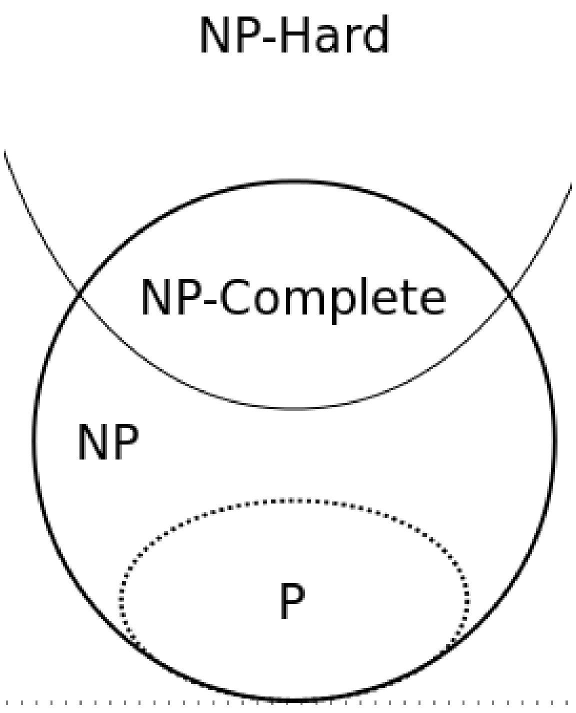
Lecture 10 cont'd: NP and Computational Intractability



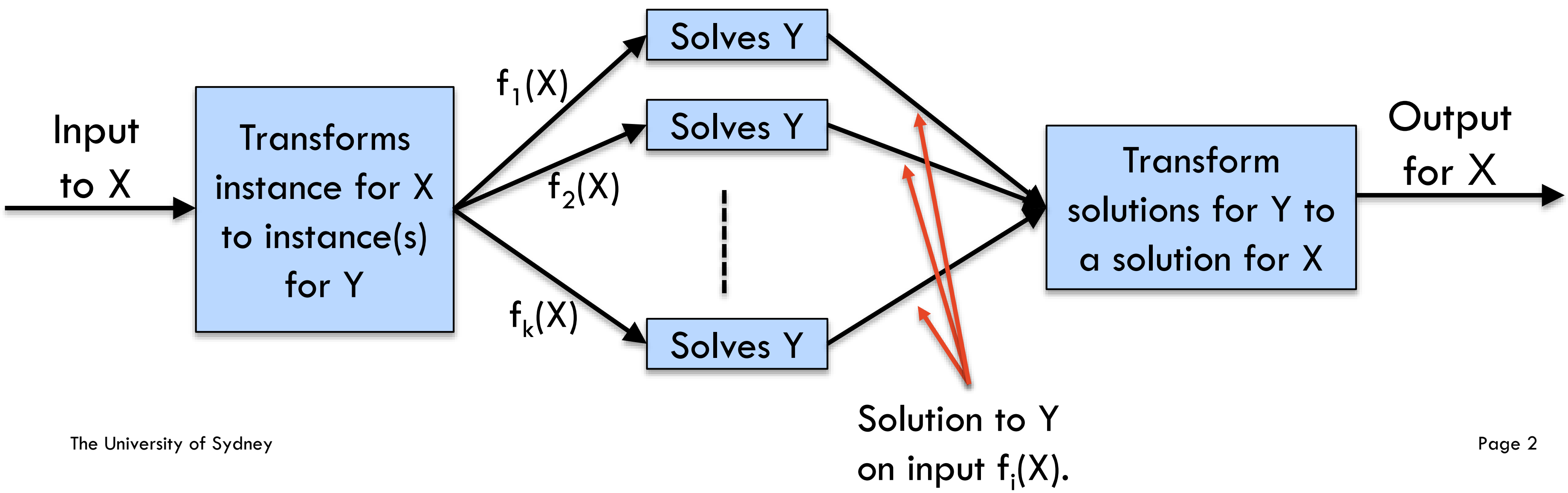
THE UNIVERSITY OF SYDNEY

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.

Summary – Lecture 10

- 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

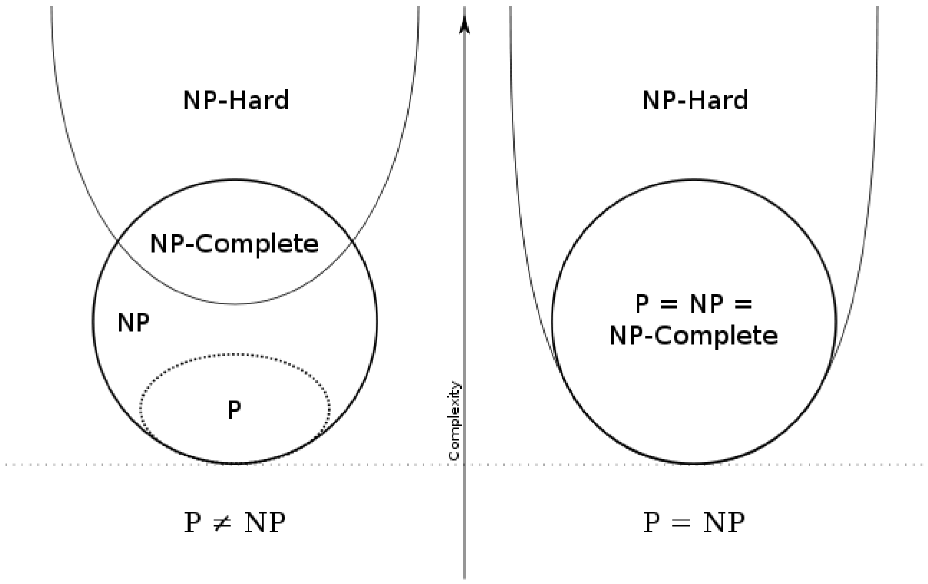
  See https://www.nada.kth.se/~viggo/wwwcompendium/

Class NP-hard

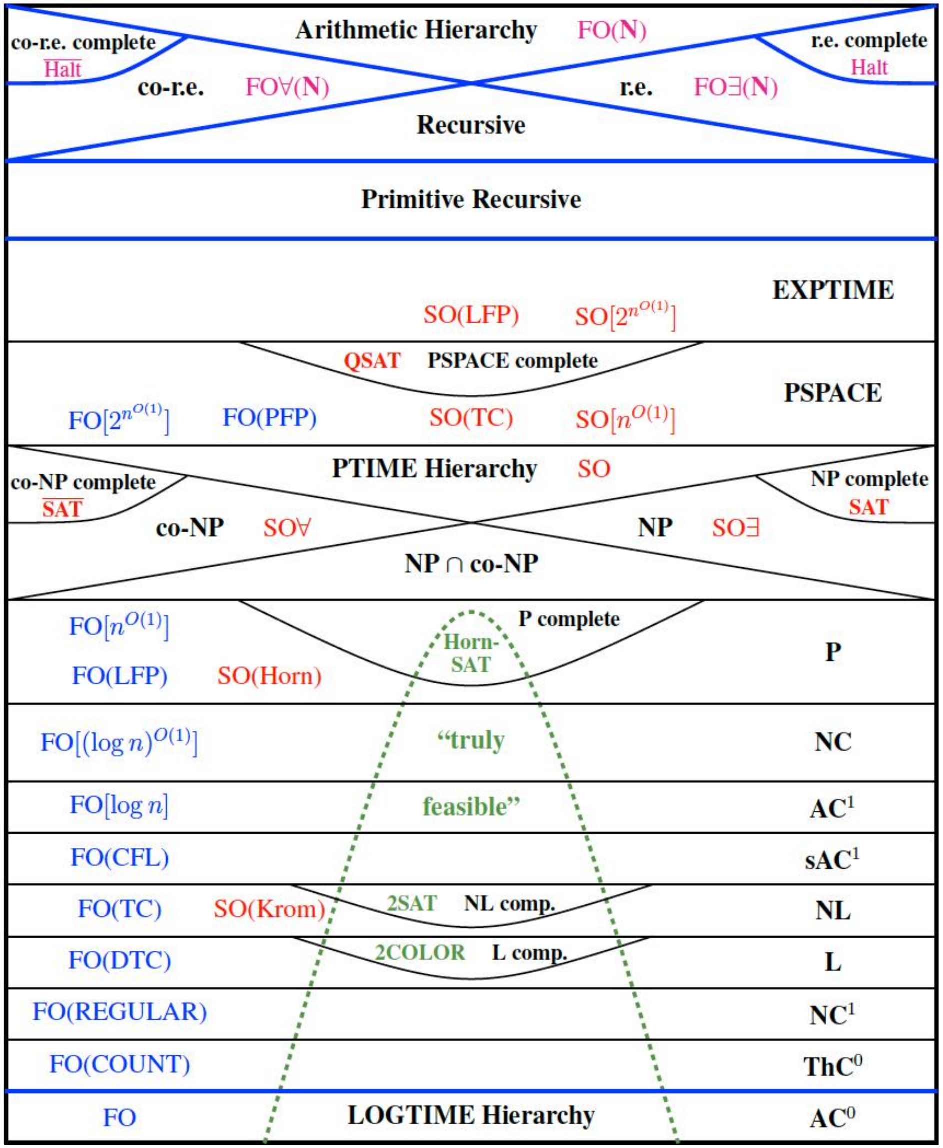
Class NP-complete: A problem in NP such that every problem in NP polynomially reduces to it.

Class NP-hard:
A decision problem such that every problem in NP polynomially reduces to it.

not necessarily in NP



Many classes?



8.5 Sequencing Problems

Six basic genres

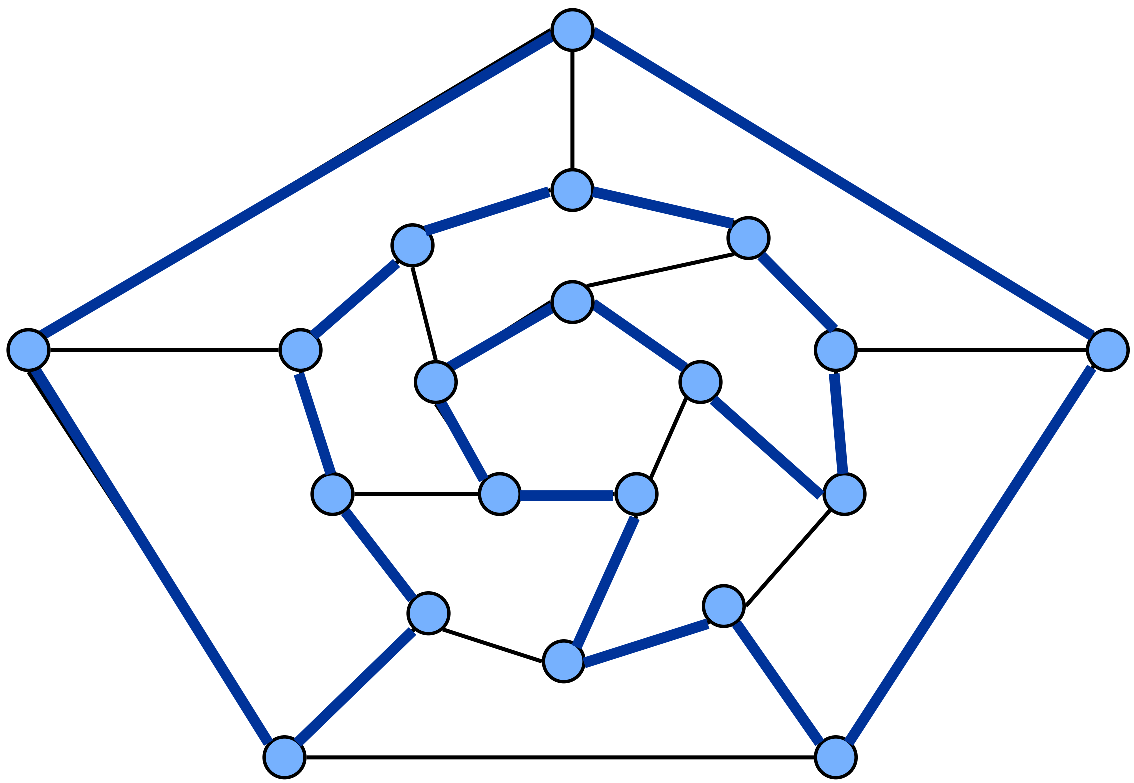
- Packing problems: SET-PACKING, INDEPENDENT SET.
- Covering problems: SET-COVER, VERTEX-COVER.
- Constraint satisfaction problems: SAT, 3-SAT.
- Sequencing problems: HAMILTONIAN-CYCLE, TSP.

  3-SAT $\leq_P$ DIR HAMILTONIAN CYCLE $\leq_P$ HAMILTONIAN CYCLE $\leq_P$ TSP

- Partitioning problems: 3D-MATCHING, 3-COLOR.
- Numerical problems: SUBSET-SUM, KNAPSACK.

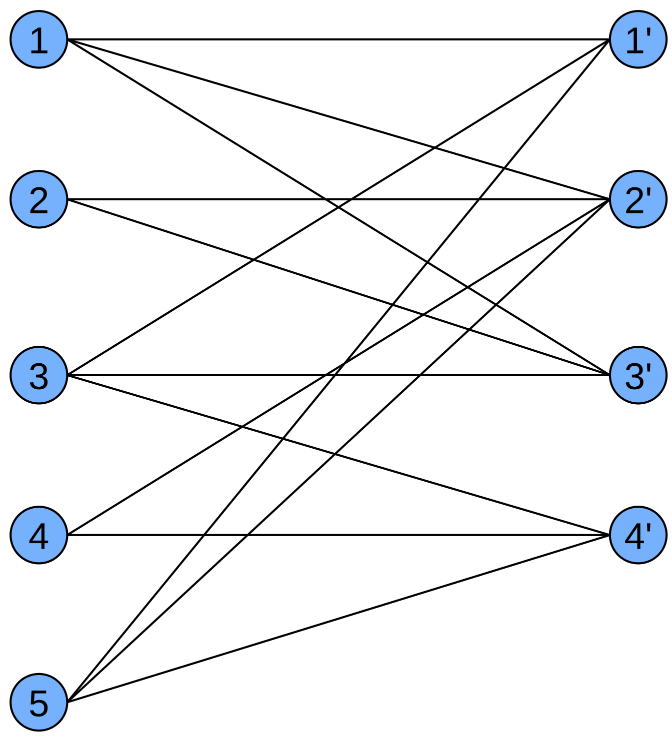
Hamiltonian Cycle

HAM-CYCLE: given an undirected graph G = (V, E), does there exist a simple cycle Γ that contains every node in V.



Hamiltonian Cycle

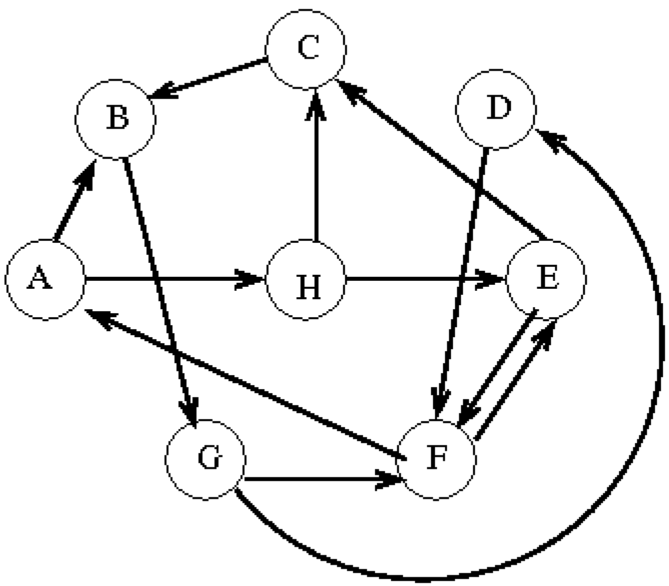
HAM-CYCLE: given an undirected graph G = (V, E), does there exist a simple cycle Γ that contains every node in V.



HAM-CYCLE $\in$ NP

Directed Hamiltonian Cycle

DIR-HAM-CYCLE: Given a directed graph $G = (V, E)$, does there exists a simple directed cycle Γ that contains every node in V?



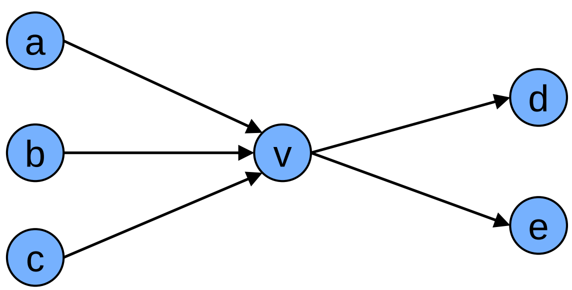
DIR-HAM-CYCLE $\in$ NP

Directed Hamiltonian Cycle

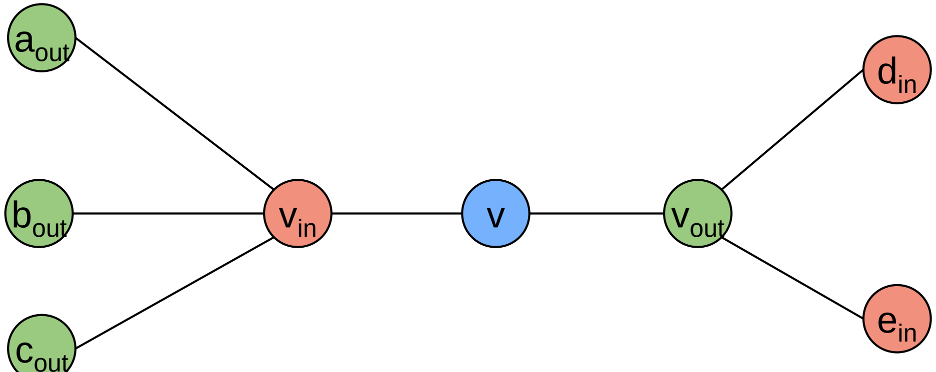
DIR-HAM-CYCLE: Given a directed graph G = (V, E), does there exists a simple directed cycle Γ that contains every node in V?

Theorem: DIR-HAM-CYCLE $\leq_P$ HAM-CYCLE.

Proof idea: Given a directed graph G = (V, E), construct an undirected graph G' with 3n vertices.

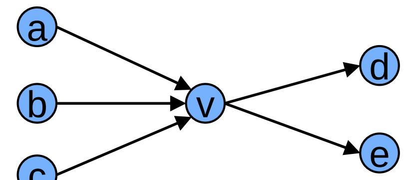


G



G'

Directed Hamiltonian Cycle



G

Claim: G has a Hamiltonian cycle iff G' does.

Proof:

$\Rightarrow$ – Suppose G has a directed Hamiltonian cycle Γ.
– Then G' has an undirected Hamiltonian cycle (same order).

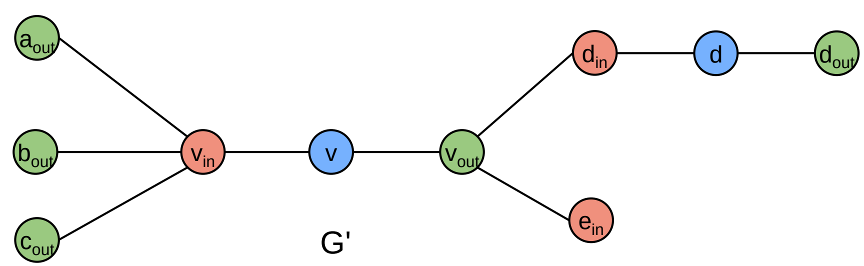
$\Leftarrow$ – Suppose G' has an undirected Hamiltonian cycle Γ'.
– Γ' must visit nodes in G' using one of two orders:

…, B, G, R, B, G, R, B, G, R, B, …

…, B, R, G, B, R, G, B, R, G, B, …

– Blue nodes in Γ' make up directed Hamiltonian cycle Γ in G, or reverse of one. ▪

DIR-HAM-CYCLE $\leq_P$ HAM-CYCLE



G'

3-SAT Reduces to Directed Hamiltonian Cycle

Theorem: 3-SAT $\leq_P$ DIR-HAM-CYCLE.

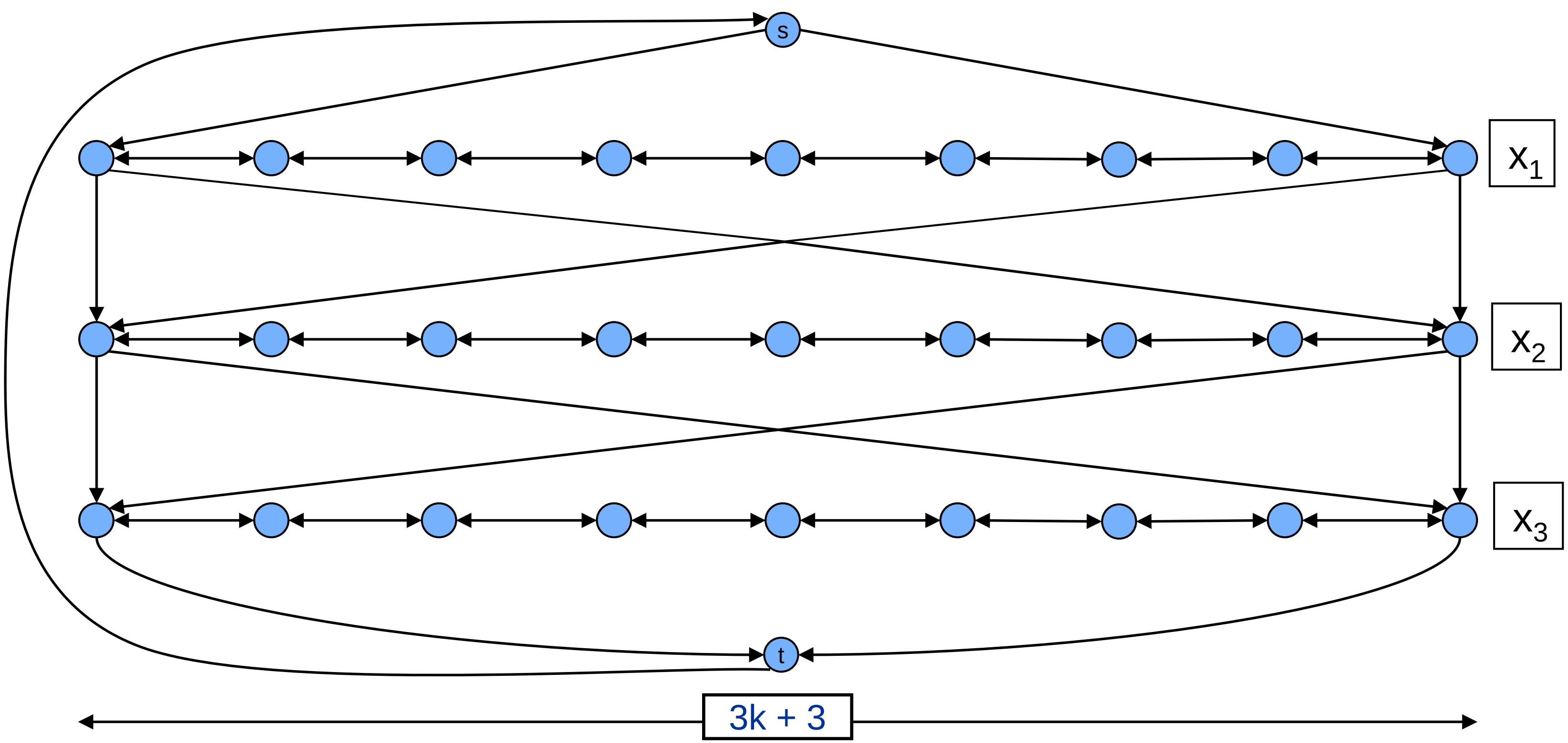
Proof: Given an instance Φ of 3-SAT, we construct an instance of DIR-HAM-CYCLE that has a Hamiltonian cycle iff Φ is satisfiable.

Construction. First, create graph that has 2^n Hamiltonian cycles which correspond in a natural way to 2^n possible truth assignments.

3-SAT Reduces to Directed Hamiltonian Cycle

Construction: Given a 3-SAT instance Φ with n variables x_i and k clauses.

- Construct G to have 2^n Hamiltonian cycles.
- Intuition: Traverse path i from left to right $\Leftrightarrow$ set variable $x_i = 1$.

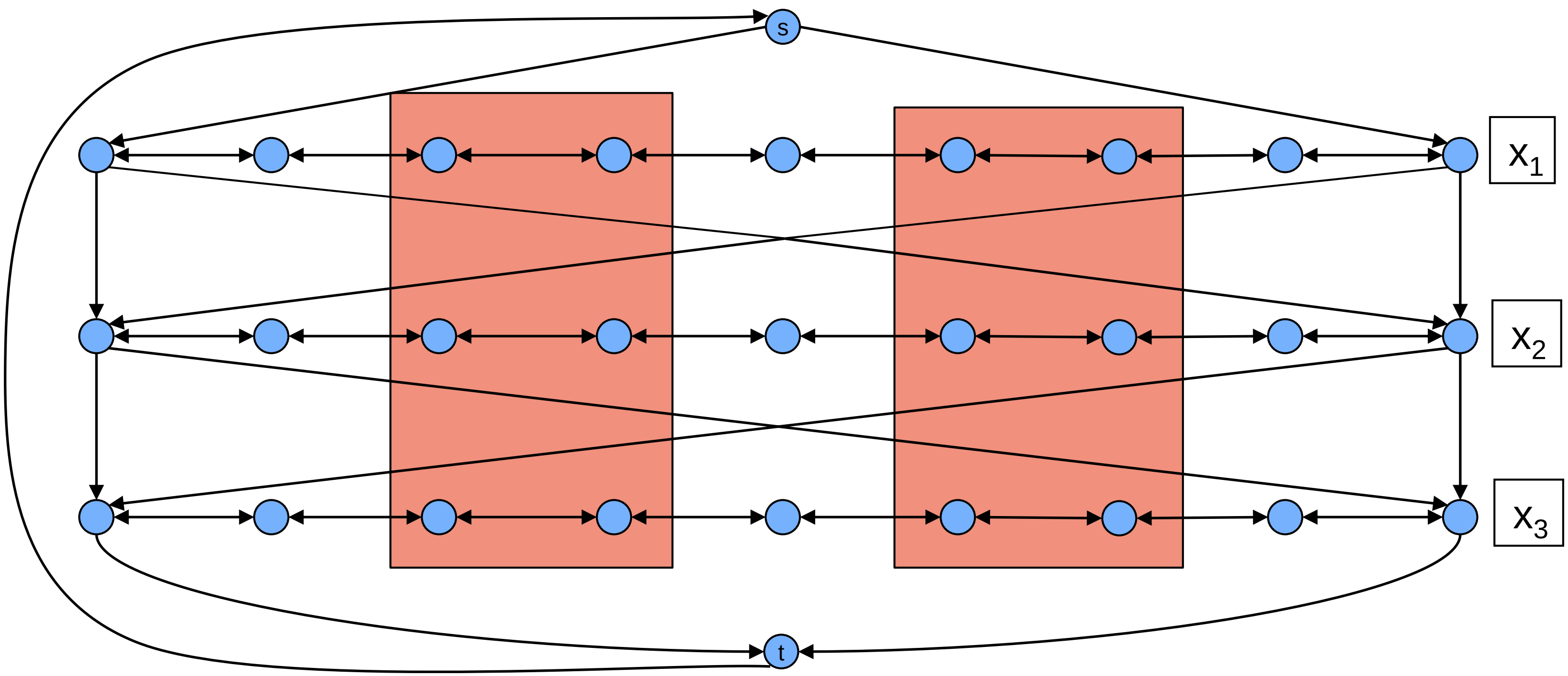


3-SAT Reduces to Directed Hamiltonian Cycle

– Construction. Given 3-SAT instance Φ with n variables x_i and k clauses.
  – For each clause: add a node and 6 edges.

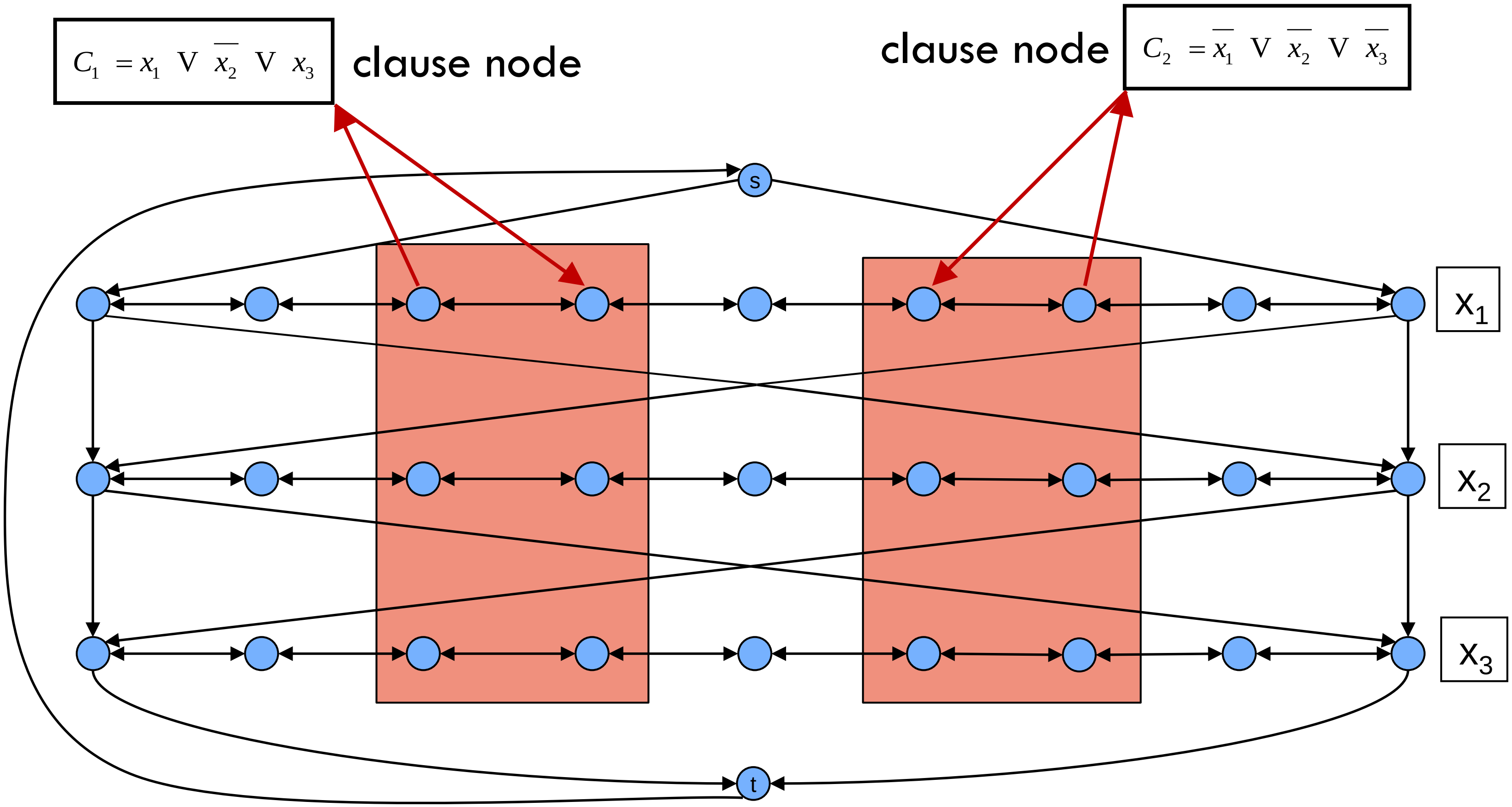
$C_1 = x_1 \vee \overline{x_2} \vee x_3$ clause node

clause node $C_2 = \overline{x_1} \vee \overline{x_2} \vee \overline{x_3}$



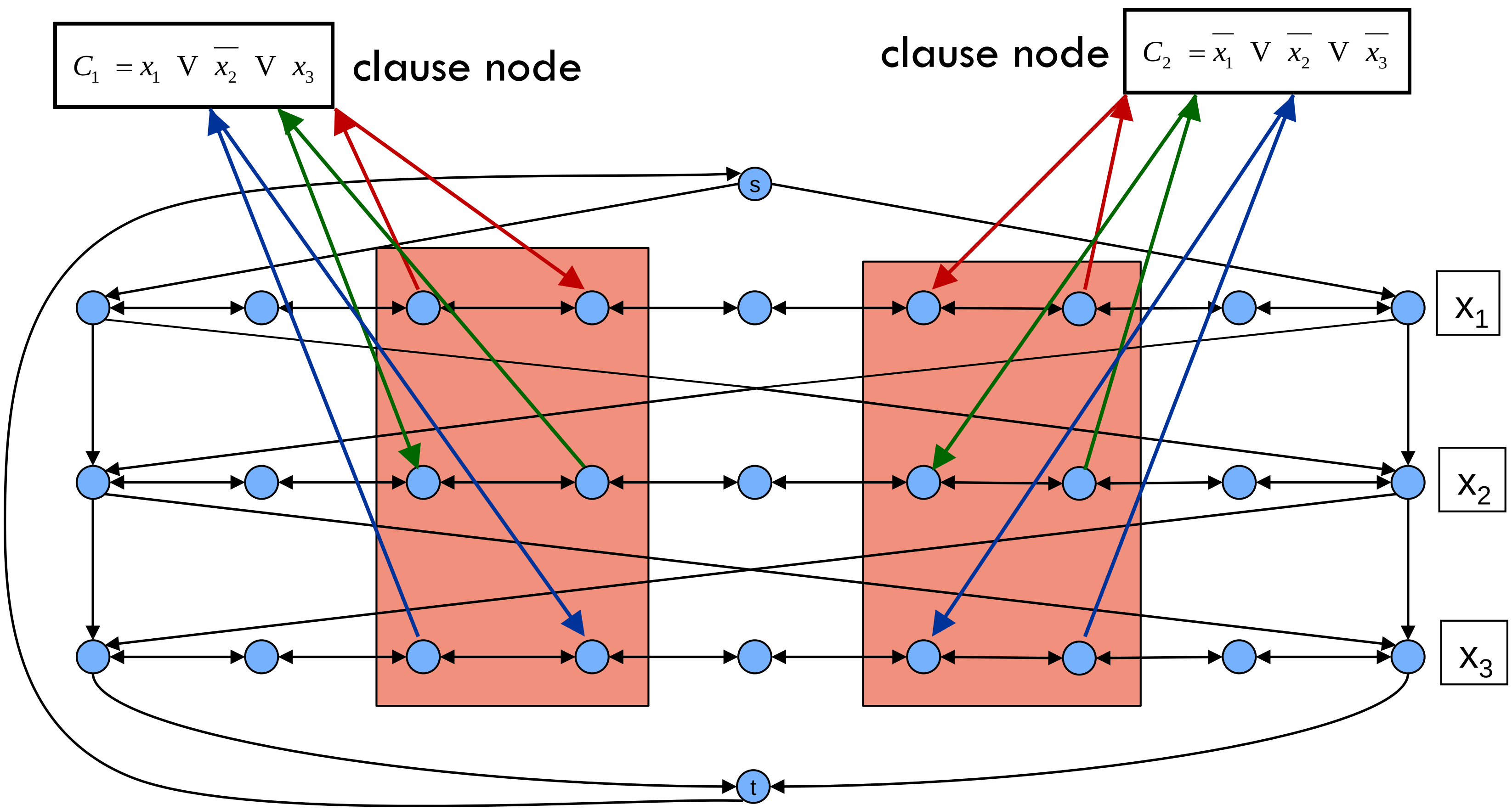
3-SAT Reduces to Directed Hamiltonian Cycle

– Construction. Given 3-SAT instance Φ with n variables x_i and k clauses.
  – For each clause: add a node and 6 edges.



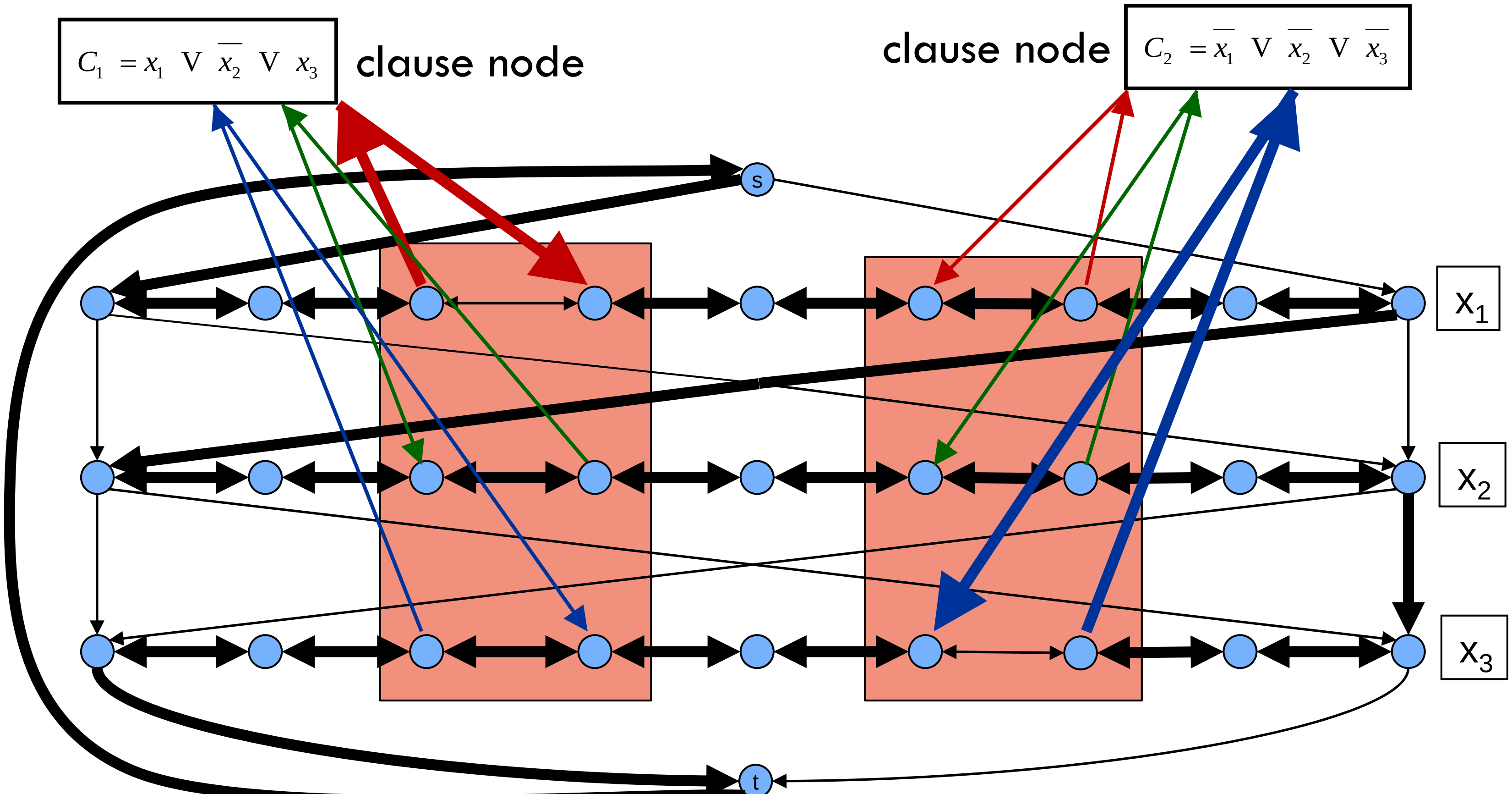
3-SAT Reduces to Directed Hamiltonian Cycle

- Construction. Given 3-SAT instance Φ with n variables x_i and k clauses.
  - For each clause: add a node and 6 edges.



3-SAT Reduces to Directed Hamiltonian Cycle

- Construction. Given 3-SAT instance Φ with n variables x_i and k clauses.
  - For each clause: add a node and 6 edges.

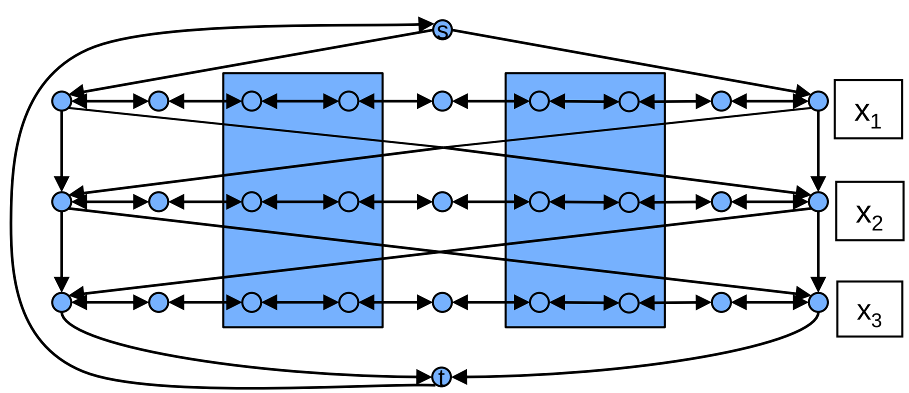


3-SAT Reduces to Directed Hamiltonian Cycle

Claim: Φ is satisfiable iff G has a Hamiltonian cycle.

Proof: $\Rightarrow$

- Suppose 3-SAT instance has satisfying assignment x*.
- Then, define Hamiltonian cycle in G as follows:
  - if $x^*_i = 1$, traverse row i from left to right
  - if $x^*_i = 0$, traverse row i from right to left
  - for each clause C_j, there will be at least one row i in which we are going in "correct" direction to splice node C_j into tour



3-SAT Reduces to Directed Hamiltonian Cycle

Claim: Φ is satisfiable iff G has a Hamiltonian cycle.

Proof: $\Leftarrow$

- Suppose G has a Hamiltonian cycle Γ.
- If Γ enters clause node C_j, it must depart on mate edge.
  - thus, nodes immediately before and after C_j are connected by an edge e in G
  - removing C_j from cycle, and replacing it with edge e yields Hamiltonian cycle on $G \setminus \{ C_j \}$
- Continuing in this way, we are left with Hamiltonian cycle Γ' in $G - \{ C_1, C_2, \ldots, C_k \}$.
- Set $x^*_i = 1$ iff Γ' traverses row i left to right.
- Since Γ visits each clause node C_j, at least one of the paths is traversed in "correct" direction, and each clause is satisfied. ▪

3-SAT Reduces to Directed Hamiltonian Cycle

Theorem: 3-SAT $\leq_P$ DIR-HAM-CYCLE.

– 3-SAT is NP-complete
– DIR-HAM-CYCLE $\in$ NP

$\Rightarrow$ DIR-HAM-CYCLE is NP-complete

Directed HAM-CYCLE reduces to HAM-CYCLE

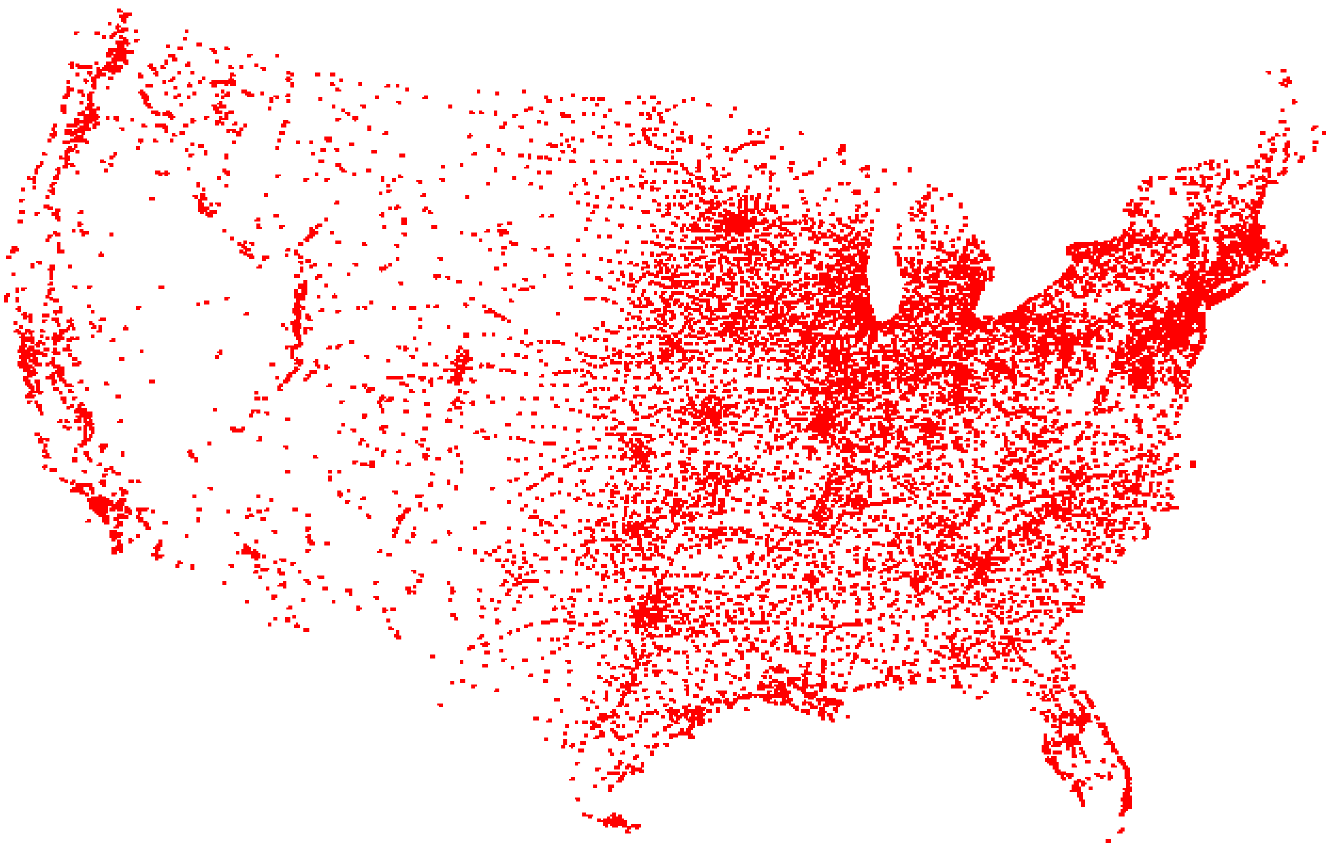
Theorem: DIR-HAM-CYCLE $\leq_P$ HAM-CYCLE.

- DIR-HAM-CYCLE is NP-complete
- HAM-CYCLE $\in$ NP

$\Rightarrow$ HAM-CYCLE is NP-complete

Travelling Salesperson Problem

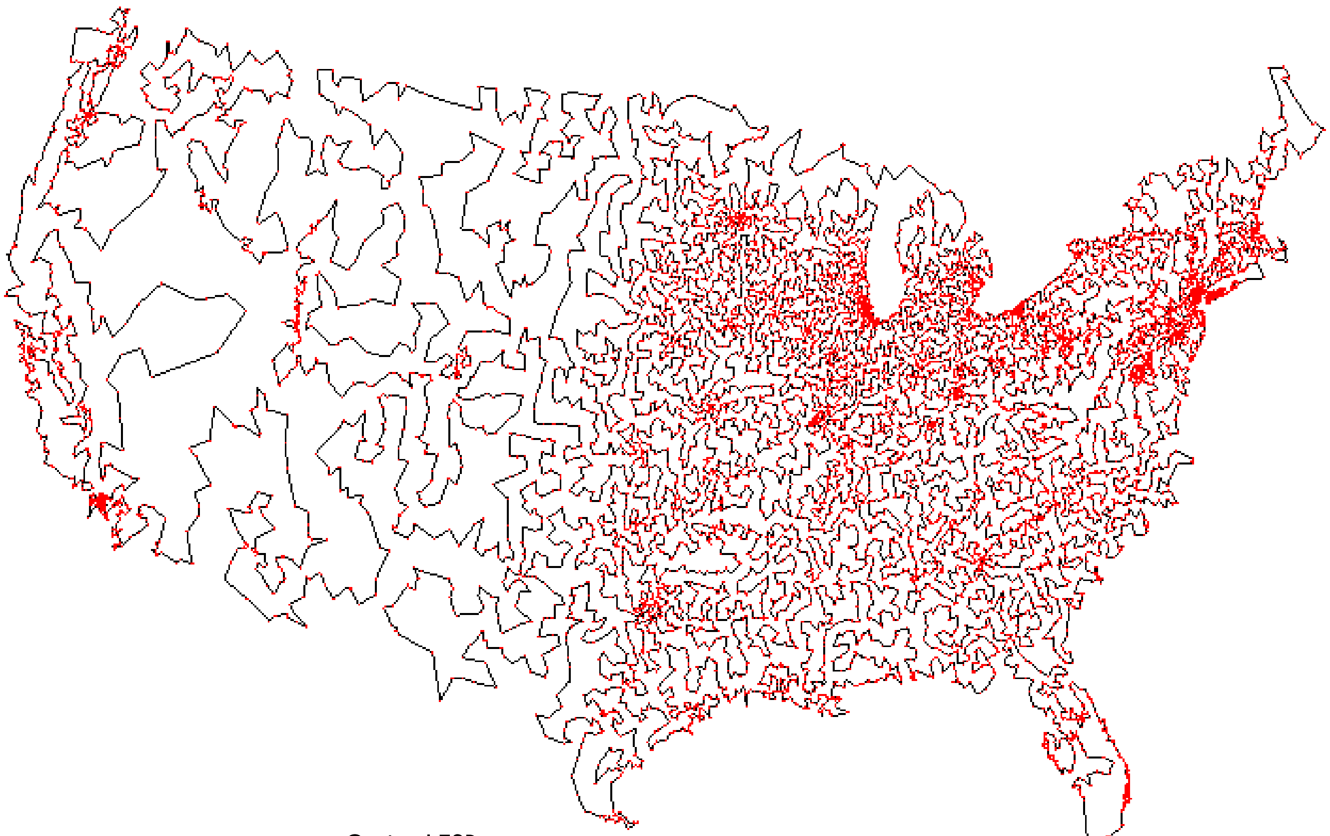
TSP: Given a set of n cities and a pairwise distance function d(u, v), is there a tour of length $\leq$ D?



All 13,509 cities in US with a population of at least 500
Reference: http://www.tsp.gatech.edu

Travelling Salesperson Problem

TSP: Given a set of n cities and a pairwise distance function d(u, v), is there a tour of length $\leq$ D?



Optimal TSP tour
Reference: http://www.tsp.gatech.edu

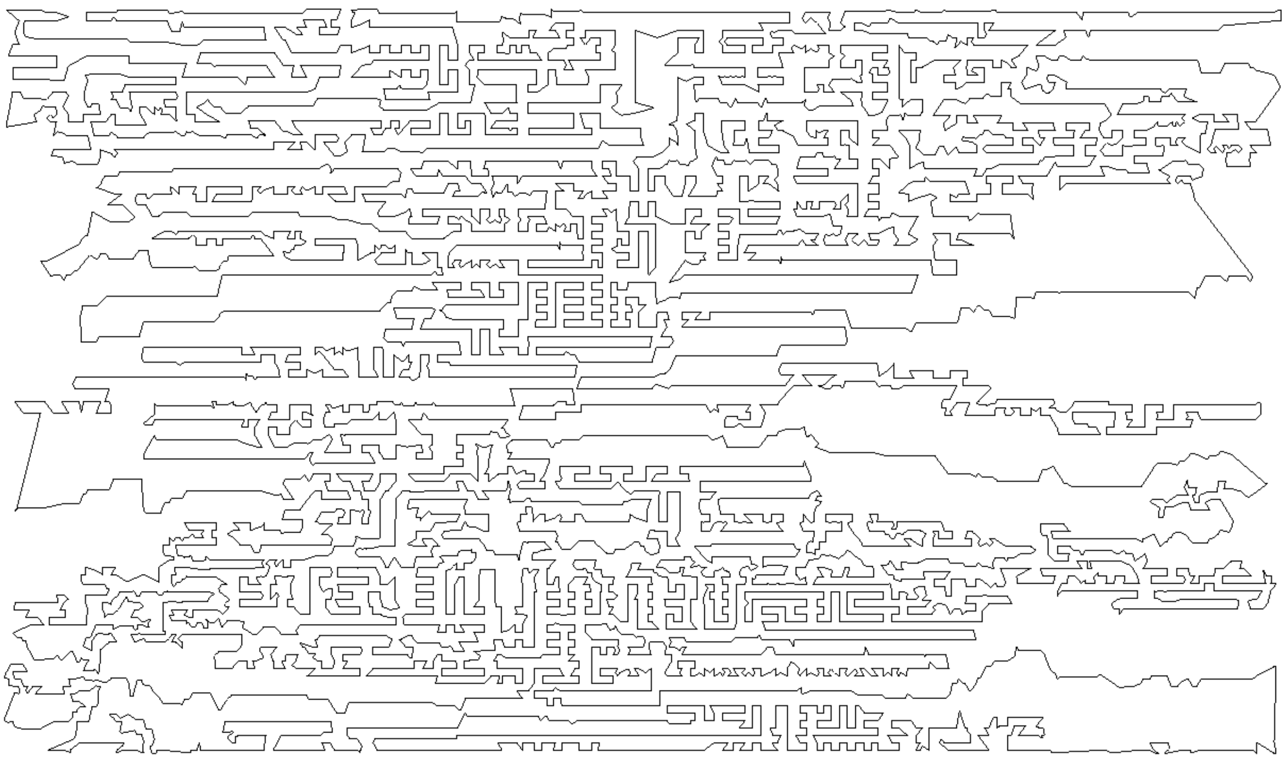
Travelling Salesperson Problem

TSP: Given a set of n cities and a pairwise distance function d(u, v), is there a tour of length $\leq$ D?

11,849 holes to drill in a programmed logic array
Reference: http://www.tsp.gatech.edu

Travelling Salesperson Problem

TSP: Given a set of n cities and a pairwise distance function d(u, v), is there a tour of length $\leq$ D?



Optimal TSP tour
Reference: http://www.tsp.gatech.edu

Travelling Salesperson Problem

TSP: Given a set of n cities and a pairwise distance function d(u, v), is there a tour of length $\leq$ D?

HAM-CYCLE: given a graph G = (V, E), does there exists a simple cycle that contains every node in V?

Theorem: HAM-CYCLE $\leq_P$ TSP.

Proof:

- Given instance G = (V, E) of HAM-CYCLE, create n cities with distance function

$$d(u, v) = \begin{cases} 1 & \text{if } (u, v) \in E \\ 2 & \text{if } (u, v) \notin E \end{cases}$$

- TSP instance has tour of length $\leq$ n iff G is Hamiltonian. ▪

TSP is NP-complete

Theorem: HAM-CYCLE $\leq_P$ TSP.

- HAM-CYCLE is NP-complete
- TSP $\in$ NP

$\Rightarrow$ TSP (decision version) is NP-complete

NP-complete games and puzzles

- Battleship
- Candy Crush Saga
- Donkey Kong
- Eternity II
- FreeCell
- Lemmings
- Minesweeper Consistency Problem
- Pokémon
- SameGame
- Sudoku
- Tetris
- Rush Hour
- Hex
- Super Mario Bros

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

Algorithms and hardness

Algorithmic techniques:

- Greedy algorithms [Lecture 3]
- Divide & Conquer algorithms [Lecture 4]
- Sweepline algorithms [Lecture 5]
- Dynamic programming algorithms [Lectures 6 and 7]
- Network flow algorithms [Lectures 8 and 9]

Hardness:

- NP-hardness [Lecture 10]: $O(n^c)$ algorithm is unlikely

Today

- How can we cope with hard problems?

Algorithms and hardness

Lots of problems that we can solve efficiently:

- MST
- Shortest path
- Scheduling
- Max flow
- …

But lots of problems that we can't solve efficiently:

- All NP-complete problems: $O(n^c)$ algorithm is unlikely

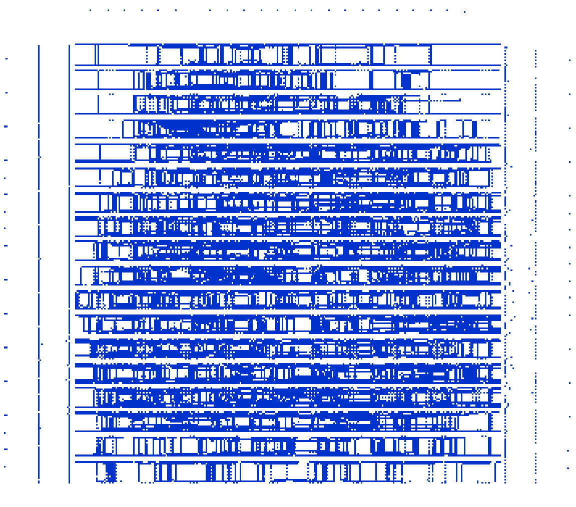
  vertex cover, independent set, 3-SAT,…
- But what if we need to solve an NP-complete problem?

Cope with NP-complete problems?

"I can't find an efficient algorithm, but neither can all these famous people."

Cope with NP-complete problems?

- Heuristics: Local search, simulated annealing, neural networks…
- Randomized algorithms: Not always correct, but a probability is guaranteed.
- Approximation algorithms: Not optimal solution, but within a guaranteed error.
- Fixed-parameter algorithms: Algorithms whose complexity depends on other parameters than n.
- Efficient exact algorithms: Euclidean TSP
- Restricted instance: trees, bipartite graphs…



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.

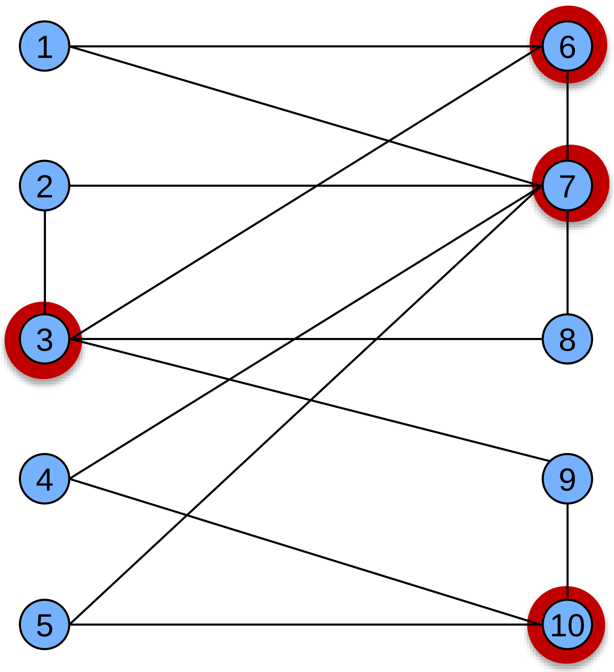
- Solve problem to optimality.
  - Approximation algorithms
  - Randomized algorithms
- Solve problem in polynomial time.
  - Exact exponential time algorithms
- Solve arbitrary instances of the problem.
  - Solve restricted classes of instances
  - Parametrized algorithms

10.1 Solving restricted instances

For example in the case when the vertex cover is small.

10.1 Finding Small Vertex Covers

VERTEX COVER: Given a graph $G = (V, E)$ and an integer k, is there a subset of vertices $S \subseteq V$ such that $|S| \leq k$, and for each edge (u, v) either $u \in S$, or $v \in S$, or both.



$k = 4$

$S = \{ 3, 6, 7, 10 \}$

Vertex Cover

VERTEX COVER: Given a graph $G = (V, E)$ and an integer k, is there a subset of vertices $S \subseteq V$ such that $|S| \leq k$, and for each edge (u, v) either $u \in S$, or $v \in S$, or both.

Vertex Cover (or Independent Set) arises naturally in many applications:

- dynamic detection of race conditions (distributed systems).
- computational biology
- Biocheimstry
- Pattern recognition
- Computer vision
- …

What should we do if we need to solve it?

Finding Small Vertex Covers

Question: What if k is small?

Brute force: $O(kn^{k+1})$.

- Try all $\binom{n}{k} \in O(n^k)$ subsets of size k.
- Takes $O(kn)$ time to check whether a subset is a vertex cover.

Finding Small Vertex Covers

Question: What if k is small?

Brute force: $O(kn^{k+1})$.

- Try all $\binom{n}{k} \in O(n^k)$ subsets of size k.
- Takes $O(kn)$ time to check whether a subset is a vertex cover.

Aim: Limit exponential dependency on k, e.g., to $O(2^k\, kn)$.

Example: $n = 1000$, $k = 10$.

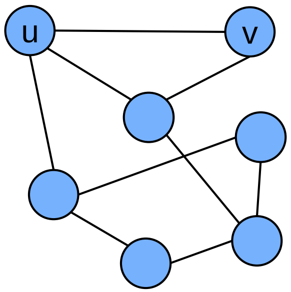
- Brute force. $kn^{k+1} = 10^{34} \Rightarrow$ infeasible.
- Better. $2^k\, kn = 10^7 \Rightarrow$ feasible.

Remark. If k is a constant, algorithm is poly-time; if k is a small constant, then it's also practical.

Finding Small Vertex Covers

Theorem: Let (u,v) be an edge of G. G has a vertex cover of size $\leq$ k iff at least one of $G\setminus\{u\}$ and $G\setminus\{v\}$ has a vertex cover of size $\leq$ k-1.

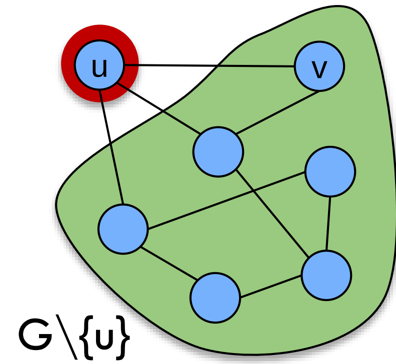
delete u and all incident edges



Finding Small Vertex Covers

Theorem: Let (u,v) be an edge of G. G has a vertex cover of size $\leq$ k iff at least one of $G \setminus \{ u \}$ and $G \setminus \{ v \}$ has a vertex cover of size $\leq$ k-1.

delete u and all incident edges



Proof:

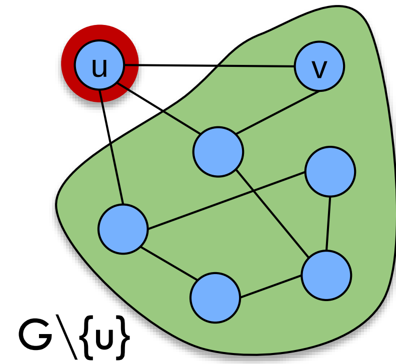
$\Rightarrow$

- Suppose G has a vertex cover S of size $\leq$ k.
- S contains either u or v (or both). Assume it contains u.
- $S \setminus \{ u \}$ is a vertex cover of $G \setminus \{ u \}$.

Finding Small Vertex Covers

Theorem: Let (u,v) be an edge of G. G has a vertex cover of size $\leq$ k iff at least one of $G \setminus \{u\}$ and $G \setminus \{v\}$ has a vertex cover of size $\leq$ k-1.

delete u and all incident edges



Proof:

$\Rightarrow$

- Suppose G has a vertex cover S of size $\leq$ k.
- S contains either u or v (or both). Assume it contains u.
- $S \setminus \{u\}$ is a vertex cover of $G \setminus \{u\}$.

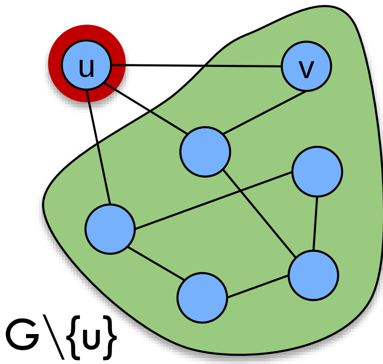
$\Leftarrow$

- Suppose S is a vertex cover of $G \setminus \{u\}$ of size $\leq$ k-1.
- Then $S \cup \{u\}$ is a vertex cover of G of size k. ▪

Finding Small Vertex Covers

Theorem: Let (u,v) be an edge of G. G has a vertex cover of size $\leq$ k iff at least one of $G \setminus \{u\}$ and $G \setminus \{v\}$ has a vertex cover of size $\leq$ k-1.

delete u and all incident edges



Proof:

$\Rightarrow$

- Suppose G has a vertex cover S of size $\leq$ k.
- S contains either u or v (or both). Assume it contains u.
- $S \setminus \{u\}$ is a vertex cover of $G \setminus \{u\}$.

$\Leftarrow$

- Suppose S is a vertex cover of $G \setminus \{u\}$ of size $\leq$ k-1.
- Then $S \cup \{u\}$ is a vertex cover of G of size k. ▪

Observation: If G has a vertex cover of size k, it has $\leq$ k(n-1) edges.

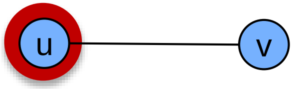
Proof: Each vertex covers at most n-1 edges. ▪

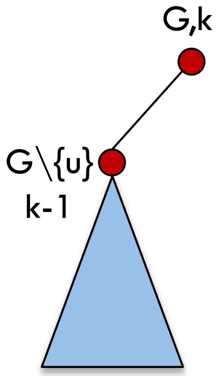
Finding Small Vertex Covers: Algorithm

Theorem: The following algorithm determines if G has a vertex cover of size $\leq$ k in O(2^k kn) time.

```
boolean Vertex-Cover(G,k) {
   if (G contains no edges)    return true
   if (G contains > k(n-1) edges) return false

   let (u,v) be any edge of G
   a = Vertex-Cover(G\{u},k-1)
   b = Vertex-Cover(G\{v},k-1)
   return a or b
}
```



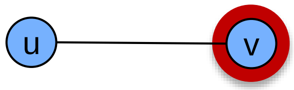


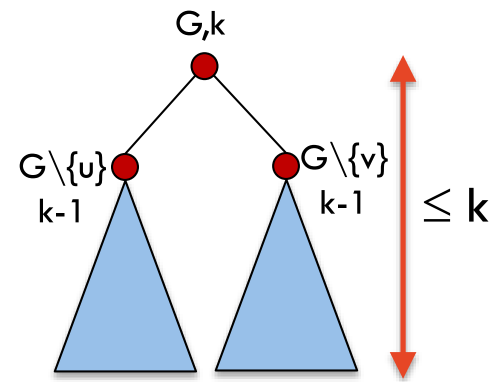
Finding Small Vertex Covers: Algorithm

Theorem: The following algorithm determines if G has a vertex cover of size $\leq$ k in $O(2^k \, kn)$ time.

```
boolean Vertex-Cover(G,k) {
   if (G contains no edges)    return true
   if (G contains > k(n-1) edges) return false

   let (u,v) be any edge of G
   a = Vertex-Cover(G\{u},k-1)
   b = Vertex-Cover(G\{v},k-1)
   return a or b
}
```



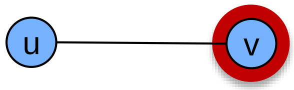


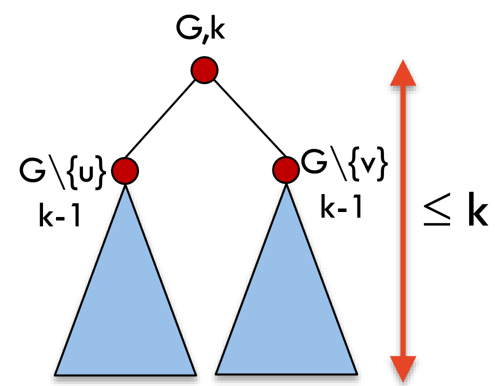
Finding Small Vertex Covers: Algorithm

Theorem: The following algorithm determines if G has a vertex cover of size $\leq k$ in $O(2^k kn)$ time.

```
boolean Vertex-Cover(G,k) {
   if (G contains no edges)    return true
   if (G contains > k(n-1) edges) return false

   let (u,v) be any edge of G
   a = Vertex-Cover(G\{u},k-1)
   b = Vertex-Cover(G\{v},k-1)
   return a or b
}
```





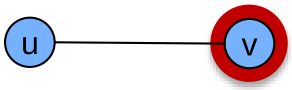
Proof:

– Correctness follows from previous two claims.

– There are $\leq 2^{k+1}$ nodes in the recursion tree; each invocation takes $O(kn)$ time. ▪

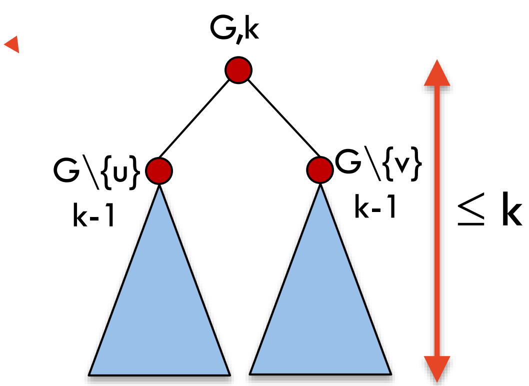
Finding Small Vertex Covers: Algorithm

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?



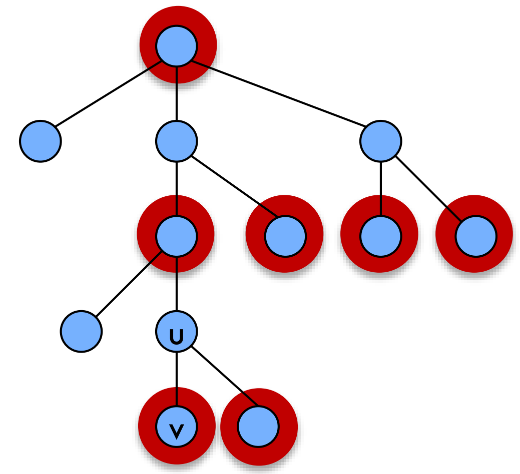
10.2 Solving NP-Hard Problems on restricted input instances

For example special cases of graphs:

- trees,
- bipartite graphs,
- planar graphs,
- …

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.

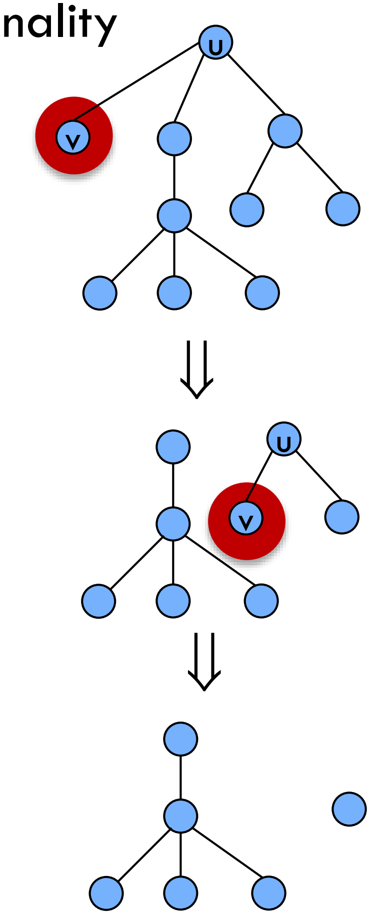
Key observation: If v is a leaf, there exists a maximum size independent set containing v.

Proof: [exchange argument]

- Consider a max cardinality independent set S.
- If $v \in S$, we're done.
- If $u \notin S$ and $v \notin S$, then $S \cup \{v\}$ is independent $\Rightarrow$ S not maximum.
- IF $u \in S$ and $v \notin S$, then $S \cup \{v\}/\{u\}$ is independent. (exchange) ▪

Independent Set on Trees: Greedy Algorithm

Theorem: The following greedy algorithm finds a maximum cardinality independent set in forests (and hence trees).




```
Independent-Set-In-A-Forest(F) {
   S ← ∅
   while (F has at least one edge) {
      Let e = (u,v) be an edge such that v is a leaf
      Add v to S
      Delete from F nodes u and v, and all edges
         incident to them.
   }
   return S
}
```

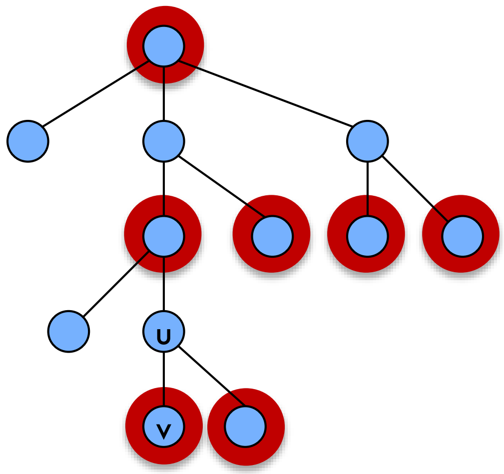
Proof: Correctness follows from the previous key observation. ▪

Remark. Can implement in O(n) time by considering nodes in postorder.

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.

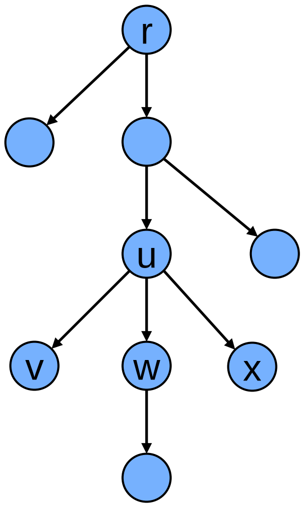


Theorem:

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

Weighted Independent Set on Trees

Weighted independent set on trees. Given a tree and node weights $w_v > 0$, find an independent set S that maximizes $\Sigma_{v \in S}\ w_v$.

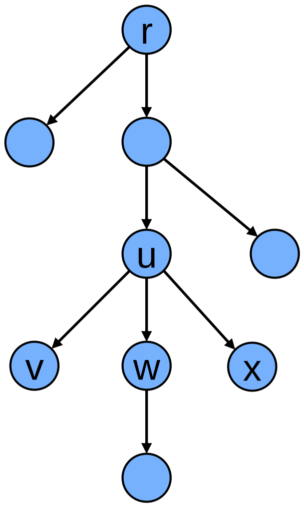


children(u) = { v, w, x }

Weighted Independent Set on Trees

Weighted independent set on trees. Given a tree and node weights $w_v > 0$, find an independent set S that maximizes $\Sigma_{v \in S} w_v$.

Observation: If (u, v) is an edge such that v is a leaf node, then either OPT includes u, or it includes all leaf nodes incident to u.



children(u) = { v, w, x }

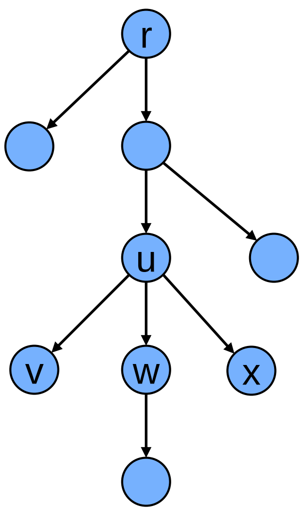
Weighted Independent Set on Trees: DP

Weighted independent set on trees. Given a tree and node weights $w_v > 0$, find an independent set S that maximizes $\Sigma_{v \in S}\, w_v$.

Observation: If (u, v) is an edge such that v is a leaf node, then either OPT includes u, or it includes all leaf nodes incident to u.

Dynamic programming solution: Root tree at some node, say r.

- $OPT_{in}(u)$ = max weight independent set rooted at u, containing u.
- $OPT_{out}(u)$ = max weight independent set rooted at u, not containing u.



children(u) = { v, w, x }

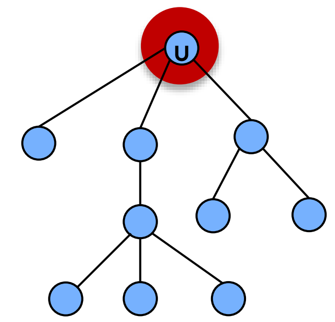
$$OPT_{in}(u) = w_u + \sum_{v \in \text{children}(u)} OPT_{out}(v)$$

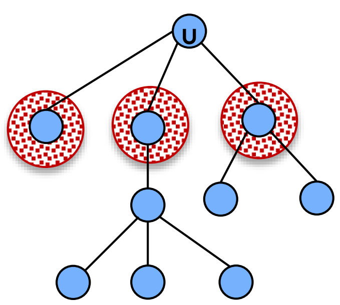
$$OPT_{out}(u) = \sum_{v \in \text{children}(u)} \max\{OPT_{in}(v),\ OPT_{out}(v)\}$$

Weighted Independent Set on Trees: DP

Theorem: The dynamic programming algorithm find a maximum weighted independent set in trees in O(n) time.

```
Weighted-Independent-Set-In-A-Tree(T) {
   Root the tree at a node r
   foreach (node u of T in postorder) {
      if (u is a leaf) {
         M_in [u] = w_u
         M_out[u] = 0}
      else {
         M_in [u] = Σ_{v∈children(u)} M_out[v]+w_v
         M_out[u] = Σ_{v∈children(u)} max(M_out[v],M_in[v])}
   }
   return max(M_in[r], M_out[r])
}
```





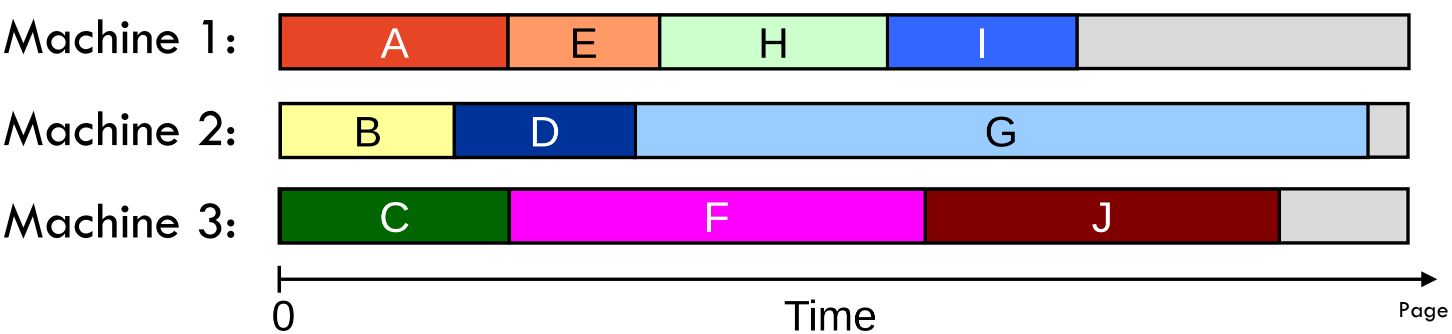
Proof: Takes O(n) time since we visit nodes in postorder and examine each edge exactly once. ▪

11.1 Approximation algorithms: Load Balancing

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.



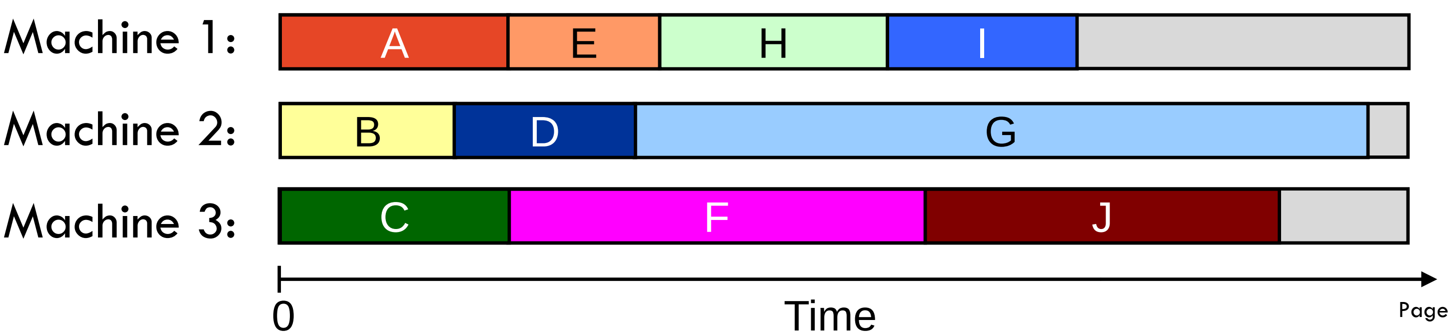
Load Balancing

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}



Load Balancing

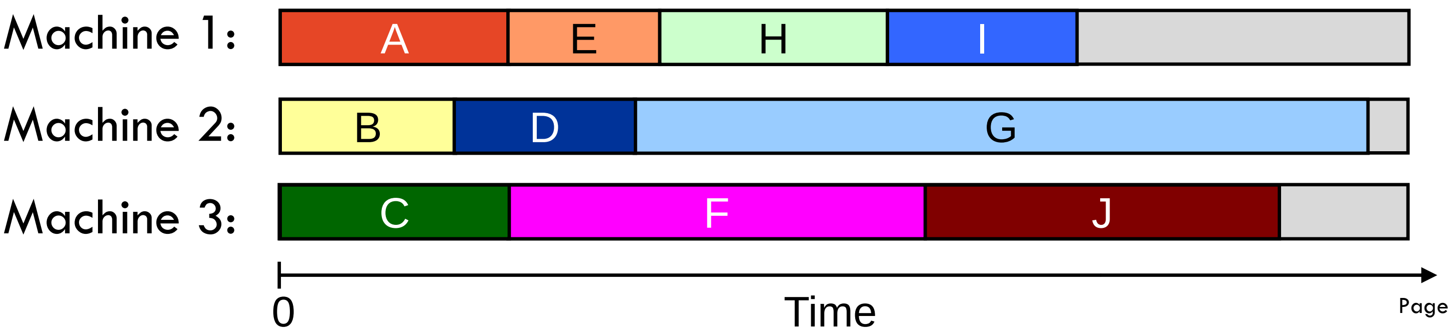
Input: m identical machines; n jobs, job j has processing time t_j.

- 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 $L_i = \sum_{j \in J(i)} t_j$.
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.



Load Balancing: List Scheduling

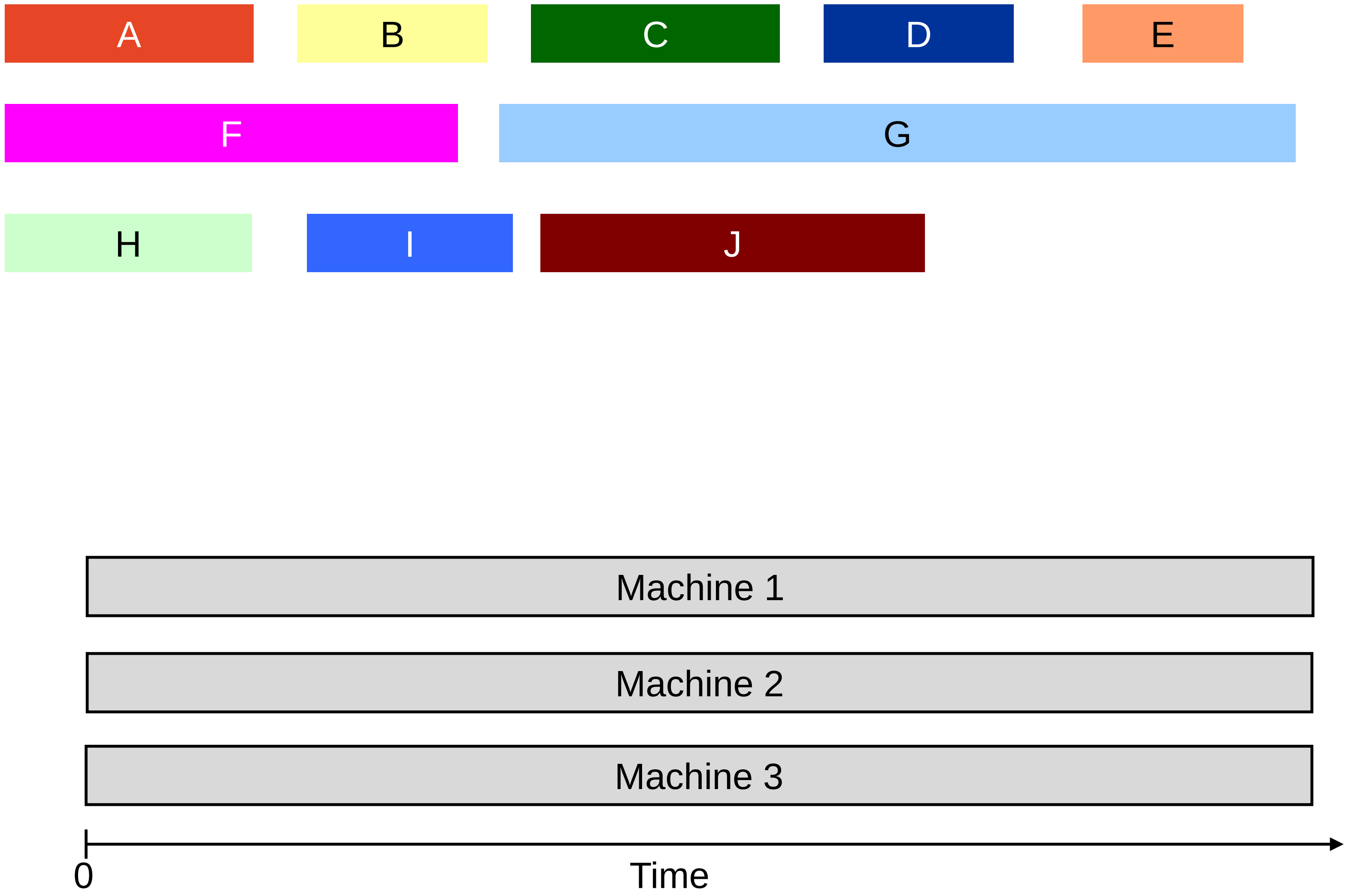
List-scheduling algorithm:

- Consider n jobs in some fixed order.
- Assign job j to machine whose load is smallest so far.

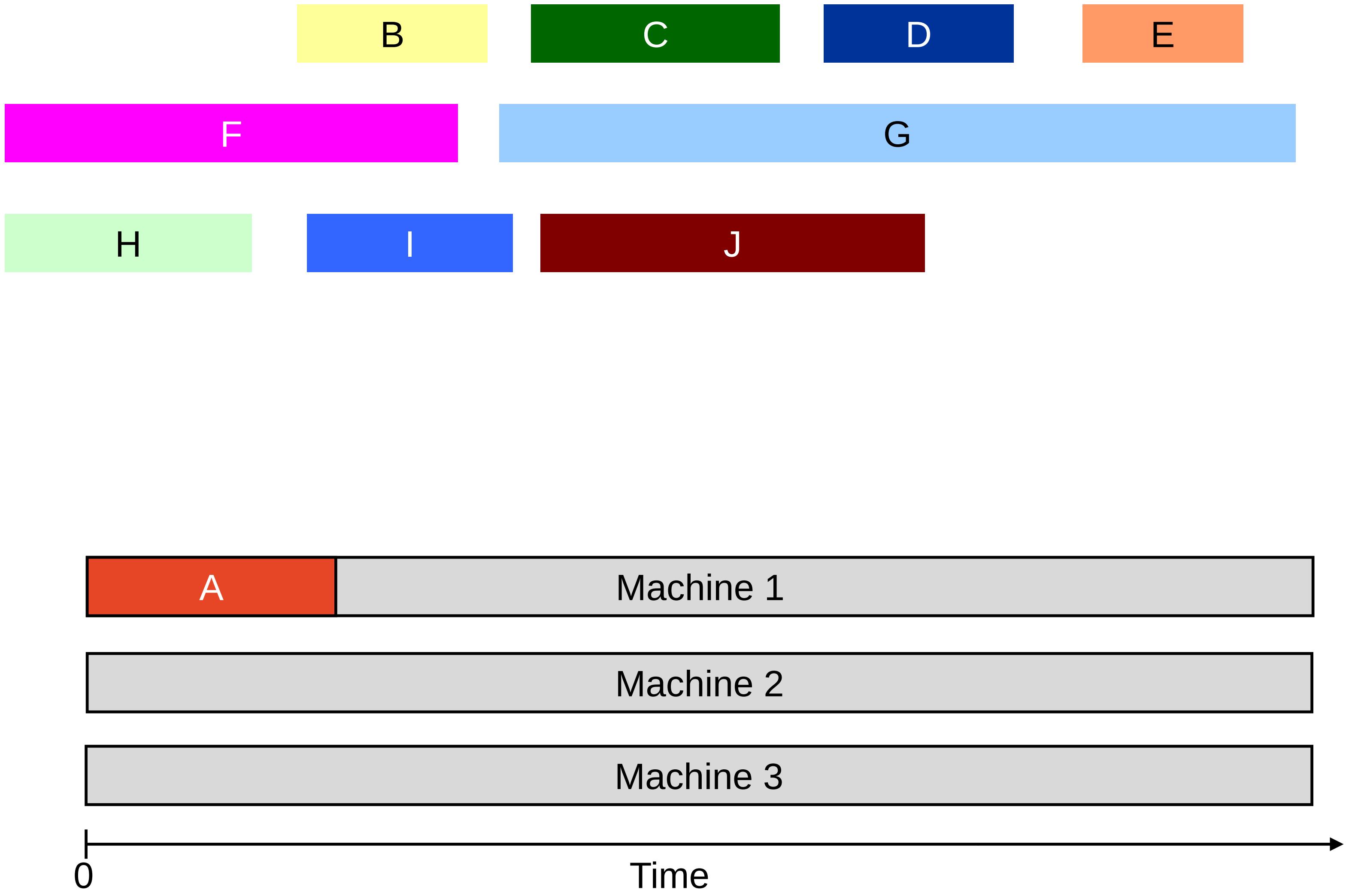
```
List-Scheduling(m, n, t1,t2,…,tn) {
   for i = 1 to m {
      Li ← 0              ← load on machine i
      J(i) ← ∅            ← jobs assigned to machine i
   }
   for j = 1 to n {
      i = argmink Lk           ← machine i has smallest load
      J(i) ← J(i) ∪ {j}        ← assign job j to machine i
      Li ← Li + tj             ← update load of machine i
   }
}
```

Implementation: O(n log n) using a priority queue.

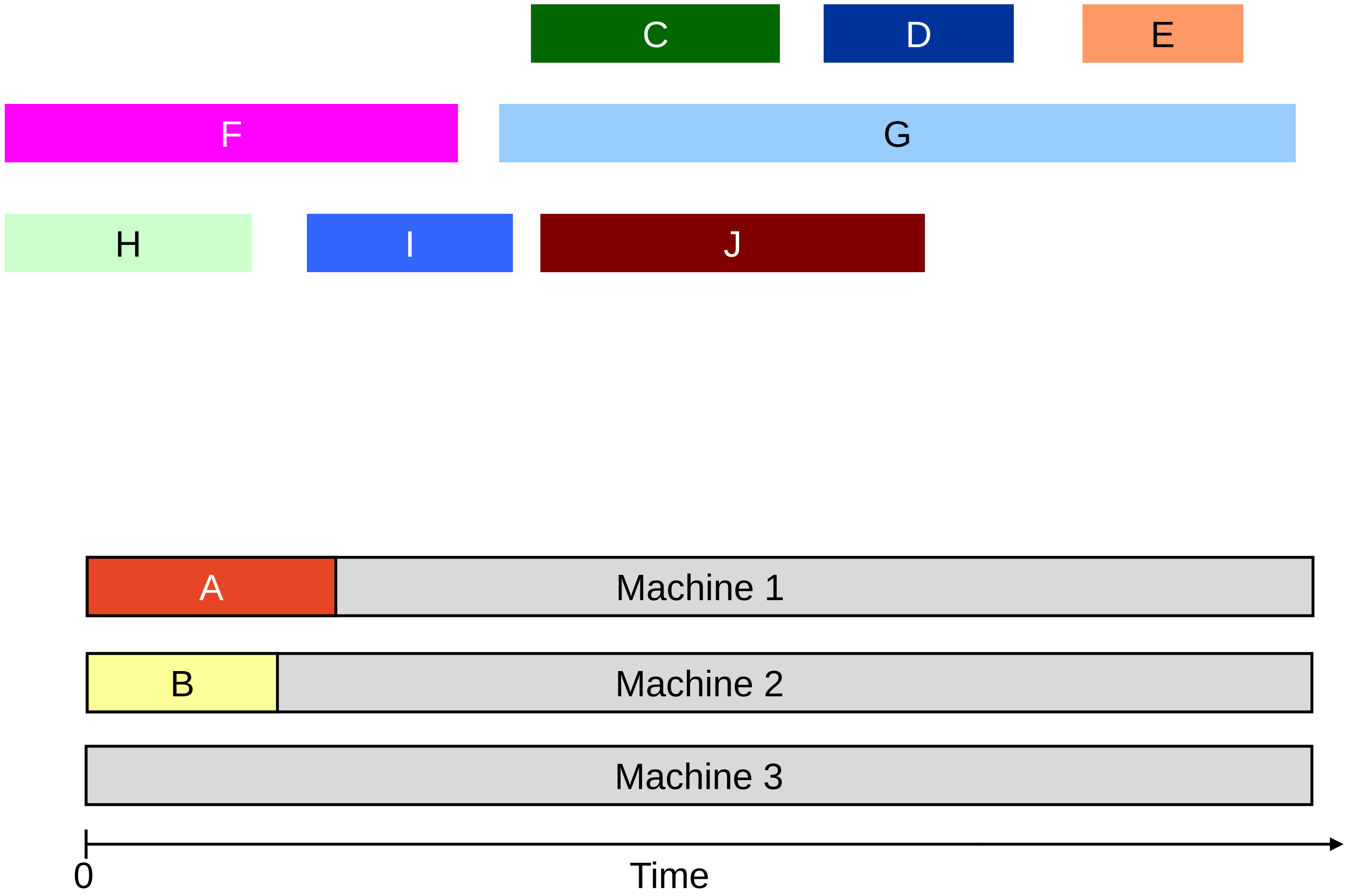
Load Balancing: List Scheduling



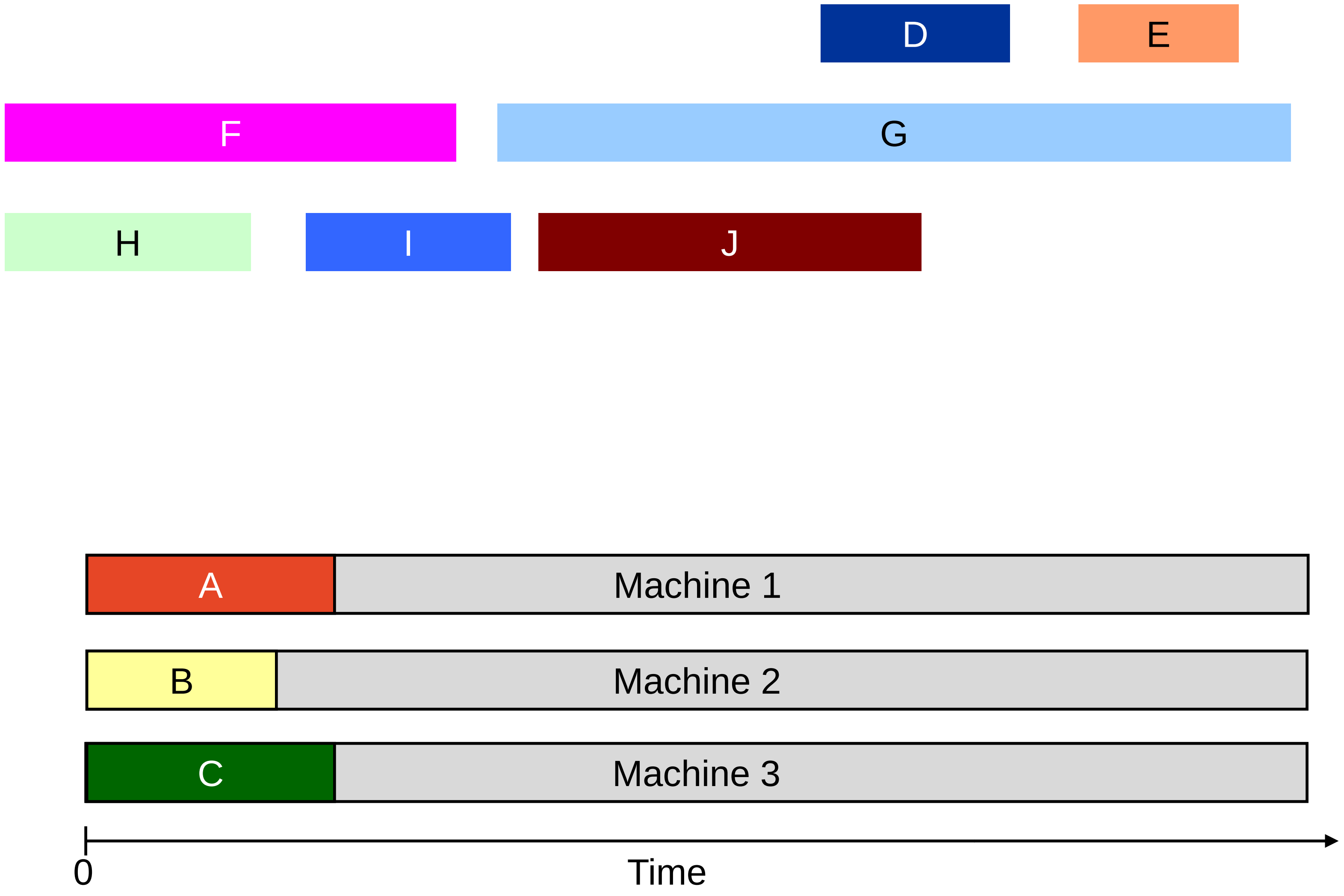
Load Balancing: List Scheduling



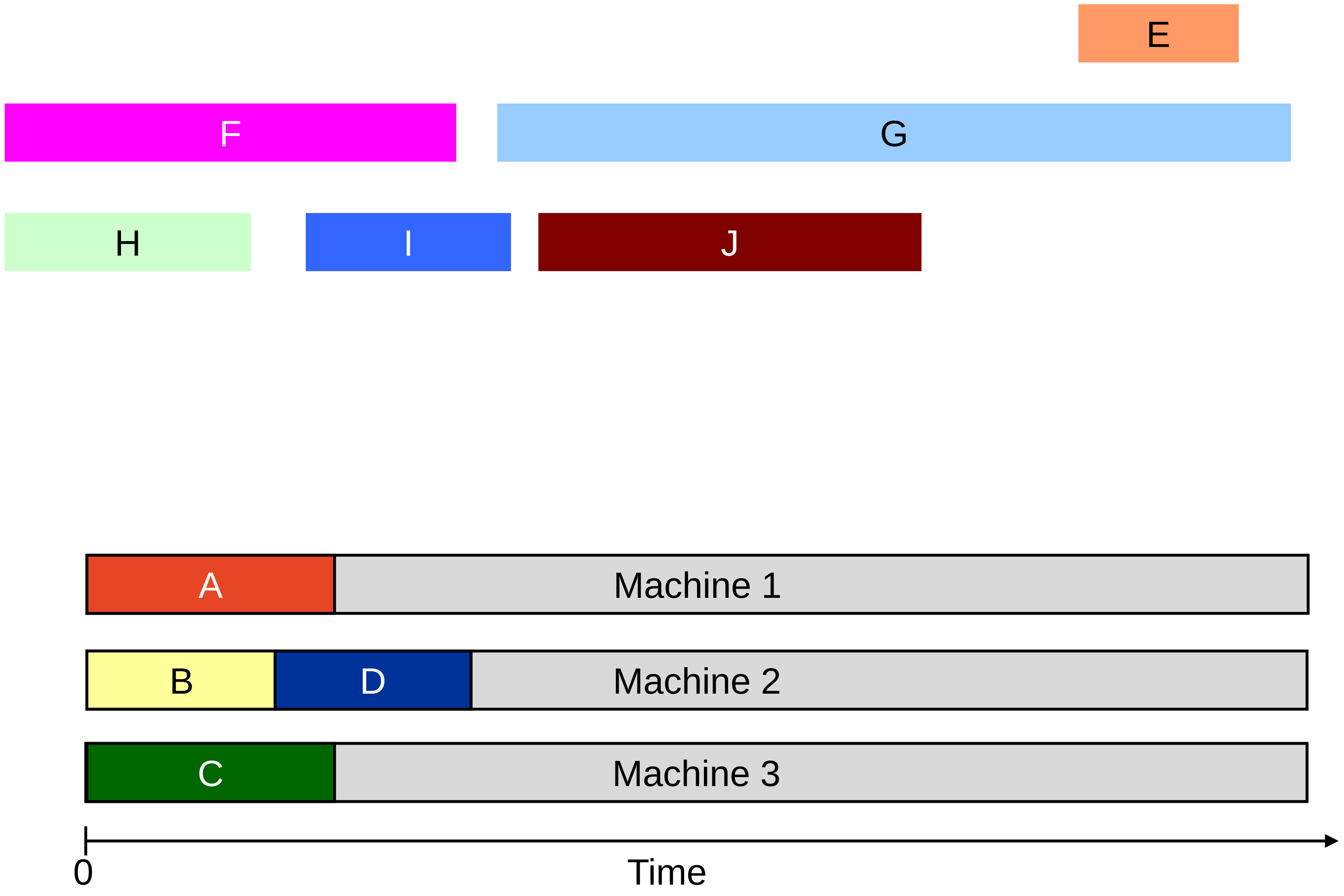
Load Balancing: List Scheduling



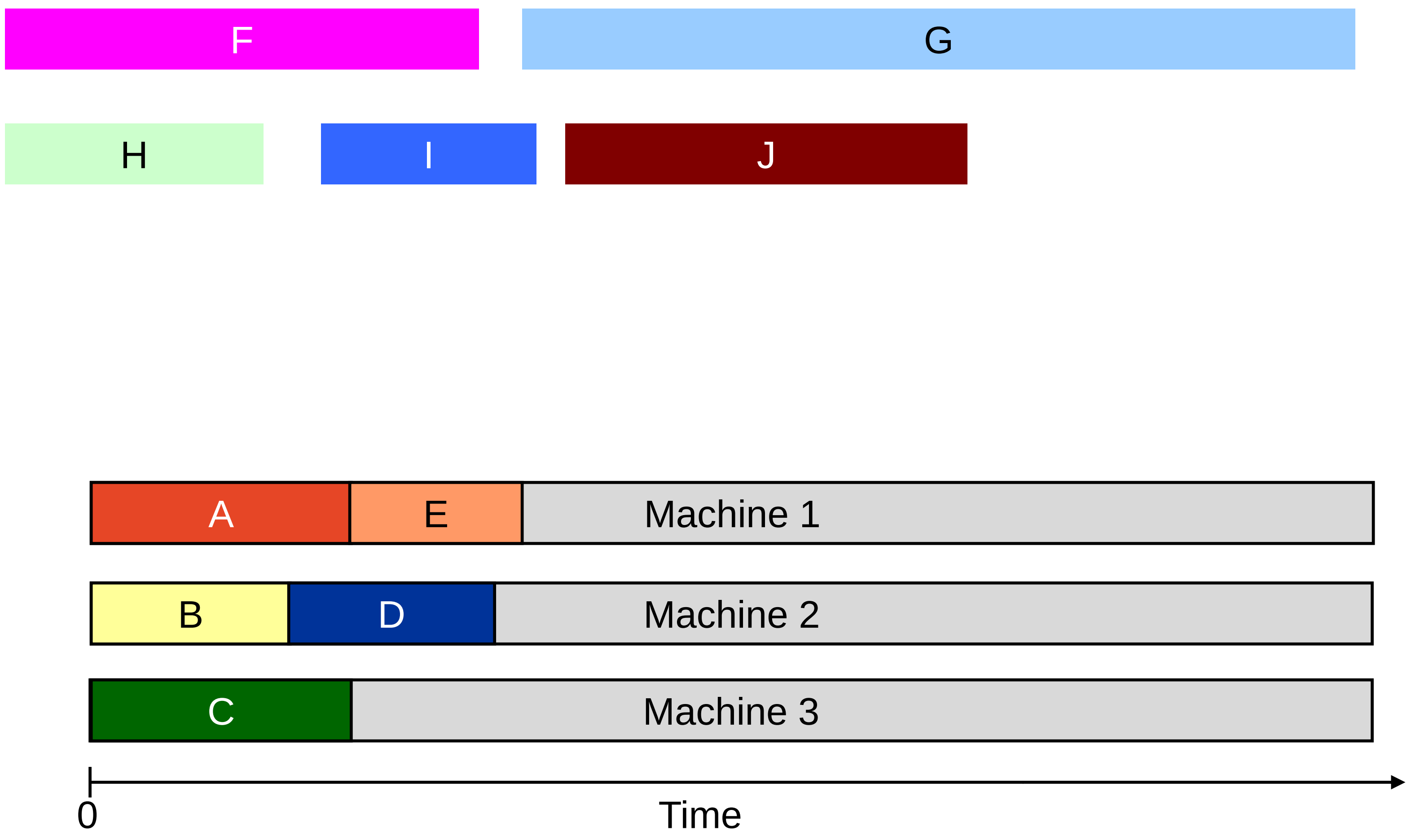
Load Balancing: List Scheduling



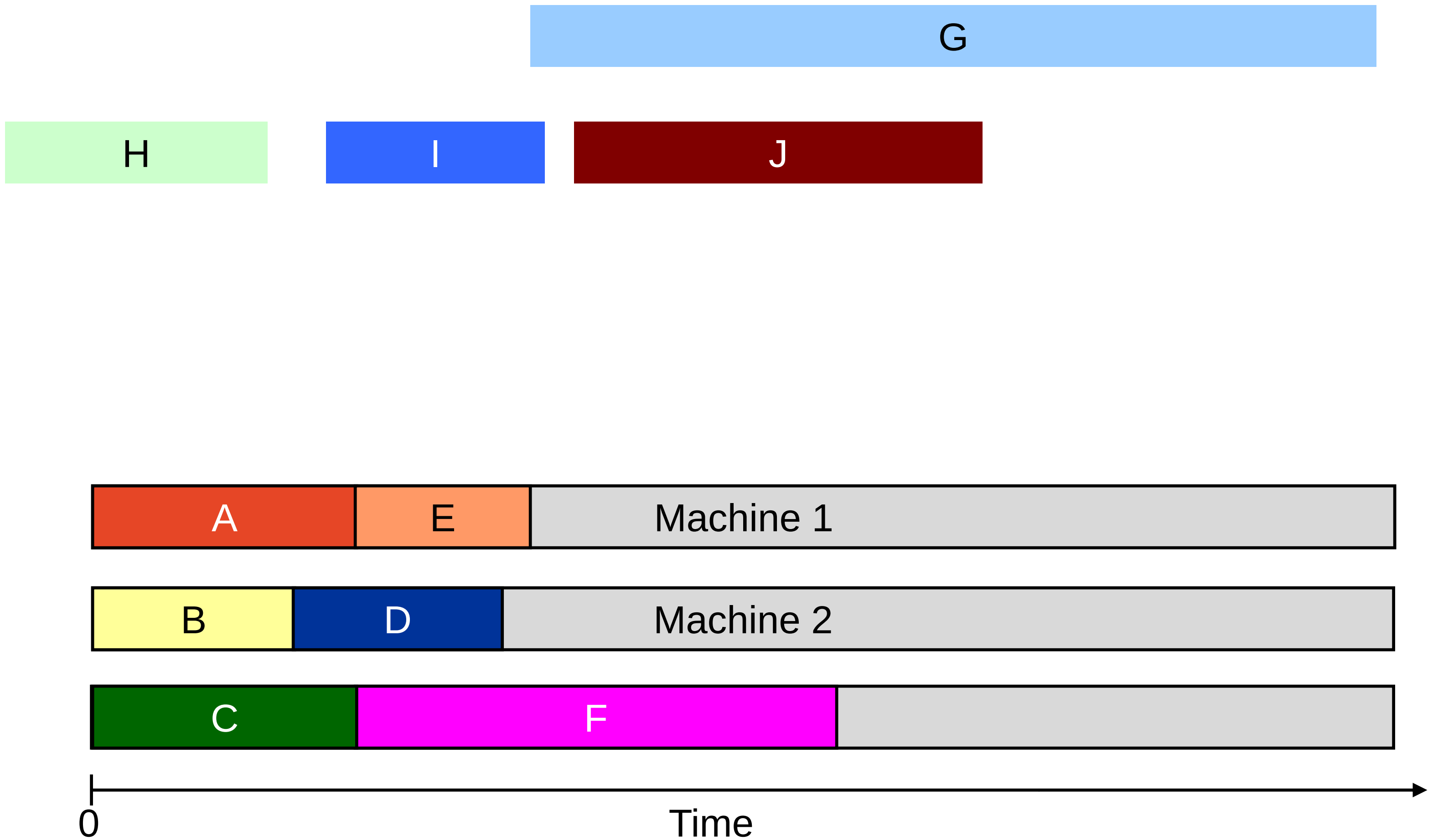
Load Balancing: List Scheduling



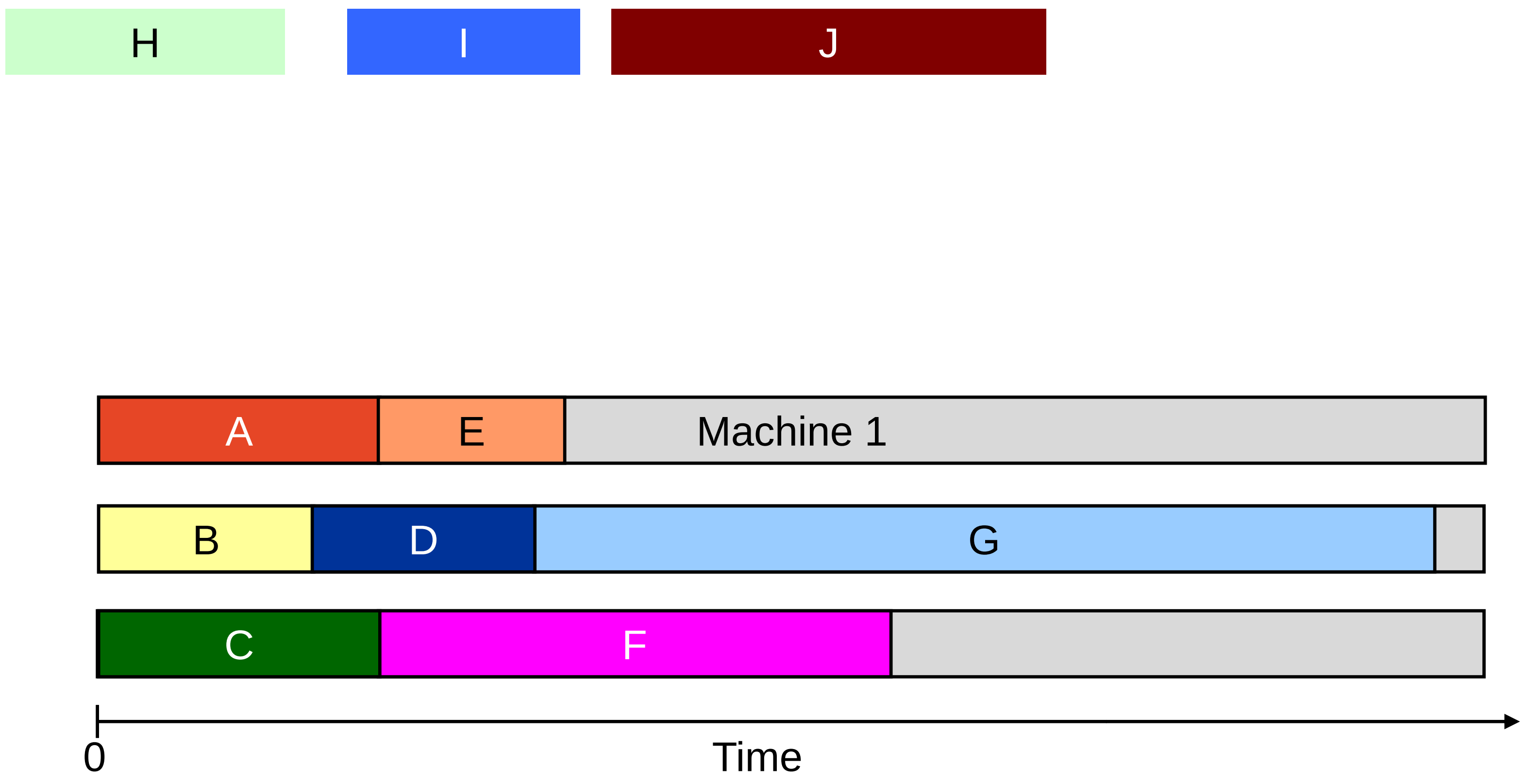
Load Balancing: List Scheduling



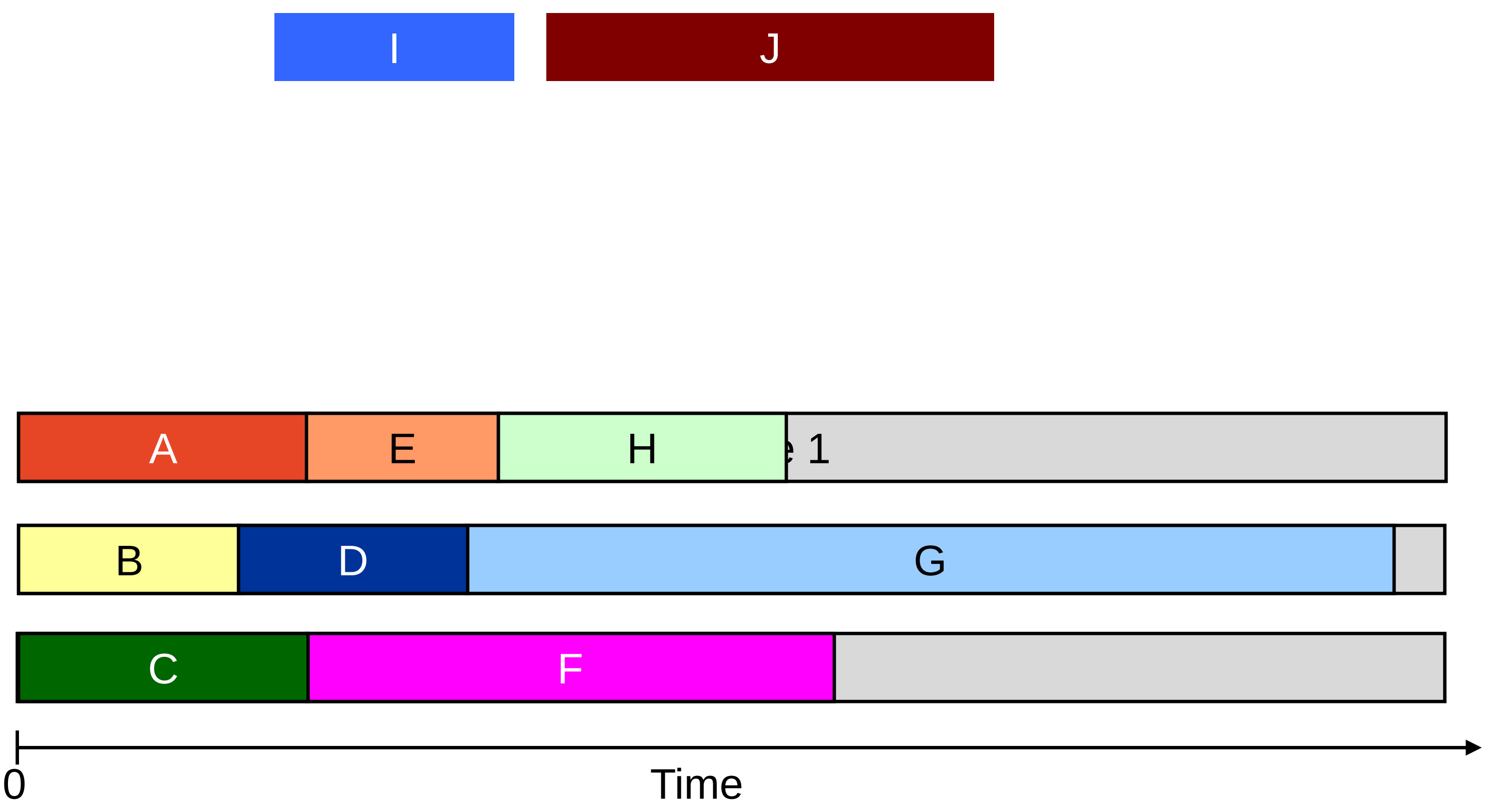
Load Balancing: List Scheduling



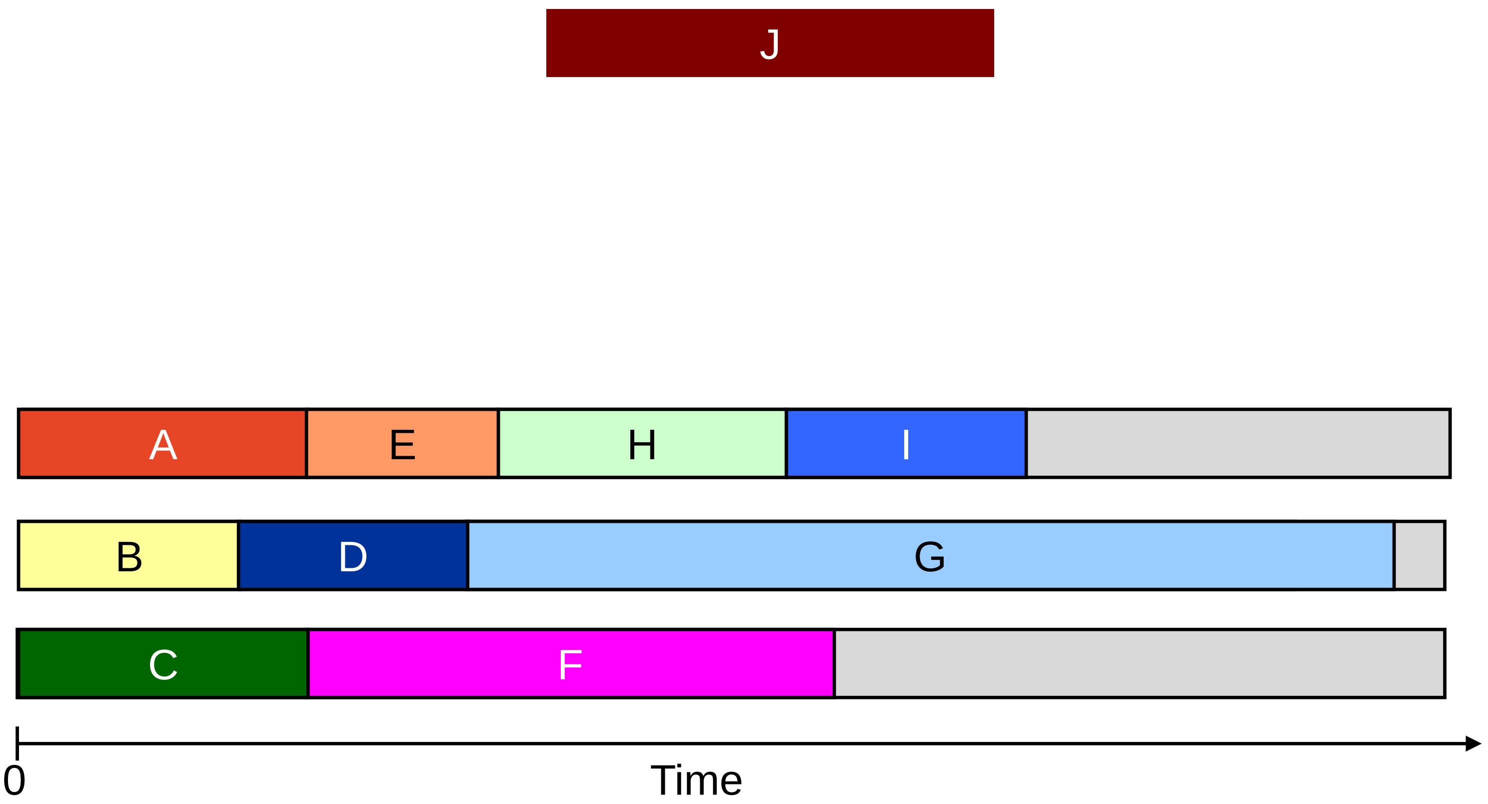
Load Balancing: List Scheduling



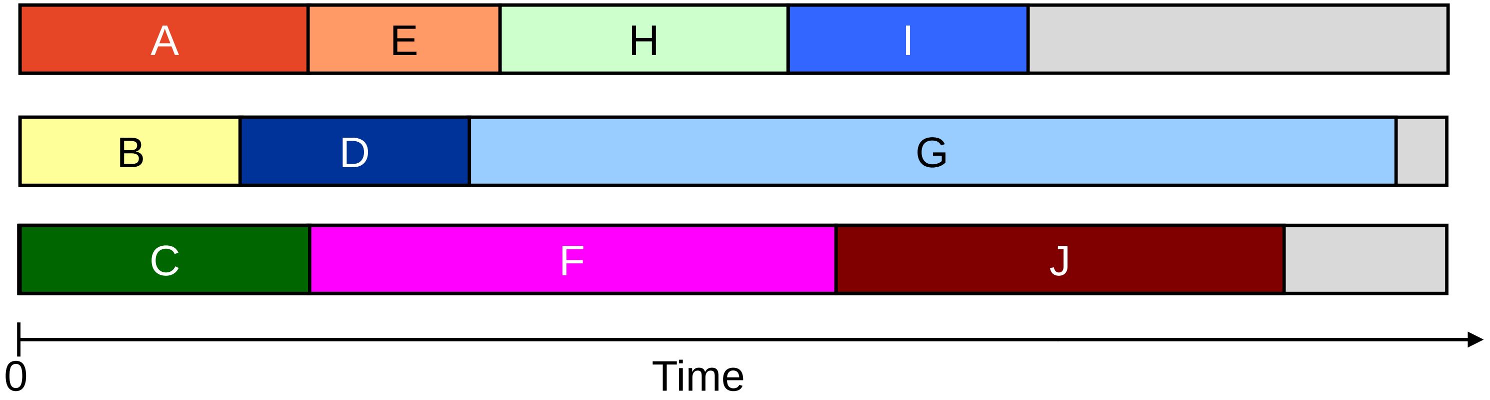
Load Balancing: List Scheduling



Load Balancing: List Scheduling



Load Balancing: List Scheduling



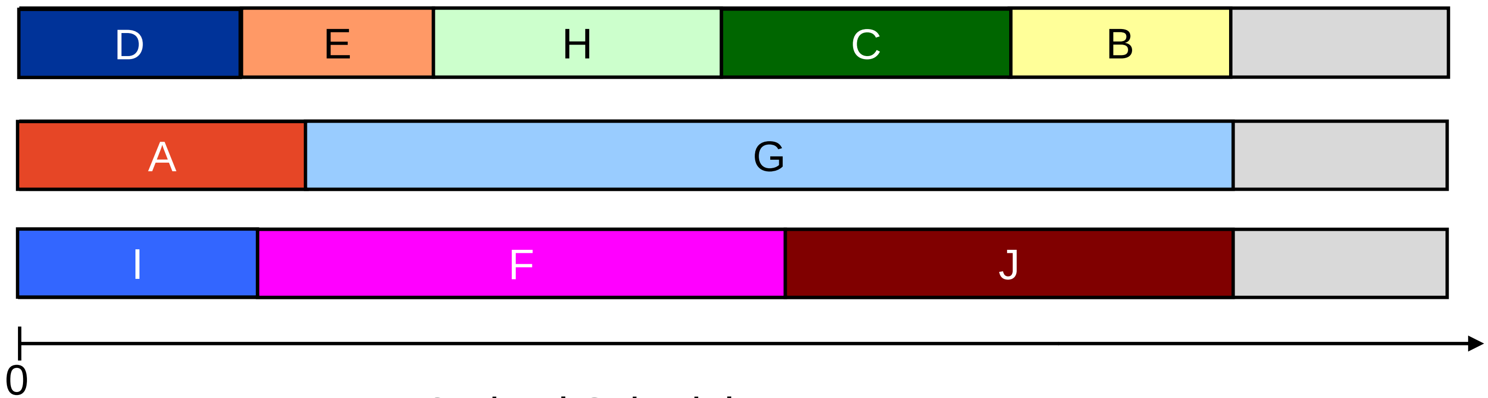
Is this a good schedule?

– The schedule may not be optimal (minimum makespan).

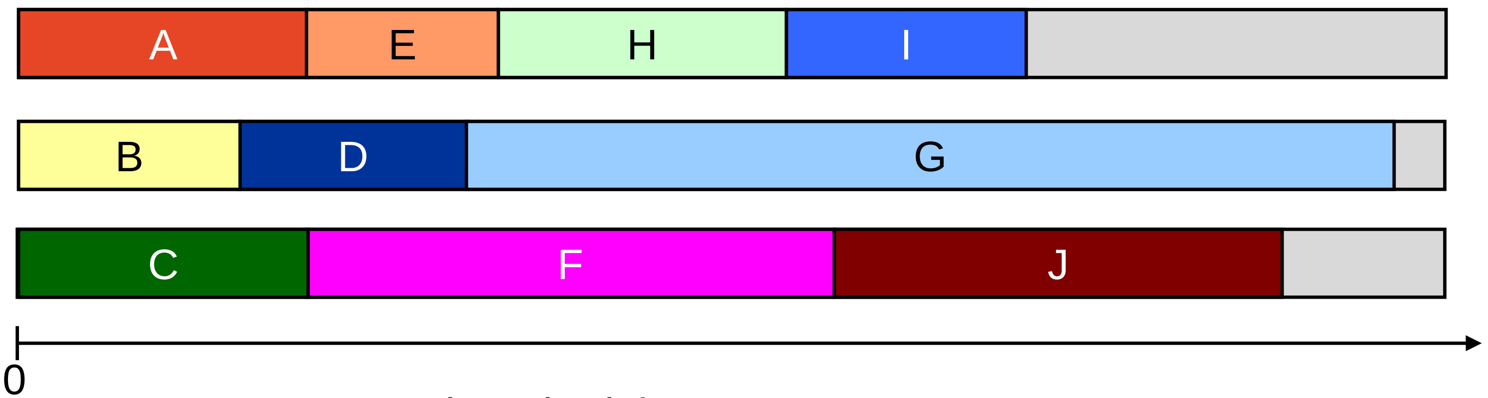
– How do we prove that statement?

– We only need to provide a counterexample.

Load Balancing: List Scheduling



Optimal Schedule



List schedule

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*.

Lemma 1: The optimal makespan $L^* \geq \max_j t_j$.

Proof: Some machine must process the most time-consuming job. ▪

Lemma 2: The optimal makespan $L^* \geq \frac{1}{m} \sum_j t_j$.

Proof:

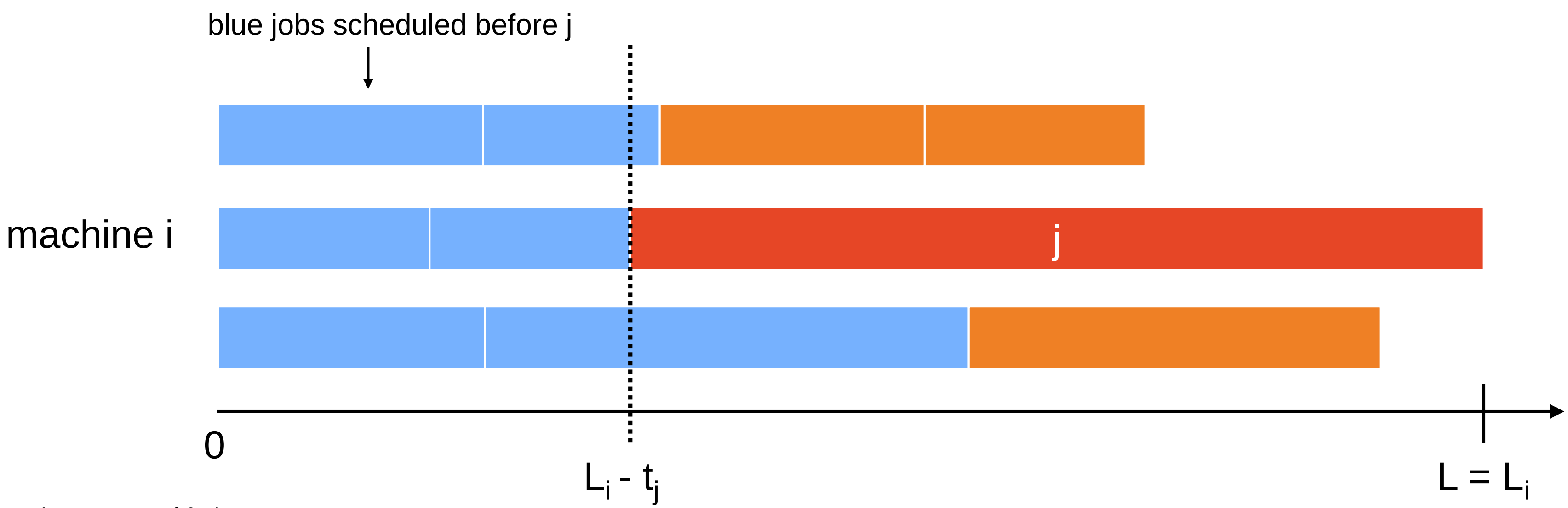
- The total processing time is $\Sigma_j t_j$.
- One of m machines must do at least a 1/m fraction of total work. ▪

Load Balancing: List Scheduling Analysis

Theorem: Greedy algorithm is a 2-approximation.

Proof: Consider load L_i of bottleneck machine i.

- Let j be last job scheduled on machine i.
- When job j assigned to machine i, i had smallest load. Its load before assignment is $L_i - t_j \Rightarrow L_i - t_j \leq L_k$ for all $1 \leq k \leq m$.



Load Balancing: List Scheduling Analysis

Theorem: Greedy algorithm is a 2-approximation.

Proof: Consider load L_i of bottleneck machine i.

- Let j be last job scheduled on machine i.
- When job j was assigned to machine i, i had smallest load. Its load before assignment is $L_i - t_j \Rightarrow L_i - t_j \leq L_k$ for all $1 \leq k \leq m$.
- Sum inequalities over all k and divide by m:

$$\begin{aligned} L_i - t_j &\leq \tfrac{1}{m} \sum_k L_k \\ &= \tfrac{1}{m} \sum_k t_k \\ \text{Lemma 1} \longrightarrow &\leq L^* \end{aligned}$$

- Now

$$L_i = \underbrace{(L_i - t_j)}_{\leq L^*} + \underbrace{t_j}_{\leq L^*} \leq 2L^*.$$

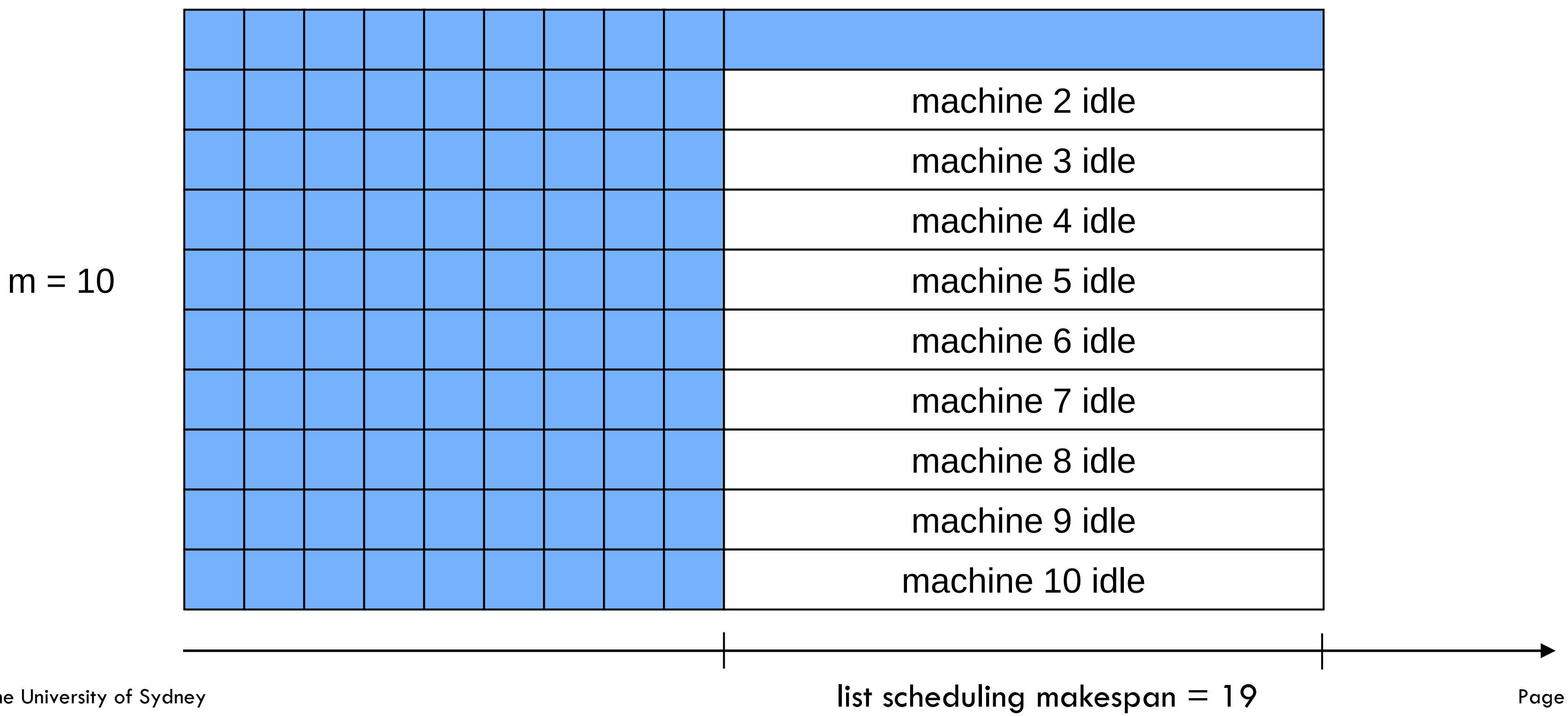
Lemma 2

Load Balancing: List Scheduling Analysis

Question: Is our analysis tight?

Answer: Yes…more or less.

Example: m machines, m(m-1) jobs of length 1, one job of length m



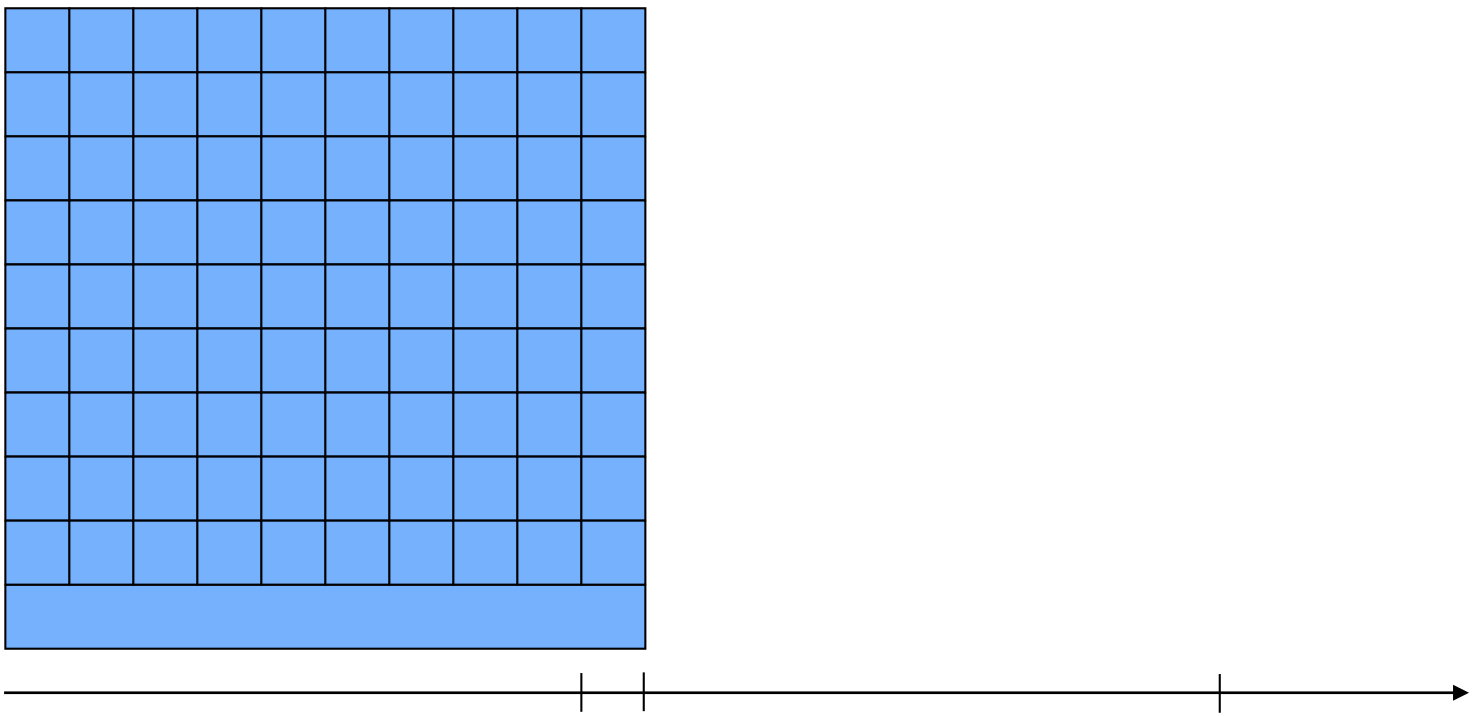
Load Balancing: List Scheduling Analysis

Question: Is our analysis tight?

Answer: Yes…more or less.

Example: m machines, m(m-1) jobs of length 1, one job of length m

m = 10



optimal makespan = 10

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.