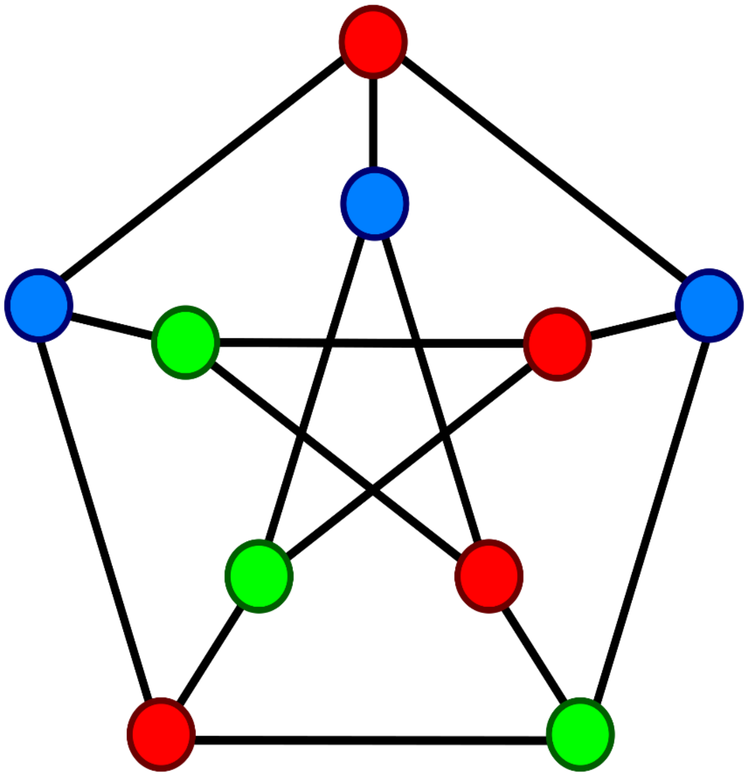
Lecture 12: Summary

Joachim Gudmundsson



THE UNIVERSITY OF SYDNEY

comp2007 – Overview I

- Graphs
  - Definitions and properties
  - Graph traversal
  - Applications: min-link path, bipartitness…

- Greedy algorithms
  - Greedy technique
  - Standard correctness proof: exchange argument
  - Applications: Scheduling, MST, Dijkstra (incl. properties)

comp2007 – Overview II

- Divide-and-Conquer algorithms
  - General technique: break, solve and combine
  - Recursion: How to state and solve a recursion
  - Standard correctness proof: Induction
  - Applications: Mergesort, Inversions,

- Sweepline algorithms
  - General technique: “imaginary” sweep line, process event points
  - Standard correctness proof:
    - Changes only at event points
    - Maintain invariant
  - Applications: Convex hull, segment intersection

comp2007 – Overview III

- Dynamic programming
  - General technique: break, solve, combine
    - Define states
    - State recursion
  - Correctness proof: Induction
  - Applications: Knapsack, weighted scheduling, RNA, Bellman-Ford,…

- Flow networks
  - Properties of flow network: max flow, min cut, integer lemma,…
  - General technique: reduce to a flow network
  - Correctness proof: Solution for X $\Leftrightarrow$ Solution for FN
  - Applications: matching, edge-disjoint paths, circulation,…

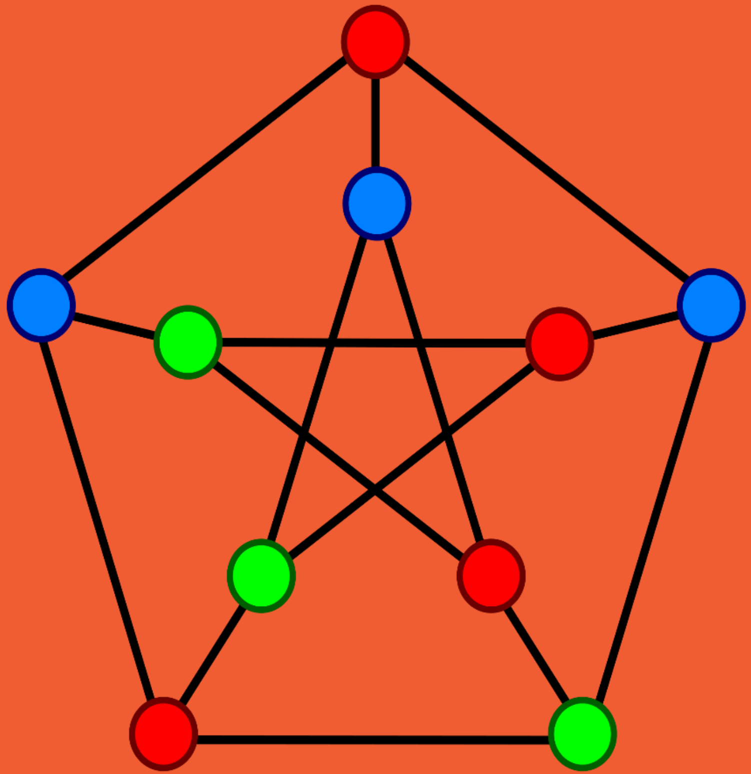
comp2007 – Overview IV

- Complexity
  - Polynomial-time reductions!
  - Classes: P, NP, NP-complete, NP-hard
  - How to prove that a problem belongs to P/NP/NP-complete
  - Understand the NP-complete problems in lecture 10.

- Coping with hardness
  - Understand the basic concepts:
    - Exponential time algorithms
    - Restricted instances
    - Approximation algorithms

Lecture 2: Graphs

Joachim Gudmundsson



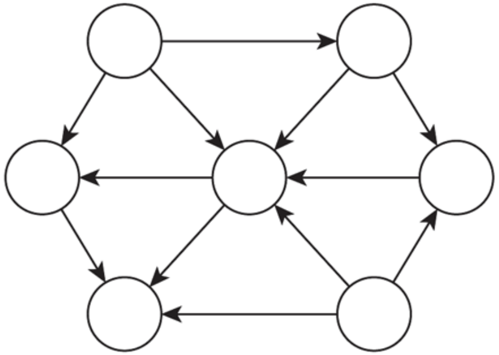
THE UNIVERSITY OF SYDNEY

Lecture 2: Graphs

- Definitions
- Representations
- Graph traversal:
  - Breadth First Search (incl. layers)
  - Depth First Search
- Applications: bipartiteness, min-link paths, …

Basic Definitions

- Graphs (directed/undirected)
- Tree (rooted/unrooted)
- Path (simple), connectivity, cycle…



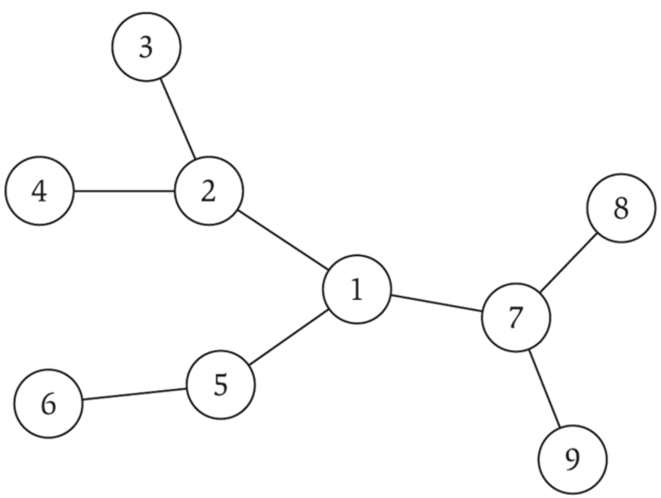
Trees

Definition: An undirected graph is a **tree** if it is connected and does not contain a cycle.

Number of edges in a tree?

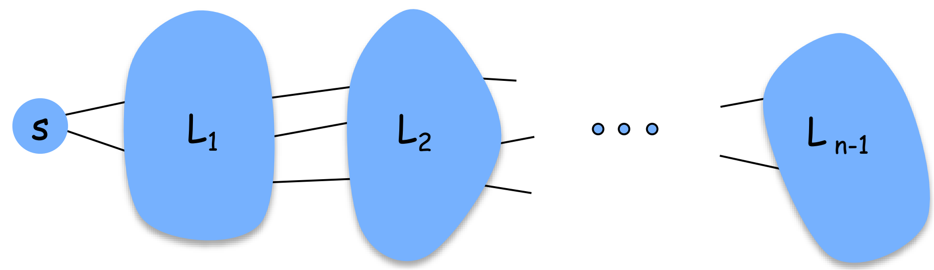
Theorem: Let G be an undirected graph on n nodes. Any two of the following statements imply the third.

– G is connected.
– G does not contain a cycle.
– G has n-1 edges.



Graph traversal: Breadth First Search

BFS intuition. Explore outward from s in all possible directions, adding nodes one "layer" at a time.



BFS algorithm.

– $L_0 = \{ s \}$.
– L_1 = all neighbors of L_0.
– L_2 = all nodes that do not belong to L_0 or L_1, and that have an edge to a node in L_1.
– L_{i+1} = all nodes that do not belong to an earlier layer, and that have an edge to a node in L_i.

Theorem: For each i, L_i consists of all nodes at distance exactly i from s. There is a path from s to t iff t appears in some layer.

Application: BFS applied to Shortest paths

The shortest path between two nodes u,v in an unweighted graph G, is the path with the minimum number of edges that connects u and v (if it exists).

Compute the shortest paths from a given node s to all other nodes using BFS.

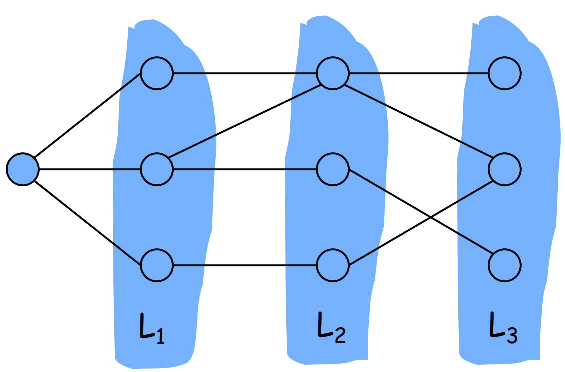
Application: BFS applied to bipartiteness

Lemma: If a graph G is bipartite, it cannot contain an odd length cycle.

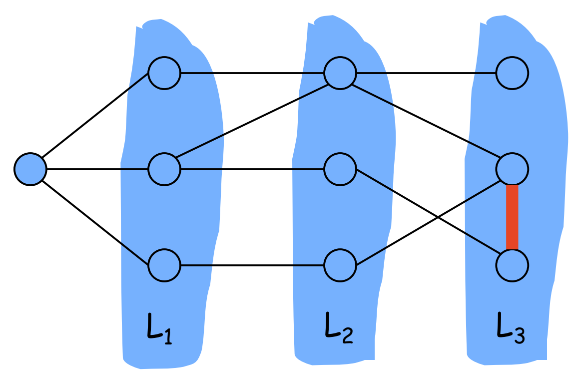
Lemma: Let G be a connected graph, and let $L_0, \ldots, L_k$ be the layers produced by BFS starting at node s. Exactly one of the following holds.

(i) No edge of G joins two nodes of the same layer, and G is bipartite.

(ii) An edge of G joins two nodes of the same layer, and G contains an odd-length cycle (and hence is not bipartite).



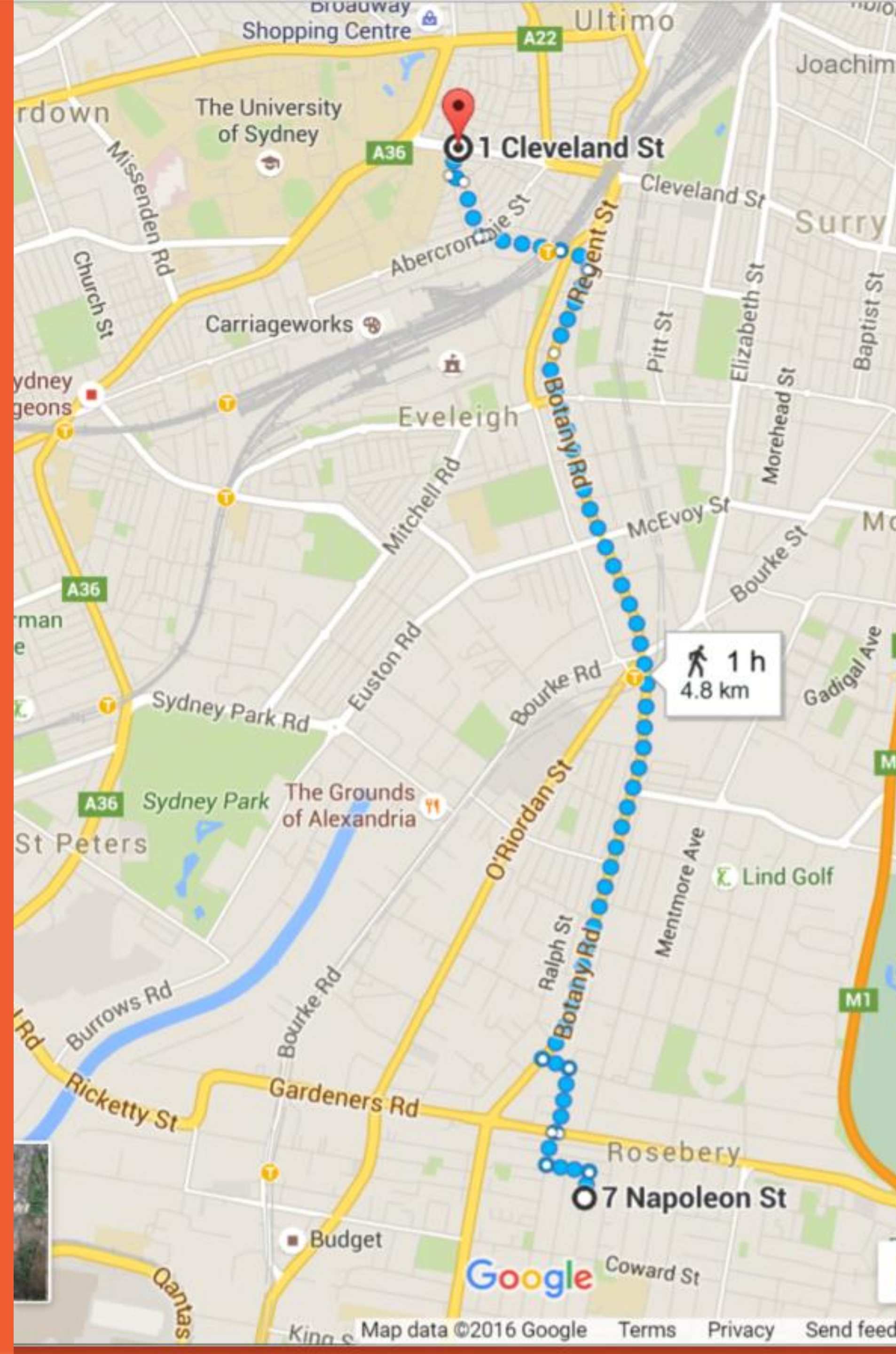
Case (i)



Case (ii)

Lecture 3:
Greedy algorithms

THE UNIVERSITY OF SYDNEY



Greedy algorithms

A greedy algorithm is an algorithm that follows the problem solving heuristic of making the locally optimal choice at each stage with the hope of finding a global optimum.

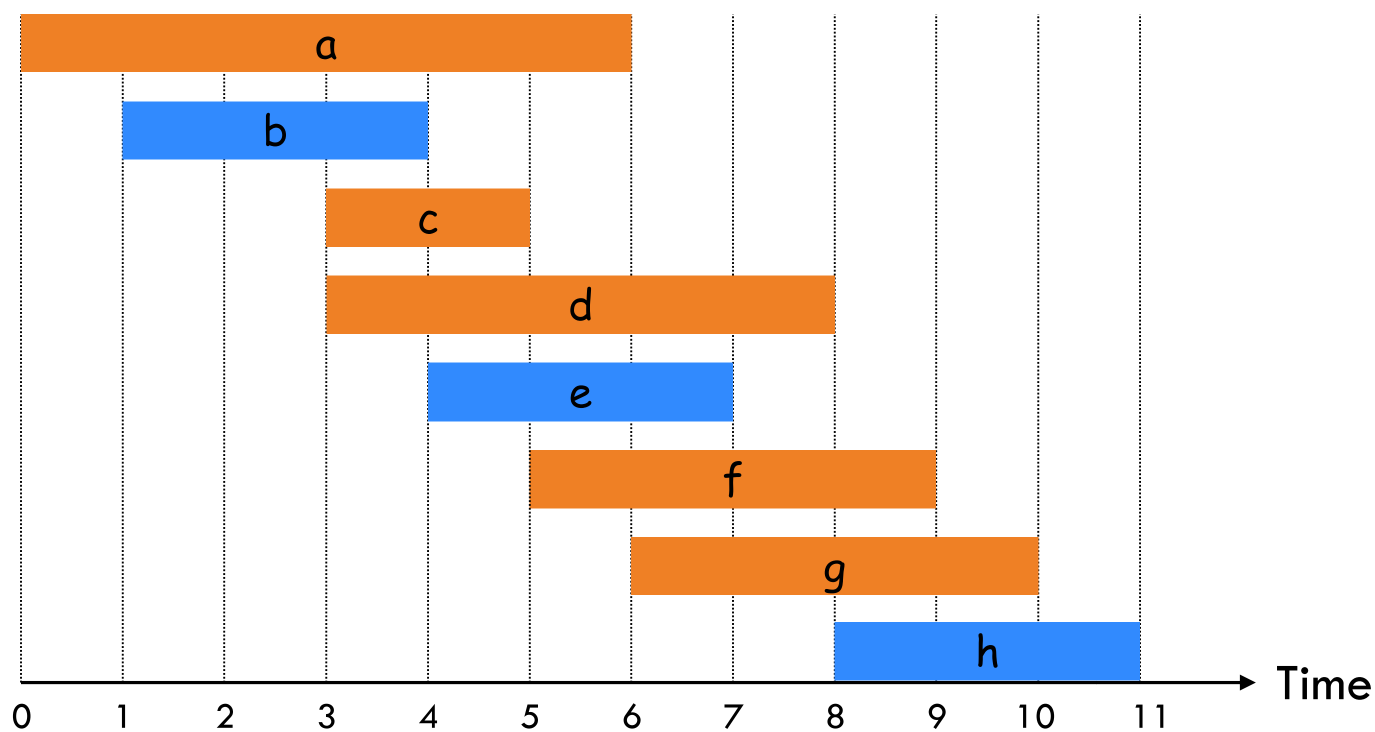
Greedy

Examples of problems that can be solved using a greedy approach:

– Interval scheduling/partitioning
– Scheduling to minimize lateness
– Shortest path
– Minimum spanning trees

Interval Scheduling

- Interval scheduling.
  - **Input:** Set of n jobs. Each job i starts at time s_i and finishes at time f_i.
  - Two jobs are compatible if they don't overlap in time.
  - **Goal:** find maximum subset of mutually compatible jobs.



Interval Scheduling: Greedy Algorithm

Greedy algorithm. Consider jobs in increasing order of finish time. Take each job provided it is compatible with the ones already taken.

```
Sort jobs by finish times so that f₁ ≤ f₂ ≤ ... ≤ fₙ.

  jobs selected
A ← ∅
for j = 1 to n {
   if (job j compatible with A)
      A ← A ∪ {j}
}
return A
```

Implementation. O(n log n).

- Remember job j* that was added last to A.
- Job j is compatible with A if $s_j \geq f_{j^*}$.

Greedy algorithms: Analysis

1. Define your solution X and an optimal solution OPT.

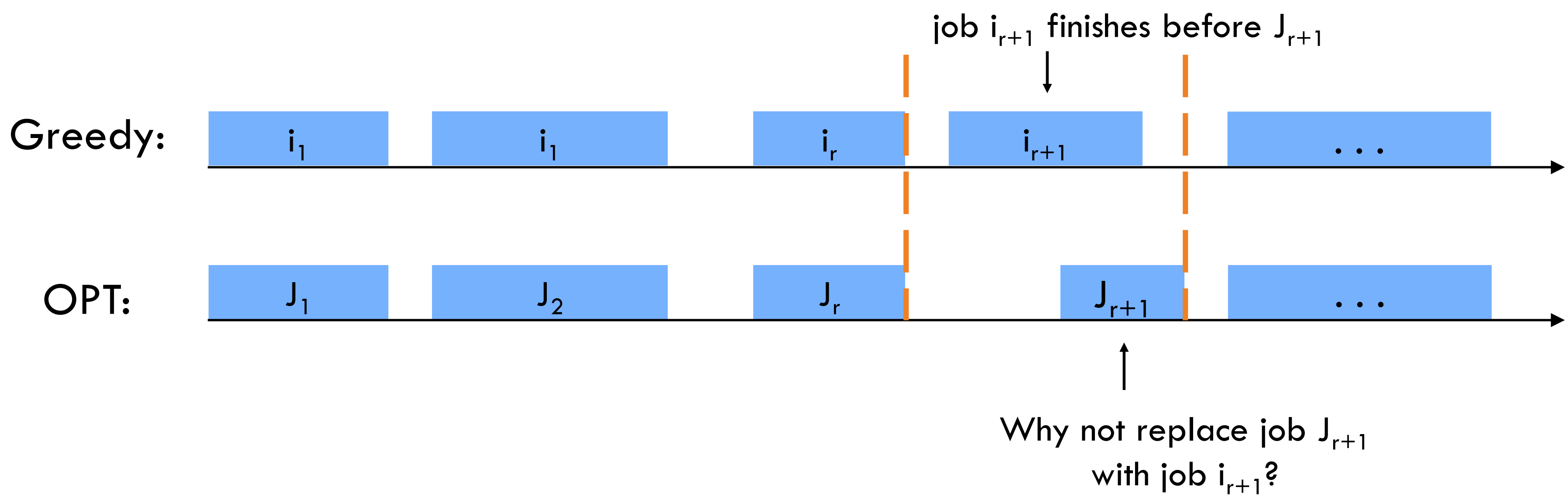
2. Compare solutions. If X$\neq$OPT then they must differ in a specific way.

3. Exchange pieces. Transform OPT by exchanging some piece of OPT for some piece of X.

4. Iterate. Argue optimality.

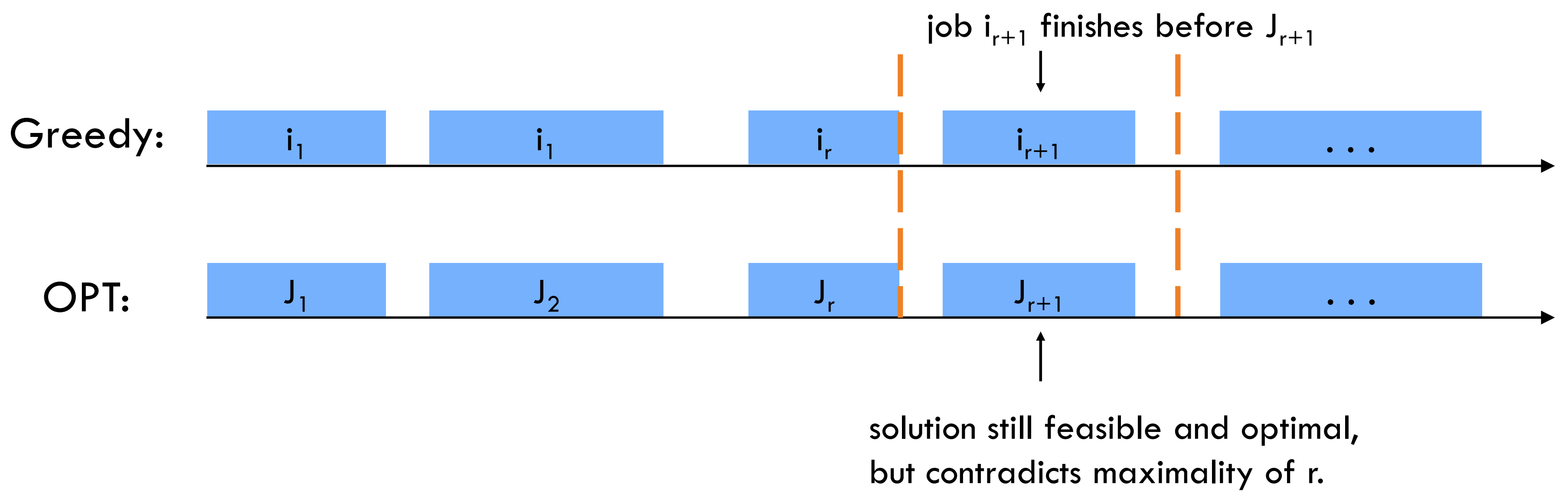
Interval Scheduling: Analysis

- **Theorem:** Greedy algorithm [Earliest finish time] is optimal.
- **Proof:** (by contradiction)
  - Assume greedy is not optimal, and let's see what happens.
  - Let $i_1, i_2, \ldots i_k$ denote the set of jobs selected by greedy.
  - Let $J_1, J_2, \ldots J_m$ denote the set of jobs in an optimal solution with $i_1 = J_1, i_2 = J_2, \ldots, i_r = J_r$ for the largest possible value of r.



Interval Scheduling: Analysis

- **Theorem:** Greedy algorithm [Earliest finish time] is optimal.
- **Proof:** (by contradiction)
  - Assume greedy is not optimal, and let's see what happens.
  - Let i_1, i_2, ... i_k denote the set of jobs selected by greedy.
  - Let J_1, J_2, ... J_m denote the set of jobs in an optimal solution with $i_1 = J_1$, $i_2 = J_2$, ..., $i_r = J_r$ for the largest possible value of r.

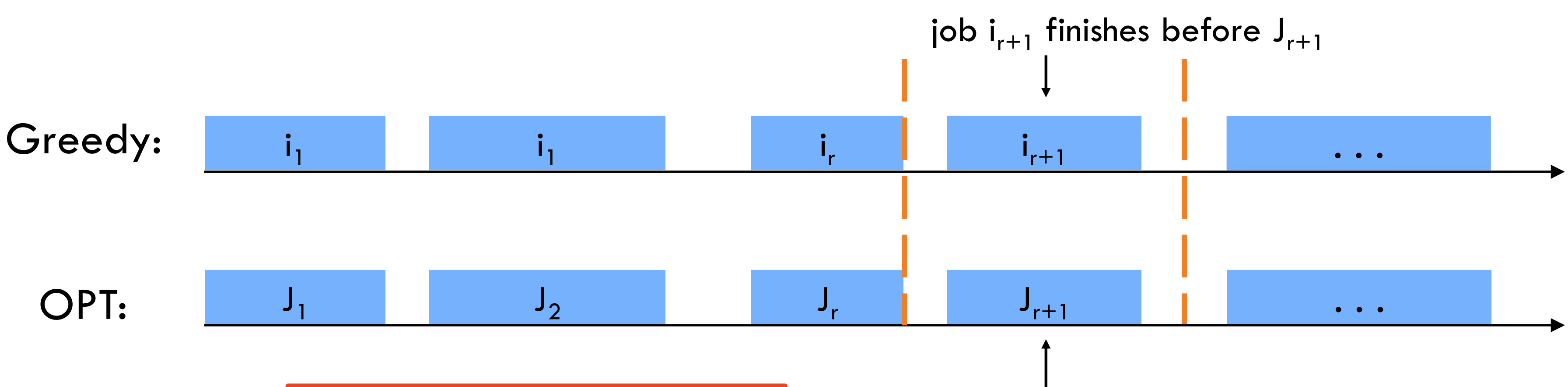


Interval Scheduling: Analysis

– **Theorem:** Greedy algorithm [Earliest finish time] is optimal.

– **Proof:** (by contradiction)

  – Assume greedy is not optimal, and let's see what happens.
  – Let $i_1, i_2, \ldots i_k$ denote the set of jobs selected by greedy.
  – Let $J_1, J_2, \ldots J_m$ denote the set of jobs in an optimal solution with $i_1 = J_1, i_2 = J_2, \ldots, i_r = J_r$ for the largest possible value of r.

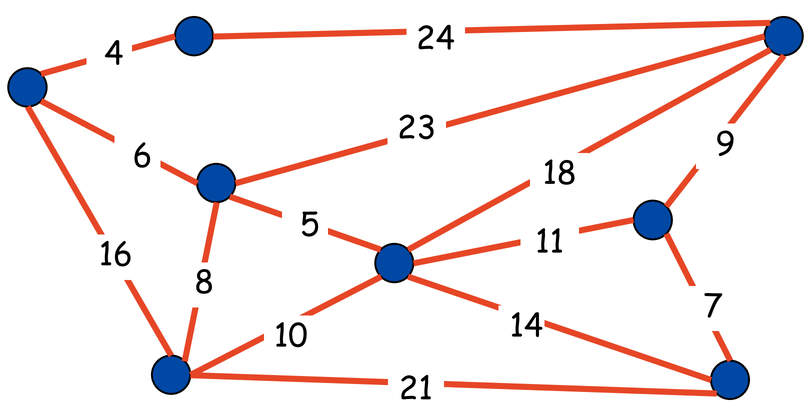


Exchange argument!

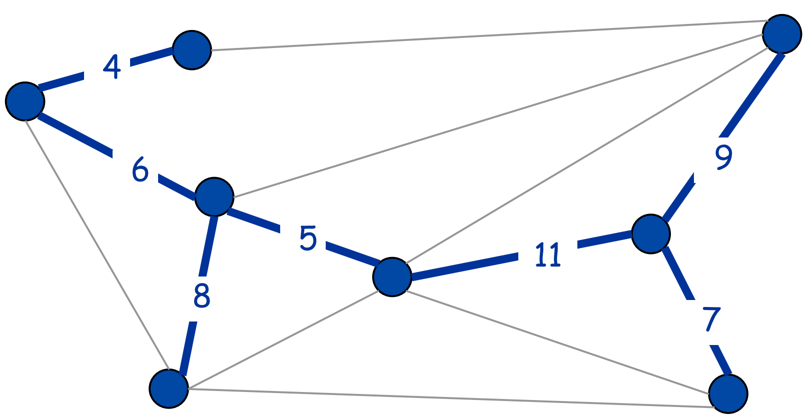
solution still feasible and optimal,
but contradicts maximality of r.

Minimum Spanning Tree

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



$G = (V, E)$



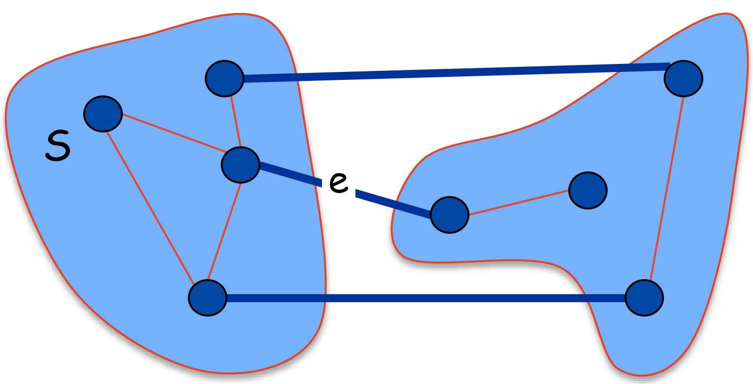
$T, \; \Sigma_{e \in T} c_e = 50$

MST properties

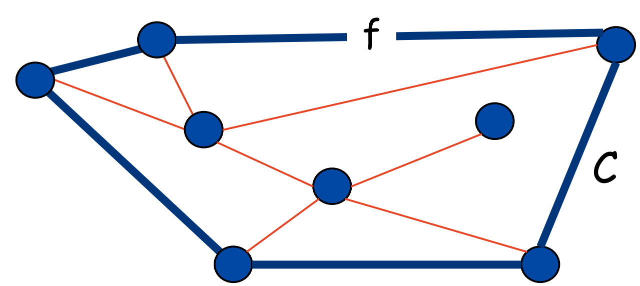
– **Simplifying assumption.** All edge costs c_e are distinct.

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

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



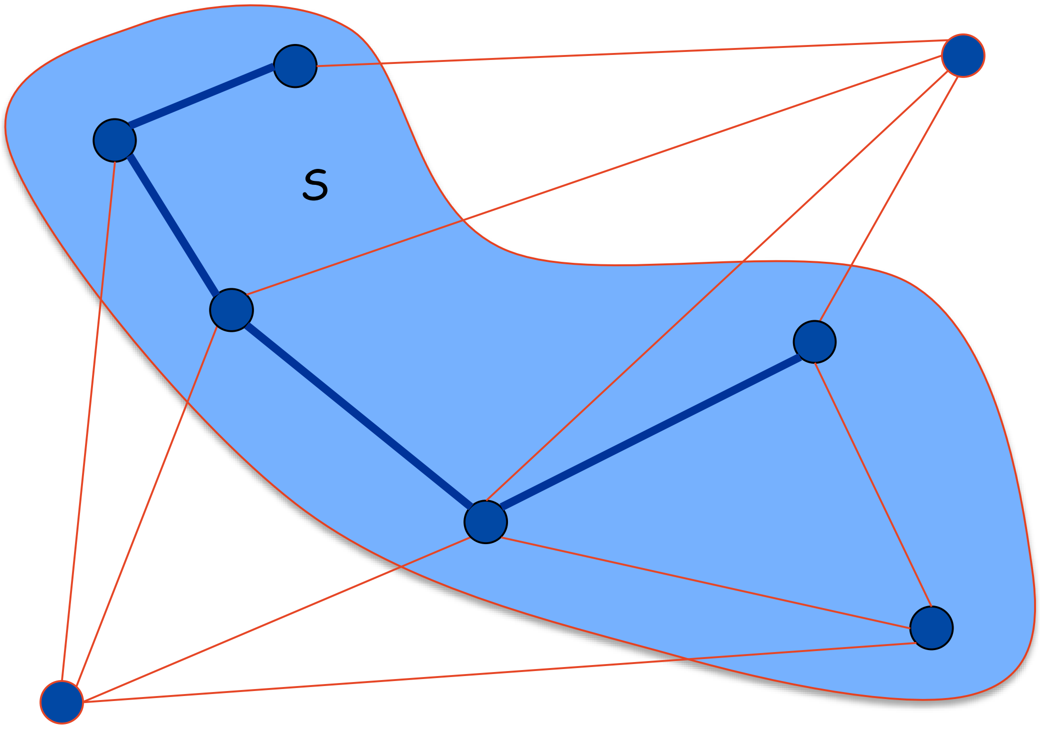
e is in the MST



f is not in the MST

Prim's Algorithm

- Prim's algorithm. [Jarník 1930, Dijkstra 1957, Prim 1959]
  - Initialize S = any node.
  - Apply cut property to S.
  - Add min cost edge in cutset corresponding to S to T, and add one new explored node u to S.



Summary: Greedy algorithms

A greedy algorithm is an algorithm that follows the problem solving heuristic of making the locally optimal choice at each stage with the hope of finding a global optimum.

Problems

- Interval scheduling/partitioning
- Scheduling: minimize lateness
- Minimum spanning tree (Prim's algorithm)
- Shortest path in graphs (Dijkstra's algorithms)
- …

Lecture 4:
Divide & Conquer

THE UNIVERSITY OF SYDNEY

Divide-and-Conquer

The divide-and-conquer strategy solves a problem by:

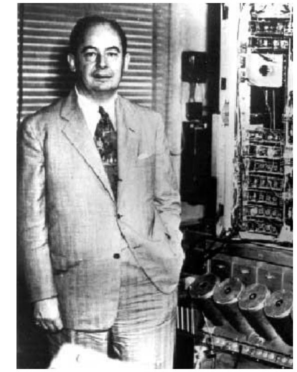
1. Breaking it into subproblems that are themselves smaller instances of the same type of the original problem.
2. Recursively solving these subproblems.
3. Appropriately combining (merging) their answers.

Most common usage.

– Break up problem of size n into two equal parts of size ½n.
– Solve two parts recursively.
– Combine two solutions into overall solution in linear time.

Mergesort

- Mergesort.
  - Divide array into two halves.
  - Recursively sort each half.
  - Merge two halves to make sorted whole.



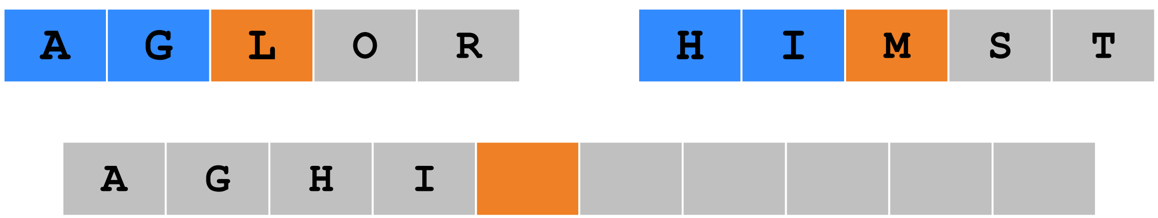
Jon von Neumann (1945)

A	L	G	O	R	I	T	H	M	S			
A	L	G	O	R	I	T	H	M	S		divide	O(1)
A	G	L	O	R	H	I	M	S	T		sort	2T(n/2)
A	G	H	I	L	M	O	R	S	T		merge	O(n)

Merging

- Merging. Combine two pre-sorted lists into a sorted whole.

- How to merge efficiently?
  - Linear number of comparisons.
  - Use temporary array.



Mergesort

- Mergesort.
  - Divide array into two halves.
  - Recursively sort each half.
  - Merge two halves to make sorted whole.

Jon von Neumann (1945)

A	L	G	O	R	I	T	H	M	S

A	L	G	O	R	I	T	H	M	S	divide	O(1)
A	G	L	O	R	H	I	M	S	T	sort	2T(n/2)

A	G	H	I	L	M	O	R	S	T	merge	O(n)

$$\Rightarrow \quad T(n) = O(n) + 2T(n/2)$$

Counting Inversions: Divide-and-Conquer

- Divide-and-conquer.
  - Divide: separate list into two pieces.
  - Conquer: recursively count inversions in each half.
  - Combine: count inversions where a_i and a_j are in different halves, and return sum of three quantities.

1	5	4	8	10	2	6	9	12	11	3	7

Divide: O(1).

1	5	4	8	10	2

6	9	12	11	3	7

Conquer: 2T(n/2)

5 blue-blue inversions

8 green-green inversions

9 blue-green inversions

5-3, 4-3, 8-6, 8-3, 8-7, 10-6, 10-9, 10-3, 10-7

Combine: ???

Total = 5 + 8 + 9 = 22.

Counting Inversions: Combine

Combine: count blue-green inversions

- Assume each half is sorted.
- Count inversions where a_i and a_j are in different halves.
- Merge two sorted halves into sorted whole.

3	7	10	14	18	19

5 blue-blue inversions

2	11	16	17	23	25
6	3	2	2	0	0

8 green-green inversions

How many blue-green inversions?

Counting Inversions: Combine

Combine: count blue-green inversions

- Assume each half is sorted.
- Count inversions where a_i and a_j are in different halves.
- Merge two sorted halves into sorted whole.

3	7	10	14	18	19

2	11	16	17	23	25
6	3	2	2	0	0

13 blue-green inversions: $6 + 3 + 2 + 2 + 0 + 0$

Count: $O(n)$

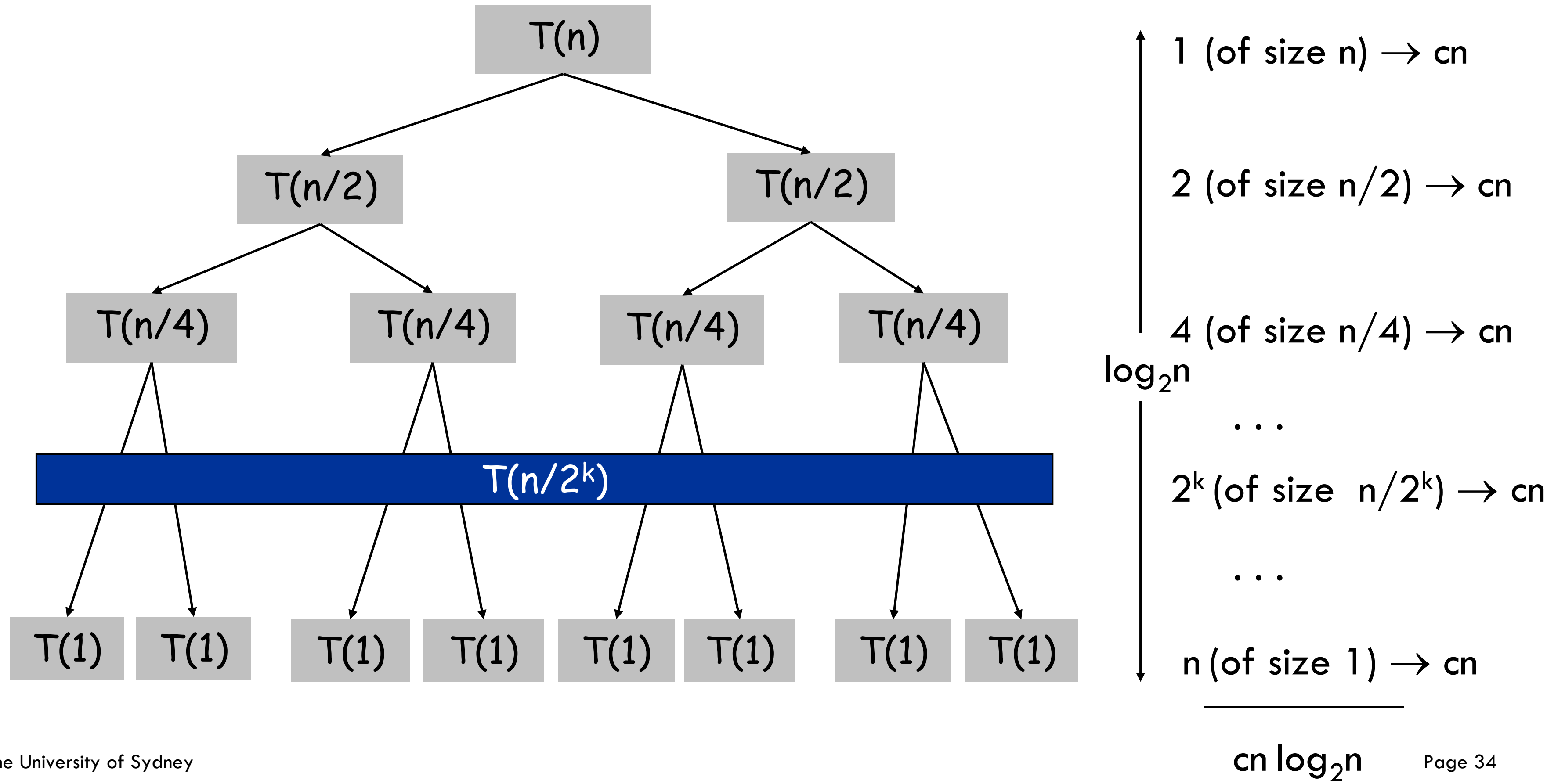
2	3	7	10	11	14	16	17	18	19	23	25

Merge: $O(n)$

Time: $T(n) = 2T(n/2) + O(n) = O(n \log n)$

Proof by unrolling

$$T(n) = \begin{cases} c & \text{if } n = 1 \\ \underbrace{2T(n/2)}_{\text{sorting both halves}} + \underbrace{cn}_{\text{merging}} & \text{otherwise} \end{cases}$$



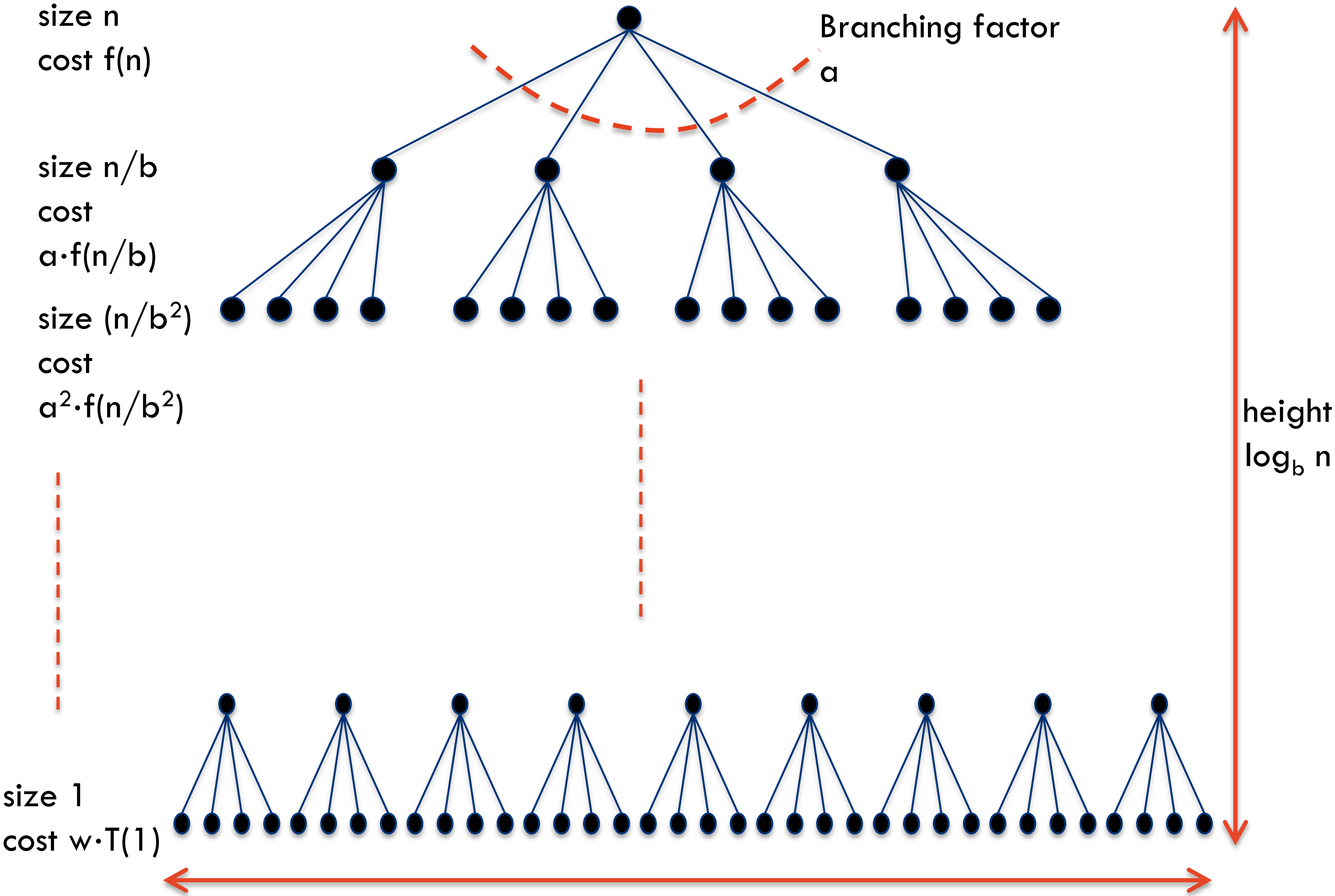
The master method

The master method applies to recurrences of the form

$$T(n) = a \cdot T(n/b) + f(n) \ ,$$

where $a \geq 1$, $b > 1$, and f is asymptotically positive.

$T(n) = a \cdot T(n/b) + f(n)$



width $w = a^{\log_b n} = n^{\log_b a}$

Summary: Divide-and-Conquer

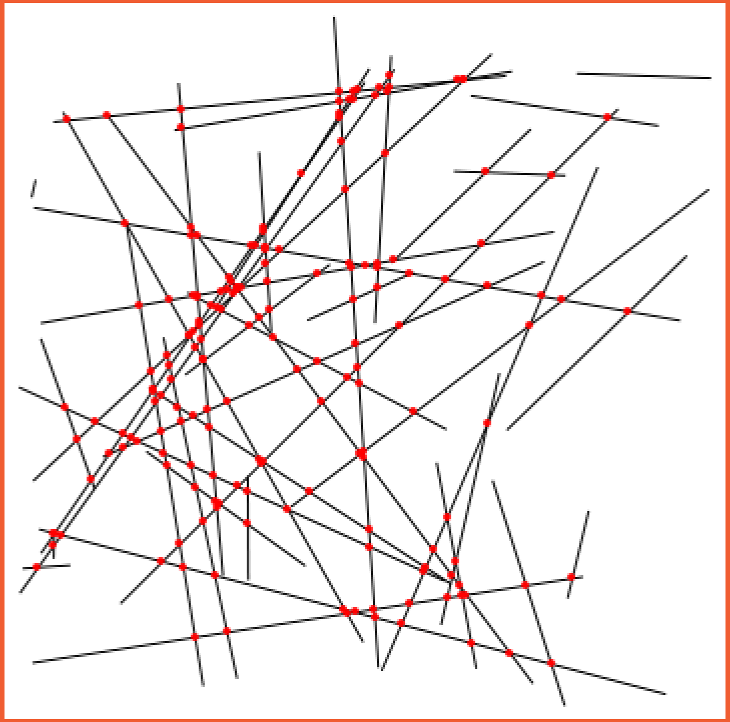
- **Algorithm:**
  - Break up problem into several parts.
  - Solve each part recursively.
  - Combine solutions to sub-problems into overall solution.

- **Complexity analysis: Solve recursion**

- **Correctness: Induction**

- **Problems**
  - Merge Sort
  - Closest pair
  - Multiplication

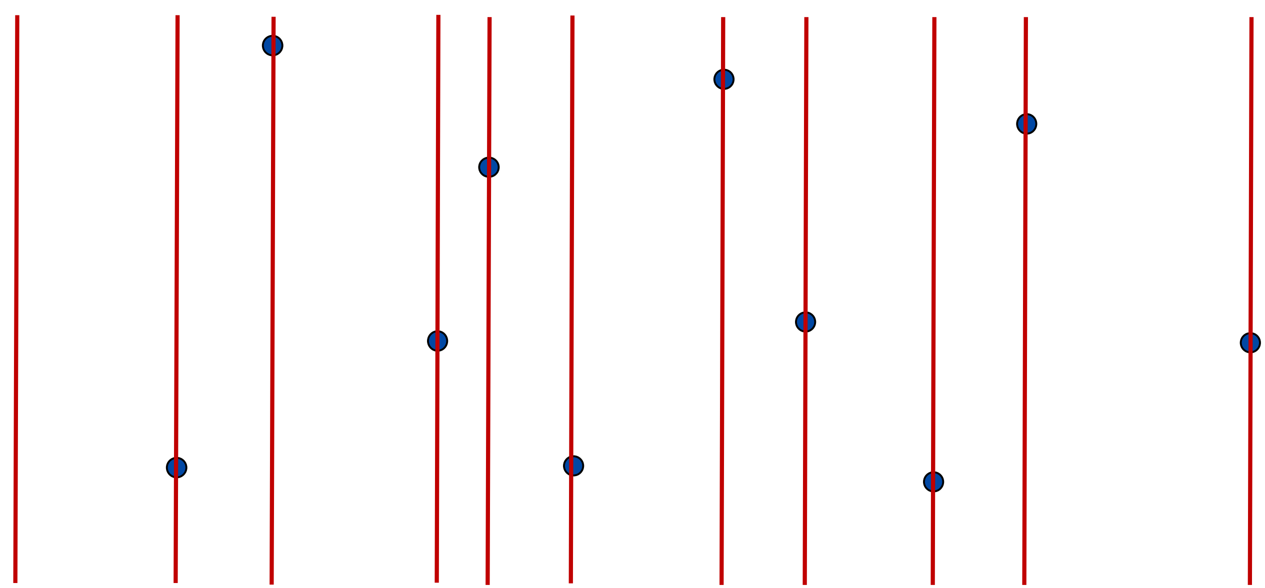
Lecture 5:
Sweepline technique
(and computational geometry)



THE UNIVERSITY OF SYDNEY

Design technique

- Simulate sweeping a vertical line from left to right across the plane.
- Events: Discrete points in "time" when sweep line status needs to be updated.
- Sweep line status: Store information along with the sweep line.
- Maintain invariant: At any point in time, to the left of sweep line everything is clean, i.e., properly processed.



Design technique

- Simulate sweeping a vertical line from left to right across the plane.
- Events: Discrete points in "time" when sweep line status needs to be updated.
- Sweep line status: Store information along with the sweep line.
- Maintain invariant: At any point in time, to the left of sweep line everything is clean, i.e., properly processed.

Algorithm Generic_Plane_Sweep:

Initialize **sweep line status** S at time $x=-\infty$

Store initial events in **event queue** Q, a priority queue ordered by x-coordinate

while $Q \neq \varnothing$

    // extract next event e:

    e = Q.extractMin();

    // handle event:

    Update sweep line status

    Discover new upcoming events and insert them into Q

Plane sweep algorithm: intersection detection

Plane sweep (general method):

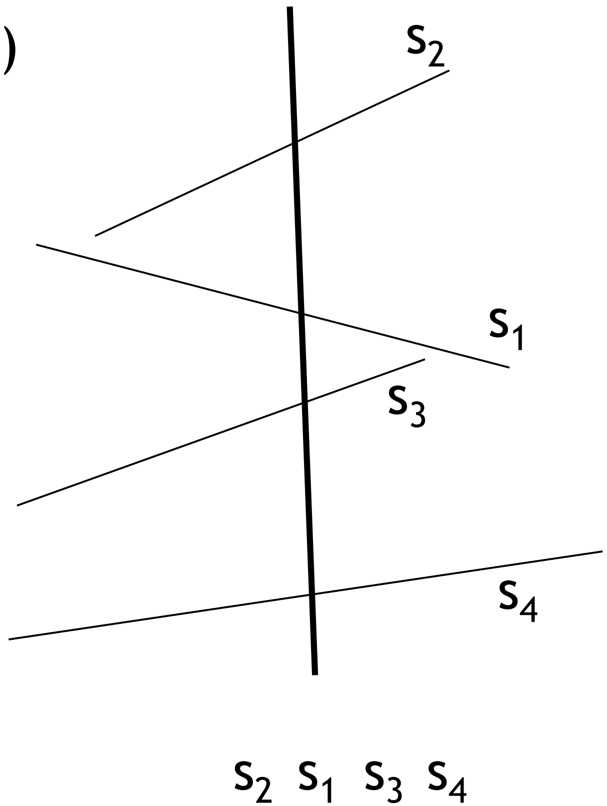
1. Sweep the input from left to right and stop at event points
2. Maintain invariant (status and "cleanliness")
3. At each event point restore invariant

Event points?

end points of the segments

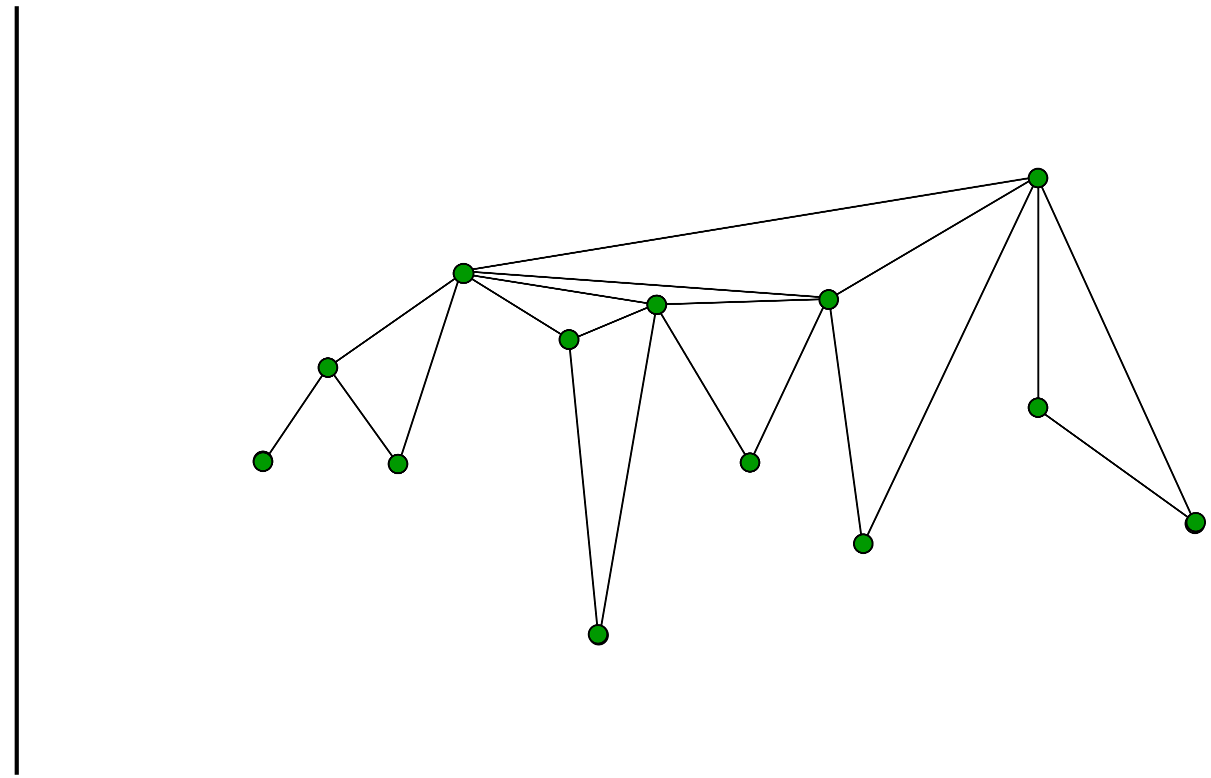
Invariant:

› The order of the segments along the sweep line
› No intersections to the left of the sweepline



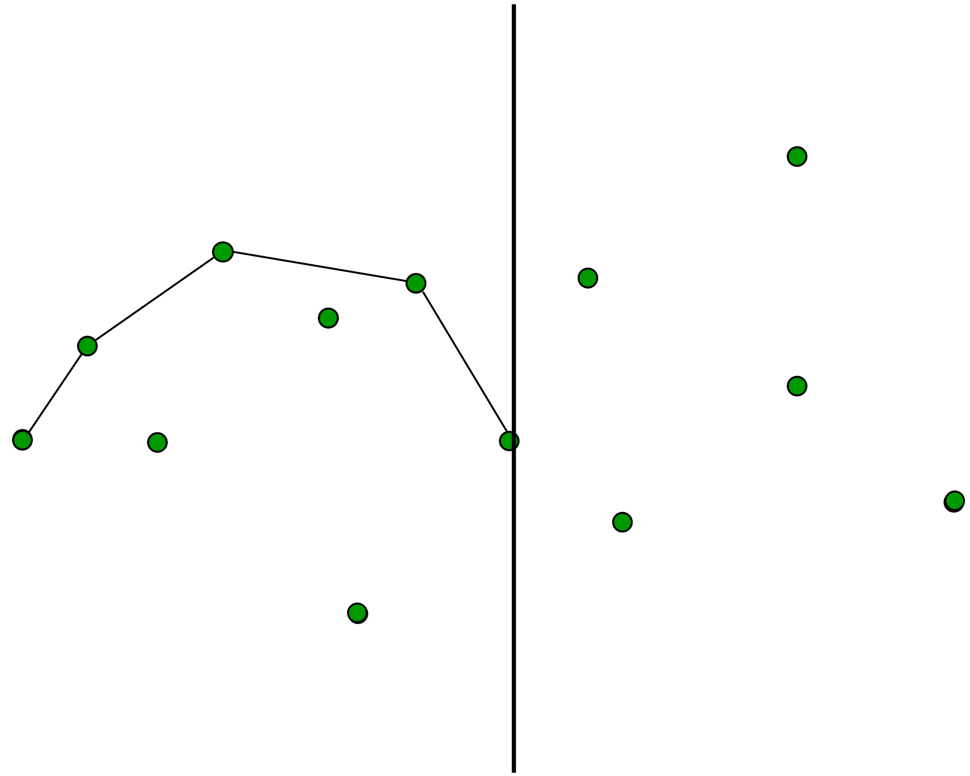
Convex hull – sweep line approach

Idea: Maintain hull while adding the points one by one, from left to right ⇔ sweep the point from left to right



Convex hull - sweep line approach

Event points: Input points
Sweep line status: The current upper hull to the left of the sweepline
Invariant: Valid hull to the left of sweepline



Summary: Sweepline

- **Sweepline.**
  - Define event points.
  - Define an invariant.
  - Prove invariant.

- **Correctness:** Usually follows immediately from invariant.

- **Problems**
  - Segment intersection
  - Convex hull
  - …

Lectures 6-7:
Dynamic Programming

THE UNIVERSITY OF SYDNEY

Dynamic programming

1. Define subproblems.

2. Write a recurrence (include base cases).

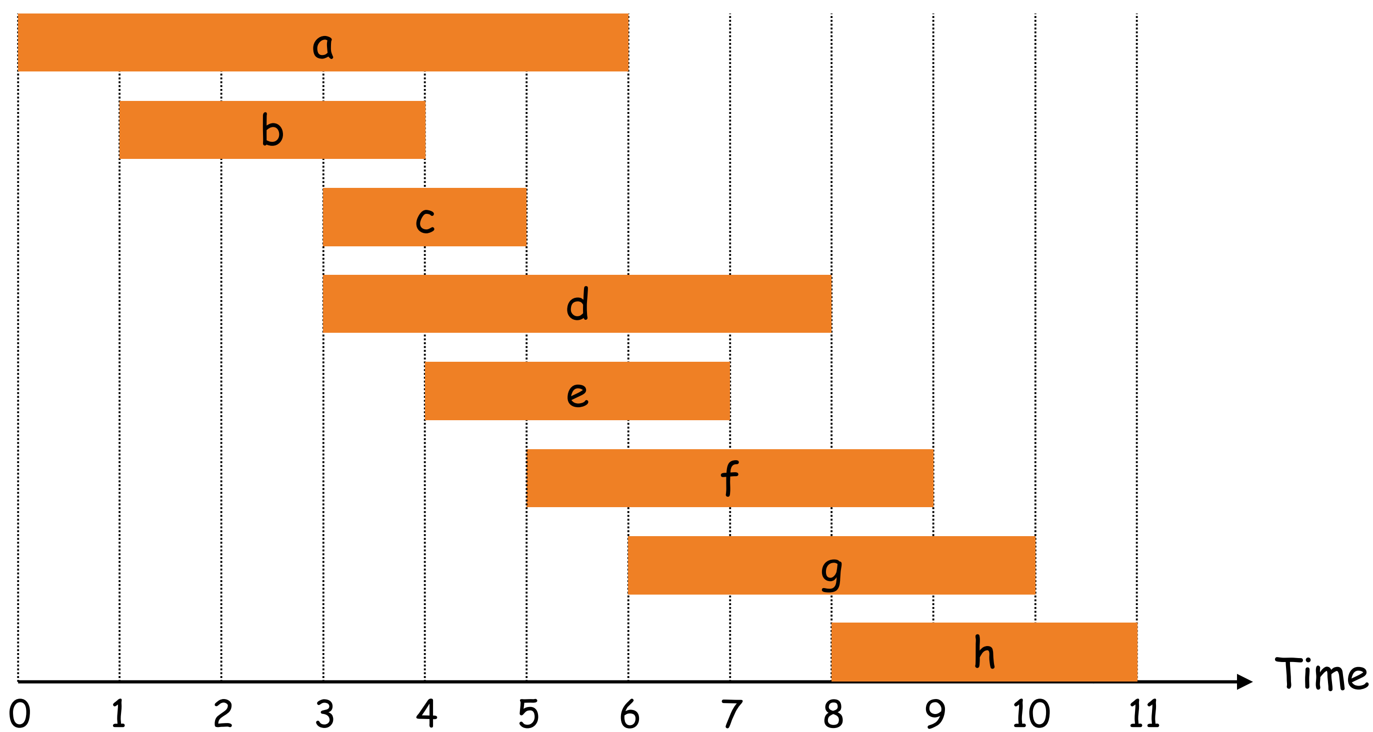
3. Prove that the recurrence is correct. Usually case-by-case. May require an induction proof, but usually easy to prove.

4. Prove the algorithm evaluates the recurrence. Values computed in correct order.

5. Prove the algorithm is correct.

Weighted Interval Scheduling

- Weighted interval scheduling problem.
  - Job j starts at s_j, finishes at f_j, and has weight or value v_j .
  - Two jobs compatible if they don't overlap.
  - Goal: find maximum weight subset of mutually compatible jobs.

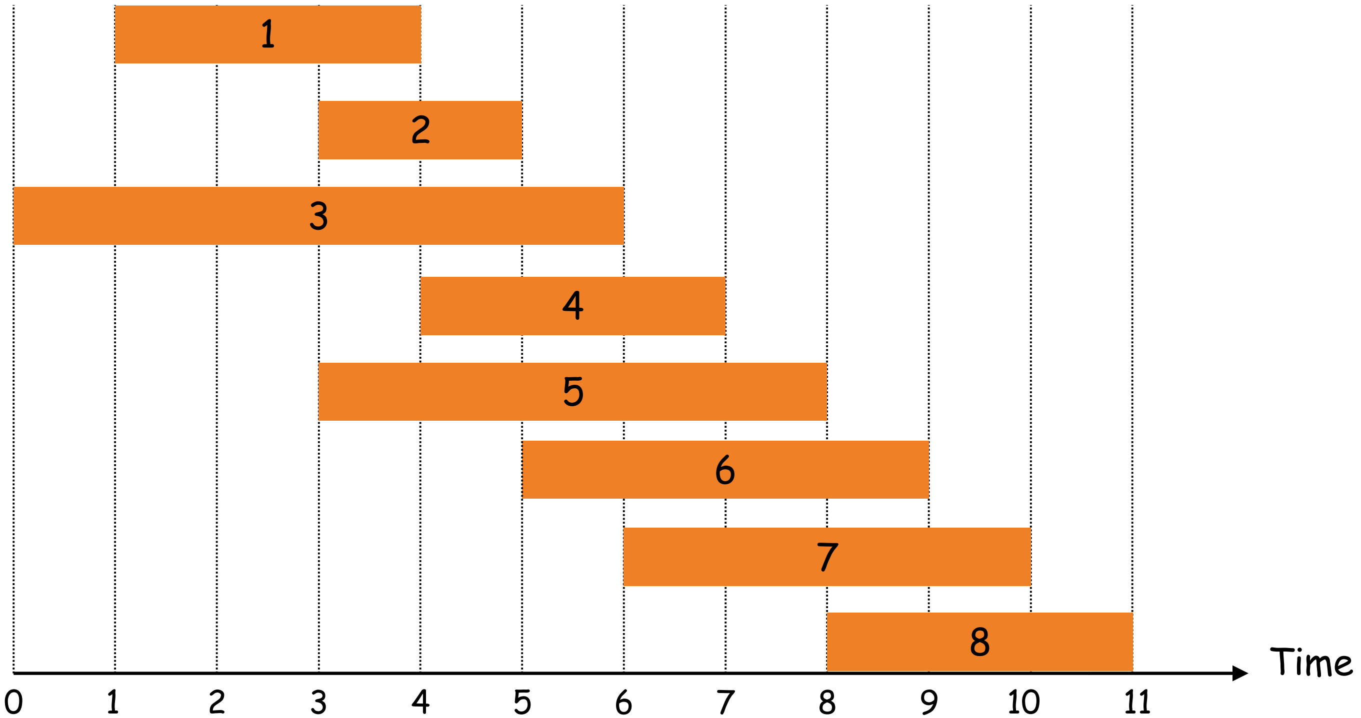


Weighted Interval Scheduling

Notation. Label jobs by finishing time: $f_1 \leq f_2 \leq \ldots \leq f_n$.

Def. $p(j)$ = largest index $i < j$ such that job i is compatible with j.

Ex: $p(8) = 5$, $p(7) = 3$, $p(2) = 0$.



Dynamic Programming – Step 1

Step 1: Define subproblems

OPT(j) = value of optimal solution to the problem consisting of job requests 1, 2, ..., j.

Dynamic Programming – Step 2

Step 2: Find recurrences

- **Case 1:** OPT selects job j.
  - can't use incompatible jobs { p(j) + 1, p(j) + 2, ..., j - 1 }
  - must include optimal solution to problem consisting of remaining compatible jobs 1, 2, ..., p(j)

- **Case 2:** OPT does not select job j.
  - must include optimal solution to problem consisting of remaining compatible jobs 1, 2, ..., j-1

$$OPT(j) = \max \{ \underbrace{v_j + OPT(p(j))}_{\text{Case 1}}, \underbrace{OPT(j-1)}_{\text{Case 2}} \}$$

Dynamic Programming – Step 3

Step 3: Solve the base cases

$$OPT(0) = 0$$

$$OPT(j) = \begin{cases} 0 & \text{if } j = 0 \\ \max \{ v_j + OPT(p(j)),\ OPT(j-1) \} & \text{otherwise} \end{cases}$$

Done…more or less

Knapsack Problem

- **Knapsack problem.**
  - Given n objects and a "knapsack."
  - Item i weighs $w_i > 0$ kilograms and has value $v_i > 0$.
  - Knapsack has capacity of W kilograms.
  - **Goal:** fill knapsack so as to maximize total value.

W = 11

Item	Value	Weight
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

Dynamic Programming – Step 1

Step 1: Define subproblems

OPT(i, w) = max profit with subset of items 1, …, i with weight limit w.

Dynamic Programming – Step 2

Step 2: Find recurrences

- **Case 1:** OPT does not select item i.
  - OPT selects best of { 1, 2, …, i-1 } using weight limit w

- **Case 2:** OPT selects item i.
  - new weight limit = $w - w_i$
  - OPT selects best of { 1, 2, …, i–1 } using this new weight limit

If $w_i > w$

$$OPT(i,w) = OPT(i-1,w)$$

otherwise

$$OPT(i,w) = \max \{ v_i + OPT(i-1,w-w_i),\ OPT(i-1,w) \}$$

Recap: Dynamic Programming – Step 3

Step 3: Solve the base cases

$$OPT(0,w) = 0$$

$$OPT(i,w) = \begin{cases} 0 & \text{if } i = 0 \\ OPT(i-1,w) & \text{if } i > 0 \text{ and } w_i > w \\ \max\{OPT(i-1,w), v_i + OPT(i-1, w-w_i)\} & \text{otherwise} \end{cases}$$

Done…more or less

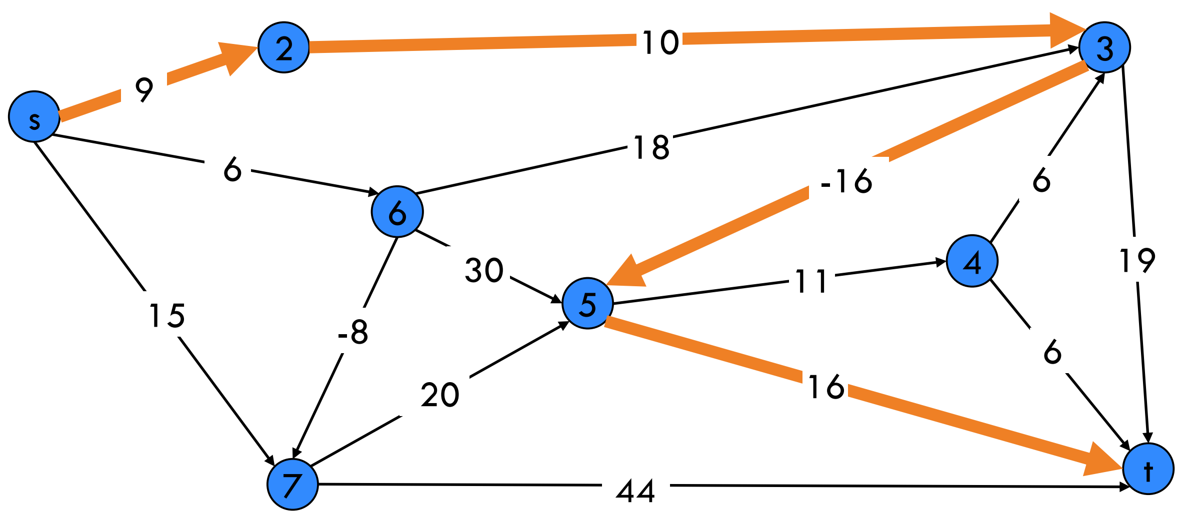
Knapsack Problem: Running Time

- **Running time:** $\Theta(nW)$.
  - Not polynomial in input size!
  - "Pseudo-polynomial."
  - Decision version of Knapsack is NP-complete.

Shortest Paths

– **Shortest path problem.** Given a directed graph $G = (V, E)$, with edge weights c_{vw}, find shortest path from node s to node t.

allow negative weights



Shortest Paths: Dynamic Programming

Step 1: OPT(i, v) = length of shortest v-t path P using at most i edges.

Step 2:

Case 1: P uses at most i-1 edges.

- OPT(i, v) = OPT(i-1, v)

Case 2: P uses exactly i edges.

- if (v, w) is first edge, then OPT uses (v, w), and then selects best w-t path using at most i-1 edges

Step 3: OPT(0,t) = 0 and OPT(0,v≠t) = ∞

$$OPT(i,v) = \begin{cases} 0 & \text{if } i=0 \text{ and } v=t \\ \infty & \text{if } i=0 \text{ and } v \neq t \\ \min\{OPT(i-1,v),\ \min\limits_{(v,w)\in E} [OPT(i-1,w)+c_{vw}]\} & \text{otherwise} \end{cases}$$

Dynamic Programming Summary I

Algorithm - 3 steps:

1. Defining subproblems
2. Finding recurrences
3. Solving the base cases

Dynamic Programming Summary II

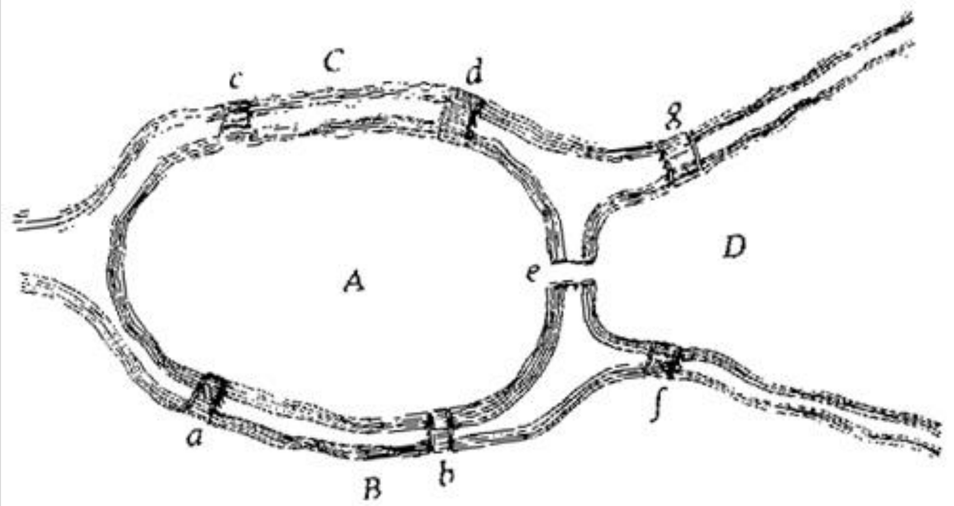
- **1D dynamic programming**
  - Weighted interval scheduling
  - Segmented Least Squares
  - Maximum-sum contiguous subarray
  - Longest increasing subsequence

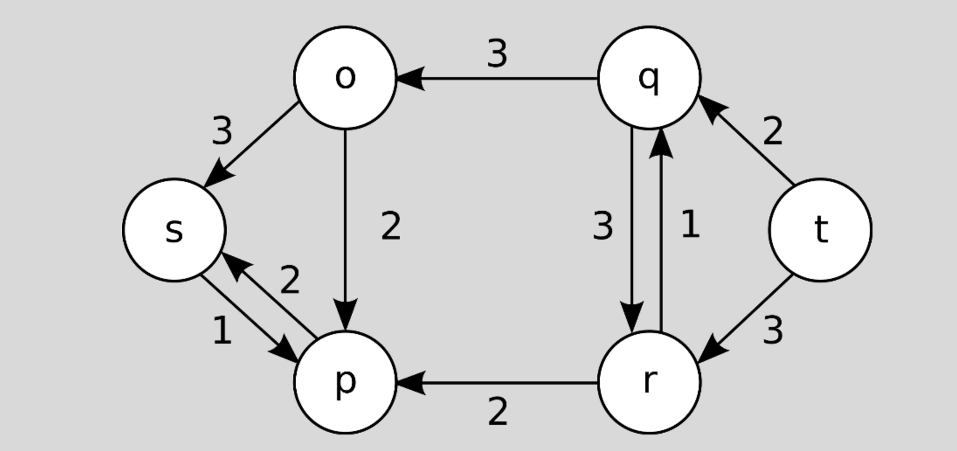
- **2D dynamic programming**
  - Knapsack
  - Shortest path

- **Dynamic programming over intervals**
  - RNA Secondary Structure

Lectures 8-9:
Flow networks

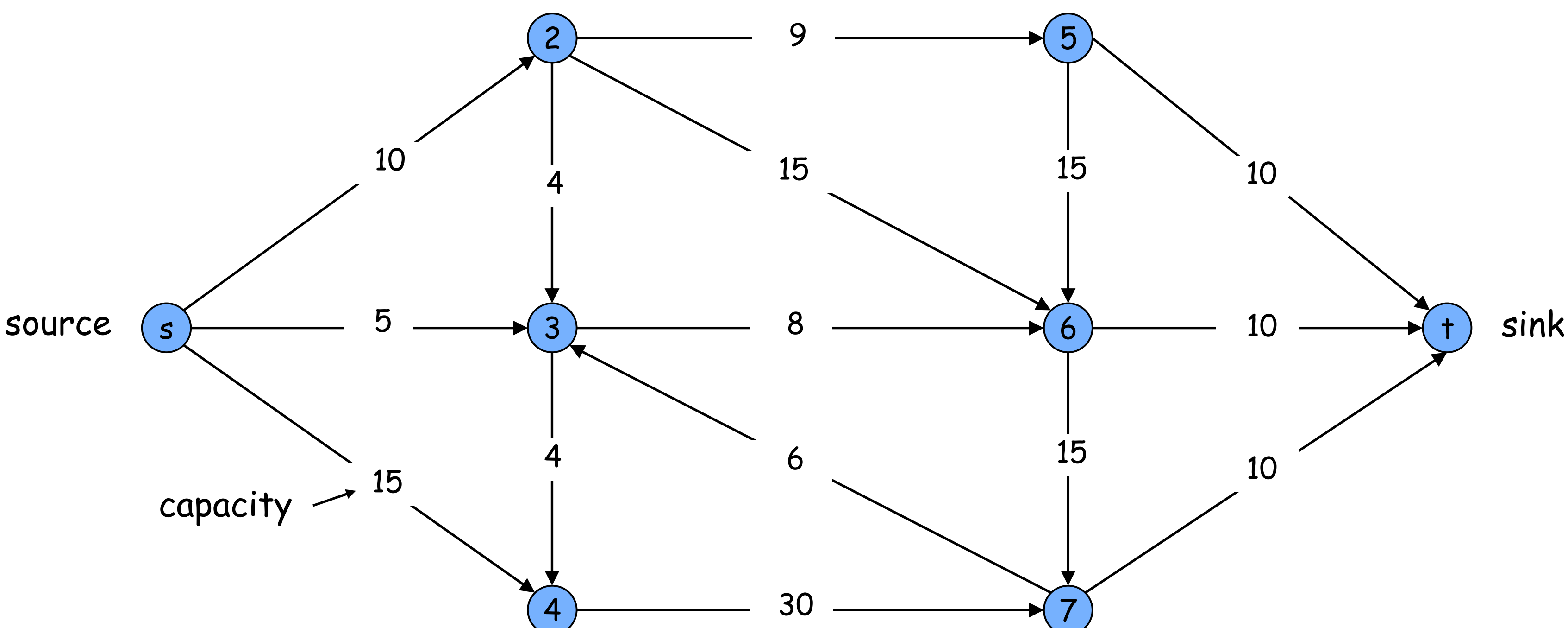
THE UNIVERSITY OF SYDNEY





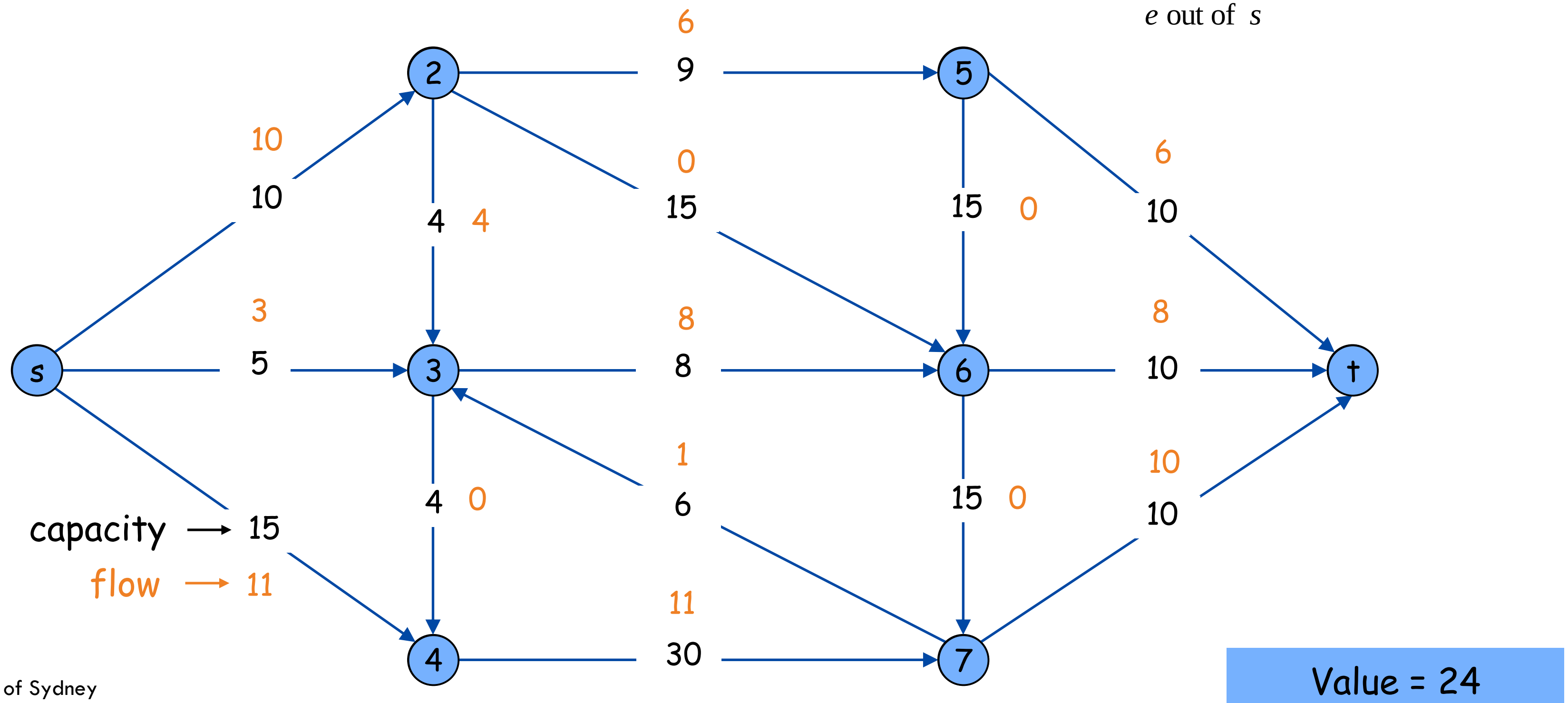
Flow network

- Abstraction for material flowing through the edges.
- $G = (V, E)$ = directed graph, no parallel edges.
- Two distinguished nodes: s = source, t = sink.
- $c(e)$ = capacity of edge e.



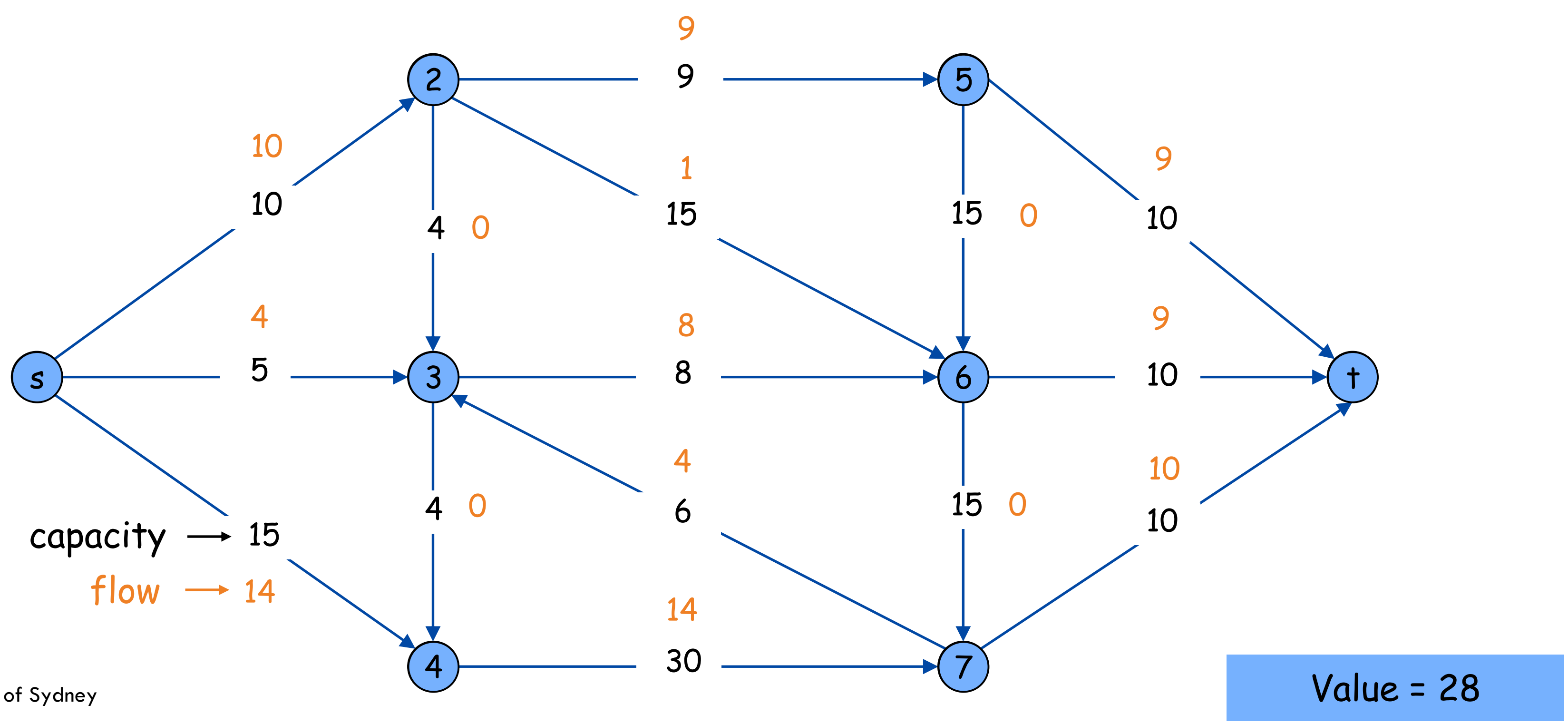
Flows

- **Definition:** An s-t flow is a function that satisfies:
  - For each e ∈ E: $0 \le f(e) \le c(e)$ (capacity)
  - For each v ∈ V – {s, t}: $\sum_{e \text{ in to } v} f(e) = \sum_{e \text{ out of } v} f(e)$ (conservation)
- **Definition:** The value of a flow f is: $v(f) = \sum_{e \text{ out of } s} f(e)$.



Maximum Flow Problem

– Max flow problem. Find s-t flow of maximum value.

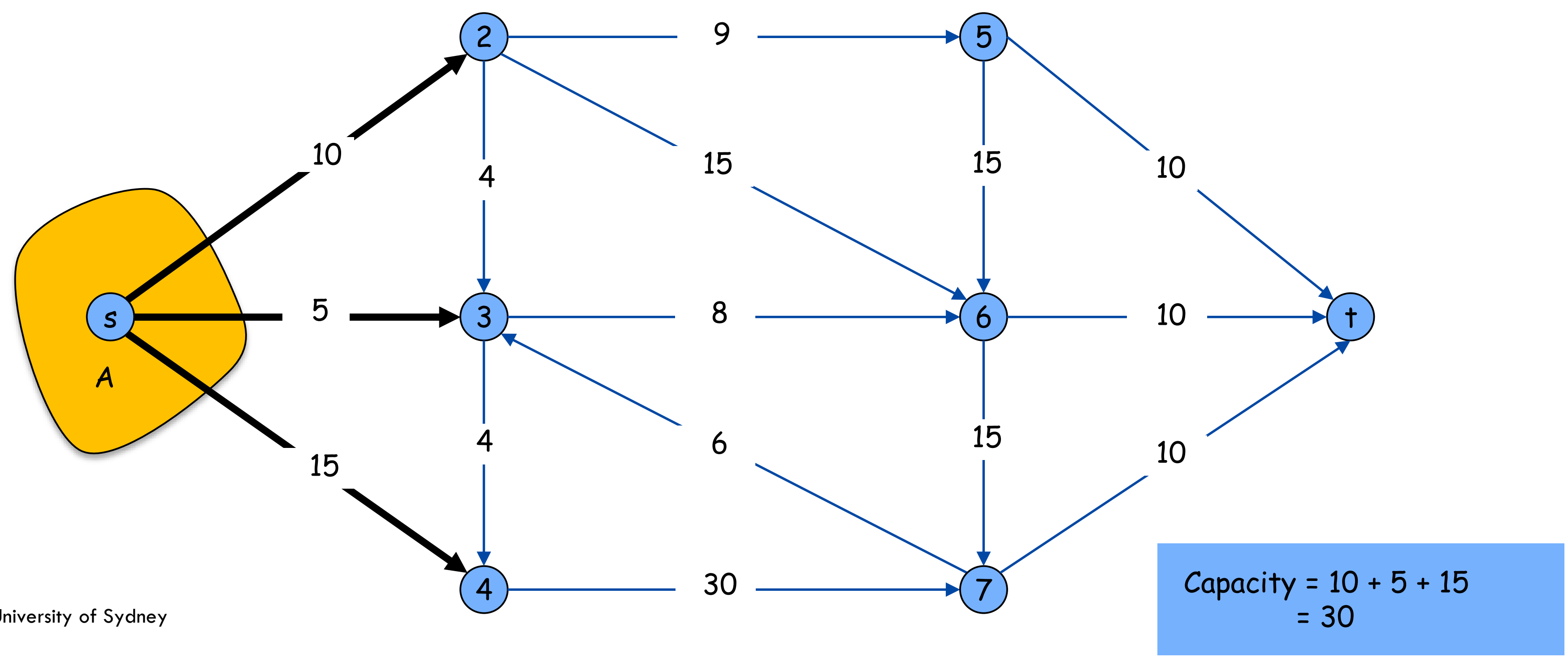


Cuts

Definitions:

– An s-t cut is a partition (A, B) of V with s ∈ A and t ∈ B.

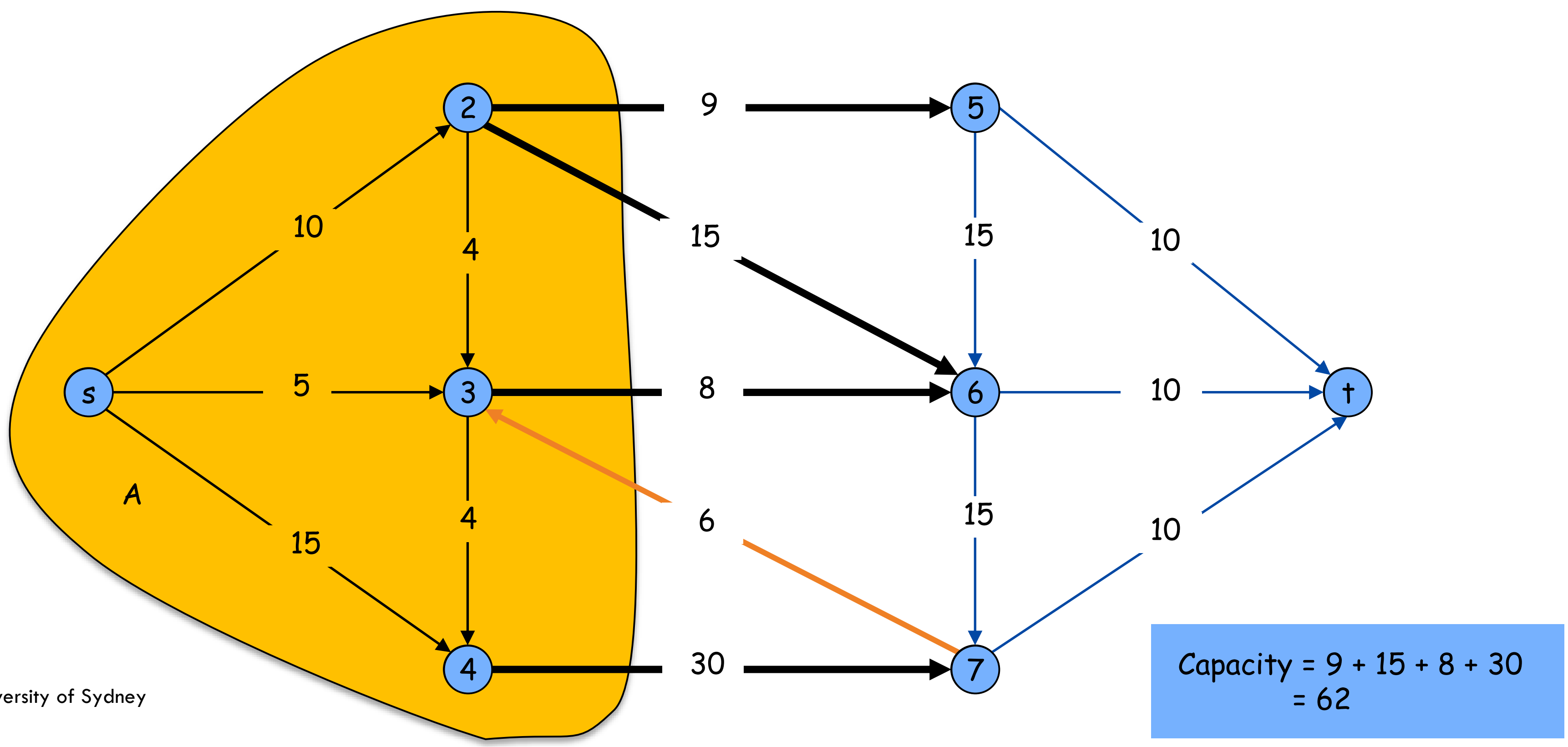
– The capacity of a cut (A, B) is: $cap(A, B) = \sum_{e \text{ out of } A} c(e)$



Cuts

Definitions:

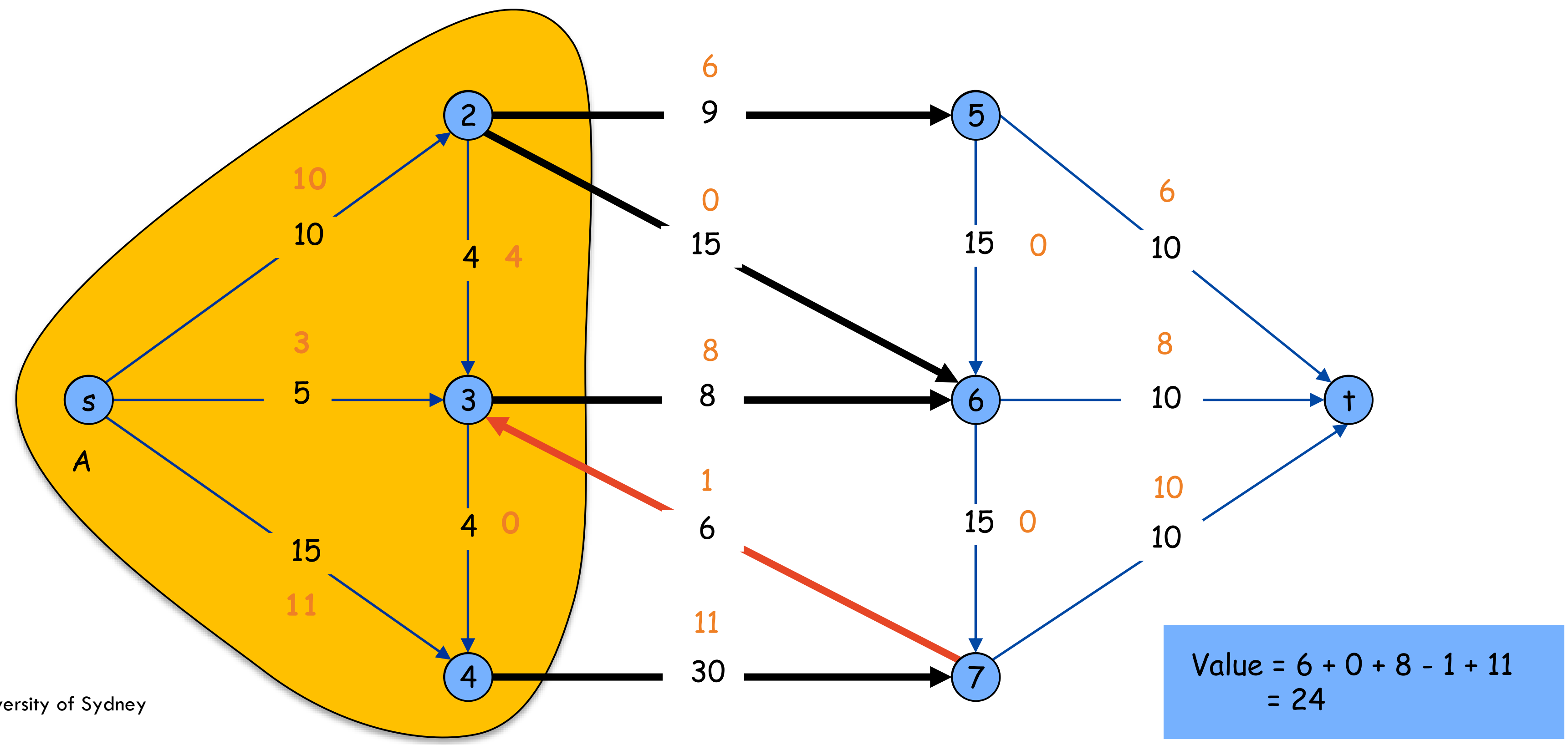
- An s-t cut is a partition (A, B) of V with s ∈ A and t ∈ B.
- The capacity of a cut (A, B) is: $cap(A, B) = \sum_{e \text{ out of } A} c(e)$



Flows and Cuts

– Flow value lemma. Let f be any flow, and let (A, B) be any s-t cut. Then, the net flow sent across the cut is equal to the amount leaving s.

$$\sum_{e \text{ out of } A} f(e) \;-\; \sum_{e \text{ in to } A} f(e) \;=\; v(f)$$



Ford-Fulkerson

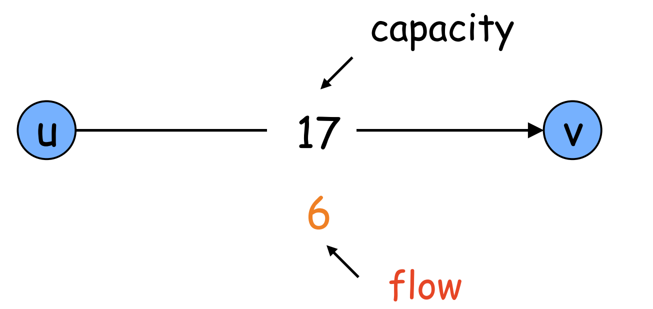
```
Ford-Fulkerson(G,s,t) {
   foreach e ∈ E
        f(e) ← 0
   Gf ← residual graph

   while (there exists augmenting path P in Gf){
      f ← Augment(f,P)
      update Gf
   }
   return f
}
```

```
Augment(f,P) {
   b ← bottleneck(P,f)
   foreach e =(u,v) ∈ P {
      if e is a forward edge then
         increase f(e) in G by b
      else (e is a backward edge)
         decrease f(e) in G by b
   }
   return f
}
```

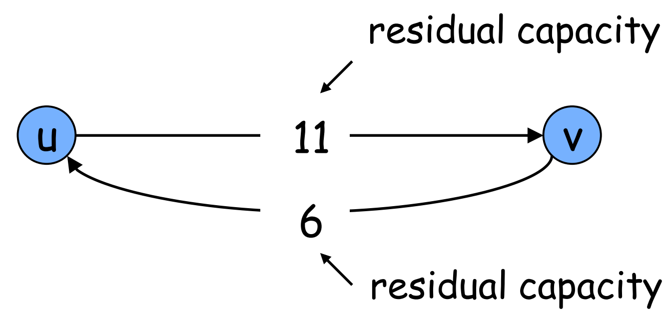
Residual Graph

- Original edge: $e = (u, v) \in E$.
  - Flow $f(e)$, capacity $c(e)$.



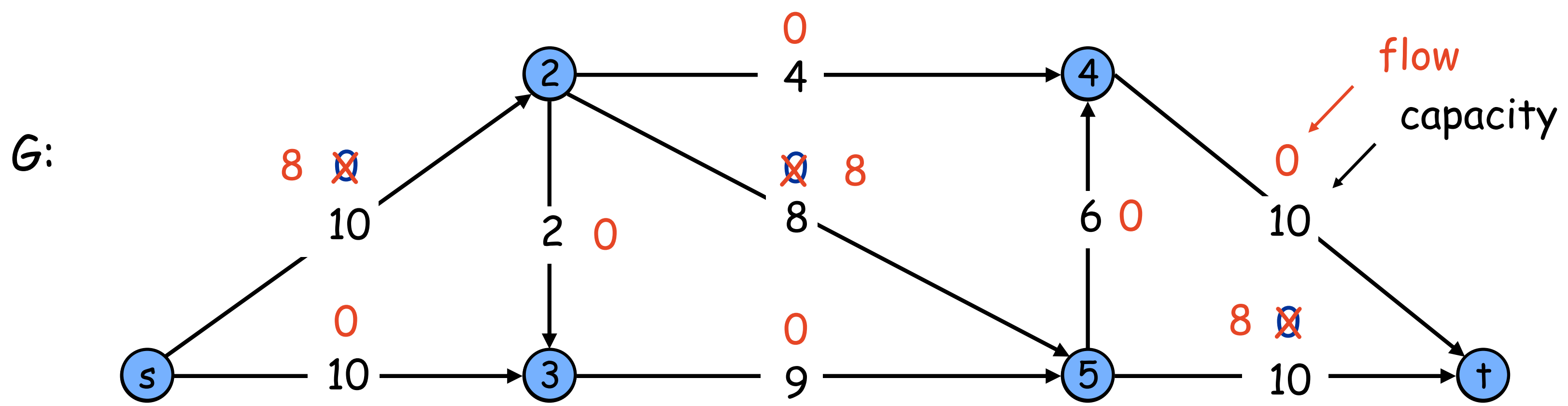
- Residual edge.
  - "Undo" flow sent.
  - $e = (u, v)$ and $e^R = (v, u)$.
  - Residual capacity:

$$c_f(e) = \begin{cases} c(e) - f(e) & \text{if } e \in E \\ f(e) & \text{if } e^R \in E \end{cases}$$

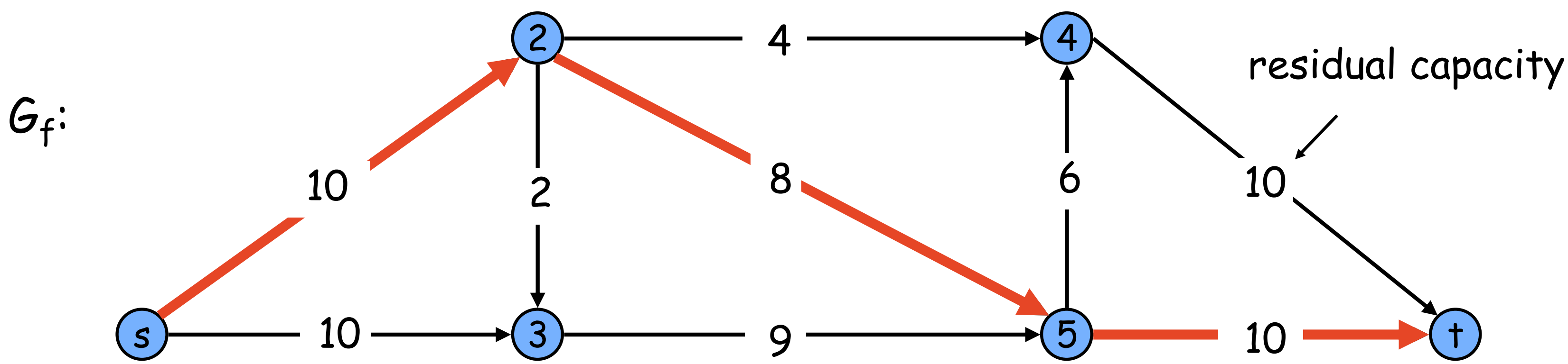


- Residual graph: $G_f = (V, E_f)$.
  - Residual edges with positive residual capacity.
  - $E_f = \{e : f(e) < c(e)\} \cup \{e^R : c(e) > 0\}$.

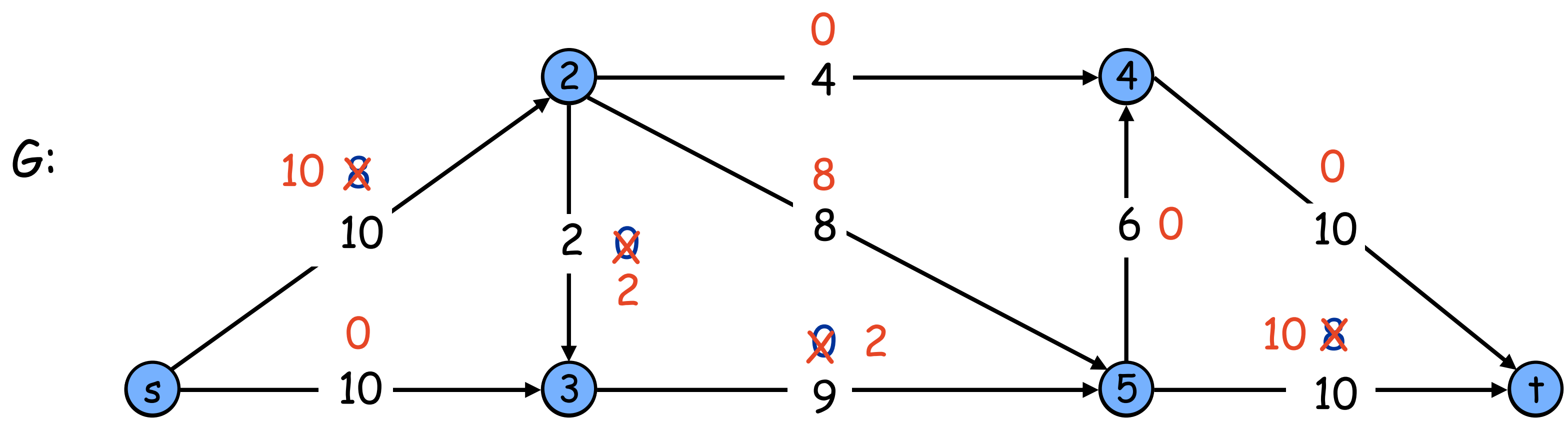
Ford-Fulkerson Algorithm



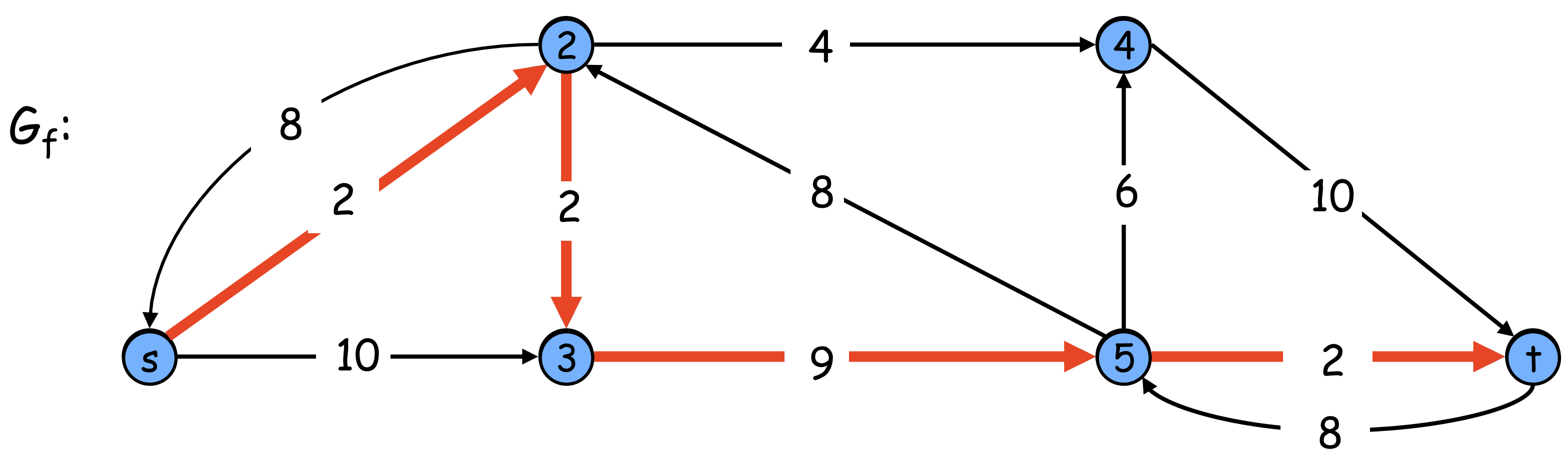
Flow value = 0



Ford-Fulkerson Algorithm



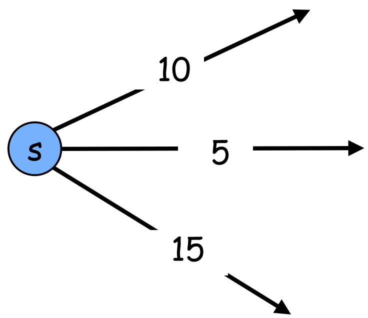
Flow value = 8



Max-Flow Min-Cut Theorem

- Augmenting path theorem: Flow f is a max flow if and only if there are no augmenting paths in the residual graph.
- Max-flow min-cut theorem: The value of the max flow is equal to the value of the min cut. [Ford-Fulkerson 1956]
- Integrality. If all capacities are integers then every flow value f(e) and every residual capacities c_f (e) remains an integer throughout the algorithm.

Running Time



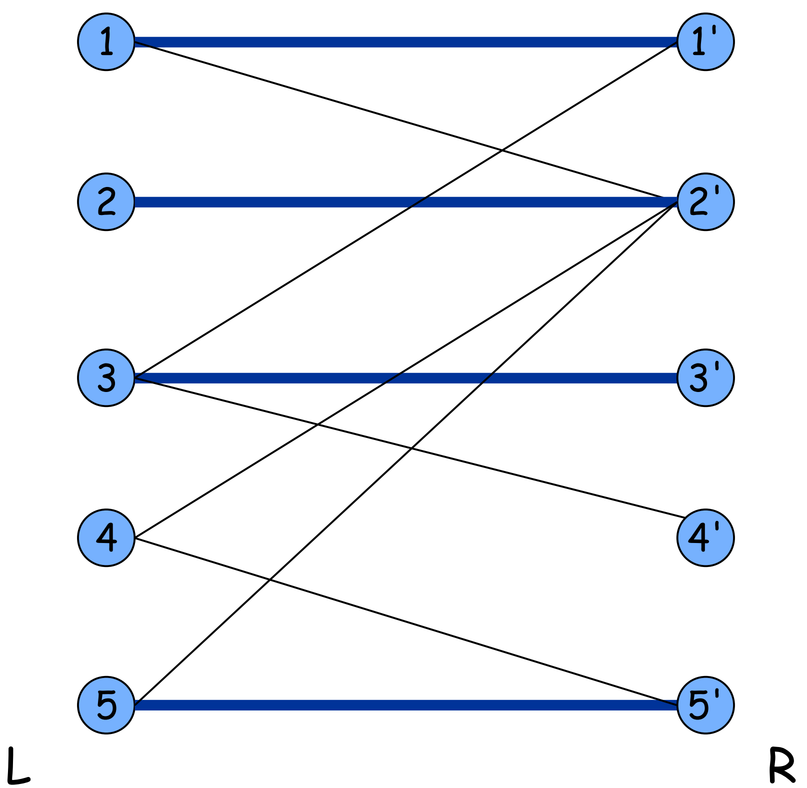
Notation: $C = \sum_{e \text{ out of } s} c(e)$

Observation: C is an upper bound on the maximum flow.

Theorem: Ford-Fulkerson runs in O(Cm) time.

Bipartite Matching

– Input: undirected, bipartite graph $G = (L \cup R, E)$.
– $M \subseteq E$ is a matching if each node appears in at most one edge in M.
– Max matching: find a max cardinality matching.



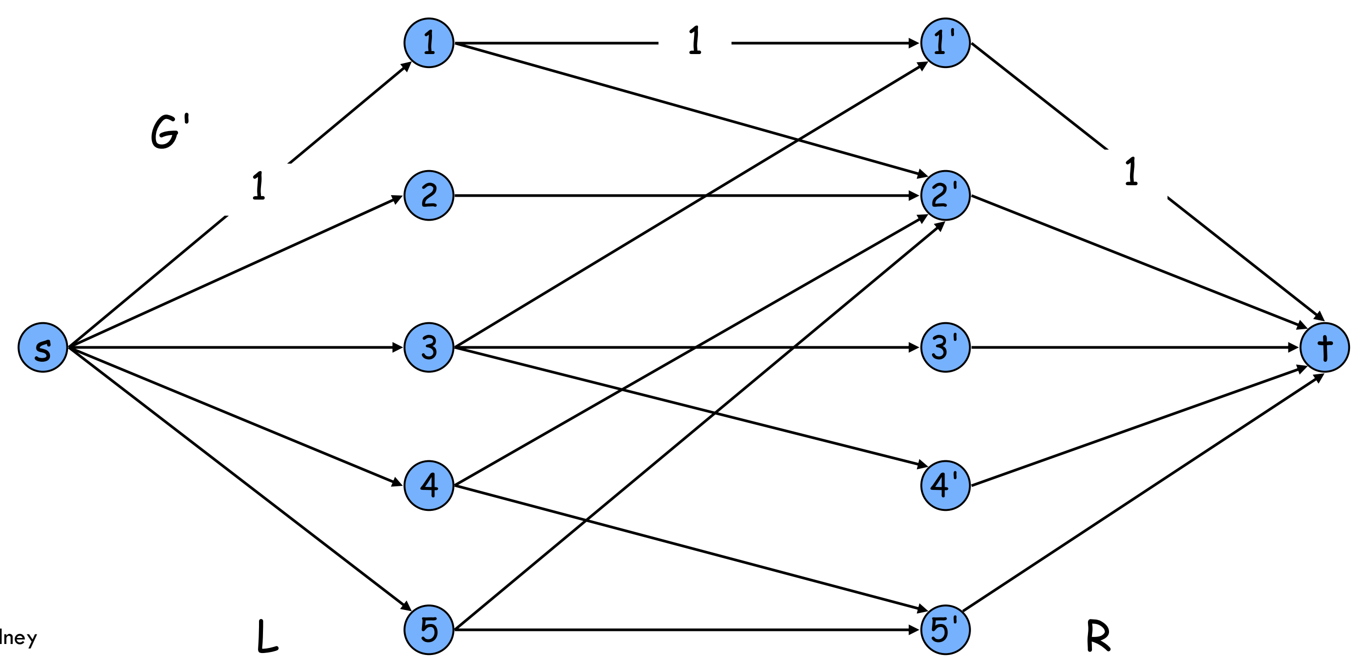
max matching

1-1', 2-2', 3-3', 5-5'

Bipartite Matching

Max flow formulation.

- Create digraph $G' = (L \cup R \cup \{s, t\}, \ E')$.
- Direct all edges from L to R, and assign unit capacity.
- Add source s, and unit capacity edges from s to each node in L.
- Add sink t, and unit capacity edges from each node in R to t.

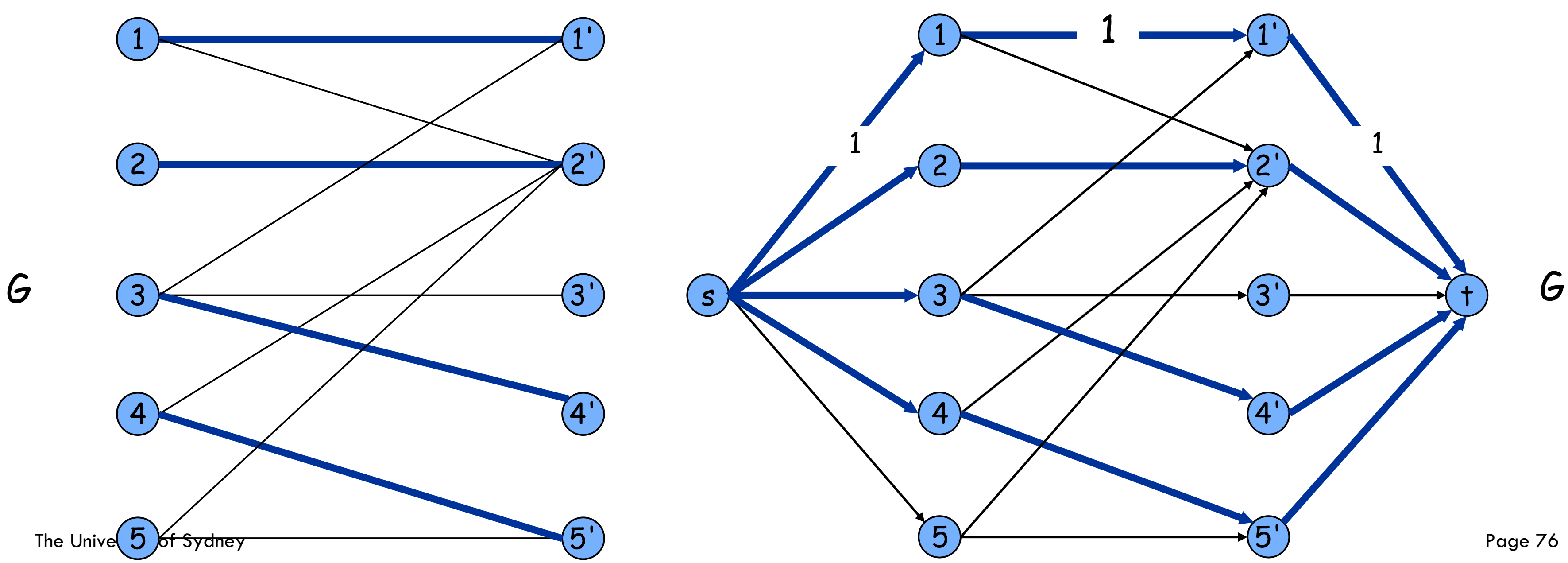


Bipartite Matching: Proof of Correctness

Theorem: Max cardinality matching in G $\Leftrightarrow$ value of max flow in G'.

Proof: $\Rightarrow$

- Assume max matching M has cardinality k.
- Consider a flow f that sends 1 unit along each of the k paths, defined by the edges in M.
- f is a flow, and it has value k. ■

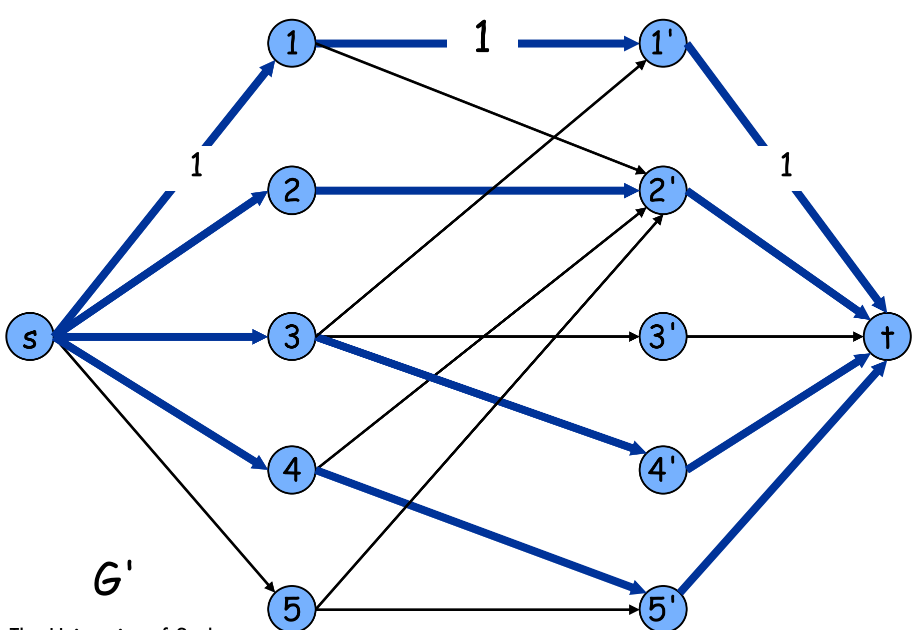


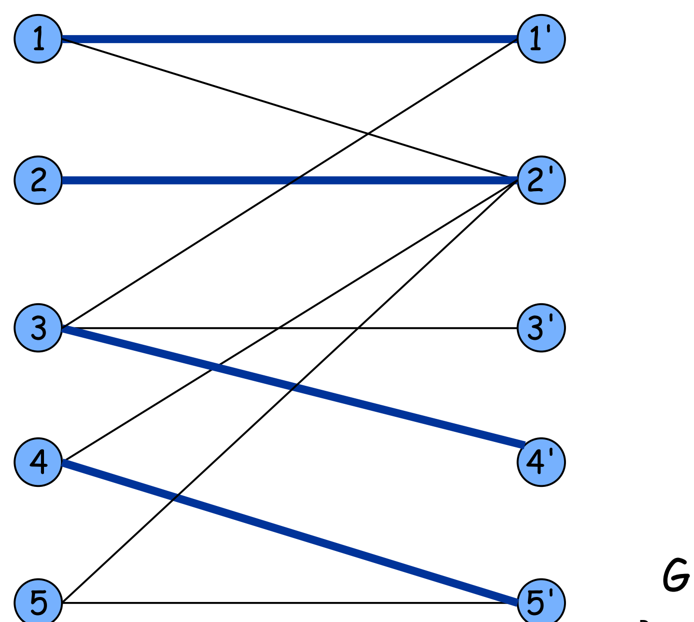
Bipartite Matching: Proof of Correctness

Theorem: Max cardinality matching in G $\Leftrightarrow$ value of max flow in G'.

Proof: $\Leftarrow$

- Let f be a max flow in G' of value k.
- Integrality theorem $\Rightarrow$ k is integral so f(e) is 0 or 1.
- Consider M = set of edges from L to R with f(e) = 1.
  - each node in L and R participates in at most one edge in M
  - $|M| = k$: consider cut $(L \cup s, R \cup t)$



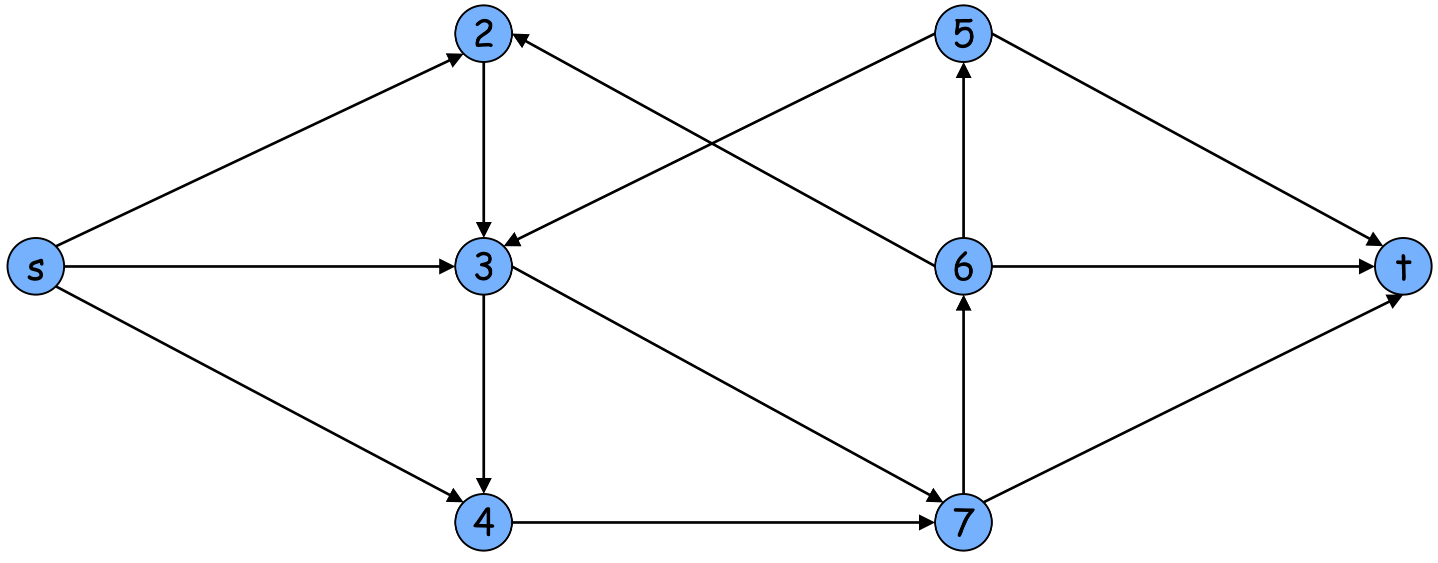


Edge Disjoint Paths

Disjoint path problem:

Given a digraph G = (V, E) and two nodes s and t, find the max number of edge-disjoint s-t paths.

Definition: Two paths are edge-disjoint if they have no edge in common.

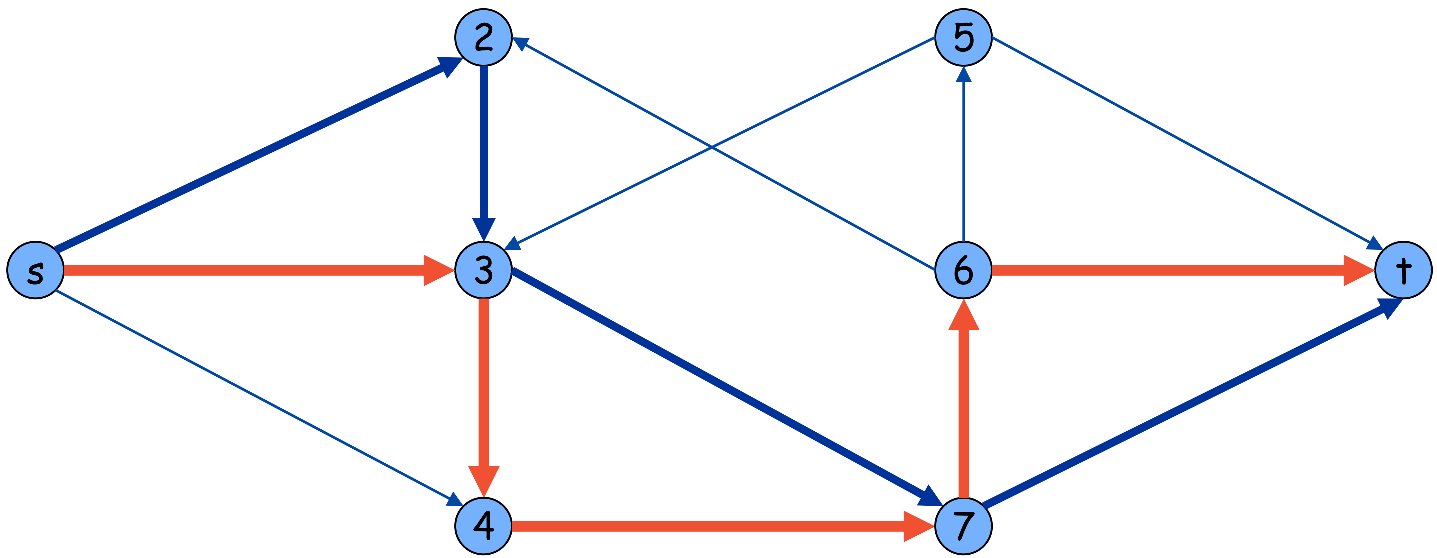


Edge Disjoint Paths

Disjoint path problem:

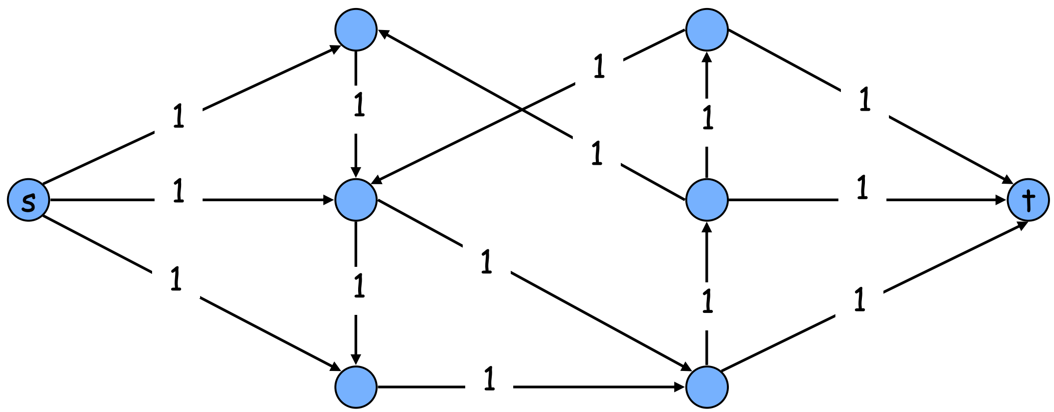
Given a digraph G = (V, E) and two nodes s and t, find the max number of edge-disjoint s-t paths.

Definition: Two paths are edge-disjoint if they have no edge in common.



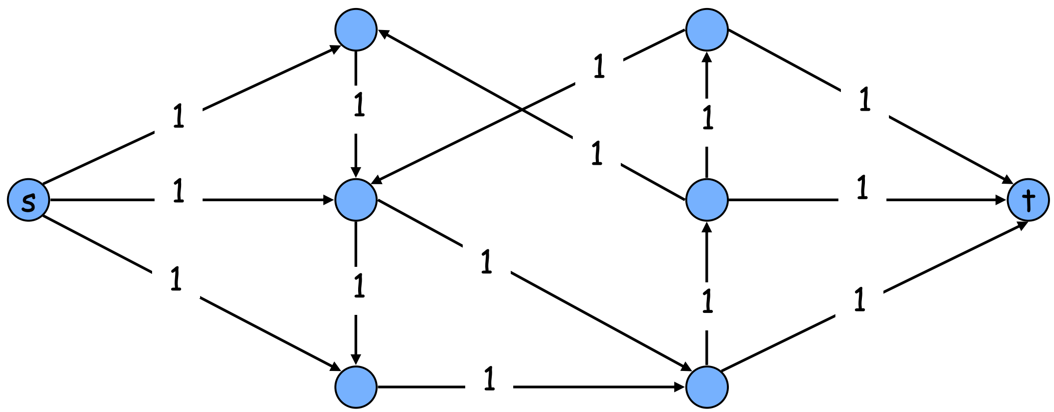
Edge Disjoint Paths

Max flow formulation: assign unit capacity to every edge.



Edge Disjoint Paths

Max flow formulation: assign unit capacity to every edge.



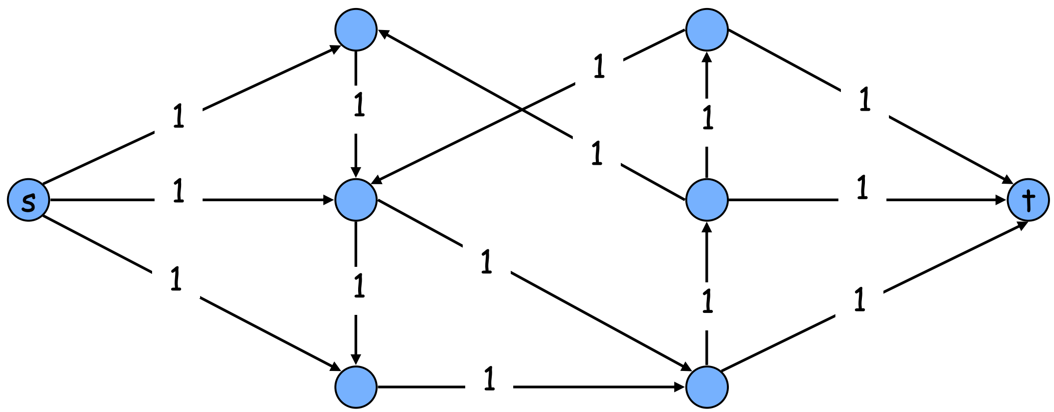
Theorem: Max number edge-disjoint s-t paths equals max flow value.

Proof: $\Rightarrow$

- Suppose there are k edge-disjoint paths $P_1, \ldots, P_k$.
- Set $f(e) = 1$ if e participates in some path P_i ; else set $f(e) = 0$.
- Since paths are edge-disjoint, f is a flow of value k.

Edge Disjoint Paths

Max flow formulation: assign unit capacity to every edge.



Theorem: Max number edge-disjoint s-t paths equals max flow value.

Proof: $\Leftarrow$

- Suppose max flow value is k.
- Integrality theorem $\Rightarrow$ there exists 0-1 flow f of value k.
- Consider edge (s, u) with f(s, u) = 1.
  - by conservation, there exists an edge (u, v) with f(u, v) = 1
  - continue until reach t, always choosing a new edge
- Produces k (not necessarily simple) edge-disjoint paths. ▪

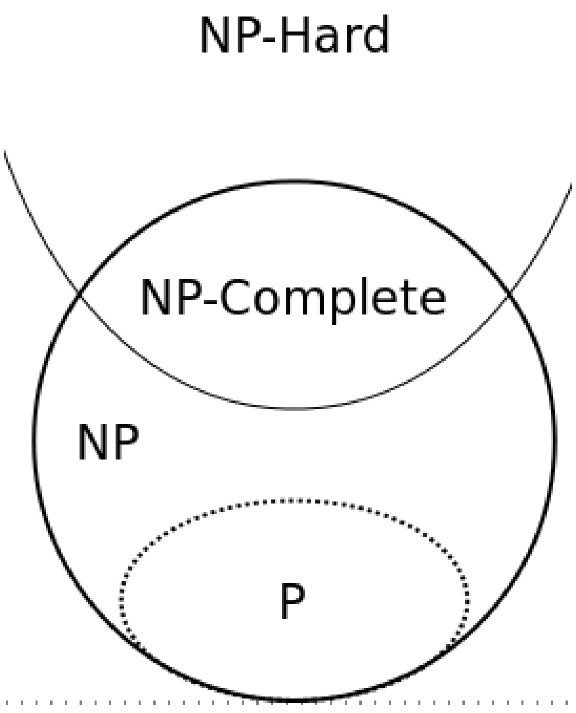
Summary: Flow networks

- **Properties**
  - Max-flow min-cut theorem
  - Integrality lemma
  - …

- **Ford-Fulkerson's algorithm**

- **Problems**
  - Max flow
  - Min cut
  - Matching
  - Disjoint edge paths
  - …

Lecture 10: NP and Computational Intractability



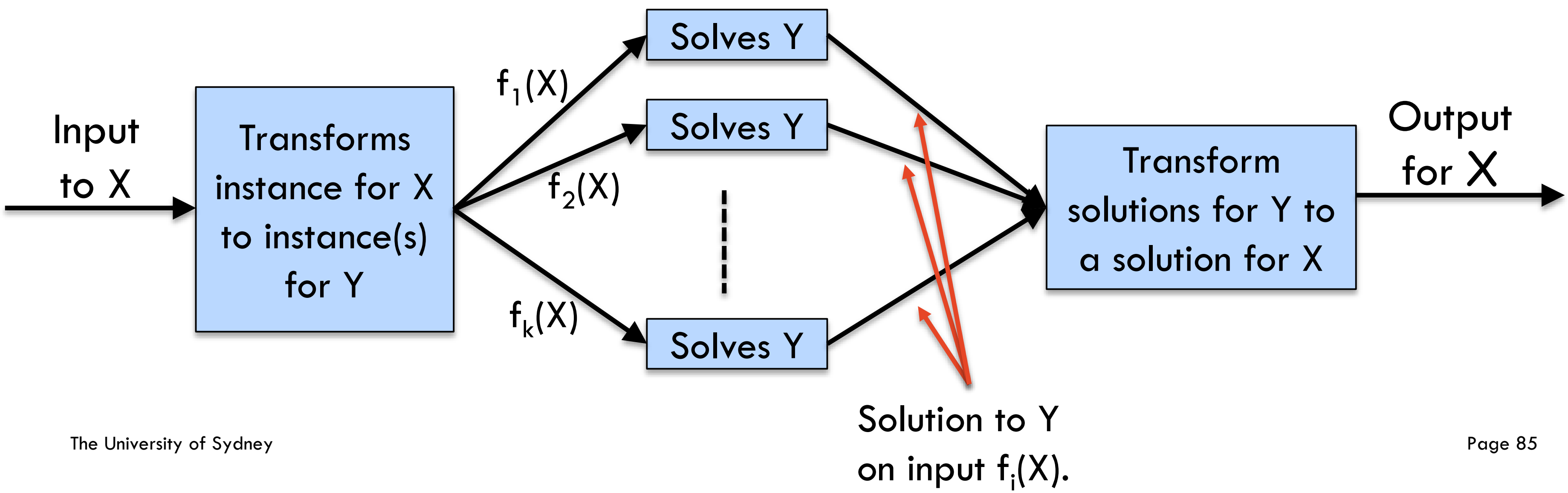
THE UNIVERSITY OF SYDNEY

Polynomial-Time Reduction

Suppose we could solve problem Y in polynomial-time. What else could we solve in polynomial time?

Reduction. Problem X polynomial reduces to problem Y, denoted $X \leq_P Y$, if arbitrary instances of problem X can be solved using:

– Polynomial number of standard computational steps, plus
– Polynomial number of calls to an oracle that solves problem Y.



Polynomial-Time Reduction

Purpose. Classify problems according to relative difficulty.

1. Design algorithms. If $X \leq_P Y$ and Y can be solved in polynomial-time, then X can also be solved in polynomial time.
2. Establish intractability. If $X \leq_P Y$ and X cannot be solved in polynomial-time, then Y cannot be solved in polynomial time.

Reduction By Simple Equivalence

Basic reduction strategies.

- Reduction by simple equivalence.
- Reduction from special case to general case.
- Reduction by encoding with gadgets.

Summary

- Polynomial time reductions
  3-SAT $\leq_P$ DIR HAMILTONIAN CYCLE $\leq_P$ HAMILTONIAN CYCLE $\leq_P$ TSP
  3-SAT $\leq_P$ INDEPENDENT-SET $\leq_P$ VERTEX-COVER $\leq_P$ SET-COVER

- Complexity classes:
  P: Decision problems for which there is a poly-time algorithm.
  NP: Decision problems for which there is a poly-time certifier.
  NP-complete: A problem in NP such that every problem in NP polynomial reduces to it.
  NP-hard: A problem such that every problem in NP polynomial reduces to it.

- Lots of problems are NP-complete
  See https://www.nada.kth.se/~viggo/wwwcompendium/

Lecture 11:
Coping with hardness

THE UNIVERSITY OF
SYDNEY

Coping With NP-Completeness

Question: What should I do if I need to solve an NP-complete problem?

Answer: Theory says you're unlikely to find poly-time algorithm.

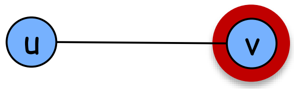
Must sacrifice one of three desired features.

- Solve problem to optimality.
  - Approximation algorithms
  - Randomized algorithms
- Solve problem in polynomial time.
  - Exact exponential time algorithms
- Solve arbitrary instances of the problem.
  - Solve restricted classes of instances
  - Parametrized algorithms

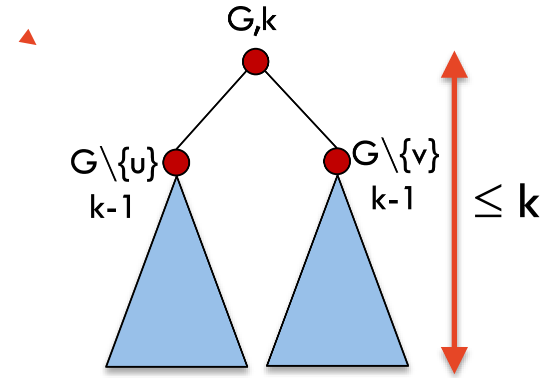
Finding Small Vertex Covers: Algorithm

Theorem: Vertex cover can be solved in $O(2^k \, kn)$ time.

This is fine as long as k is (a small) constant.



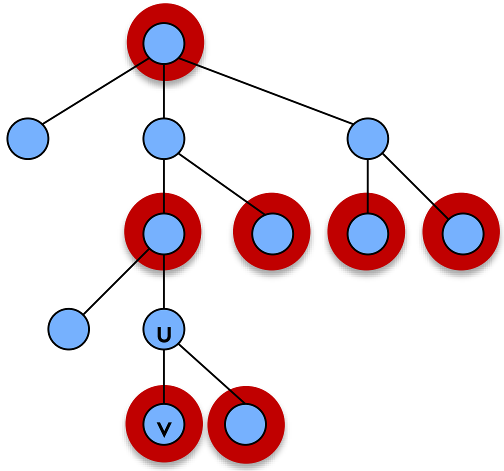
What if k is not a small constant?



Independent Set on Trees

INDEPENDENT-SET: Given a graph $G = (V, E)$ and an integer k, is there a subset of vertices $S \subseteq V$ such that $|S| \geq k$, and for each edge at most one of its endpoints is in S?

Problem: Given a tree, find a maximum IS.



Theorem:

INDEPENDENT-SET on trees can be solved in $O(n)$ time.

Approximation algorithms: Load Balancing

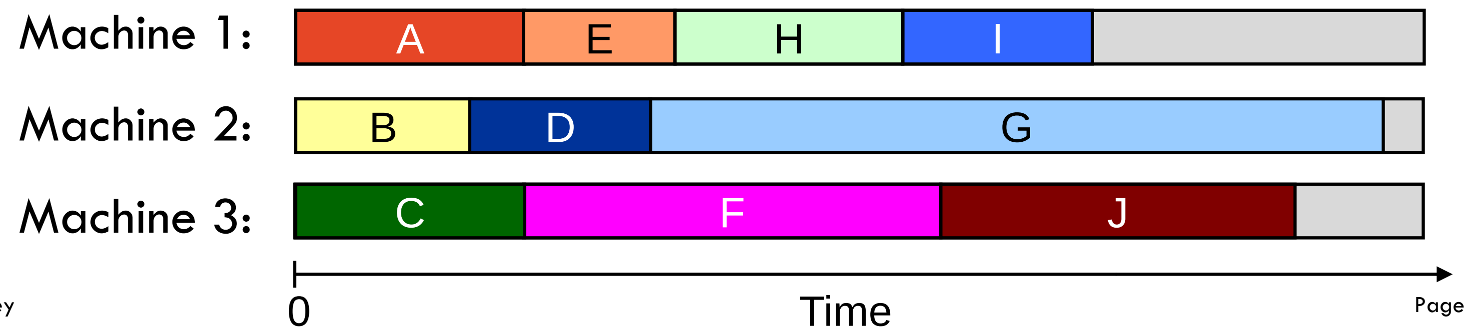
Input: m identical machines; n jobs, job j has processing time t_j.

- Job j must run contiguously on one machine.
- A machine can process at most one job at a time.

Definition: Let J(i) be the subset of jobs assigned to machine i. The load of machine i is $L_i = \Sigma_{j \in J(i)} t_j$.
Example: J(1) = {A,E,H,I}, J(2) = {B,D,G}, J(3)={C,F,J}

Definition: The makespan is the maximum load on any machine $L = \max_i L_i$.

Load balancing: Assign each job to a machine to minimize makespan.



How far off can the schedule be from optimal?

Is there an approximation guarantee?

$$\text{Approximation ratio} = \frac{\text{Cost of apx solution}}{\text{Cost of optimal solutions}}$$

An approximation algorithm for a minimization problem requires an approximation guarantee:

- Approximation ratio $\leq$ c
- Approximation solution $\leq$ c $\cdot$ value of optimal solution

Load Balancing: List Scheduling Analysis

Theorem: [Graham, 1966]

Greedy algorithm is a 2-approximation.

- First worst-case analysis of an approximation algorithm.
- Need to compare resulting solution with optimal makespan L*.

Summary

NP-complete problems show up in many applications. There are different approaches to cope with it:

- Approximation algorithms
- Restricted cases (trees, bipartite graphs, small solution…)
- Randomized algorithms
- …

Each approach has its pros and cons.

Exam

Time: 10 minutes reading time

2.5 hours writing

Number of problems: 6 (ordered from easiest to hardest…imo)

For Question 6 there is one for comp2007

and one for comp2907.

More algorithms?

- COMP3530: Discrete Optimization

  Lecturer: Julian Mestre

- COMP5045: Computational Geometry

  Lecturer: Joachim

Please remember to fill in the unit of study evaluation

https://student-surveys.sydney.edu.au/students/complete/

What was good? What was bad?

Thanks for taking the class!

Good luck on the exam!