INF421: Design and Analysis of Algorithms (X‘17)

Lecture 5: Graph Algorithms 1

Benjamin Doerr

<u>Outline:</u>
Quizzes
Using graphs in algorithmics even where you don’t see them
Bipartite matching
Minimum spanning tree problem
- Kruskal’s algorithm, union-find data structure
- Prim’s algorithm, priority queue
[Dijkstra’s algorithm: chapter 5 of poly, but lecture 6]

Where Are We (in INF421)?

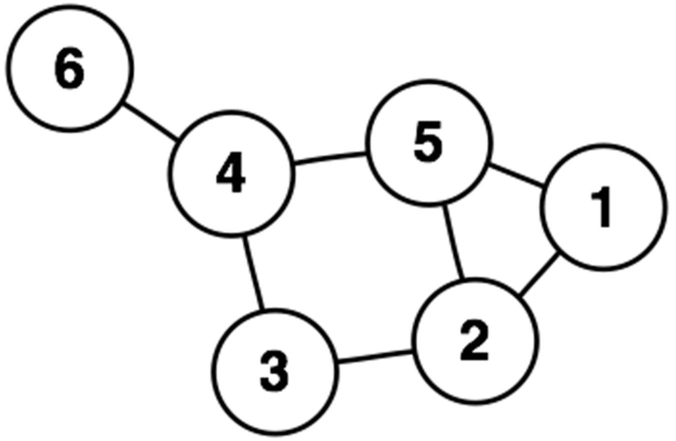
- Course targets
  - Understand the most important algorithms and algorithmic problems
  - Learn how to design and analyze algorithms yourself
  - Understand different algorithmic paradigms

- Course outline
  - Stable marriages, what does it mean to analyze an algorithm
  - 3 basic algorithm design principles: divide and conquer, dynamic programming, greedy algorithms (lectures 2-4)
  - Graph algorithms: Spanning trees, shortest paths, flows (lectures 5-7)
  - 2 advanced algorithm design principles: linear programming, randomized algorithms (chapters 8 and 9, but lectures 10 and 8)
  - Heuristics (chapter 10, lecture 9)

Here!

Quiz 1: Greedily Coloring a Graph

- Problem: Color the vertices of a graph with few colors, but in a way that neighboring vertices get different colors!
  - example: here you need three colors, because the triangle 1-2-5 cannot be colored with fewer colors



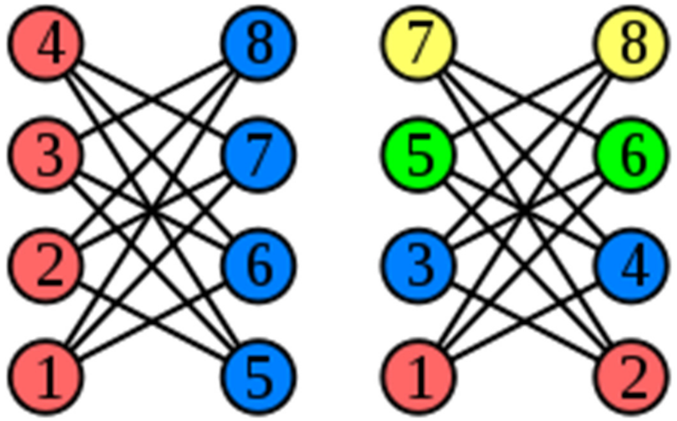
- Greedy-Algorithm:
  - traverse the vertices in any given order and color the current vertex with the smallest color $\in \mathbb{N}$ that is not yet used at a neighbor.
- Analysis: If every vertex has at most Δ neighbors, then the greedy algorithm uses at most $\Delta + 1$ colors.
  - Proof: At each coloring step, at least one color in $[1 \mathinner{..} \Delta + 1]$ is not yet used at a neighbor.

Quiz 2: Greedy “Optimal”?

- Question: Is the greedy algorithm “optimal” (in the sense that there are graphs with max-degree Δ that need $\Delta + 1$ colors)?

- Yes!
  - Complete graphs on n vertices have $\Delta = n - 1$ and need n colors.
  - Cycles of odd length have $\Delta = 2$ and need 3 colors
  - [Side remark: These are the only such examples and there is a clever greedy algorithm that colors all other graphs with Δ colors (Brook’s theorem)]

- Note: For some graphs, the greedy algorithm with some orderings gives very bad results.



Quiz 3 (scheduling): Later in this lecture

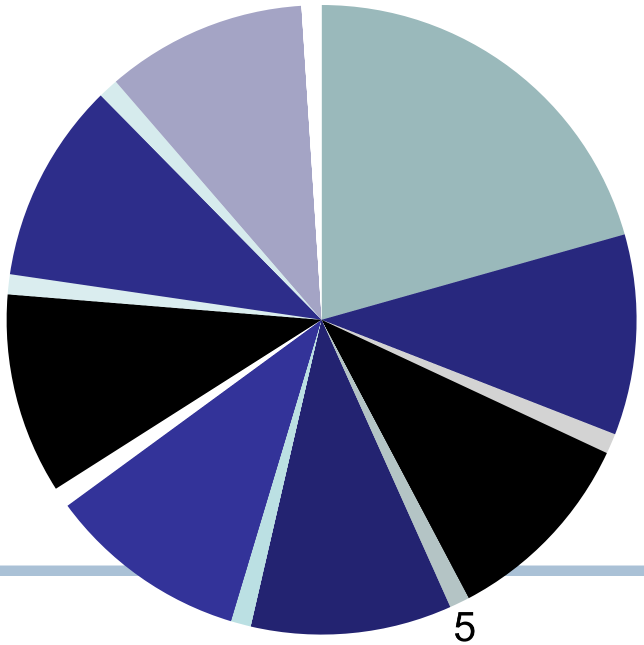
Quiz 4: Greedily Sharing a Pizza

- Reminder pizza game: You and your best study buddy regularly share a pizza. The rules are as follows: Your buddy cuts the pizza into pieces, you choose the first piece, and then alternatingly you both take pieces until everything is eaten. Only pieces adjacent to those already taken may be claimed (hence, apart from the first and the last piece, you always have exactly two choices).

- Greedy algorithm for selecting pieces: Always take the largest (allowed) piece!

- Claim: For any $\gamma > 0$, this algorithm does not guarantee you a share of at least γ !

- Proof: Bad instance with $2n + 1$ pieces: Let $\alpha = 1/n$.
  Let the pieces have size $2, 1, \alpha, 1, \alpha, 1, \dots, \alpha, 1, \alpha$.
  - greedy player gets the piece of size 2 and all n pieces of size α → total of 3
  - size of pizza: $n\alpha + n \cdot 1 + 2 = n + 3$



Topics Today

- Graphs and graph algorithms: A central topic in algorithmics
  - There's a graph around every corner – why graph algorithms are really central in algorithmics (even when you do not expect this)
    - example: maximum bipartite matching
  - A classic problem and two classic algorithms: Minimum spanning trees
    - Kruskal's algorithm
    - Prim's algorithm
  - [poly, but next lecture] Dijkstra's algorithm (with correctness proof, missing in INF411)

- Data structures: From rough pseudocode to the most efficient algorithm
  - Reminder: Priority queues
  - New: Union-find data structure

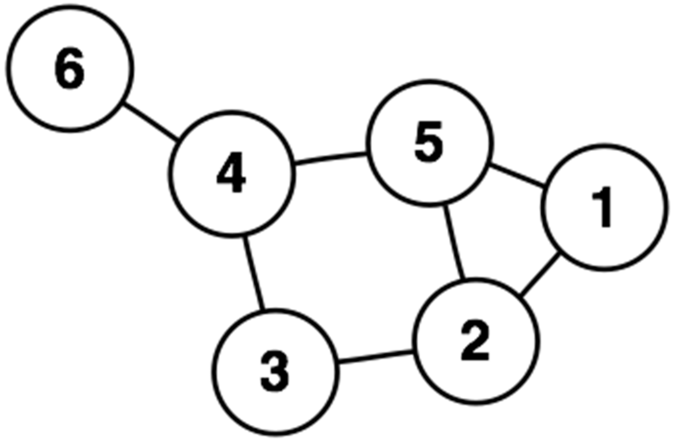
Targets for This Lecture/Chapter

- Graph thinking:
  - learn the graph language as a standardized way to formulate combinatorial optimization problems
  - learn how to connect graph theory and algorithm design
  - learn how to analyze graph algorithms
- Use the methods from Chapter 1 to 4 to develop good algorithms:
  - advanced greedy algorithms
  - think first, then program: use rough pseudo-code to analyze the algorithm, then fix the implementation details (data structures etc)
- Data structures:
  - choose the right data structure for your algorithm

Part 0: Reminder Graphs

- (Undirected) *graph* G: pair (V, E)
  - V finite set, *vertices* (sommets) or *nodes* (nœuds)
  - $E \subseteq \binom{V}{2} \coloneqq \{e \subseteq V \mid |e| = 2\}$ *edges* (arêtes)
  - Can be drawn nicely, but the way we draw a graph has no meaning!
    → All that matters is the combinatorial structure (is there an edge between u and v or not)

- Language:
  - $\{u, v\} \in E$: u, v are *neighbors/adjacent*
  - $N(v)$: set of all neighbors of v
  - $d(v) \coloneqq |N(v)|$ *degree* of v
  - lots of intuitive notation: *paths*, *connected*,…



- Convention: $n \coloneqq |V|$, $m \coloneqq |E|$

Reminder Storing Graphs: Two Common Representations

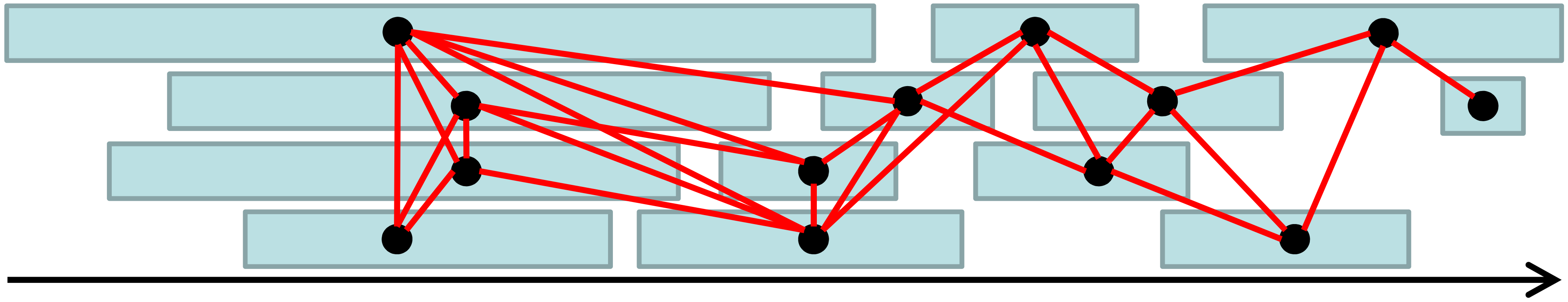
- Adjacency matrix: square matrix indicating for each pair of vertices whether there is an edge between them or not
  - check whether u, v are neighbors: $\Theta(1)$ time
  - enumerate all neighbors of v: $\Theta(n)$ time
  - memory usage: $O(n^2)$
- Adjacency lists: for each vertex, we have a list of its neighbors
  - check whether u, v are neighbors: $\Theta(d(v))$ time
  - enumerate all neighbors of v: $O(d(v))$ time
  - memory usage: $O(\sum_{v \in V} d(v)) = O(m)$
    → good for sparse graph (all real-world graphs are sparse)
    → adjacency lists are the typical format to store graphs ☺
- Convention: If we do not specify the representation, we assume that the input graph is represented via adjacency lists

Part 1: There’s a Graph Around Every Corner (in Algorithmics) …

- Graphs and graph algorithms are central in algorithmics not only because of problems like shortest paths, where the graph is obvious, but equally well because
  - we can re-phrase our algorithmic problem in graph-theoretic language
  - we can use deep results from graph theory at some stage of the design and analysis of our algorithm

- Two examples on the following slides:
  - interval scheduling = independent set problem in interval graphs
  - assignment problems = matching problems in bipartite graphs

Interval Scheduling

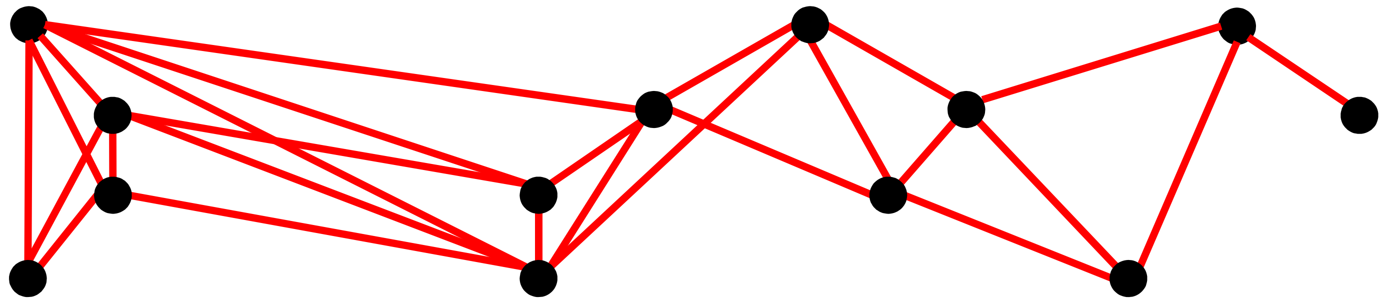


- Problem: Find a largest set of non-overlapping intervals (→ lecture 4)
- *Conflict graph:*
  - vertices: intervals
  - edges indicate overlap (conflict)

Graphs that can be constructed in this fashion are called *interval graphs*

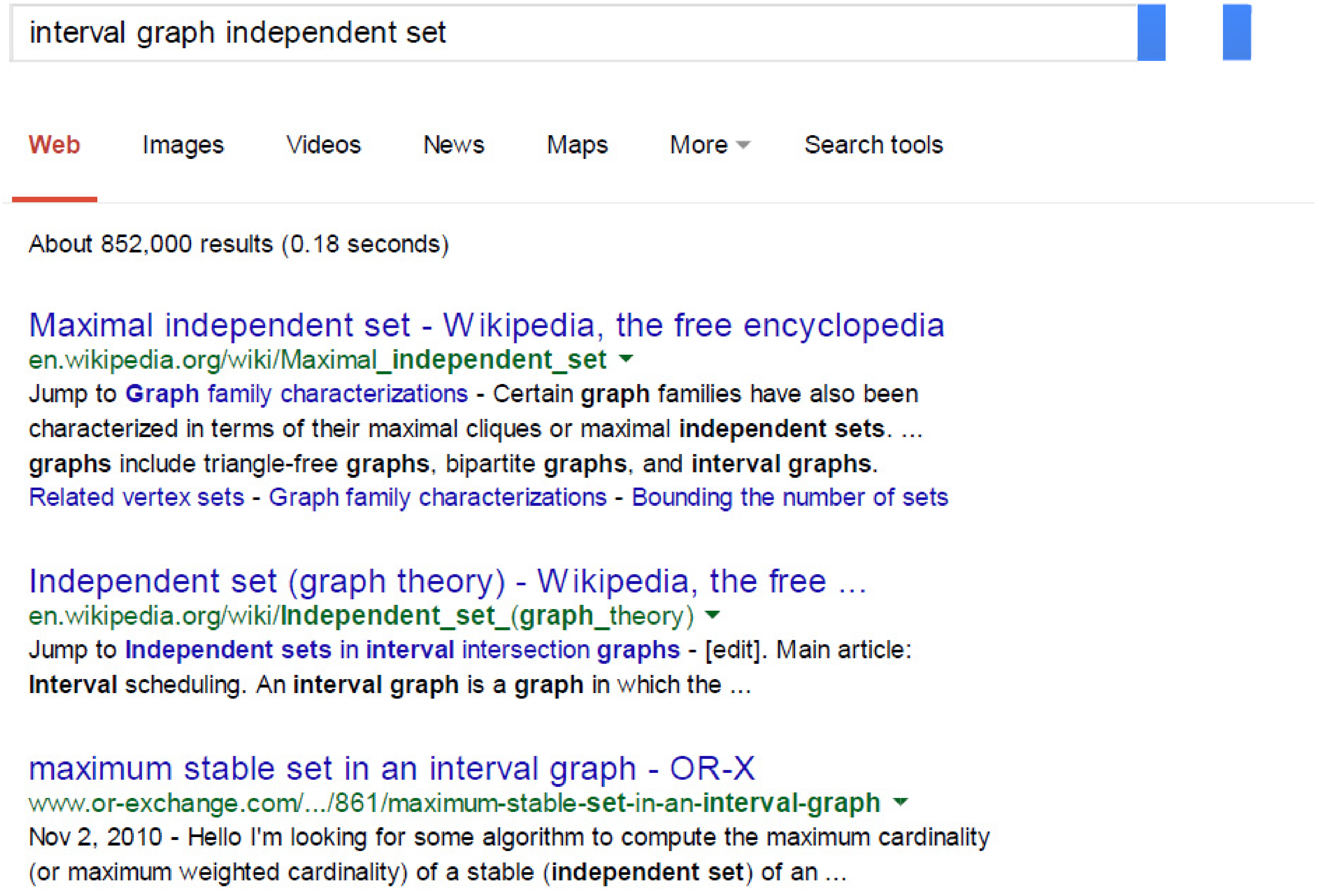
- Observation: The interval scheduling problem asks for a maximum set of vertices that contains no edge *(maximum independent set)* in this graph.

Interval Scheduling vs. Finding Maximum Independent Sets



- Solution for our problem: Take a good independent set algorithm?
  - No! *Independent set* (in general graphs) is an NP-complete problem ☹

- Why then care for graphs and independent sets?
  - may help our understanding (higher level of abstraction)
  - makes it easy to find information…



About 852,000 results (0.18 seconds)

Maximal independent set - Wikipedia, the free encyclopedia
en.wikipedia.org/wiki/Maximal_**independent_set**
Jump to **Graph** family characterizations - Certain **graph** families have also been characterized in terms of their maximal cliques or maximal **independent sets**. ... **graphs** include triangle-free **graphs**, bipartite **graphs**, and **interval graphs**.
Related vertex sets - Graph family characterizations - Bounding the number of sets

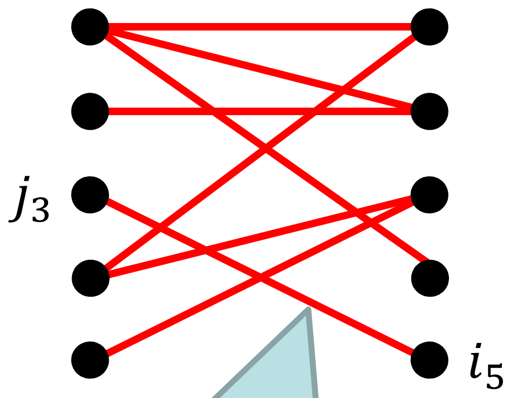
Independent set (graph theory) - Wikipedia, the free ...
en.wikipedia.org/wiki/**Independent_set_(graph_**theory)
Jump to **Independent sets** in **interval** intersection **graphs** - [edit]. Main article: **Interval** scheduling. An **interval graph** is a **graph** in which the ...

maximum stable set in an interval graph - OR-X
www.or-exchange.com/.../861/maximum-stable-**set**-in-an-**interval-graph**
Nov 2, 2010 - Hello I'm looking for some algorithm to compute the maximum cardinality (or maximum weighted cardinality) of a stable (**independent set**) of an ...

2nd Example: Assignment Problem

- Problem (quiz 3):
  - set J of n jobs
  - set M of m non-identical machines: machine i can only process a subset $J_i \subseteq J$ of the jobs
  - each machine can take one job only
  - task: schedule as many jobs as possible!

- Graph representation: $\{j, i\} \in E \iff$ job j can be processed on machine i



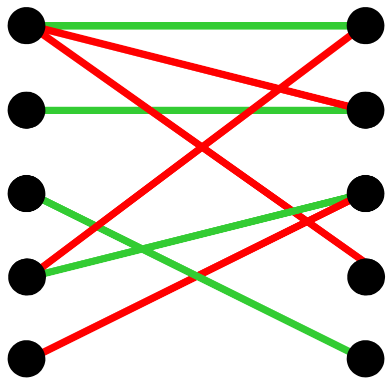
Job j_3 can be processed on machine i_5

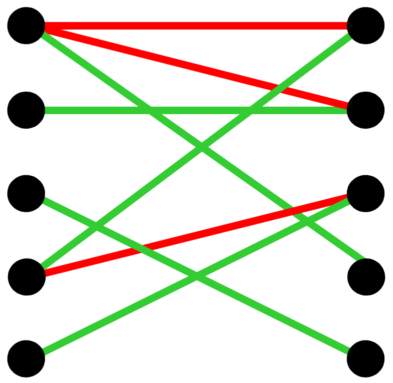
Equivalent problem: Find a maximum set of edges that pair-wise do not intersect *(maximum matching)*.

Advantages: (i) graphical representation
(ii) standardized problem formulation
(iii) use graph theory to solve the problem

Finding Maximum Matchings

- Problem: Find a maximum matching (edges that do not intersect) in a bipartite graph (2 classes of vertices, edges go from one to the other)
- Simple greedy algorithm: Go through the jobs (in some order). If there is a machine that could take the job, take the first machine (in some order).
  - Clearly computes a maximal matching M (one that is not contained in a larger matching)
  - Not always a maximum matching M^* (one of maximal cardinality)
  - Claim: $|M| \geq 0.5\ |M^*|$



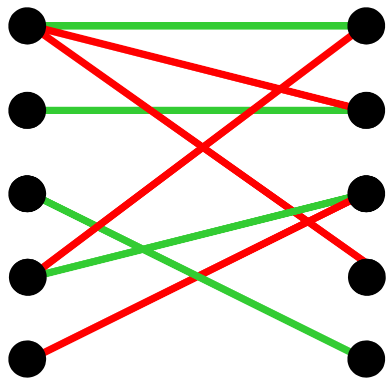


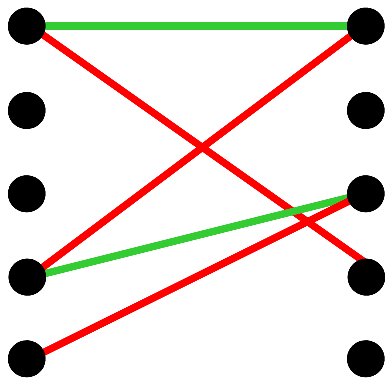
Lemma: For any maximal matching M, $|M| \geq 0.5\ |M^*|$

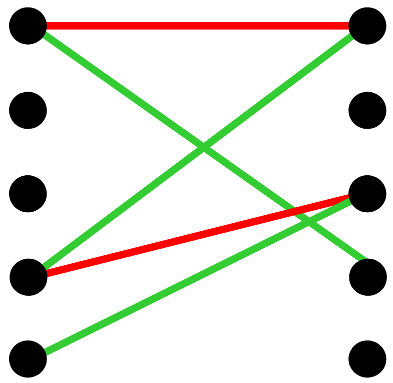
- Informal argument: A wrong edge chosen for the maximal matching M can prevent at most two wanted edges from being in the matching.

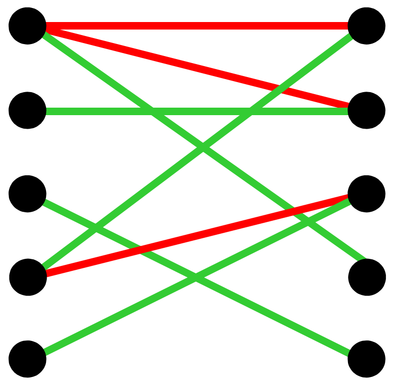
- Solid proof (making the informal argument precise):
  - Let M be a maximal matching and M^* be a maximum matching.
  - For each $e \in M^* \setminus M$, there is an $f \in M \setminus M^*$ such that $e \cap f \neq \emptyset$. Otherwise we could add e to M (contradiction maximality). Fix such an f for each e and call it $\phi(e)$.
  - $\phi: M^* \setminus M \rightarrow M \setminus M^*$ is such that each $f \in M \setminus M^*$ has at most two pre-images (there are at most two different e such that $\phi(e) = f$). Reason: $M^* \setminus M$ is a matching, i.e., edges do not intersect.
  - $|M^* \setminus M| = |\phi^{-1}(M \setminus M^*)| = \left|\bigcup_{f \in M \setminus M^*} \phi^{-1}(f)\right| \leq \sum_{f \in M \setminus M^*} |\phi^{-1}(f)| \leq 2|M \setminus M^*|$
  - $|M| = |M \cap M^*| + |M \setminus M^*| \geq |M \cap M^*| + 0.5\ |M^* \setminus M| \geq 0.5\ |M^*|$

Designing a Cool Greedy Algorithm Using Graph Theory


some matching M


augmenting path P


switching status


$M \oplus P$

- **Definition:** An *augmenting path* P (with respect to a given matching M) is a path such that
  - the first and last vertex are not in any matching edge
  - the edges alternate between matching and non-matching edges
- **Lemma:** If we switch the edge-status in an augmenting path P, then we obtain a matching $M \oplus P$ with $|M \oplus P| = |M| + 1$.

A Better Greedy Algorithm

- AugmentingPathGreedy:
  - $M := \emptyset$
  - While there is an augmenting path
    - choose an augmenting path P and set $M := M \oplus P$
  - Return M

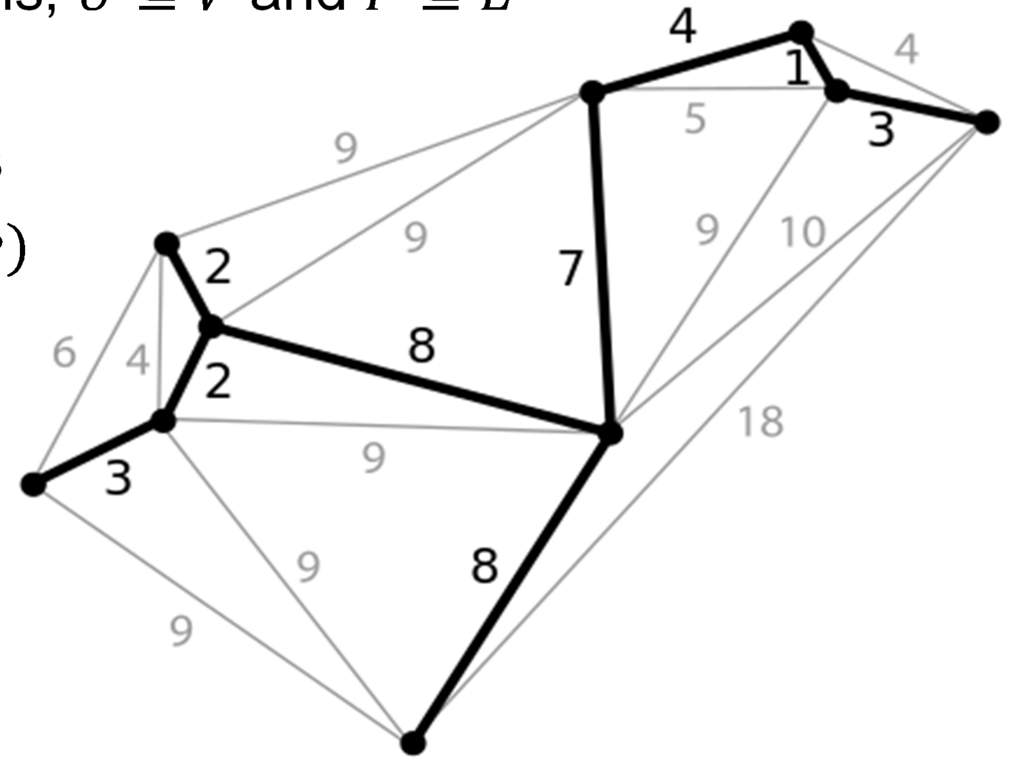
- Complexity: Augmenting paths can be computed in time $O(m+n)$ via a variant of breadth-first search
- Correctness [famous graph theory results of Claude Berge (1957)]:
  M is a maximum matching if and only if there are no augmenting paths.
- Summary (assignment problems): Using deep results from graph theory, we found an optimal algorithm for the assignment problem.
- Summary (part 1): Graphs, graph theory, and graph algorithms play a central role in the design and analysis of algorithms

Part 2: Minimum Spanning Tree Problem

- Input: Let $G = (V, E)$ be a graph. Let $w: E \to \mathbb{R}$ *edge weights.* Assume that G is *connected* (there is a path between any two vertices).

- Task: Find a *minimum spanning tree* (MST) of G
  - a subgraph $T = (U, F)$ of G, that is, $U \subseteq V$ and $F \subseteq E$
  - spanning: $U = V$
  - T is a tree: connected, no cycles
  - minimal weight $w(F) = \sum_{e \in F} w(e)$

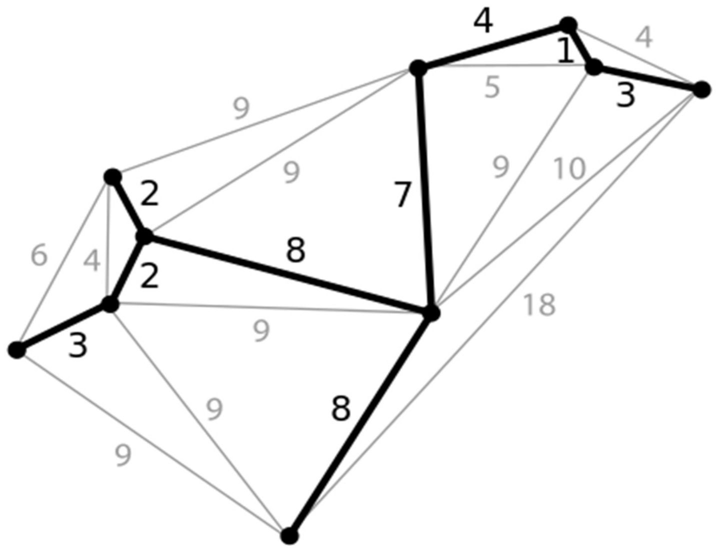
- Minimum spanning tree: Cheapest way to keep all vertices connected → network design



Reminder: Trees

- **Tree Lemma:** Let $G = (V, E)$ be a graph. The following four properties are equivalent.
  - G is a tree (connected, no cycles).
  - G is connected, but deleting any edge would make G disconnected.
  - G is cycle-free, but adding any new edge would create a cycle.
  - $m = n - 1$ and G is connected or cycle-free.

- Consequence: The MST problem consists of finding the right $n - 1$ edges.

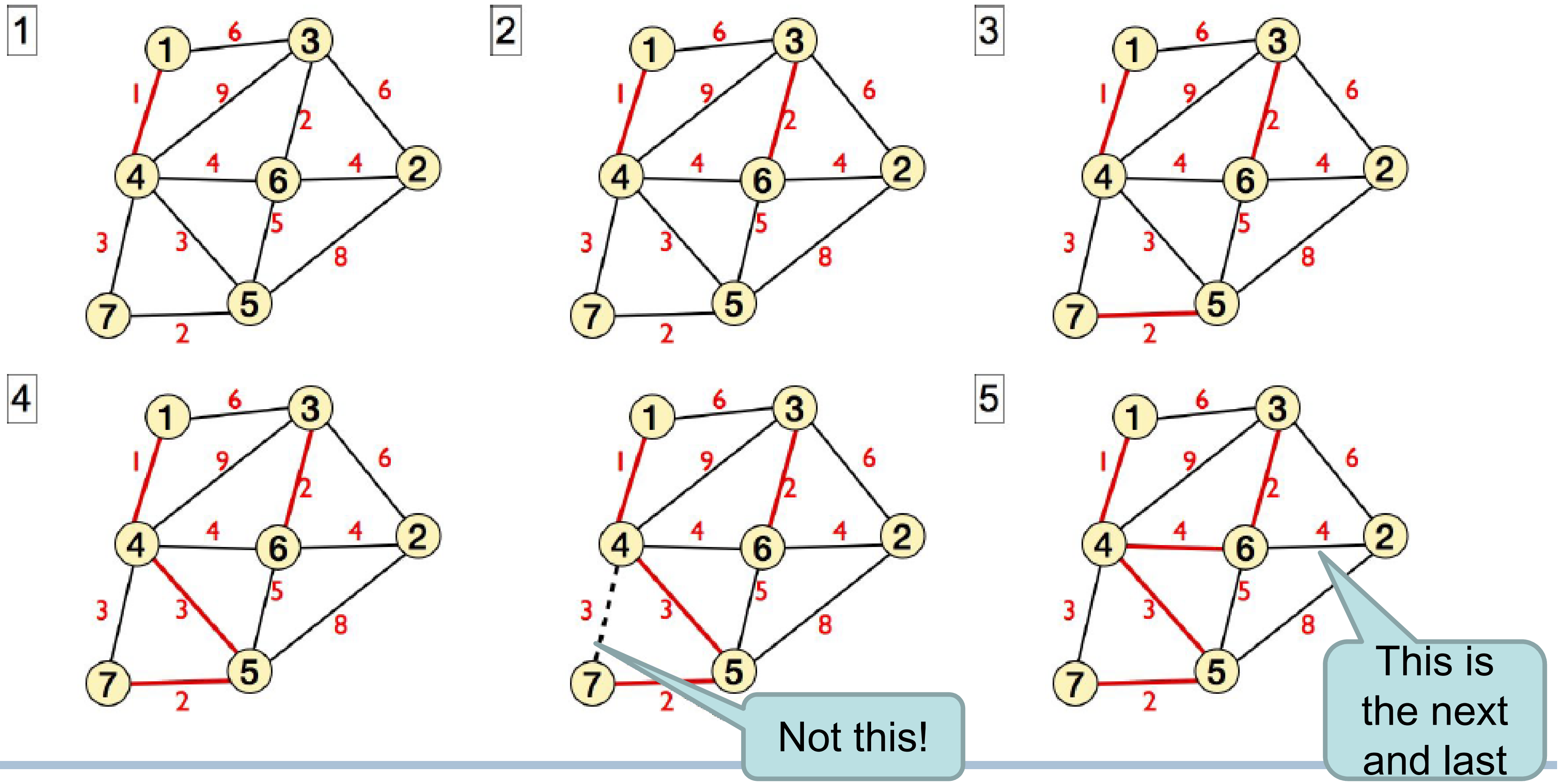


Two Greedy Algorithms for Computing MSTs

- Insight from last lecture: Greedy does not always work. But if greedy works, then often this is the perfect solution. So let’s try…
- Kruskal’s algorithm: Start with the empty graph on all vertices. Repeat adding the cheapest edge that does not create a cycle
- Prim’s algorithm: Start with any vertex and no edge. Repeat adding the cheapest outgoing edge and its other end vertex.
- [Borůvka's algorithm (1926): Start with all vertices as isolated subgraphs. Repeat (i) finding the cheapest edge (with global tie-breaking) leaving each component and (ii) adding all these edges.
  - This works as well, but we do not discuss it because it is more complicated and usually not better.
  - Also known as Sollin’s algorithm (1965)]

Kruskal's Algorithm in Pictures

- Kruskal: Start with the empty set of edges. Add edges from cheap to expensive, but discard those which create a cycle.



Kruskal's Algorithm: Rough Pseudocode

```
1 Input: An undirected connected graph $G = (V, E)$, edge weights $w : E \to \mathbb{R}$.
2 Output: The edge set $F$ of a minimum spanning tree $T = (V, F)$ of $(G, w)$.
3 Sort $E$ by increasing weight, let $e_1, \dots, e_m$ be the result;
4 $F = \emptyset$;
5 for $i = 1$ to $m$ do
6     if $(V, F \cup \{e_i\})$ contains no cycle then $F := F \cup \{e_i\}$;
7 return $(F)$;
```

- Reminder (lecture 1): Rough pseudocode…
  - is perfect to analyze correctness and solution quality (optimal solution, approximation algorithm) → Task 1
  - needs more details do discuss the runtime → Task 2

[Task 1] Analysis of Kruskal's Algorithm (1): Solution is a Tree

- Let F be the output of Kruskal's algorithm.

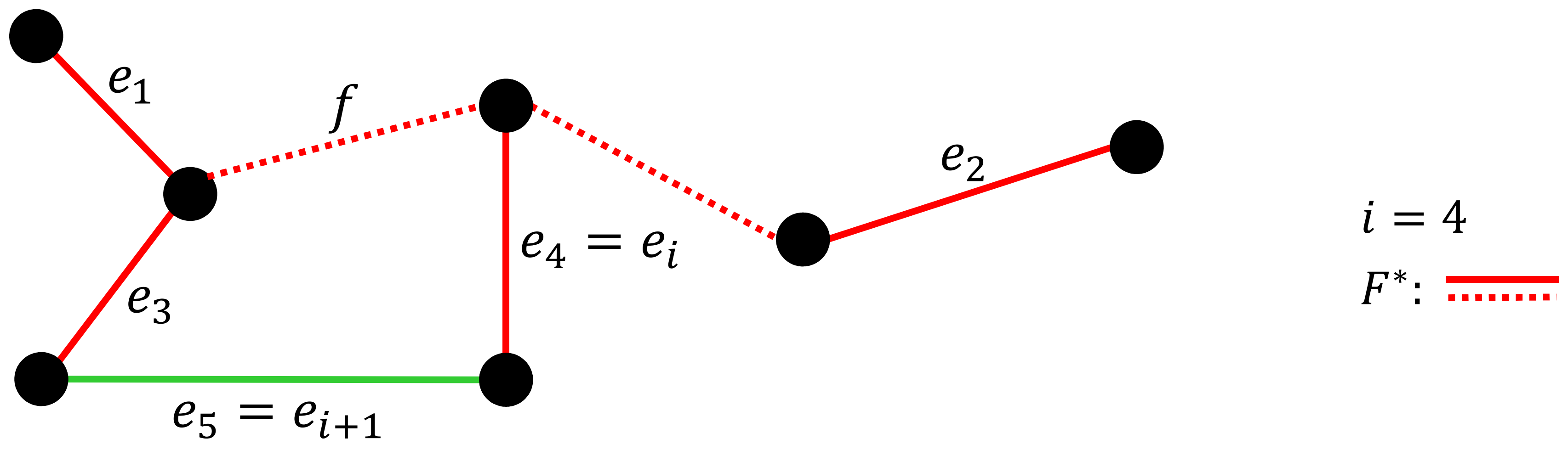
- 1st observation: (V, F) contains no cycle.
  - this is a loop invariant of the algorithm

- 2nd observation: (V, F) is connected.
- Proof:
  - Assume not. Then there is an edge $e_i \notin F$ that could be added without creating a cycle (namely one that connects two components).
  - Dynamic invariant: during a run of the algorithm, the graphs (V, F) only increase (edges are added to F).
  - Therefore, the edge e_i would also not have created a cycle in the i-th iteration.
  - Contradiction to $e_i \notin F$.

Analysis of Kruskal's Algorithm (2): Transforming Solutions Argument

2nd observation gives that these are $n-1$ edges

- **Lemma:** Let F be the output of Kruskal's algorithm. Let $e_1, \dots, e_{n-1}$ be the edges Kruskal's algorithm chose (in this order). Let F^* be a MST. Assume that F^* contains $e_1, \dots, e_i$, but not e_{i+1}. Then there is a MST F'' containing $e_1, \dots, e_{i+1}$.
- **Proof idea:**
  - $F^* \cup \{e_{i+1}\}$ contains a cycle and this cycle contains an edge $f \notin F$
  - $w(f) \geq w(e_5)$ since Kruskal took e_5 and not f
  - $F'' := F^* \cup \{e_{i+1}\} \setminus \{f\}$ is a tree, with $w(F'') \leq w(F)$, hence a MST



Analysis of Kruskal’s Algorithm (2): Transforming Solutions Argument

- **Lemma:** Let F be the output of Kruskal’s algorithm. Let $e_1, \dots, e_{n-1}$ be the edges Kruskal’s algorithm chose (in this order). Let F^* be a MST. Assume that F^* contains $e_1, \dots, e_i$, but not e_{i+1}. Then there is a MST F'' containing $e_1, \dots, e_{i+1}$.

- **Formal Proof:**
- $(V, F^* \cup \{e_{i+1}\})$ contains a cycle (tree lemma).
- Since F is cycle-free (1st observation), this cycle contains an edge $e^* \notin F$.
- Regard $F'' := F^* \setminus \{e^*\} \cup \{e_{i+1}\}$.
  - F'' is connected, because we exchanged two edges on a cycle
  - F'' has $n - 1$ edges, because we added one and took one. → F'' tree
  - $w(e^*) \geq w(e_{i+1})$, since Kruskal took e_{i+1} and not e^* (and $\{e_1, \dots, e_i, e^*\}$ is cycle-free as well)
  - $w(F'') = w(F^*) - w(e^*) + w(e_{i+1}) \leq w(F^*)$, hence F'' is also a MST

Analysis of Kruskal’s Algorithm (3): Putting Things Together

- Theorem: Kruskal’s algorithm computes a MST.

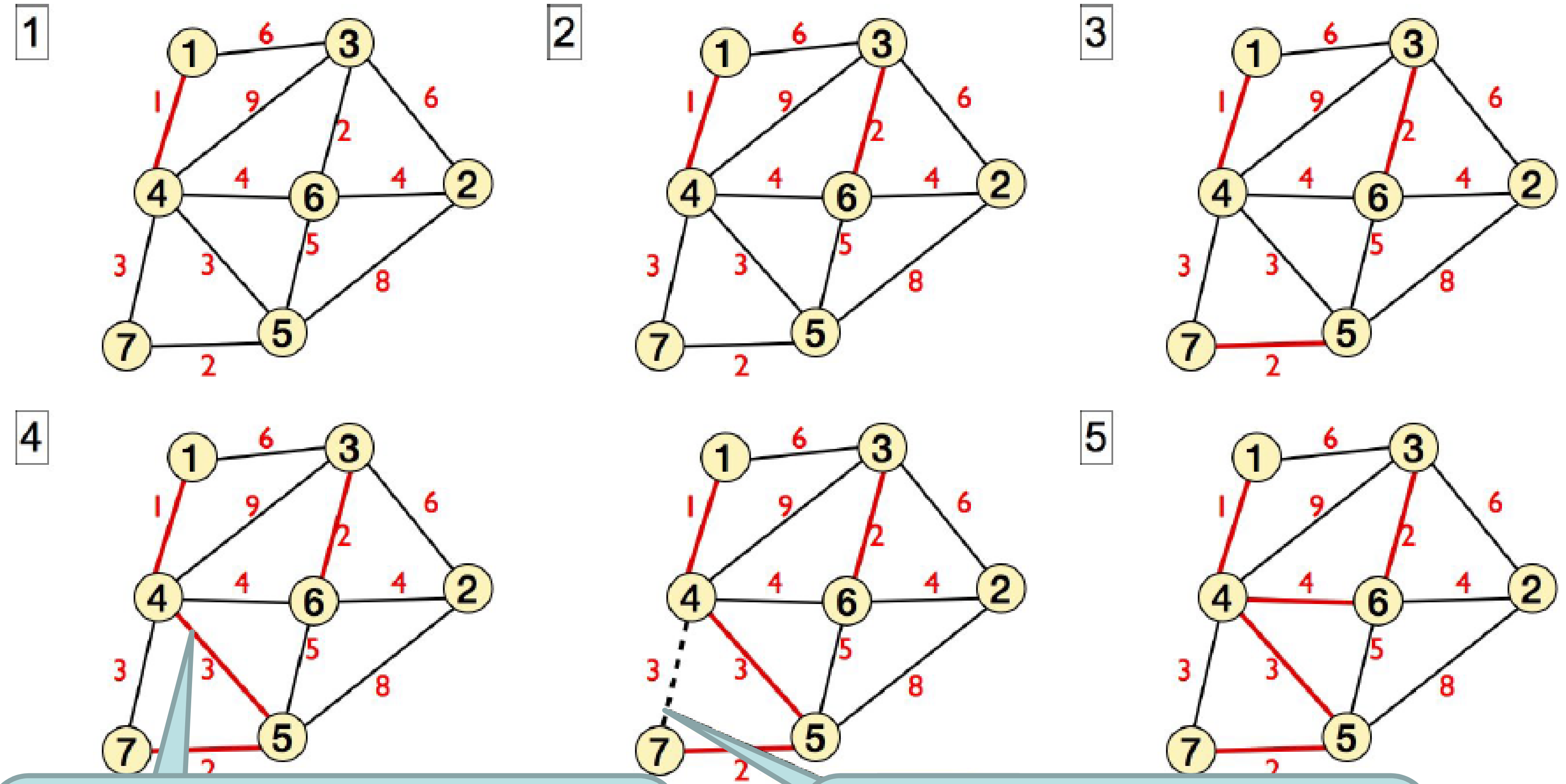
- Proof: Lemma plus induction
  - Let $F = \{e_1, \dots, e_{n-1}\}$ be the output of Kruskal’s algorithm with the edges chosen in this order.
  - Let F^* be a MST (optimal solution). Assume that F^* contains $e_1, \dots, e_i$, but not e_{i+1}.
  - Using the previous lemma and induction, for all $j \in [i+1 \,..\, n-1]$ we construct a MST (V, F_j^*) such that $e_1, \dots, e_j \in F_j^*$.
  - By construction, $F_{n-1}^* = F$ and thus (V, F) is a MST.

[Task 2:] Implementing Kruskal's Algorithm, Time Complexity

```
1 Input: An undirected connected graph $G = (V, E)$, edge weights $w : E \to \mathbb{R}$.
2 Output: The edge set $F$ of a minimum spanning tree $T = (V, F)$ of $(G, w)$.
3 Sort $E$ by increasing weight, let $e_1, \ldots, e_m$ be the result;
4 $F = \emptyset$;
5 for $i = 1$ to $m$ do
6     if $(V, F \cup \{e_i\})$ contains no cycle then $F := F \cup \{e_i\}$;
7 return $(F)$;
```

- How do we implement this algorithm efficiently?
  - Everything is straight-forward except the cycle check in line 6.
  - In general: Checking whether a graph contains a cycle can be done in time $O(m)$
    - do a graph search and see if you visit a vertex a second time.
  - Can we do better because (V, F) changes not much per iteration?

Edge Creates Cycle ⇔ End Vertices Are in Same Connected Component



Vertex 4 and vertex 5 were in different connected component {1,4} and {5,7}, hence adding {4,5} creates no cycle

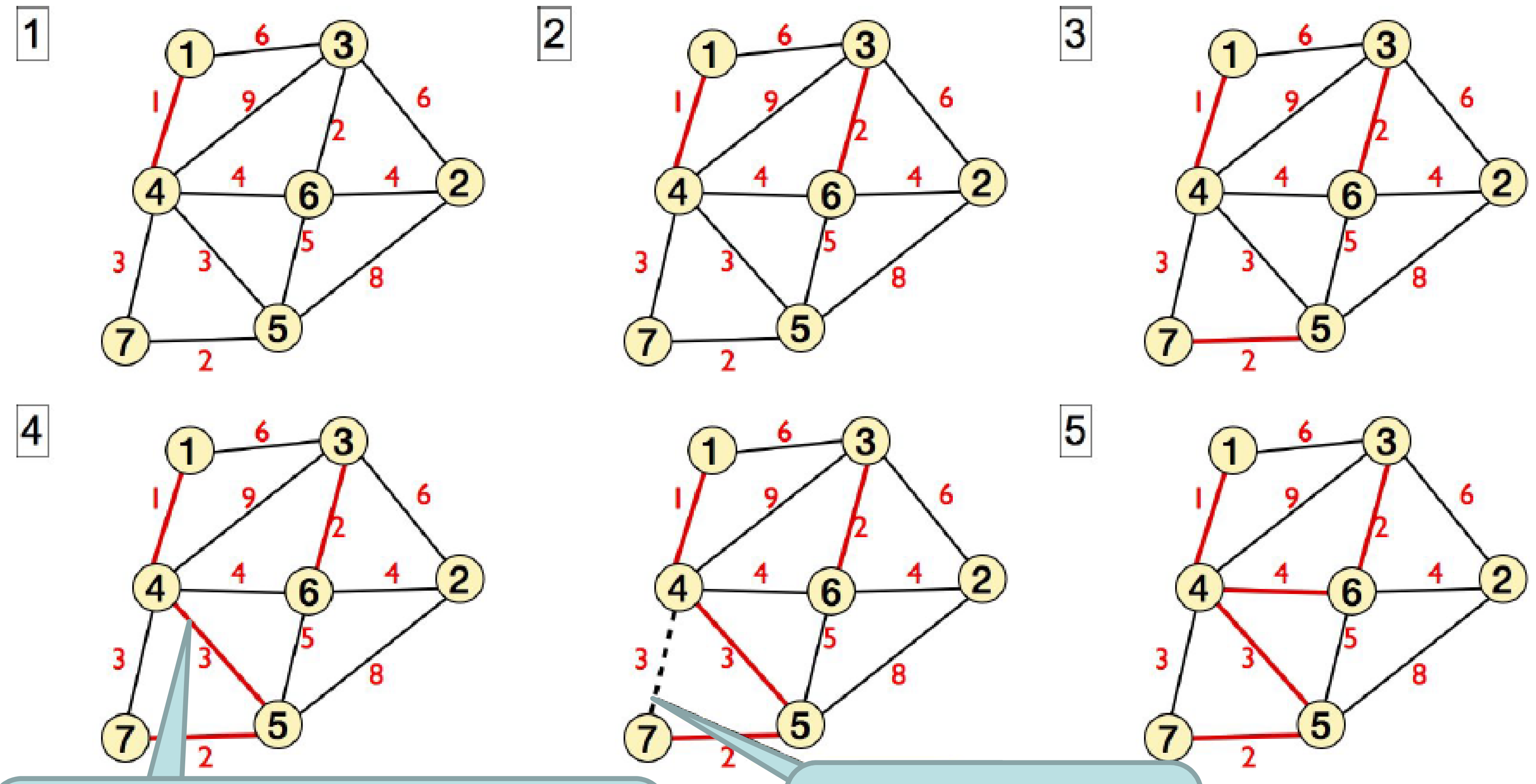
Vertex 4 and vertex 7 are in the same connected component {1,4,5,7}, hence this edge creates a cycle

Union-Find Data Structure

- Observation: We “only” need to answer questions like “are x and y in the same connected component?”
- Since the components do not change too much from one iteration to the next, maybe we can re-use insight from previous iterations.

- More precisely: We need a data structure that stores a partition $C_1, \dots, C_k$ of a given set V. The data structure should support two operations:
  - Find(x): For $x \in V$, tell me for which $j \in [1..k]$ we have $x \in C_j$
  - Union(x, y): If x and y are not in the same partition class, then modify the partition by uniting the two classes that x and y lie in.
  - → such a data structure is called *union-find data structure*.

Picture: Find(x), Union(x,y)



Find(4) ≠ Find(5), hence add the edge {4,5} and Union(4,5)

Find(4) = Find(7), hence discard the edge {4,7}

Union-Find: Representatives

- Union-find data structure: stores a partition $C_1, \ldots, C_k$ of a given set V and supports the following operations. the representative of the C_j with $x \in C_j$
  - Find(x): For $x \in V$, tell me ~~for which $j \in [1..k]$ we have $x \in C_j$~~
  - Union(x, y): If x and y are not in the same partition class, then modify the partition by uniting the two classes x and y lie in.

- How to identify partition classes? What exactly shall Find(x) return?
  - Not the whole class (highly inefficient!)
  - We do not want to number the partition classes, because a union operation would require a re-numbering
  - Best solution: For each C_i we designate a unique element of C_i as *representative* of C_i
    - since the C_i form a partition, this automatically ensures that representatives are different

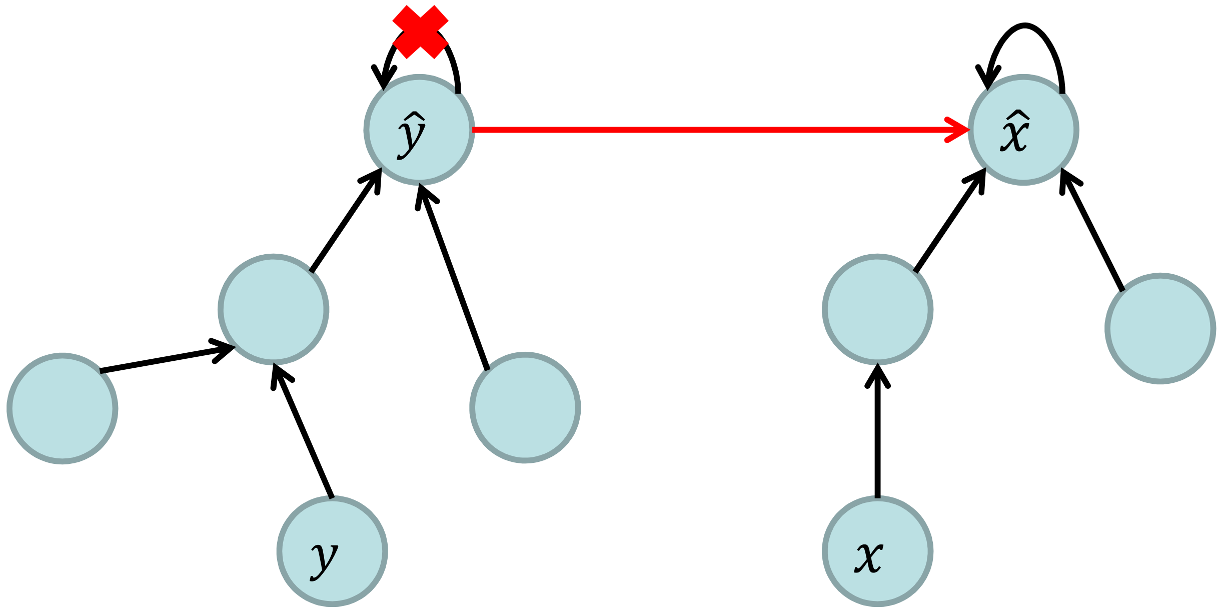
Union-Find with Arrays

- Use an array rep[V] that stores for each $x \in V$ the representative rep[x] of the partition class that x is in.
  - Find takes *constant* time
  - Union(x, y): for all $v \in V$ with rep[v]=rep[y] change the representative to rep[v]:=rep[x]. Takes *linear* time since the whole rep[.] array has to be traversed to find all elements of the class of y

- Making array-based union-find efficient:
  - array class[V]: class[v] is a list enumerating the class that v is the representative of (empty list if v is not a representative)
    - union(x, y) can now be implemented in time proportional to the size of the class of y
  - array size[V]: size[v] stores the length of the list class[v]
    - "union by size": make the smaller class a part of the bigger one
    - a single union can still take linear time (if both classes have linear size)
    - amortized analysis: whenever rep[v] changes, then the class v lies in grows by at least a factor of 2. Consequently, v will change its rep[.] at most $\log_2 n$ times. Total cost of all union operations = total number of rep[.] changes $\leq n \log_2 n$

Towards a More Elegant Union-Find Data Structure

- Insight of the previous slide: Asking for a constant-time Find operation produced lots of difficulties with the Union operation.
- Typical approach in data structures: balance the work!
  - here: ask for reasonably efficient (logarithmic time) Find and Union
- Pointer structures: Partition classes are directed trees with edges pointing towards the representative (which has a self-loop)



Find(x): Follow the arrows until a self-loop is found.
Cost: $\leq$ height of tree

Union(x, y): Find $\hat{x}$ and $\hat{y}$.
Then make $\hat{y}$ a child of $\hat{x}$

Union-Find with Trees: Details

- Each $x \in V$ has a pointer parent[x] "in the direction of its representative"
  - parent[x] = x if and only if x is the representative of its class
- Find operation:
  - Find(x) = x, if parent[x]=x
  - Find(x) = Find(parent[x]), otherwise.
- Union(x, y) operation: parent[Find(y)] := Find(x)
- Time complexities: $O(tree\ height)$
  - Find takes as long as the path to the representative is
  - Union = two Find operations plus constant time.
- To obtain low tree heights, we do *Union by rank*: the root of the tree with smaller height becomes a child of the root of the tree with larger height (details next slide)

Union by Rank: Implementation Details

- For each representative $\hat{x}$, we store the height of its tree in an array rank[$\hat{x}$]
- Union(x, y): Make the tree with smaller rank a subtree of the other!
  Let $\hat{x} \neq \hat{y}$ be the representatives of x and y
  - if rank[$\hat{x}$] > rank[$\hat{y}$], then parent[$\hat{y}$]:= $\hat{x}$
  - if rank[$\hat{x}$] < rank[$\hat{y}$], then parent[$\hat{x}$]:= $\hat{y}$
  - if rank[$\hat{x}$] = rank[$\hat{y}$], then parent[$\hat{x}$]:= $\hat{y}$ and rank[$\hat{y}$]:=rank[$\hat{y}$] + 1
- Lemma: The above implementation of Union(,) is correct, that is, if the rank array stores the correct tree heights before the union operation, then also after it.
  - Proof: Trivial check of the three cases.
- Initialization of a partition consisting of n sets containing a single element:
  - for all $x \in V$ set parent[x]:=x and rank[x]:=0

Union by Rank: Efficiency

- Lemma: If rank[$\hat{y}$] = k, then the partition class of $\hat{y}$ has size at least 2^k.
- Proof: Induction.
  - To get an element with rank k, you need a union of two elements of rank $k-1$. By induction, these sets have size at least 2^{k-1}, hence the union has size at least 2^k.
- Theorem: In the tree-based union-find data structure with union-by-rank any Find and Union operation takes time $O(\log n)$.
- Proof: Since all partition classes have at most n elements, by the lemma above at all times, all tree heights (=ranks) are at most $\log_2 n$.

Path Compression!

- Observation: If we execute Find(x), we compute the representative $\hat{x}$ of x and all the nodes v on the path from x to $\hat{x}$ – and forget this information ☹
  - why not update parent[v]:= $\hat{x}$ and point directly to the representative?

simple find:

```
if $i$ = parent[$i$] then
    return ($i$)
else
    return (Find(parent[$i$]));
```

find with path compression:

```
if $i$ = parent[$i$] then
    return ($i$)
else
    $r$ :=Find(parent[$i$]);
    parent[$i$] := $r$;
    return ($r$);
```

- Result: In union-find with union-by-rank and path compression, the amortized time complexity of Find is *InverseAckermann*(n).
- *InverseAckermann* is a function very very slowly tending to infinity:
  - *InverseAckermann*$(n) \leq 4$ when $n \leq$ number atoms in the universe.

Kruskal's Algorithm with Union-Find

```
Input: An undirected connected graph $G = ([0..n-1], E)$, edge weights $w : E \to \mathbb{R}$.
Output: The edge set $F$ of a minimum spanning tree $T = (V, F)$ of $(G, w)$.
Sort $E$ by increasing weight, let $e_1, \dots, e_m$ be the result;
U.SetUp($n$);    % sets up the union-find data structure with all vertices being singletons
$F = \emptyset$;
for $i = 1$ to $m$ do
    Let $e_i = \{x, y\}$;
    if U.Find($x$) $\neq$ U.Find($y$) then
        $F := F \cup \{e_i\}$;
        U.Union($x, y$);
return ($F$);
```

- Runtime: $O(m \log m)$ when using a union-find data structure that supports Union and Find in logarithmic time.

- Note: path compression brings no improvement here because sorting takes time $\Theta(m \log m)$

Summary Kruskal's Algorithm

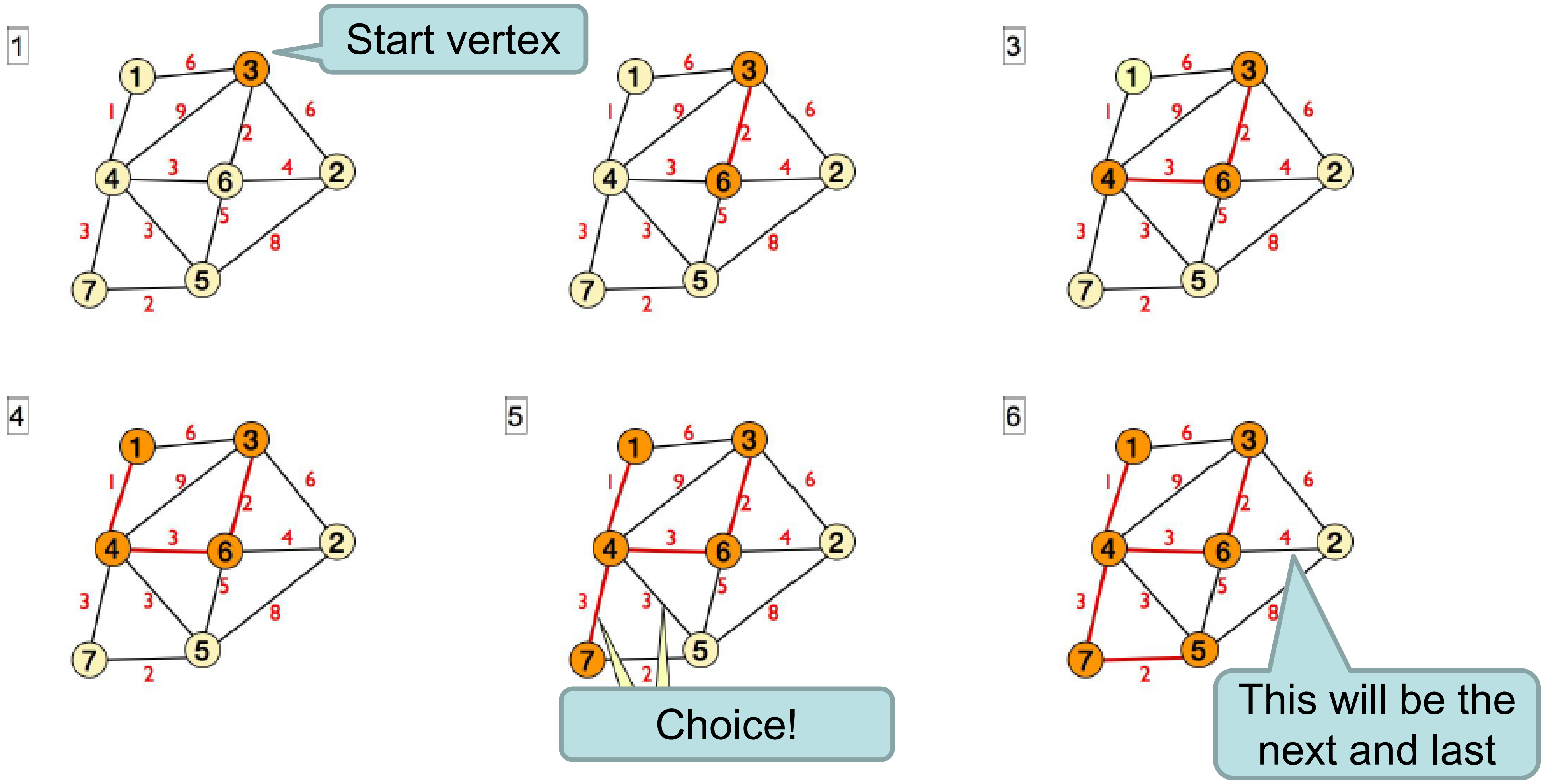
- A greedy algorithm to compute MSTs.
  - Greedy idea: Add the cheapest edge that does not create a cycle
  - Proof: Transform an arbitrary optimal solution into one that contains the first k Kruskal edges ($k = 1,2, \dots, n-1$)

- Implementation: Use a union-find data structure to efficiently check if adding an edge creates a cycle
  - Resulting runtime: $\Theta(m \log m)$, due to sorting the edges by weight

- Union-find data structures: Manage partitions.
  - pointers to representatives: fast union operation
  - union-by-rank: low tree heights
  - path compression: don't forget what you just learned

Prim's Algorithm in Pictures

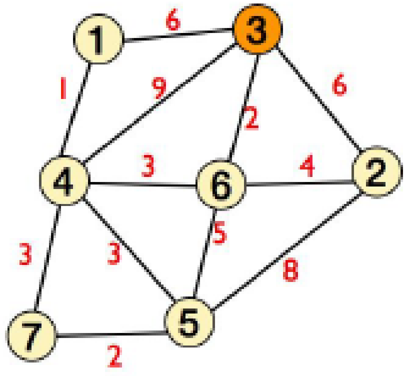
- Algorithm: Start with a tree consisting of one vertex. Repeat adding the cheapest edge from the tree to the rest until everything is connected.



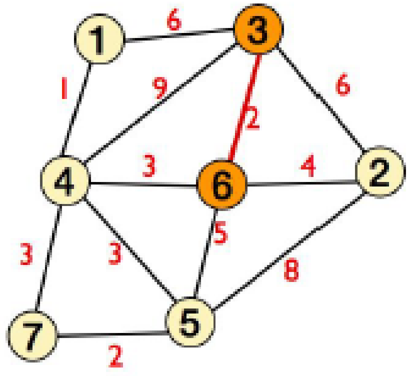
Rough Pseudocode of Prim's Algo.

```
1  Input: An undirected connected graph $G = (V, E)$, edge weights $w : E \to \mathbb{R}$.
2  Output: The edge set $F$ of a minimum spanning tree $T = (V, F)$ of $(G, w)$.
3  Let $v \in V$ be any vertex;
4  $V_T = \{v\}$;
5  $F = \emptyset$;
6  while $V_T \neq V$ do
7      Let $e = \{x, y\}$ be a cheapest edge such that $x \in V_T$ and $y \in V \setminus V_T$;
8      $V_T := V_T \cup \{y\}$;
9      $F := F \cup \{\{x, y\}\}$;
10 return $(F)$;
```

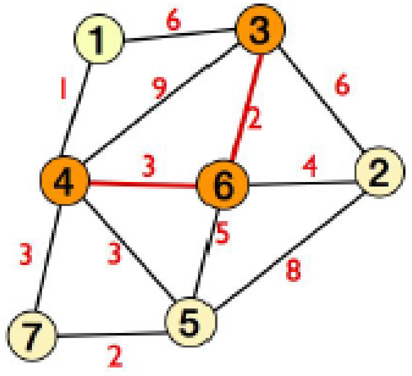
1



2



3



Correctness of Prim's Algorithm

- Trivial: When the algorithm terminates, (V, F) is a tree
  - Loop invariant: (V_T, F) is a tree.

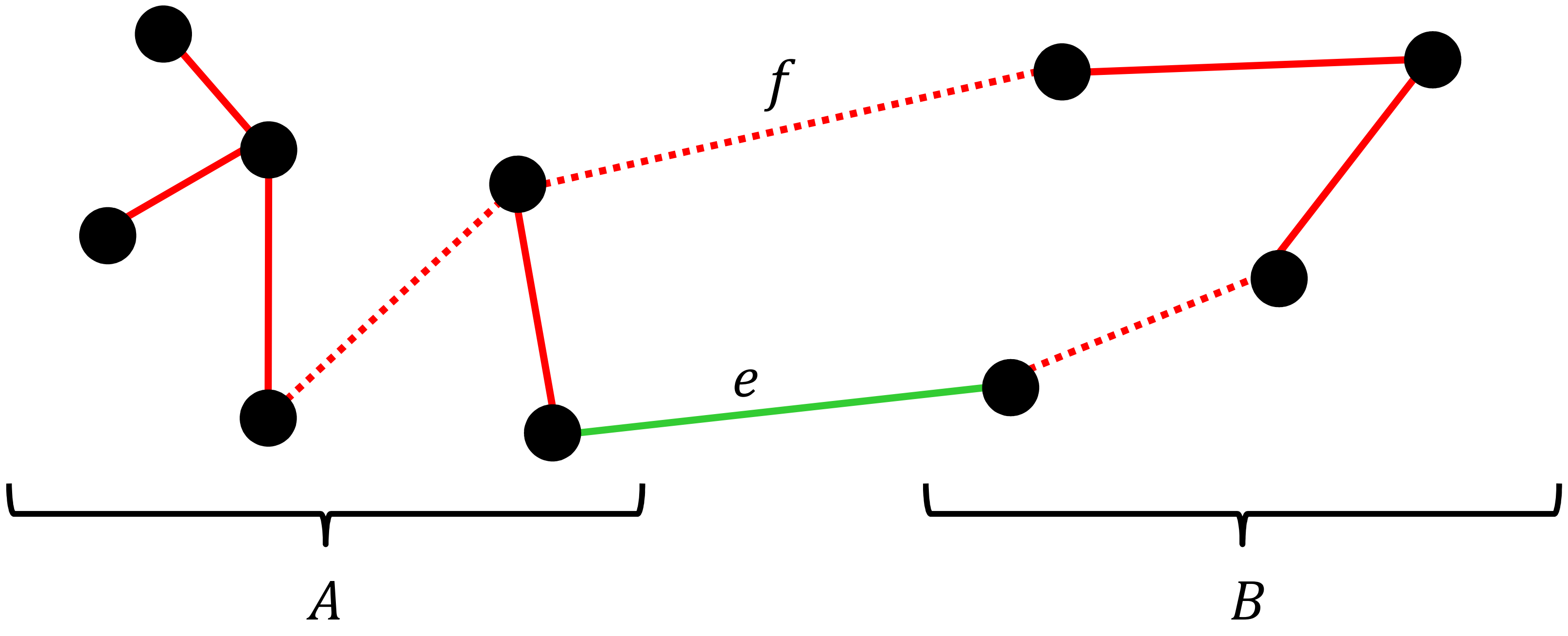
- Correctness (optimality) proof: The greedy decision can't be wrong
  → lecture 4

- Definitions:
  - $F \subseteq E$ is called *extendable* if there is a minimum spanning tree (V, F^*) such that $F \subseteq F^*$.
  - $E(A, B) \coloneqq \{e \in E \mid \exists a \in A, b \in B : e = \{a, b\}\}$

- **Cut Lemma:** Let F be extendable. Let $V = A \dot\cup B$ be a partition of V such that $F \cap E(A, B) = \emptyset$. Let $e \in E(A, B)$ with minimal weight. Then $F \cup \{e\}$ is extendable.

Cut Lemma: Proof in Pictures

- **Cut Lemma:** Let F be extendable. Let $V = A \dot\cup B$ be a partition of V such that $F \cap E(A,B) = \emptyset$. Let $e \in E(A,B)$ with minimal weight. Then $F \cup \{e\}$ is extendable.
  - 🟥 MST T containing F
  - T contains an edge $f \in E(A,B)$
  - $w(f) \geq w(e)$
  - $T \setminus \{f\} \cup \{e\}$ is a MST containing $F \cup \{e\}$



Cut Lemma: Complete Proof

- Let $T = (V, F^*)$ be a MST such that $F \subseteq F^*$ (use that F is extendable)
- Assume $e \notin F^*$ (otherwise nothing to do)
- Since (V, F^*) is a tree, $(V, F^* \cup \{e\})$ contains a cycle, which contains e.
- This cycle contains a second edge $f \in E(A, B)$.
- By construction, $f \in F^* \setminus F$ (since F does not intersect $E(A, B)$)
- Exchange argument:
  - $(V, F^* \setminus \{f\} \cup \{e\})$ is a tree that contains $F \cup \{e\}$
  - $w(e) \leq w(f)$ by definition of e (cheapest edge in $E(A, B)$)
  - Hence $w(F^* \setminus \{f\} \cup \{e\}) \leq w(F^*)$ and $(V, F^* \setminus \{f\} \cup \{e\})$ is (also) a MST. Consequently, $F \cup \{e\}$ is extendable.

Correctness of Prim's Algorithm

- **Cut Lemma:** Let F be extendable. Let $V = A \dot\cup B$ be a partition of V such that $F \cap E(A,B) = \emptyset$. Let $e \in E(A,B)$ with minimal weight. Then $F \cup \{e\}$ is extendable.

- Prim's algorithm always adds edges as in cut lemma ($A = V_T$, $B = V \setminus V_T$):

```
while $V_T \neq V$ do
    Let $e = \{x, y\}$ be a cheapest edge such that $x \in V_T$ and $y \in V \setminus V_T$;
    $V_T := V_T \cup \{y\}$;
    $F := F \cup \{\{x, y\}\}$;
```

- "F is extendable" is a loop invariant (trivially fulfilled at the start with $F = \emptyset$)

- Hence Prim's algorithm ends with an extendable set.
  But: This set is a tree! Hence it is already a MST

- Theorem: Prim's algorithm computes a MST.

Implementing Prim’s Algorithm

```
1  **Input:** An undirected connected graph $G = (V, E)$, edge weights $w : E \to \mathbb{R}$.
2  **Output:** The edge set $F$ of a minimum spanning tree $T = (V, F)$ of $(G, w)$.
3  Let $v \in V$ be any vertex;
4  $V_T = \{v\}$;
5  $F = \emptyset$;
6  **while** $V_T \neq V$ **do**
7      Let $e = \{x, y\}$ be a cheapest edge such that $x \in V_T$ and $y \in V \setminus V_T$;
8      $V_T := V_T \cup \{y\}$;
9      $F := F \cup \{\{x, y\}\}$;
10 **return** $(F)$;
```

- Only difficult point: How to find the cheapest edge in line 7?
  - Check all edges: $O(m)$ per iteration, total $O(nm)$
    → redoing much work every iteration
  - Solution: Priority queue

Priority Queues (Reminder From INF411)

- A data structure that stores pairs "(item, key)"
  - item: what you really want to store
  - key: a number representing the priority of the item

- Supported operations (in at most logarithmic time)
  - *extractmin*: delete from the queue and return the item with smallest key
  - *add (insert)*: add a new pair (item, key) to the queue
  - *decreasekey*: decrease the key of an item (increase its priority)

- Note: There are many variants of priority queues.
  - Some do not support *decreasekey*
  - Most allow an initial setup (*build*, adding n items) in time $O(n)$, either as explicit operation (example: binary heap) or because *add* takes only constant time (example: Fibonacci heap).

Adding a Priority Queue (Edges)

```
1  Input: An undirected connected graph $G = (V, E)$, edge weights $w : E \to \mathbb{R}$.
2  Output: The edge set $F$ of a minimum spanning tree $T = (V, F)$ of $(G, w)$.
3  Let $v \in V$ be any vertex;
4  $V_T = \{v\}$;
5  $F = \emptyset$;
6  while $V_T \neq V$ do
7      Let $e = \{x, y\}$ be a cheapest edge such that $x \in V_T$ and $y \in V \setminus V_T$;
8      $V_T := V_T \cup \{y\}$;
9      $F := F \cup \{\{x, y\}\}$;
10 return $(F)$;
```

- Use a priority Q to store interesting edges e with their weight $w(e)$ as key:
  - before line 6: Initialize an empty queue, add all edges incident with v
  - instead of line 7: $e =$ Q.extractmin
  - skip lines 8 and 9 if $e \notin E(V_T, V \setminus V_T)$
  - with lines 8 and 9: add all edges incident with y to the queue
- Analysis: Each edge is added and extracted at most twice: $O(m \log m)$ time

When an edge in the queue becomes obsolete (because it does not leave T anymore), we do not delete it immediately, but wait until we extract it!

Adding a Priority Queue (Vertices)

```
Input: An undirected connected graph $G = (V, E)$, edge weights $w : E \to \mathbb{R}$.
Output: The edge set $F$ of a minimum spanning tree $T = (V, F)$ of $(G, w)$.
Let $v \in V$ be any vertex;
$V_T = \{v\}$;
$F = \emptyset$;
while $V_T \neq V$ do
    Let $e = \{x, y\}$ be a cheapest edge such that $x \in V_T$ and $y \in V \setminus V_T$;
    $V_T := V_T \cup \{y\}$;
    $F := F \cup \{\{x, y\}\}$;
return $(F)$;
```

- Alternative: Store vertices y with their distance $d(y, V_T)$ from V_T as key
  - before line 6: Initialize an empty queue, add all neighbors of v
  - instead of line 7: $y =$ Q.extractmin; find $x \in V_T \cap N(y)$ with $w(\{x, y\})$ min.
  - after line 9 (in the while loop): for all $z \in N(y) \setminus V_T$ do
    - Q.add($z, w(\{y, z\})$) if z not yet in Q
    - Q.decreasekey($z, w(\{y, z\})$) if z in Q and now closer to V_T
- Analysis: one *add* and *extractmin* per vertex, one *decreasekey* per edge

Depending on the priority queue used, this might need additional elementary datastructures

Complexity of Prim's Algorithm

- Just seen (with some details suppressed → poly):
  - edge-based priority queue: $\Theta(m)$ *add* and *extractmin* operations
  - vertex-based priority queue: $\Theta(n)$ *add* and *extractmin* operations, $\Theta(m)$ *decreasekey* operations

- Simple priority queues without decreasekey: *add* and *extractmin* in logarithmic time → $O(m \log m)$ time (edge-based)
- Simple priority queues with decreasekey: *add*, *extractmin* and *decreasekey* in logarithmic time → $O(m \log m)$ time edge-based or vertex-based
- Advanced priority queues: *add* and *extractmin* in logarithmic time, *decreasekey* in (amortized) constant time
  - $O(m + n \log n)$ time for the vertex-based version of Prim's algorithm

Summary Prim's Algorithm

- Greedy algorithm to compute MSTs
  - Greedy idea: Add the cheapest outgoing edge
  - Proof: "can't be wrong" argument, cut lemma

- Implementation: Use a *priority queue* to find the cheapest outgoing edge
  - edge-based priority queue: $\Theta(m)$ add and $\Theta(m)$ extractmin operations, each taking $O(\log m)$ time (simple queue) → $O(m \log m)$ time compl.
  - vertex-based queue: $\Theta(n)$ add, $\Theta(n)$ extractmin, and $\Theta(m)$ decreasekey operations
    → $O(m \log m)$ time complexity with simple queues (allowing decr-key!)
    → $O(m + n \log n)$ time complexity with, e.g., Fibonacci heaps
  - Warning: The implicit constants are much worse for Fibonacci heaps, so in practice often simpler queues are faster (despite the theoretically worse time complexities)

Summary

- Graph theory and graph algorithms are central in algorithmics, even when no graph is visible

- Advanced greedy algorithms and correctness/optimality proofs
  - Augmenting path algorithm for Maximum Matching
  - Kruskal’s algorithm for MST
  - Prim’s algorithm for MST

- Data structures make the difference: an $O(m \log m)$ runtime or slightly better instead of the naïve $O(mn)$ runtime for MST algorithms
  - Union-find
  - Priority queues

What Next?

- Read the poly
- Enjoy the quizzes!
- PC tomorrow
- DM2 is out, deadline January 18
- Deadline PI: February 5
- See you next Monday with **graph algorithms 2 – all types of short paths**
  - single-source shortest path problems
  - all-pairs shortest paths
  - Hamilton/Euler cycles
  - traveling salesman problem