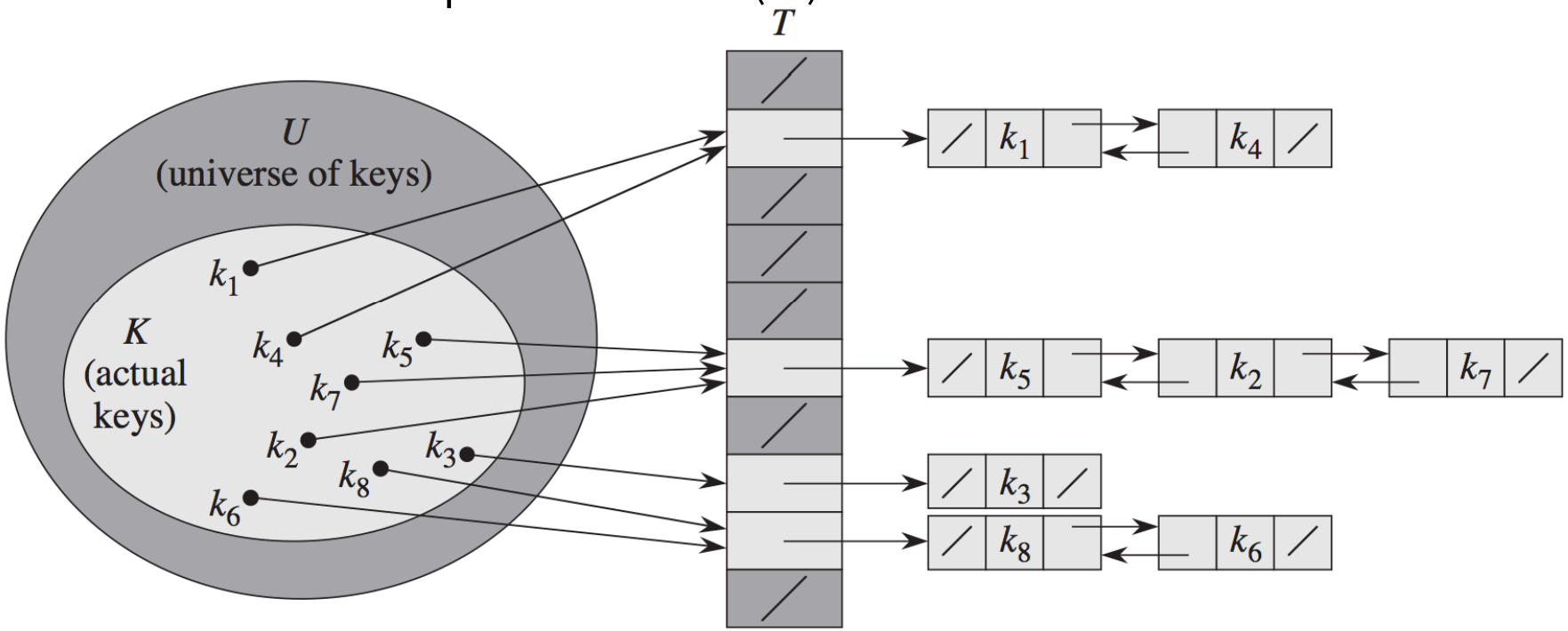
- A Bloom filter is a bit vector of size m initialized to zeros
  - **Insert(e)**: set the bit at position $H_i(e)$ to 1 for all i
  - **IsMember(e)**: return MAYBE if all bits at position $H_i(e)$ are 1s; otherwise, return NO
- Properties
  - No false negative, low false positives
  - Time efficient: O(1) insertion and membership checking
  - Space efficient: constant bits per element given FP rate
  - But still O(n) memory overhead in total
- Keep your hash functions in private to prevent attacks!

Probabilistic Data Structure: Cuckoo Hashing

Chained Hash Tables

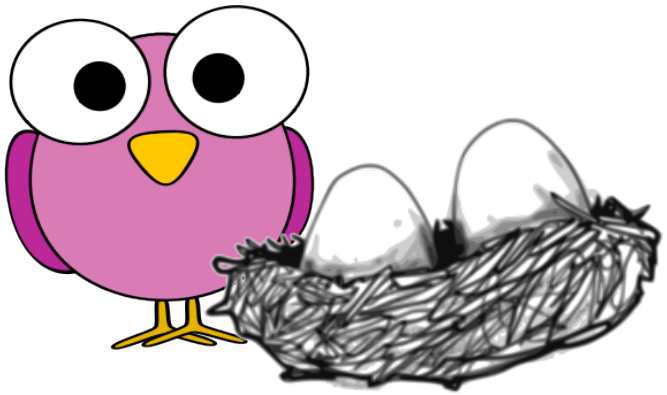
- **Hash collision**: two keys hash to the same slot
- Chaining resolves collisions
- Worst-case lookup time = Ω(n)

Can we reduce the worst-case time to O(1)?



Cuckoo hashing

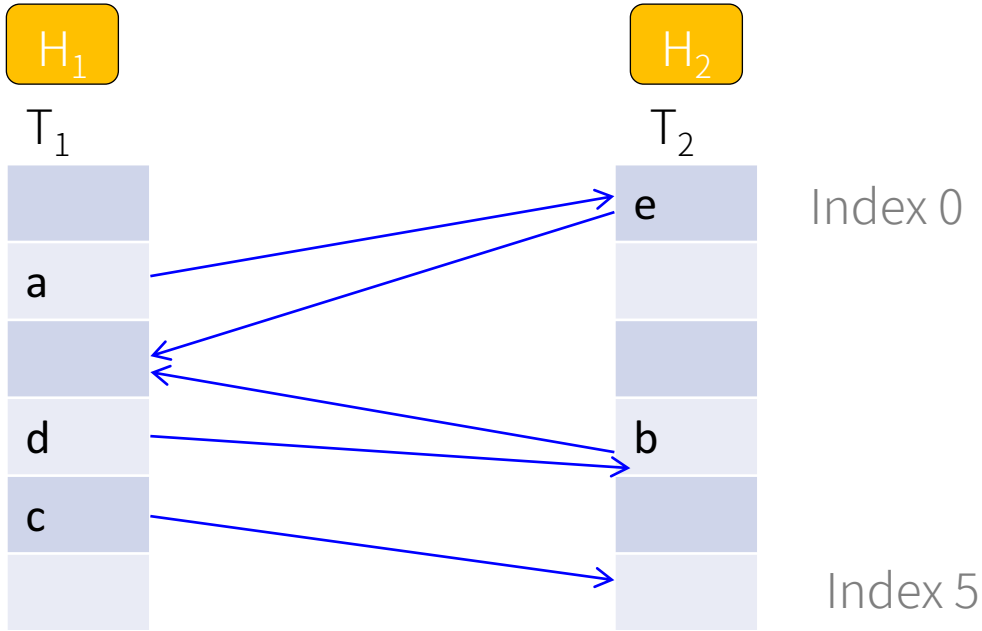
- Cuckoo hashing is a simple hash table
- Lookup: worst-case O(1)
- Delete: worst-case O(1)
- Insert: amortized, expected O(1)



"cuckoo chick pushes the other eggs or young out of the nest when it hatches; analogously, inserting a new key into a cuckoo hashing table may push an older key to a different location in the table."

Cuckoo hashing

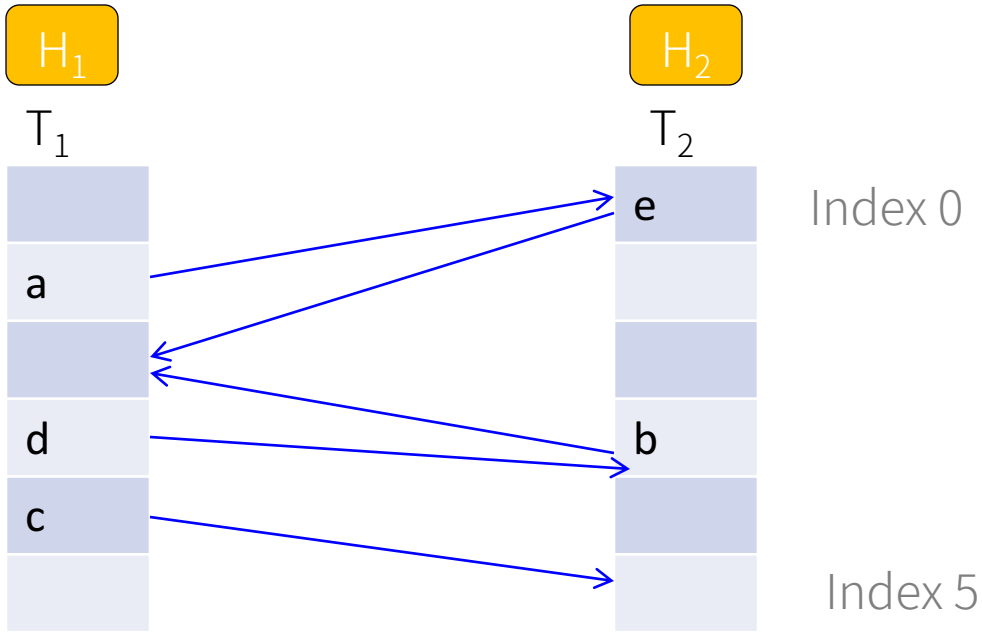
- Two tables, each of which has m slots
- Two hash functions H_1 and H_2
- Every element x will either be at position $H_1(x)$ in the first table or $H_2(x)$ in the second table



Cuckoo hashing: lookup and deletion

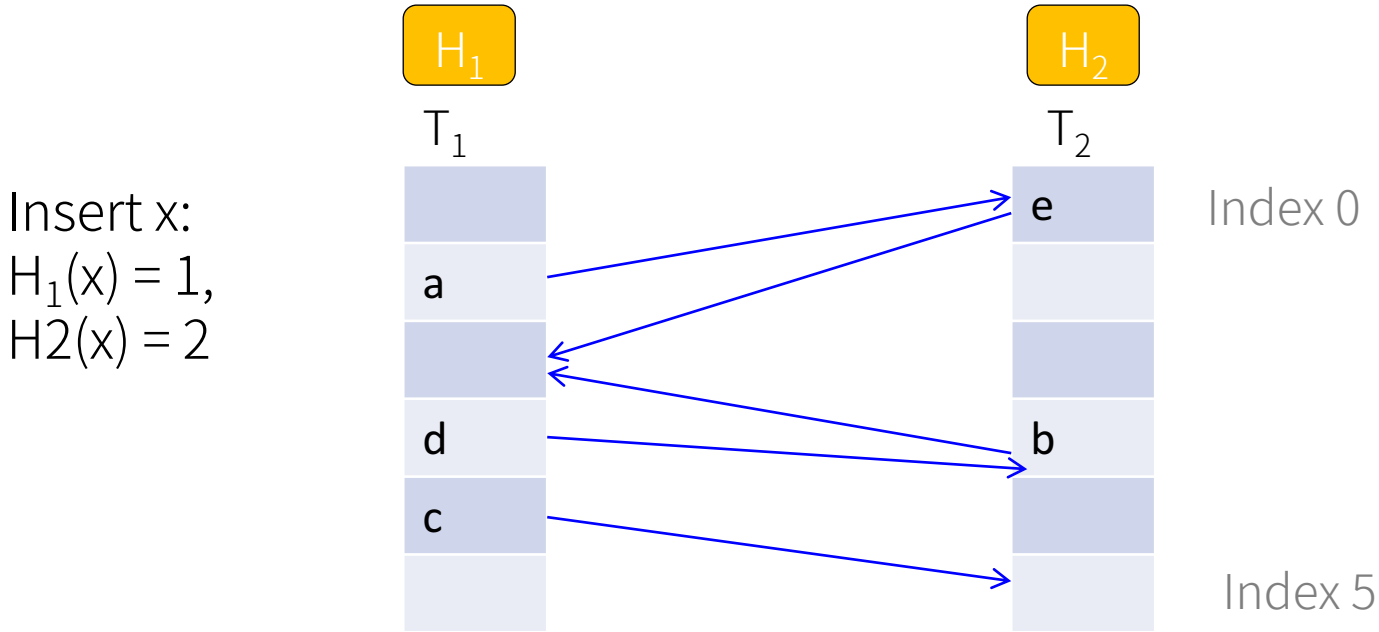
- Lookup: O(1) because only needs to check two positions
- Delete: O(1) because only needs to check two positions

Lookup c: check $H_1(c) = 4$ and $H_2(c) = 5$



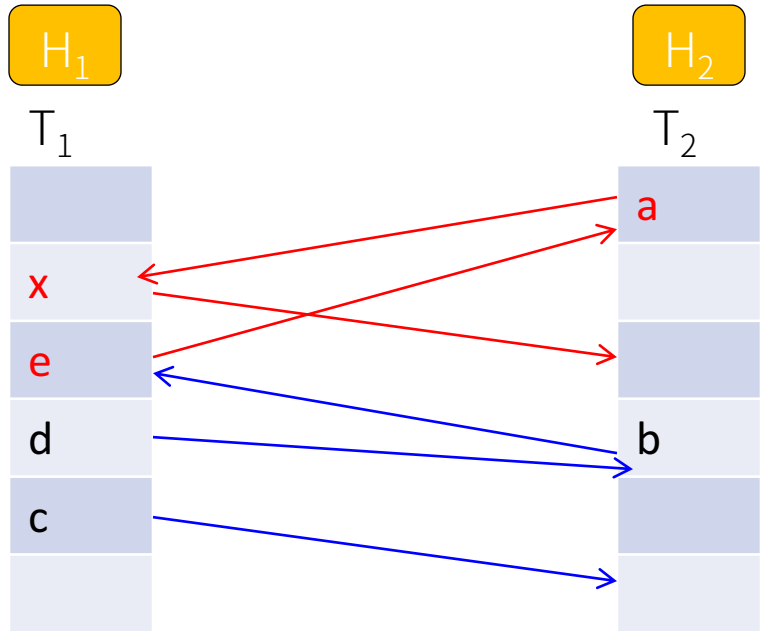
Cuckoo hashing: insertion

- To insert an element x, place x at $H_1(x)$ in table 1
- If $H_1(x)$ were occupied, evict the old element y, and try placing y into table 2
- Repeat until all elements are stored



Cuckoo hashing: insertion

- To insert an element x, place x at $H_1(x)$ in table 1
- If $H_1(x)$ were occupied, evict the old element y, and try placing y into table 2
- Repeat until all elements are stored



Insert x:
$H_1(x) = 1$,
$H2(x) = 2$

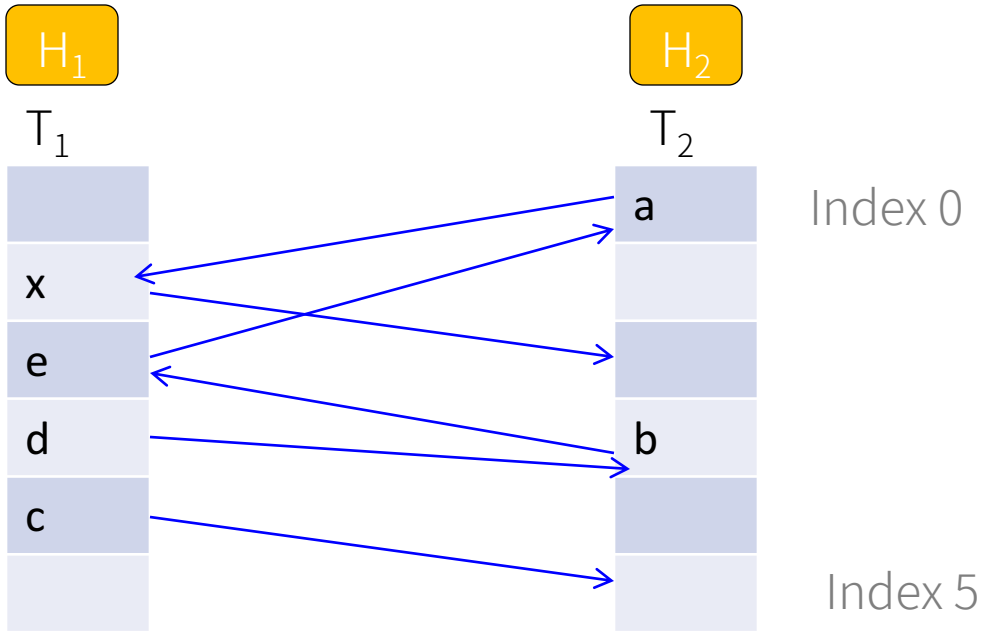
What if it doesn't stop (i.e., running into a cycle)?

Index 0

Index 5

Cuckoo hashing: cycle?

- If a cycle is detected during insertion (e.g., # of evictions > threshold), perform a ***rehash*** by choosing a new H_1 and H_2 and inserting all elements back into the tables
  - Amortized expected cost = O(1)



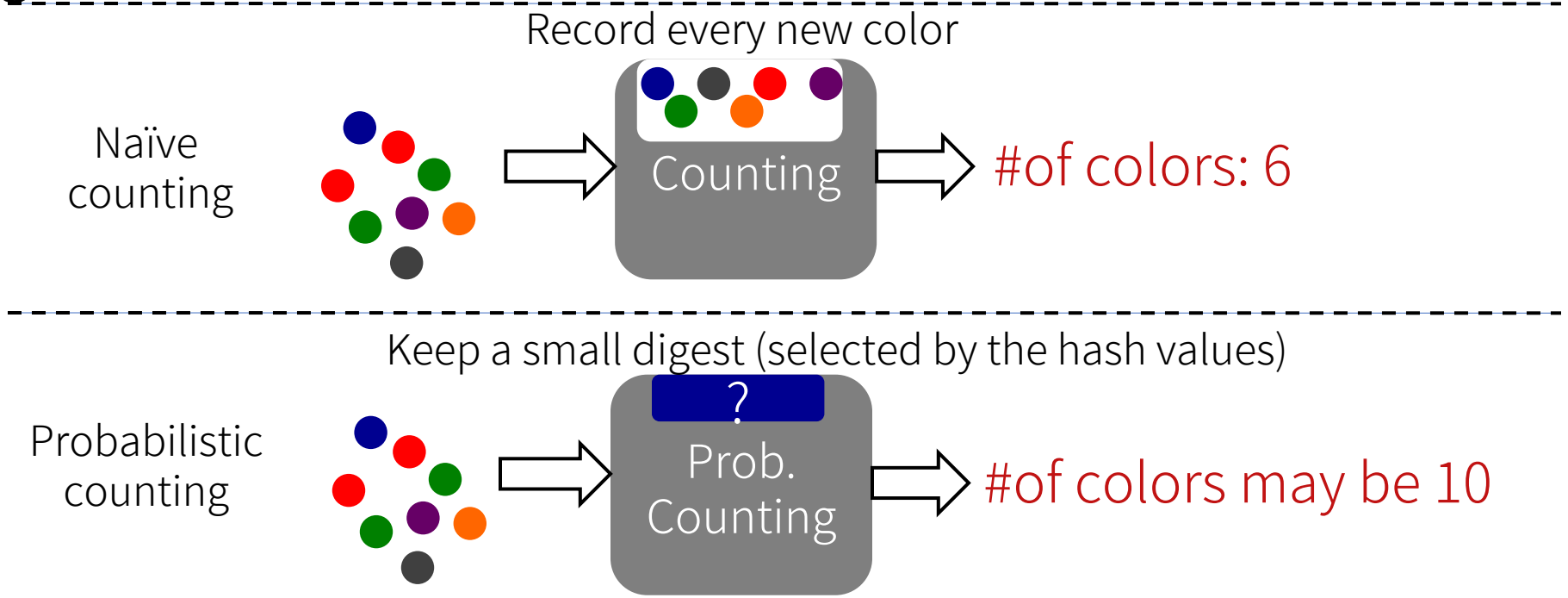
Probabilistic Counting Algorithms

Application: Estimating Number of Flows

- Record all distinct flow IDs
  - Infeasible to keep one record for every distinct flow
  - O(N) memory overhead, where N is number of flows
- Increase a counter when the incoming packet belongs to a "new" flow
  - Use Bloom Filter to test whether a flow has occurred (otherwise, it's "new")
  - Bloom Filter enables O(1) time membership checking but still, O(N) memory overhead

Background on Probabilistic Counting

- Counting with limited memory/bandwidth
- Output an accurate estimate with high probability
- e.g., count total # of colors

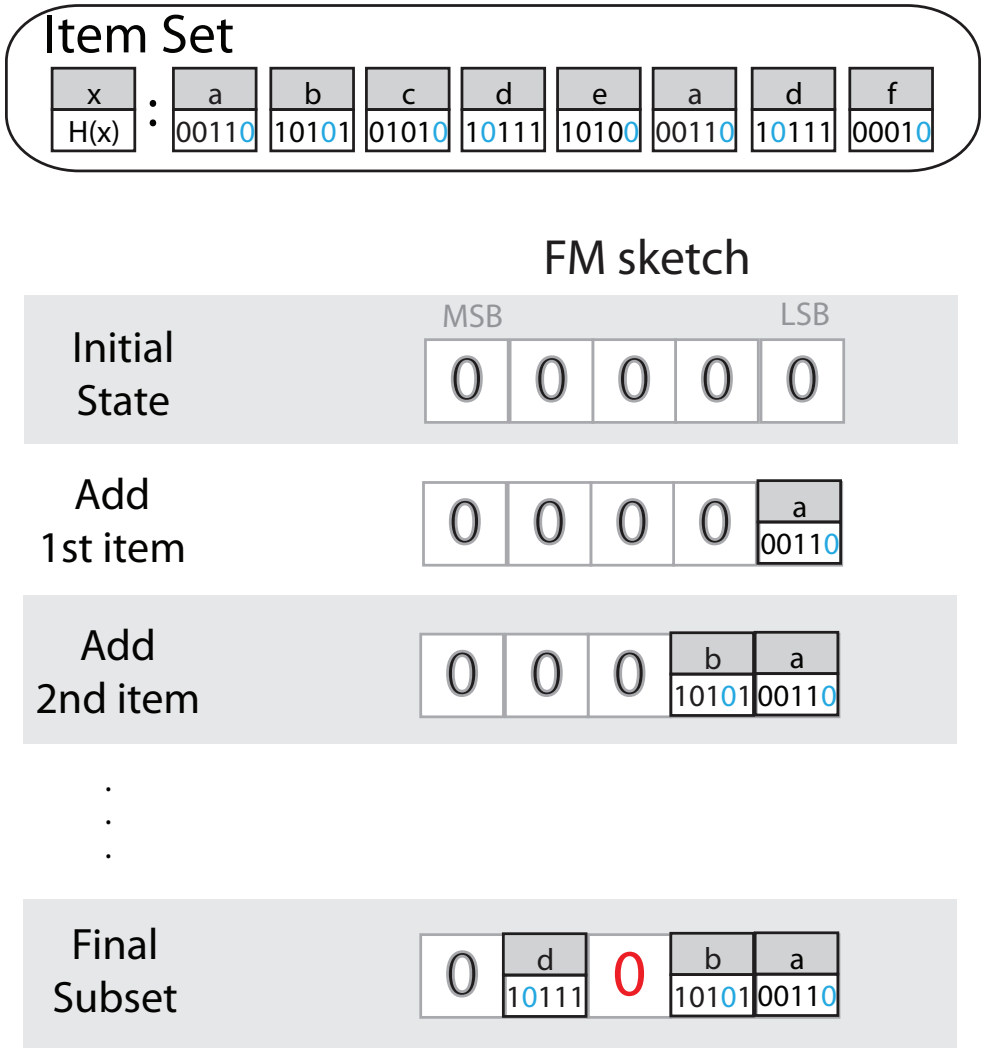


Simple Probabilistic Counting

- Transform by random functions
  - Unknown => exponential
    - Estimate by the rarest event
    - n ~ 1/p

  - Unknown => uniform
    - Estimate by the smallest value (v)
    - n ~ 1/v

FM Sketch [FM85]



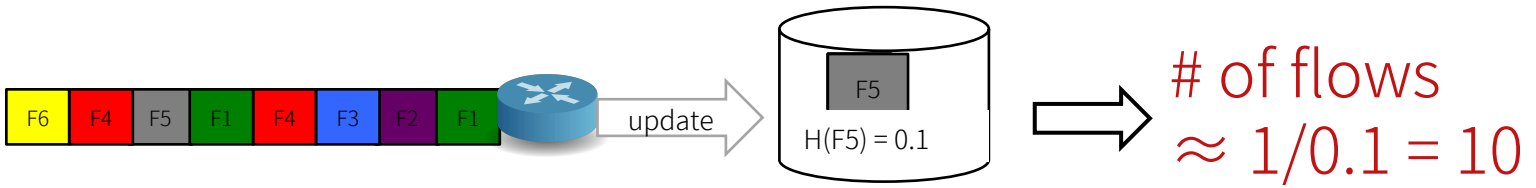
- 6 distinct items
- Place by LSB_0 of H(x)
  - Exponential distribution
- Estimate by LSB_0 in FM
  - $2^2/0.77351 \approx 5$
- Bit-level operation

Estimate by the smallest hash

A simple probabilistic counting scheme

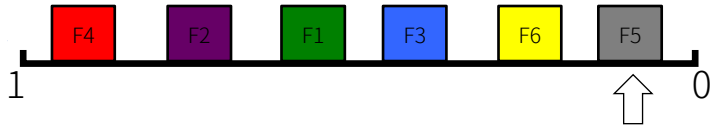
1. Hashes flow ID to generate a value in 0~1
2. Keeps the flow ID associated with the smallest hash value
3. Estimates the number of flows by the smallest value (say, v) that has been seen so far: $\tilde{n} \sim 1/v$

Example:
H(F1) = 0.6
H(F2) = 0.85
H(F3) = 0.4
H(F4) = 0.92
H(F5) = 0.1
H(F6) = 0.26



Estimate by the smallest hash

- Estimates the number of flows by the smallest value (say, v) that has been seen so far: $\tilde{n} \sim 1/v$
- Intuition behind the estimation: assume the hashes of n flows are uniformly distributed in [0,1], the smallest hash value should locate at 1/n

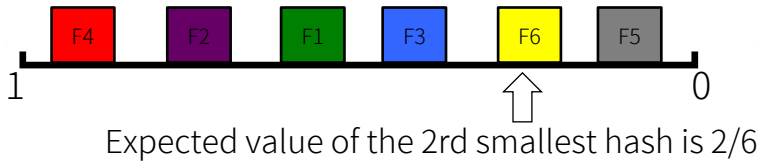


Expected value of the smallest hash is 1/6

- Estimate by the smallest is not robust enough
  - Intuition: min has a higher variance than median
- An attacker controlling only 1 input can bias the estimation

Estimate by the k-th smallest hash

- Estimate by the k-th smallest is more robust
  - Hash flow ID to generate a value in 0~1
  - Estimate the number of flows by the k-th smallest value (say, v_k) that has been seen so far: $\tilde{n} \sim k/v_k$
  - Intuition behind the estimation: if the hashes of n flows are uniformly distributed in [0,1], the k-th smallest hash value should locate at k/n



Expected value of the 2rd smallest hash is 2/6