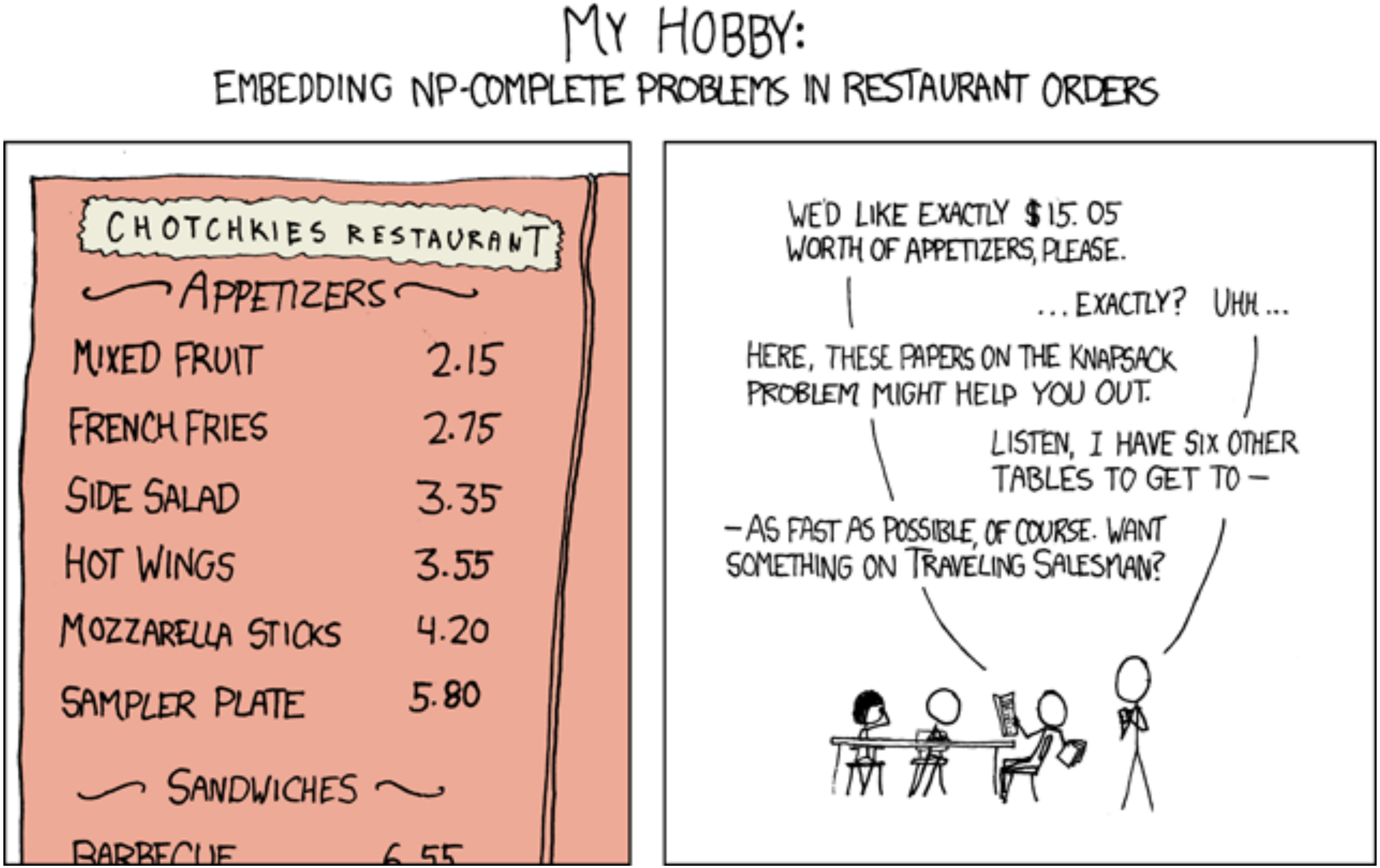
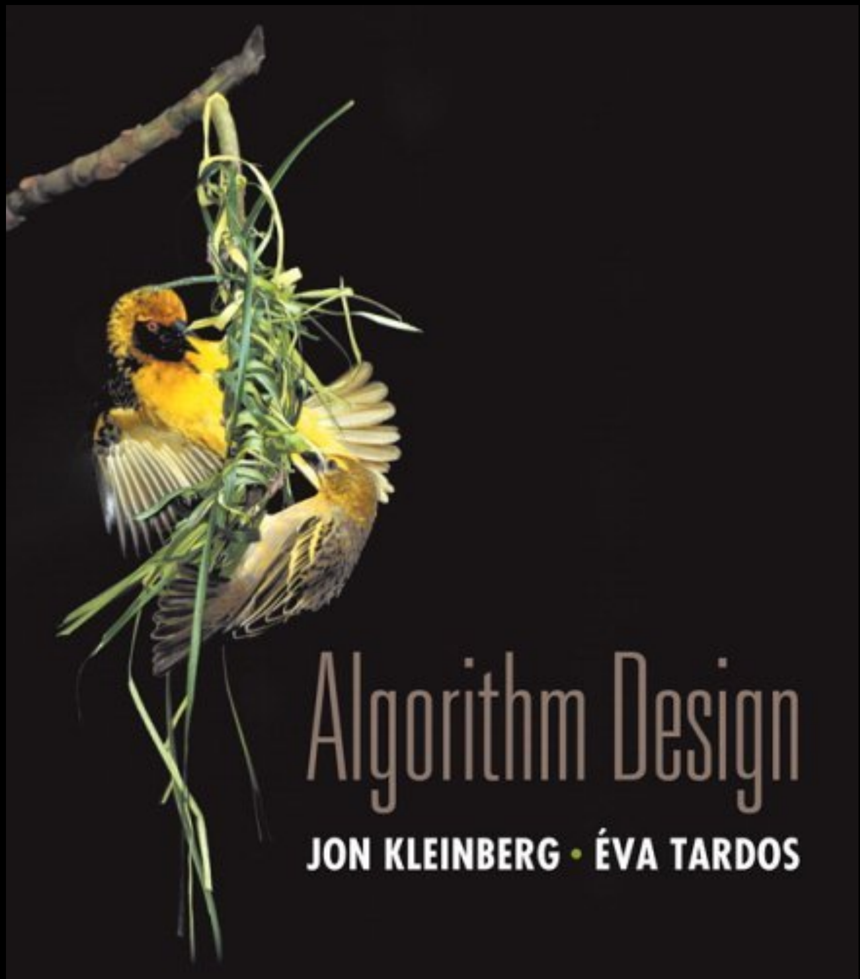

MY HOBBY:
EMBEDDING NP-COMPLETE PROBLEMS IN RESTAURANT ORDERS
CHOTCHKIES RESTAURANT
APPETIZERS
MIXED FRUIT 2.15
FRENCH FRIES 2.75
SIDE SALAD 3.35
HOT WINGS 3.55
MOZZARELLA STICKS 4.20
SAMPLER PLATE 5.80
SANDWICHES
WE'D LIKE EXACTLY $15.05 WORTH OF APPETIZERS, PLEASE.
...EXACTLY? UHH...
HERE, THESE PAPERS ON THE KNAPSACK PROBLEM MIGHT HELP YOU OUT.
LISTEN, I HAVE SIX OTHER TABLES TO GET TO –
– AS FAST AS POSSIBLE, OF COURSE. WANT SOMETHING ON TRAVELING SALESMAN?

xkcd.com/287



Chapter 11

Approximation Algorithms

Approximation Algorithms

Q. Suppose I need to solve an NP-hard problem. What should I do?
A. Theory says you're unlikely to find a poly-time algorithm.

Must sacrifice one of three desired features.

- Solve problem in poly-time.
- Solve arbitrary instances of the problem.
- Solve problem to optimality.

ρ-approximation algorithm.

- Guaranteed to run in poly-time.
- Guaranteed to solve arbitrary instance of the problem
- Guaranteed to find solution within ratio ρ of true optimum.

Challenge. Need to prove a solution's value is close to optimum, without even knowing what optimum value is!

11.1 Load Balancing

Load Balancing

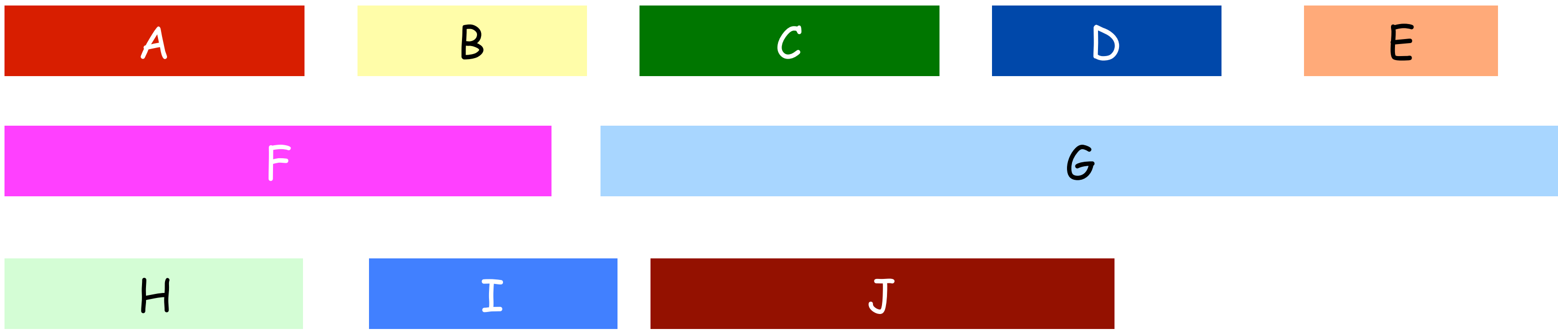
Input. m identical machines; n jobs, job j has processing time t_j.

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

Def. Let J(i) be the subset of jobs assigned to machine i. The load of machine i is $L_i = \Sigma_{j \in J(i)} t_j$.

Def. 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

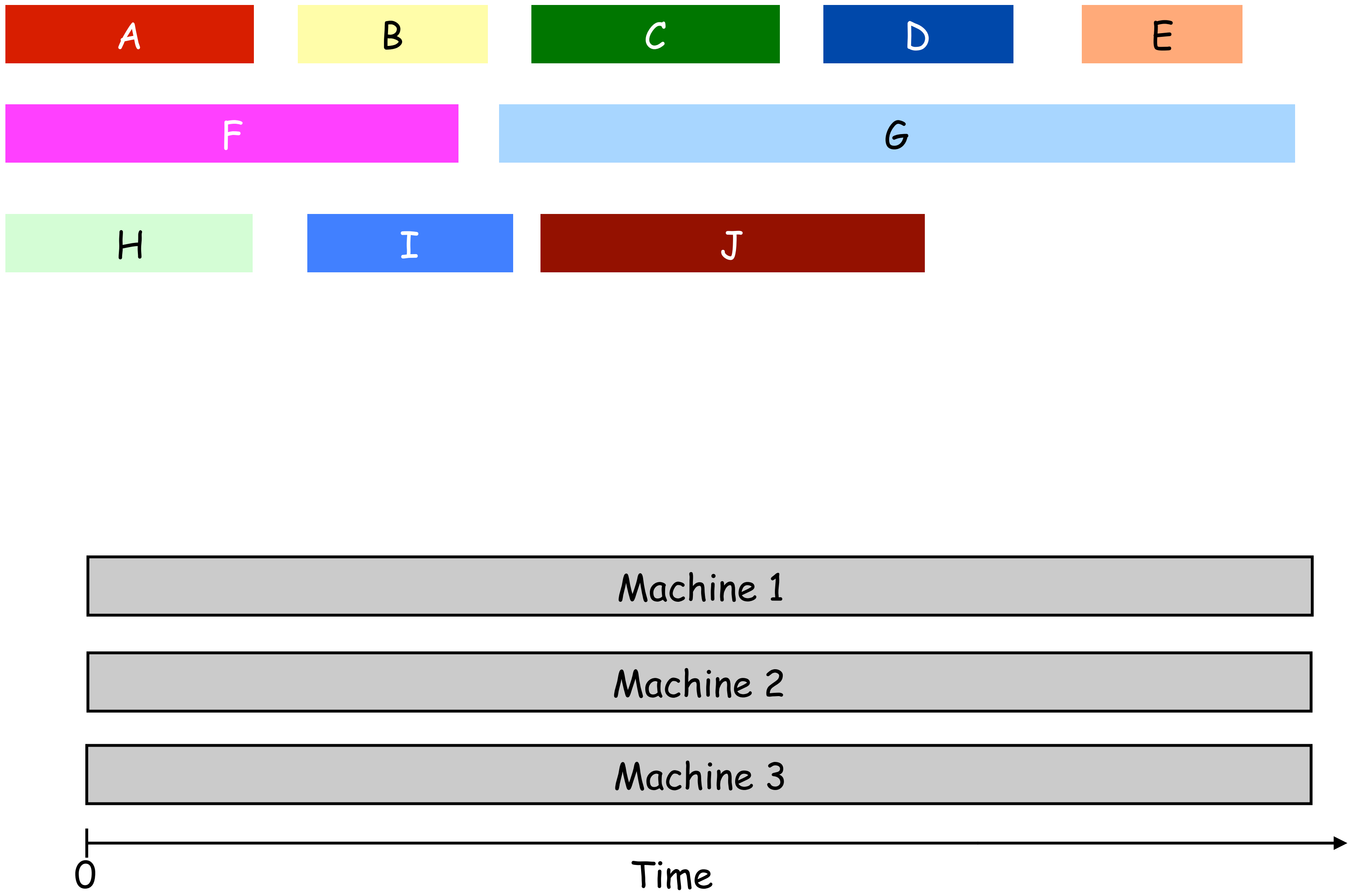
List-scheduling algorithm.

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

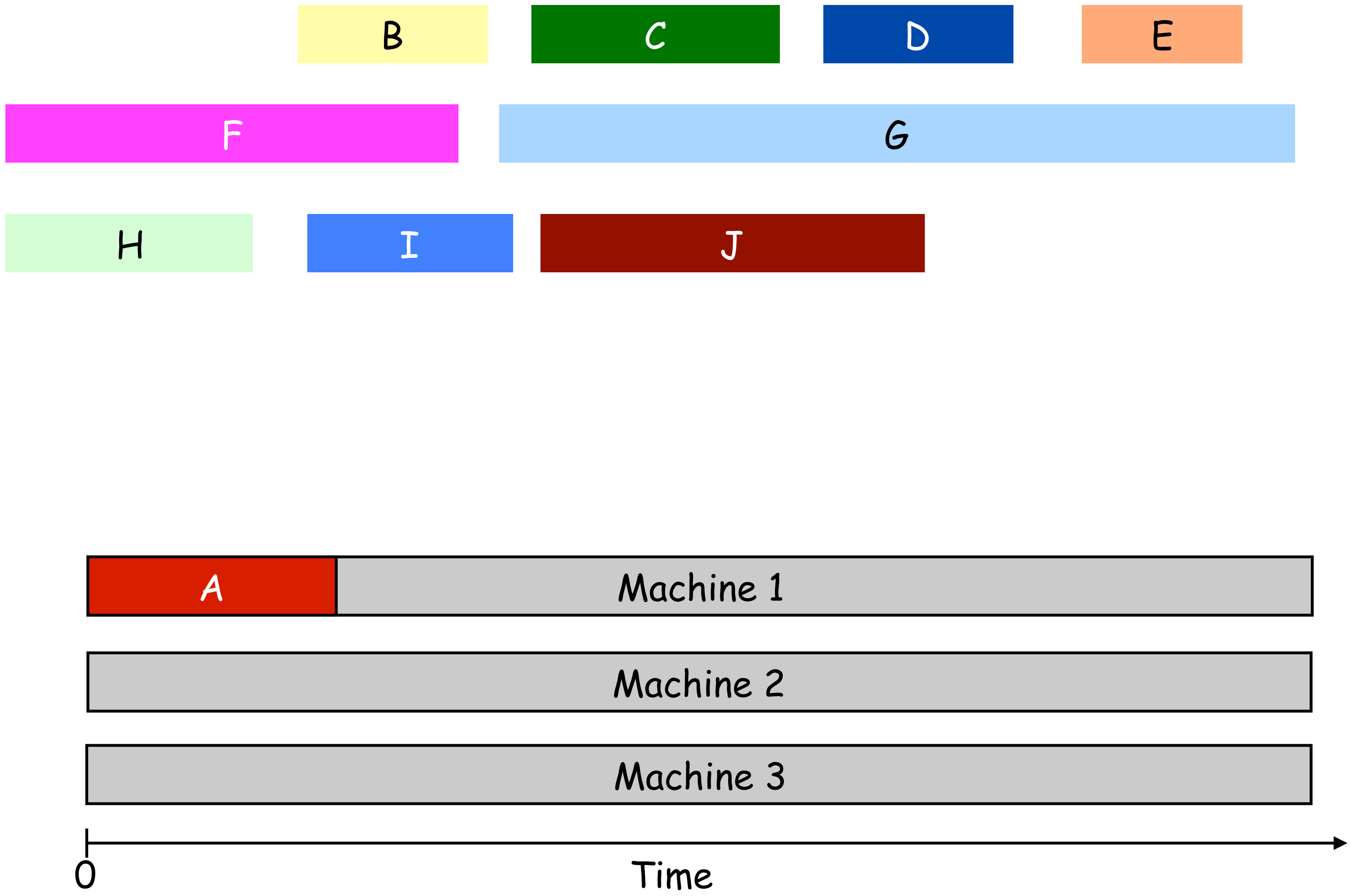
```
List-Scheduling(m, n, t_1,t_2,…,t_n) {
   for i = 1 to m {
      L_i ← 0          ← load on machine i
      J(i) ← φ         ← jobs assigned to machine i
   }

   for j = 1 to n {
      i = argmin_k L_k          ← machine i has smallest load
      J(i) ← J(i) ∪ {j}         ← assign job j to machine i
      L_i ← L_i + t_j           ← update load of machine i
   }
   return J(1), …, J(m)
}
```

Implementation. $O(n \log m)$ using a priority queue.

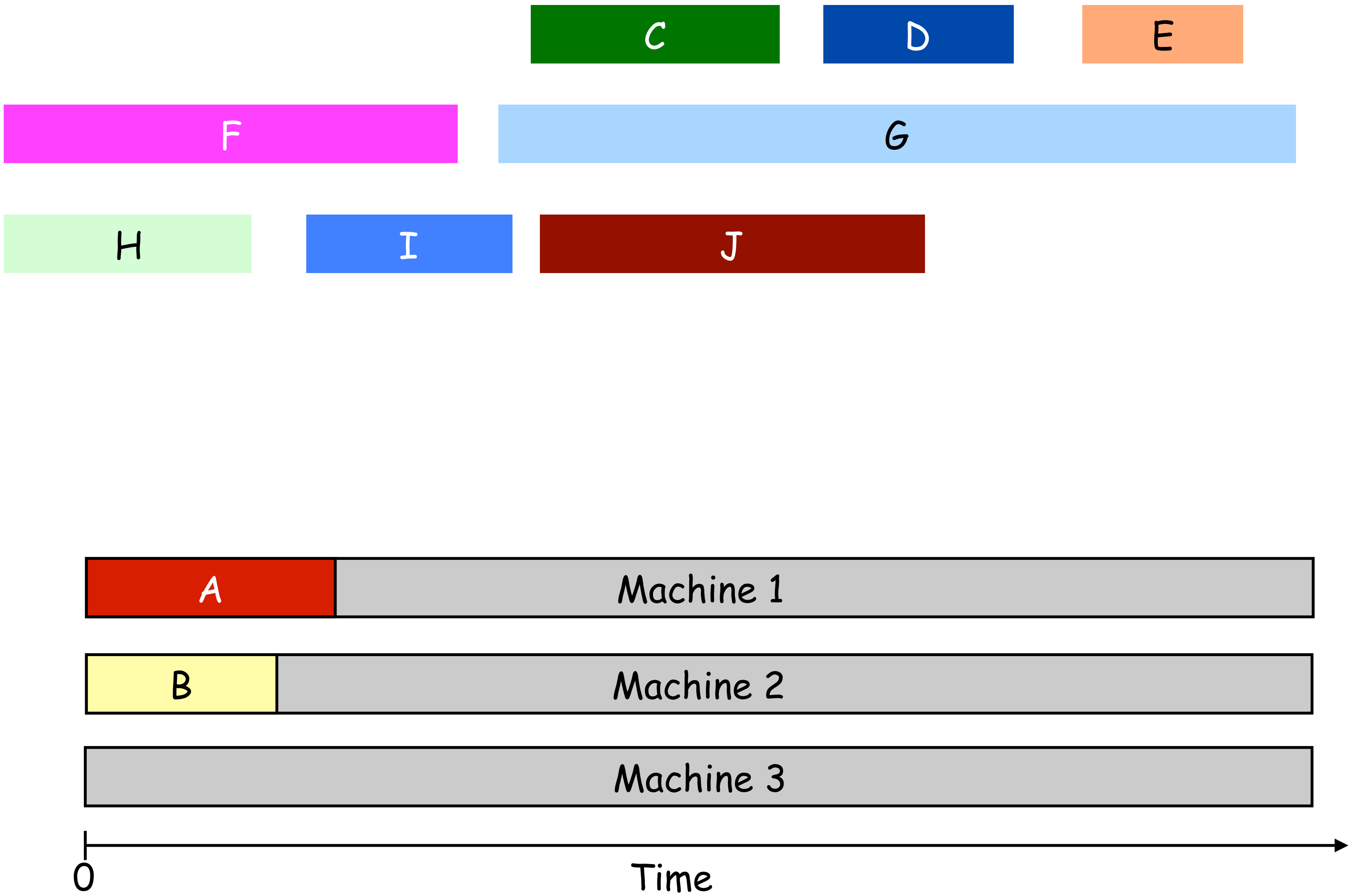
Load Balancing: List Scheduling



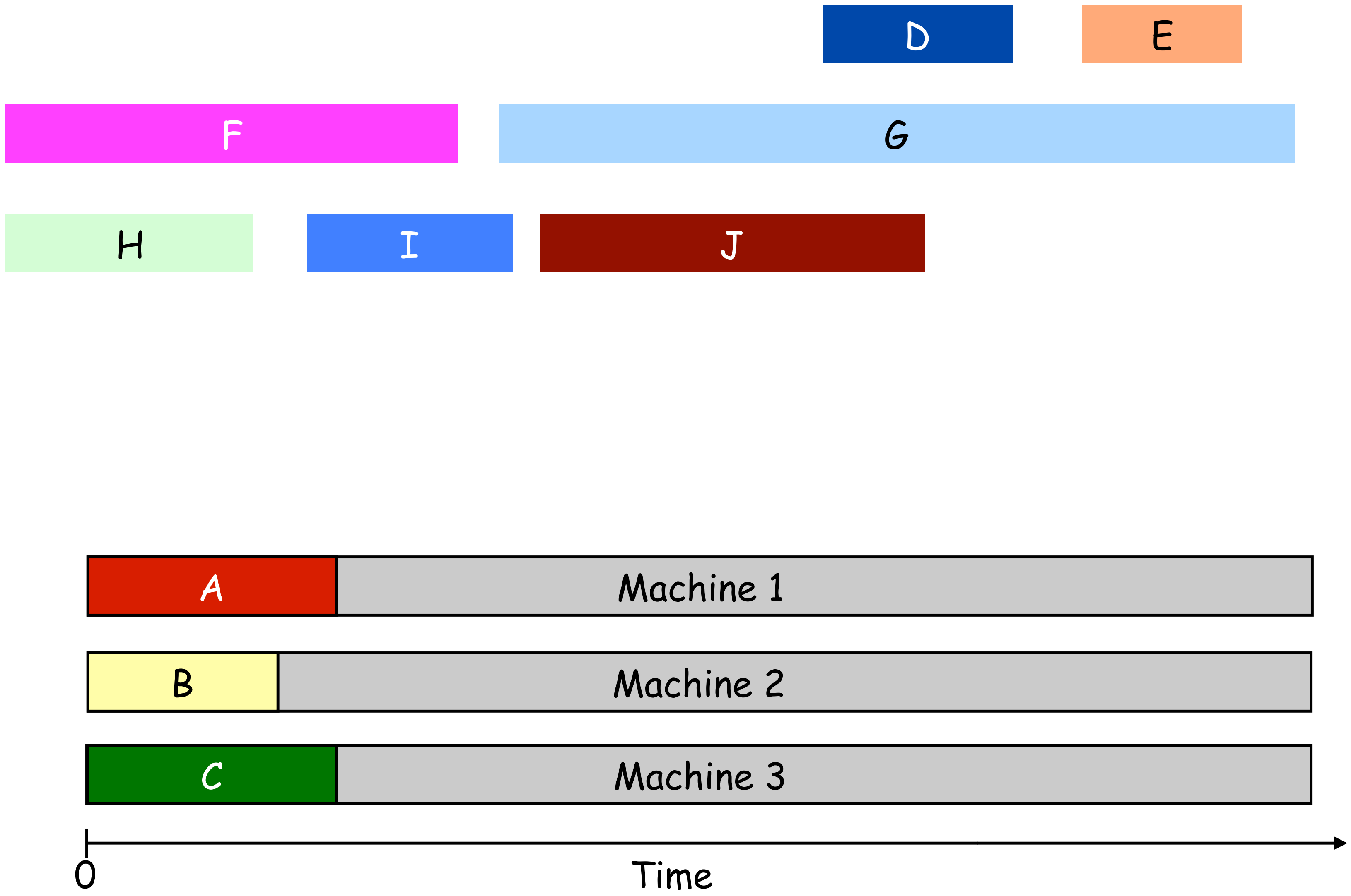
Load Balancing: List Scheduling



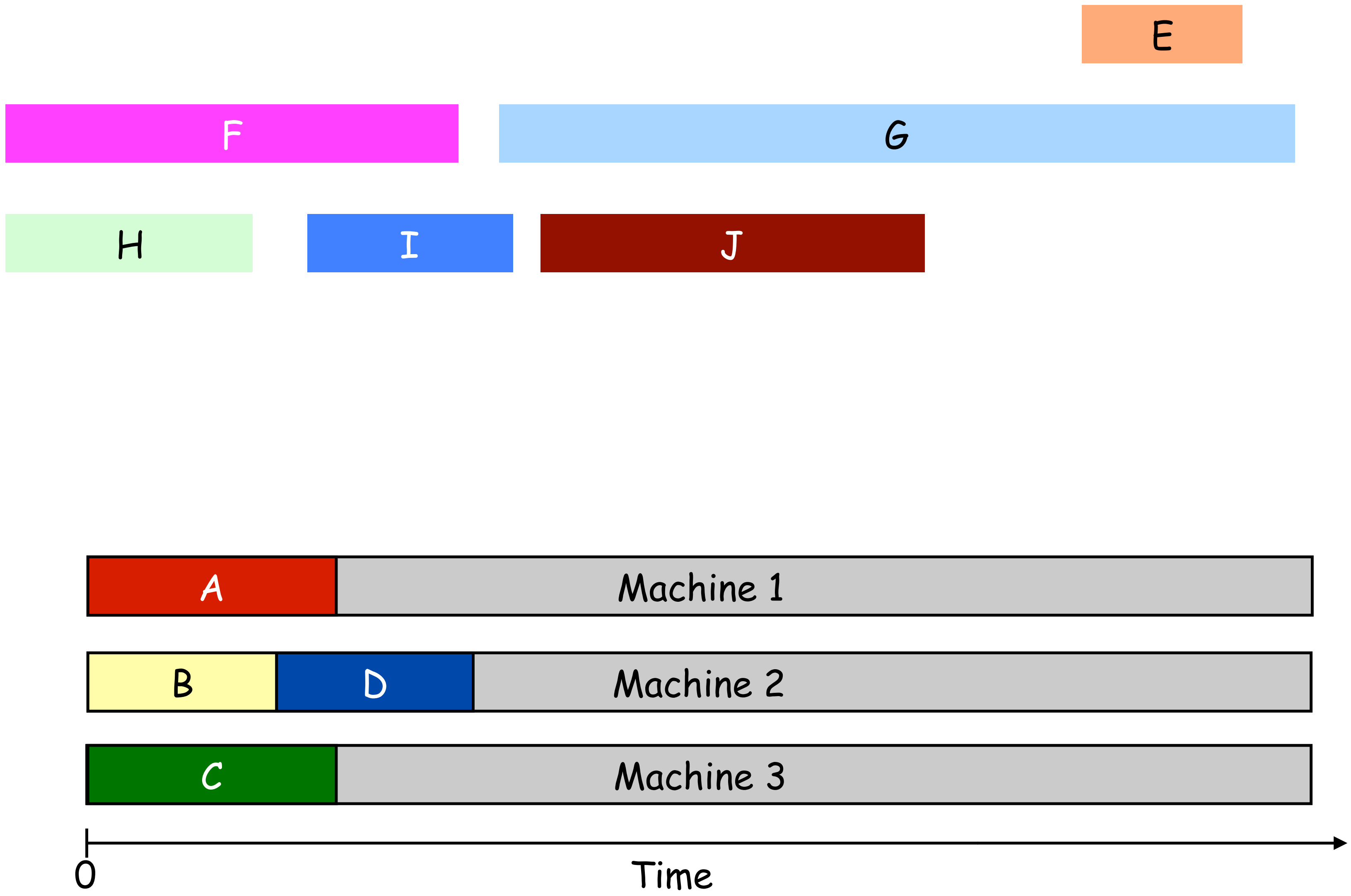
Load Balancing: List Scheduling



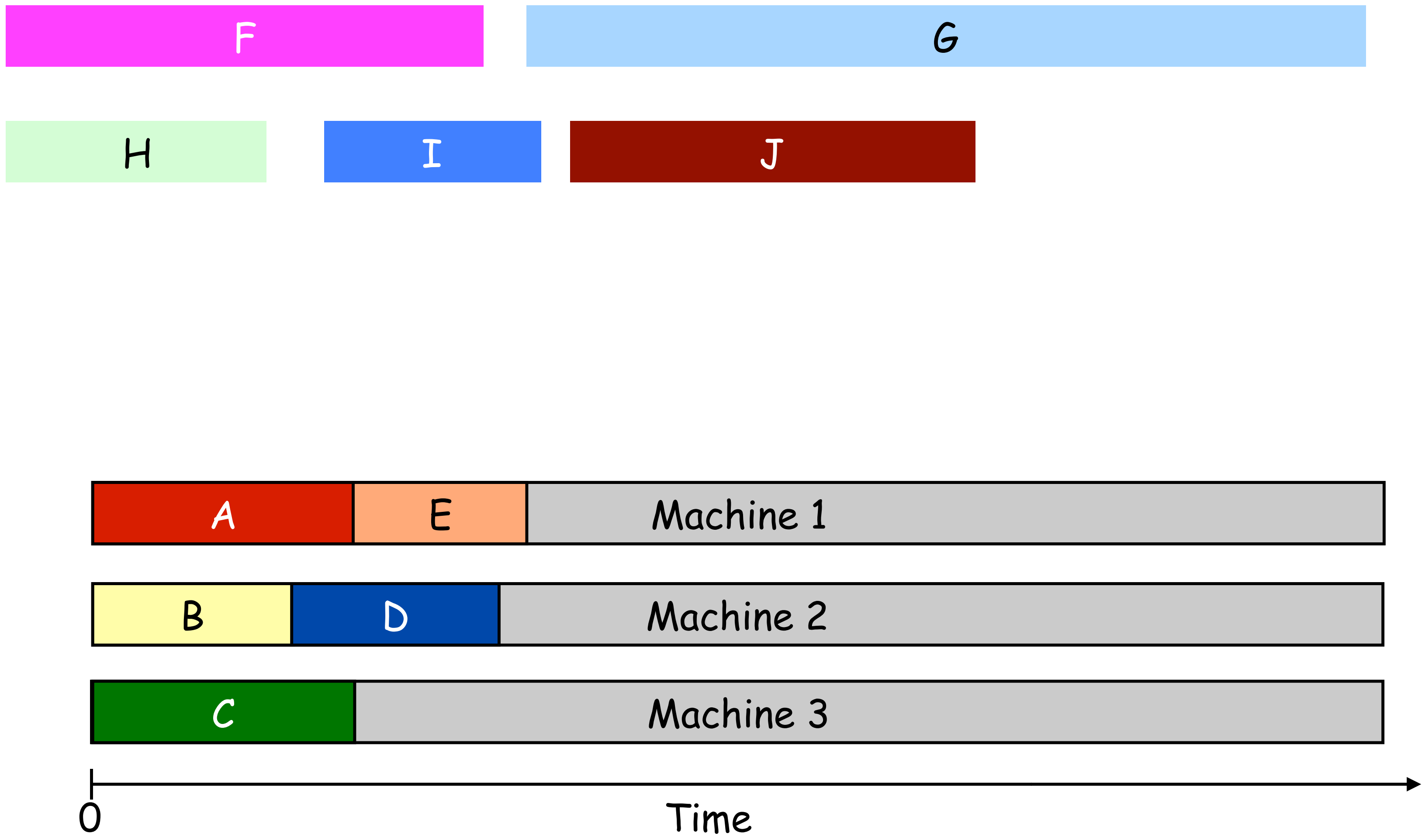
Load Balancing: List Scheduling



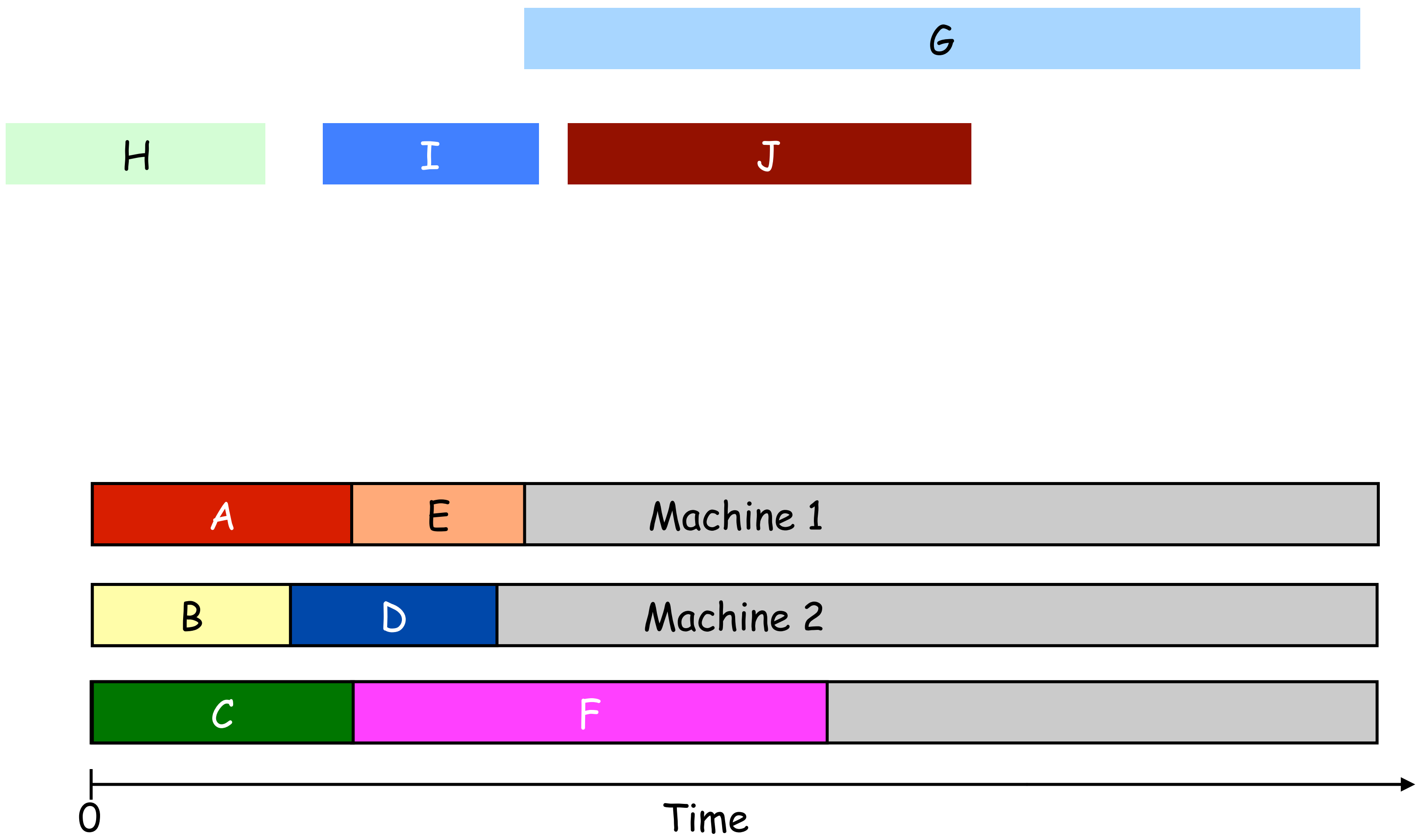
Load Balancing: List Scheduling



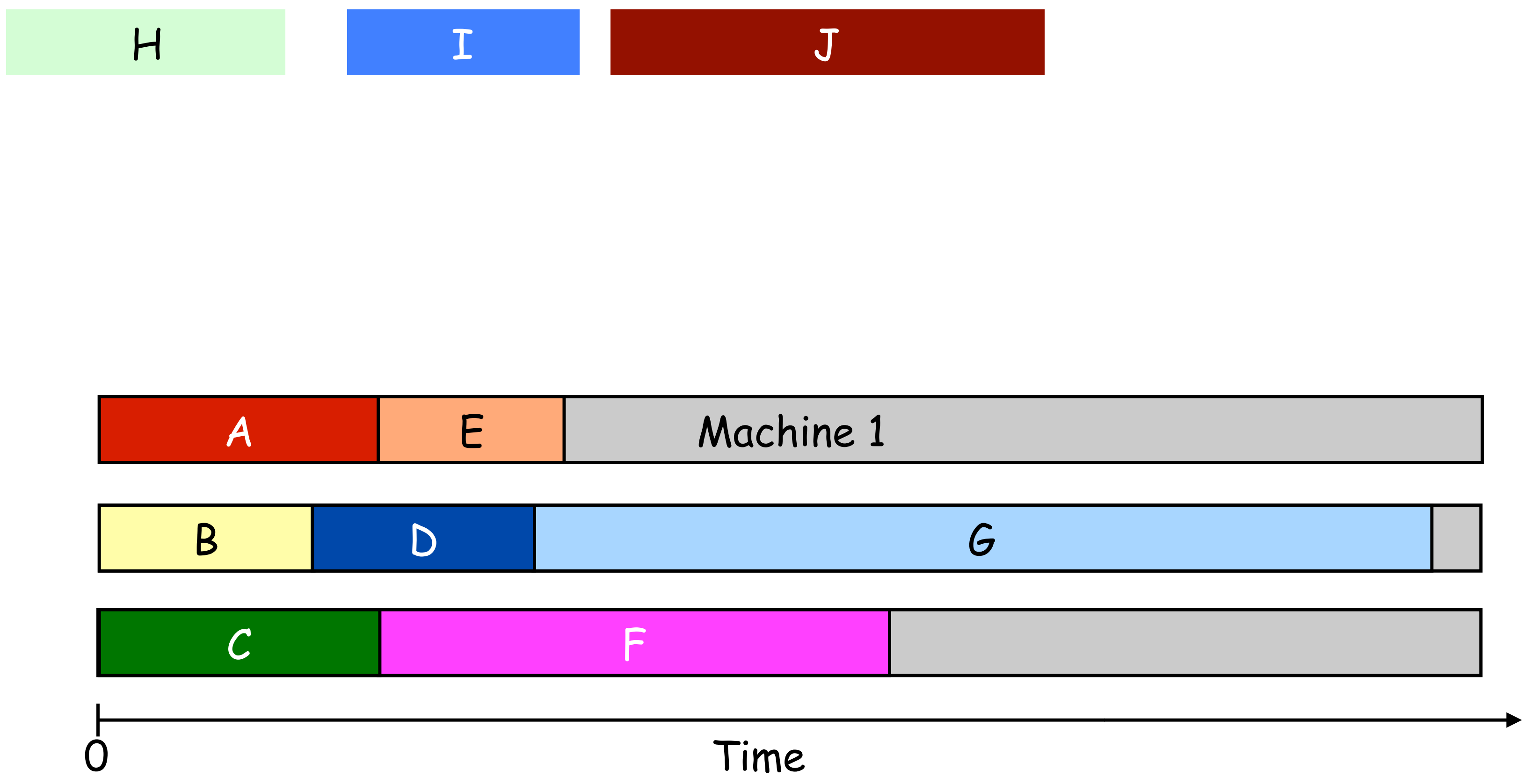
Load Balancing: List Scheduling



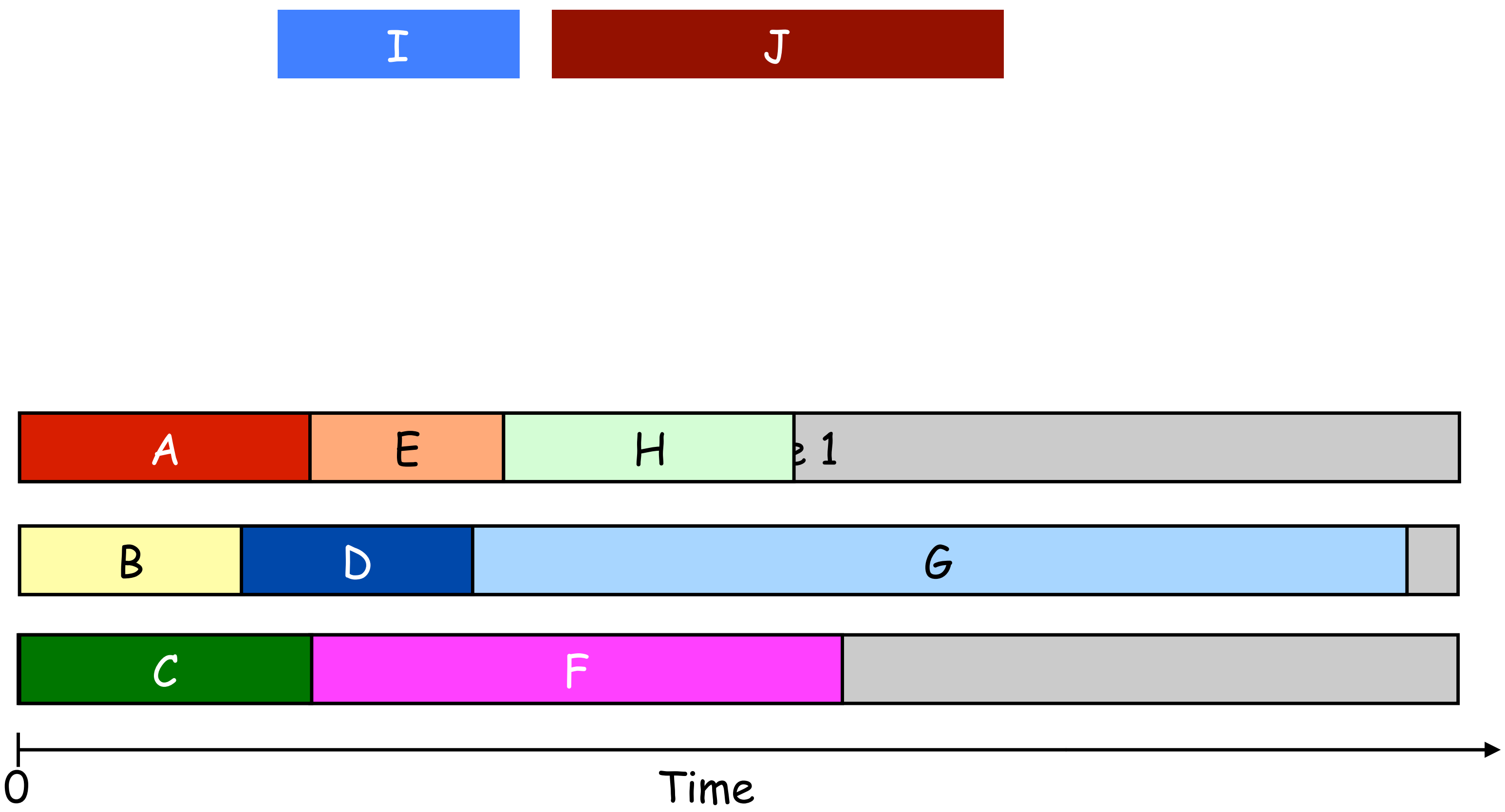
Load Balancing: List Scheduling



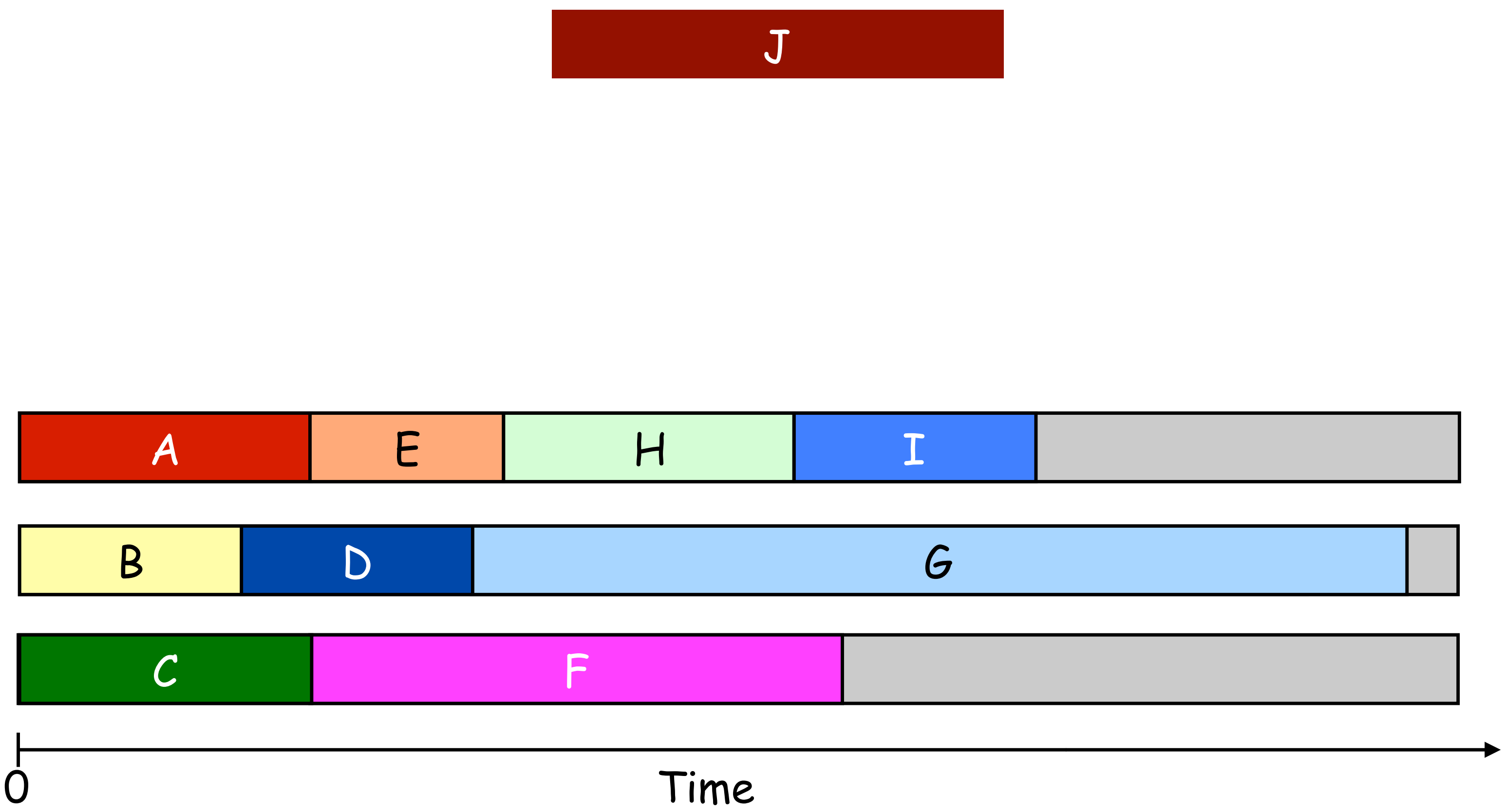
Load Balancing: List Scheduling



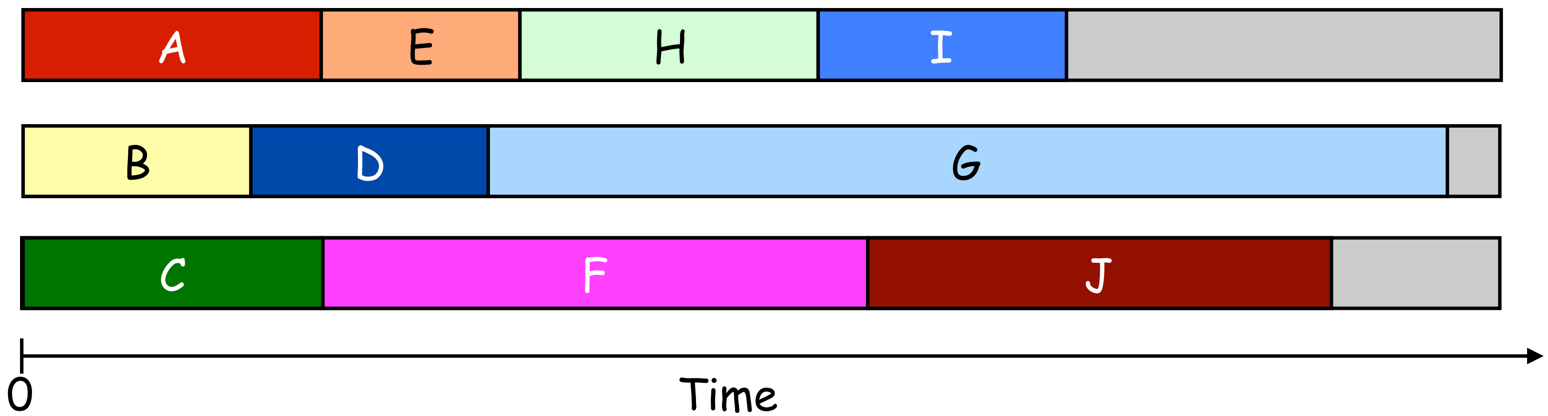
Load Balancing: List Scheduling



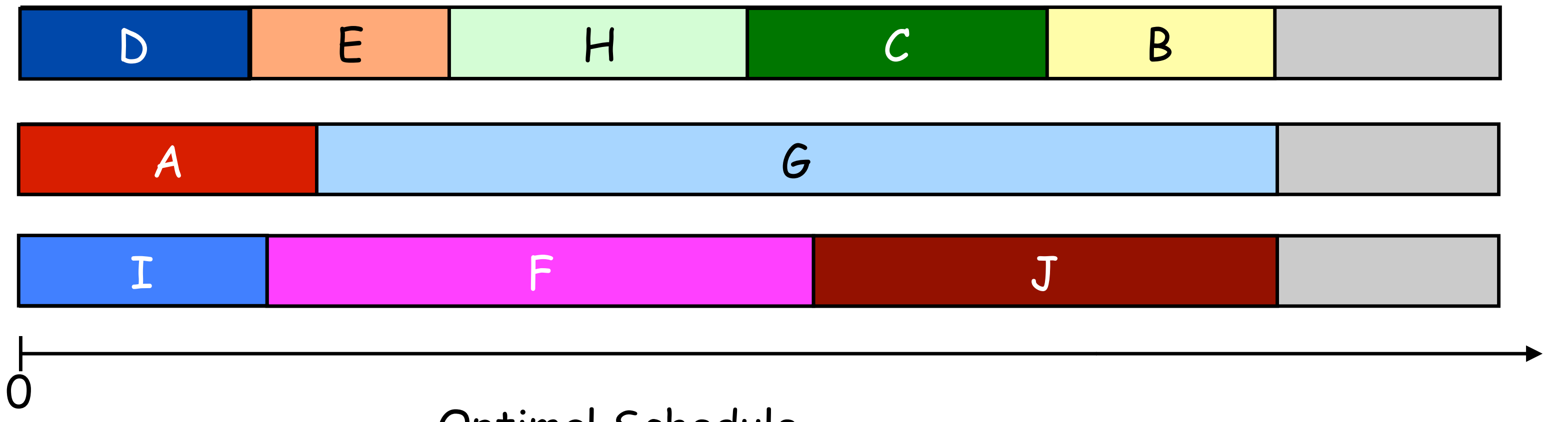
Load Balancing: List Scheduling



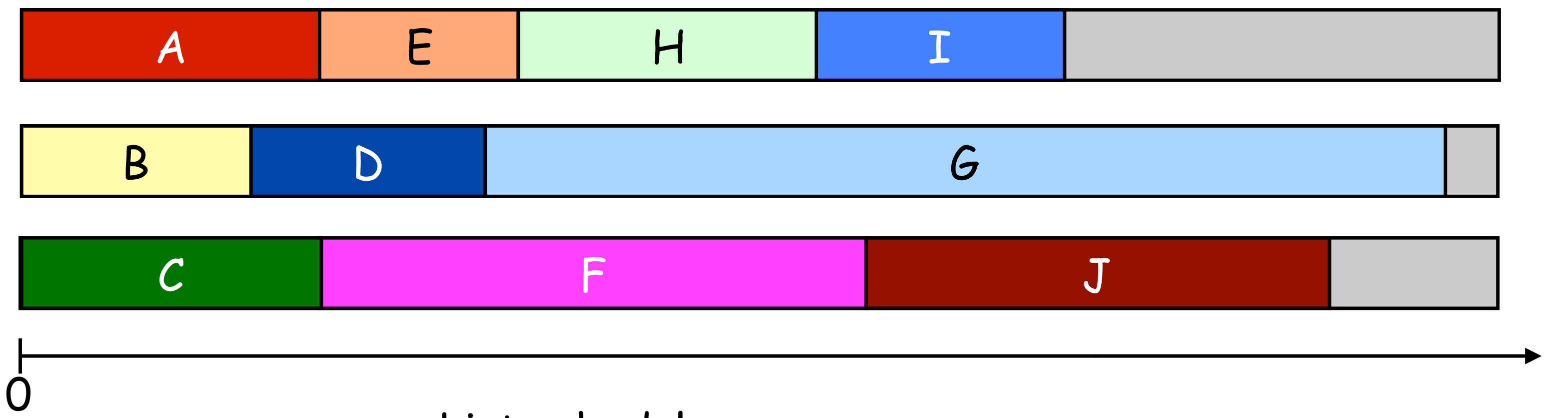
Load Balancing: List Scheduling



Load Balancing: List Scheduling



Optimal Schedule

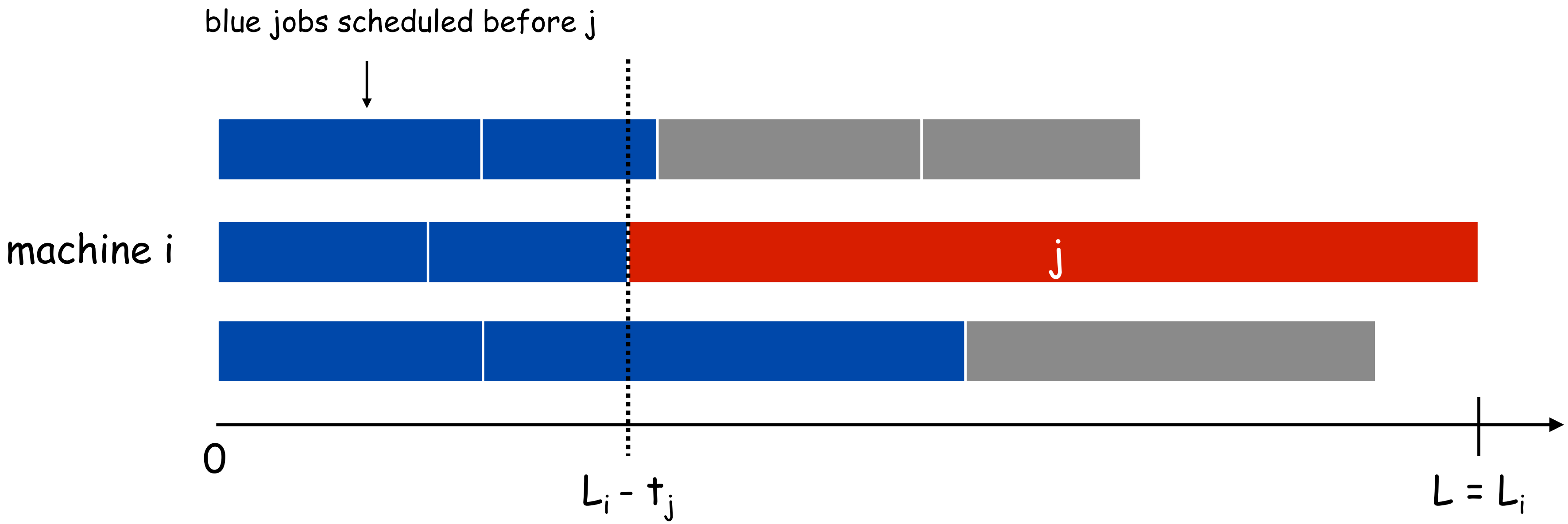


List schedule

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*.

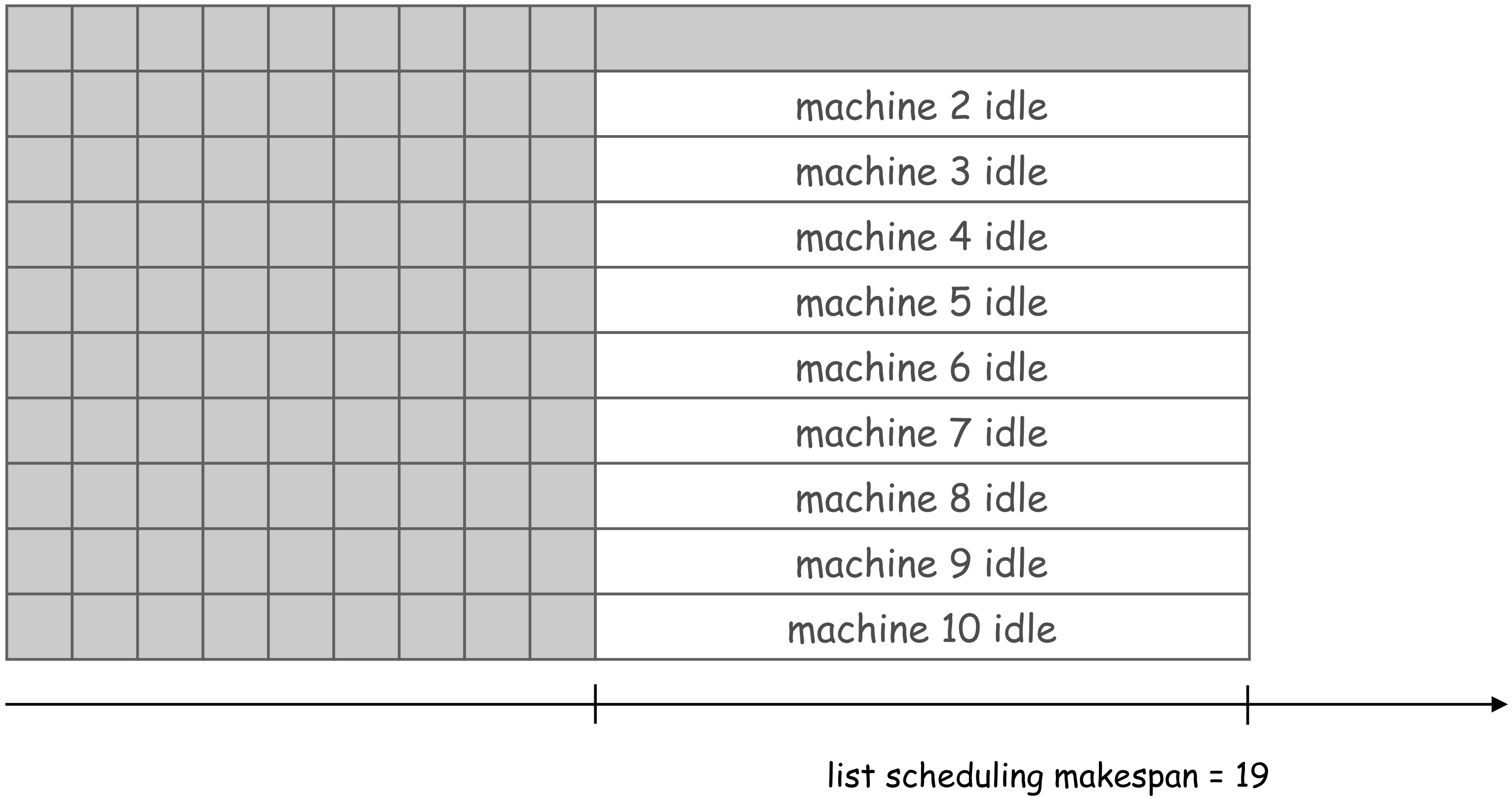


Load Balancing: List Scheduling Analysis

Q. Is our analysis tight?
A. Essentially yes.

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

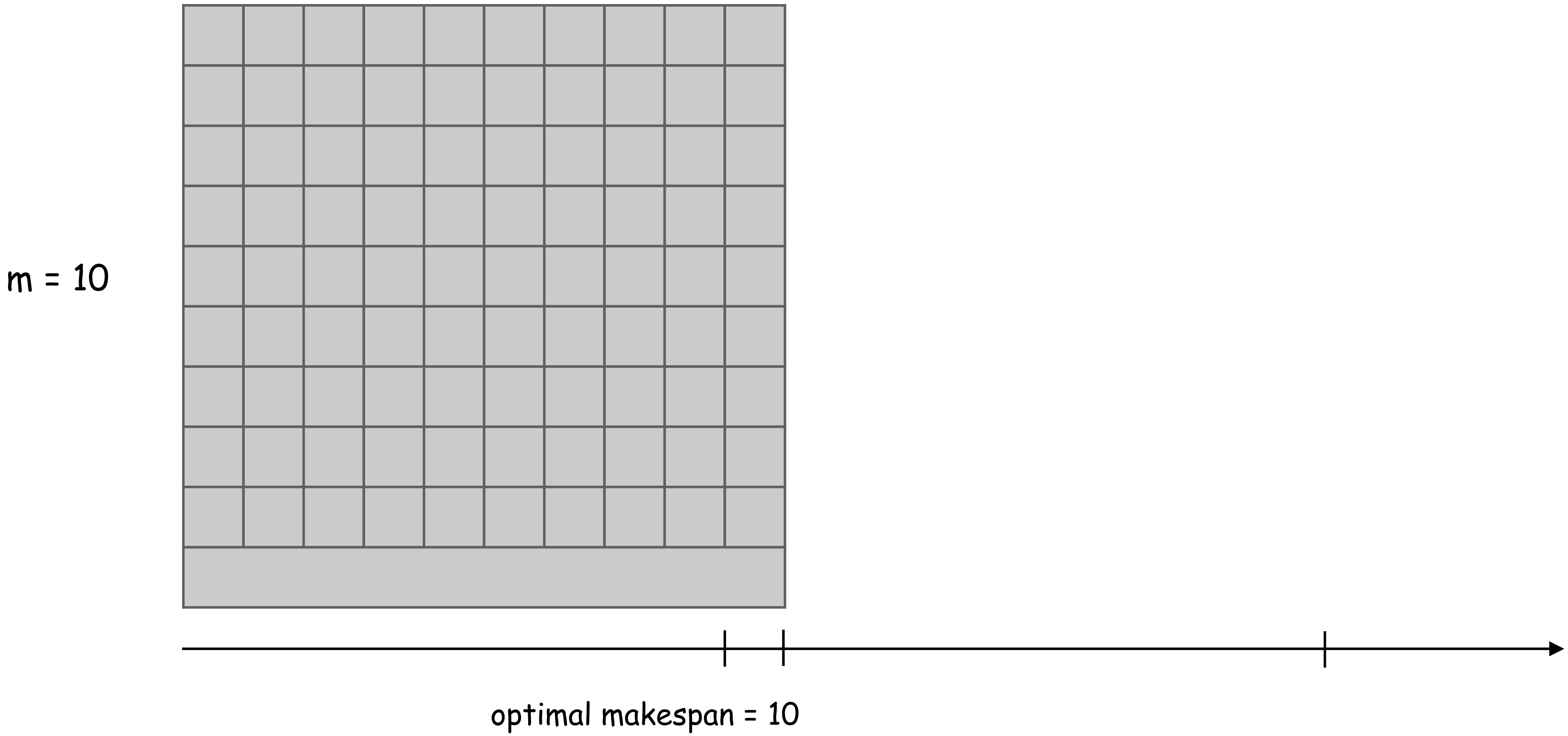
m = 10



Load Balancing: List Scheduling Analysis

Q. Is our analysis tight?
A. Essentially yes.

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



Load Balancing: LPT Rule

Longest processing time (LPT). Sort n jobs in descending order of processing time, and then run list scheduling algorithm.

```
LPT-List-Scheduling(m, n, t₁,t₂,…,tₙ) {
   Sort jobs so that t₁ ≥ t₂ ≥  … ≥ tₙ

   for i = 1 to m {
      Lᵢ ← 0               ← load on machine i
      J(i) ← φ             ← 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
   }
   return J(1), …, J(m)
}
```

Load Balancing: LPT Rule

Observation. If at most m jobs, then list-scheduling is optimal.
Pf. Each job put on its own machine. ▪

Lemma 3. If there are more than m jobs, $L^* \geq 2\, t_{m+1}$.
Pf.

- Consider first m+1 jobs $t_1, \ldots, t_{m+1}$.
- Since the t_i's are in descending order, each takes at least t_{m+1} time.
- There are m+1 jobs and m machines, so by pigeonhole principle, at least one machine gets two jobs. ▪

Theorem. LPT rule is a 3/2 approximation algorithm.
Pf. Same basic approach as for list scheduling.

$$L_i \;=\; \underbrace{(L_i - t_j)}_{\leq\, L^*} \;+\; \underbrace{t_j}_{\leq\, \frac{1}{2} L^*} \;\leq\; \tfrac{3}{2} L^*. \qquad ▪$$

Lemma 3
(by observation, can assume number of jobs > m)

Load Balancing: LPT Rule

Q. Is our 3/2 analysis tight?
A. No.

Theorem. [Graham, 1969] LPT rule is a 4/3-approximation.
Pf. More sophisticated analysis of same algorithm.

Q. Is Graham's 4/3 analysis tight?
A. Essentially yes.

Ex: m machines, n = 2m+1 jobs, 2 jobs of length m+1, m+2, …, 2m-1 and one job of length m.

Does there exist a better approximation algorithm?

A compendium of NP optimization problems

http://www.nada.kth.se/~viggo/problemlist/compendium.html

Results of the form:

- Best (theoretical) approximation algorithms and their dependence on approximation factor.
- Impossibility results like “cannot be 1.16 approximated unless P=NP”.

11.4 The Pricing Method: Vertex Cover

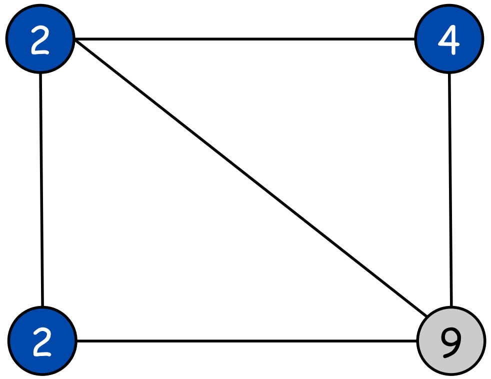
Last week:
Algorithm to decide if G has vertex cover of size $\leq k$ in $O(2^k\, kn)$ time.

What if there is no small vertex cover?

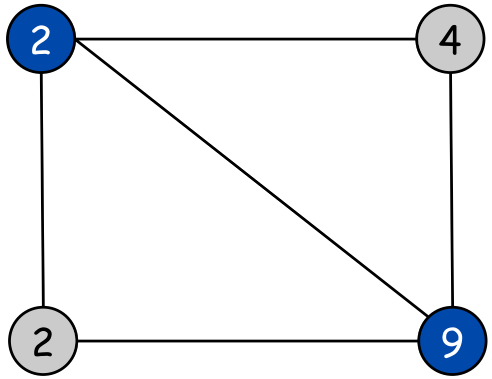
Weighted Vertex Cover

Definition. Given a graph $G = (V, E)$, a vertex cover is a set $S \subseteq V$ such that each edge in E has at least one end in S.

More general problem: Weighted vertex cover. Given a graph G with vertex weights, find a vertex cover of minimum weight.
Example instance: Vertex = worker; edge = task.

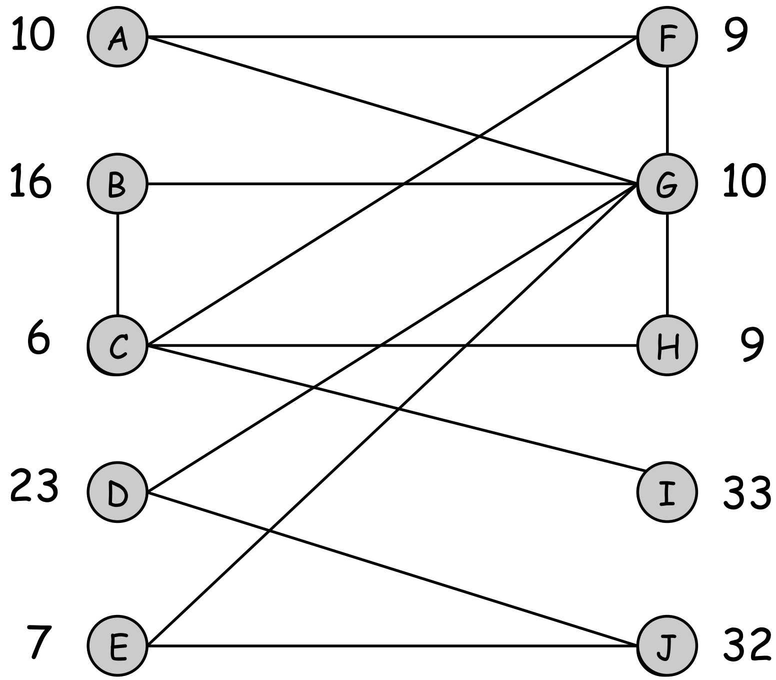


weight = 2 + 2 + 4



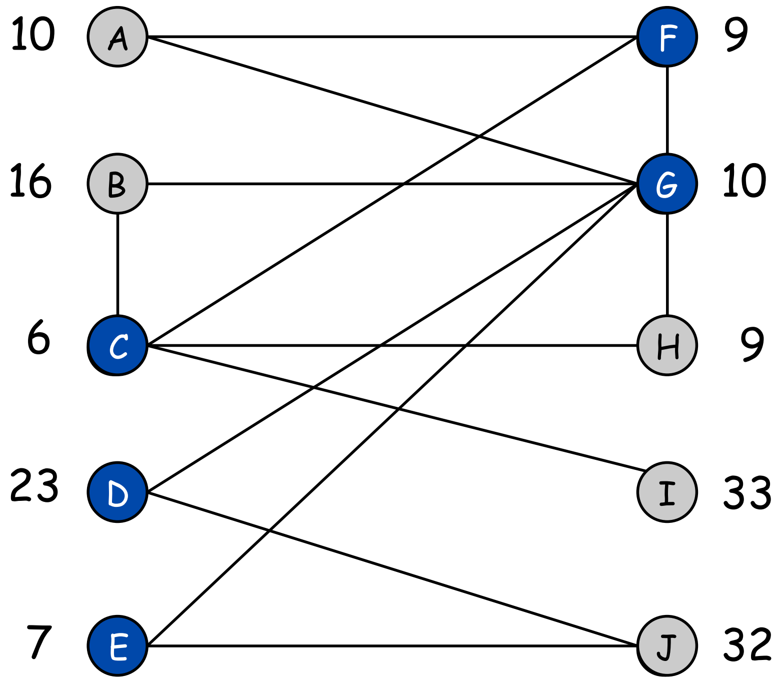
weight = 11

Example



total weight = 55

Min weight cover



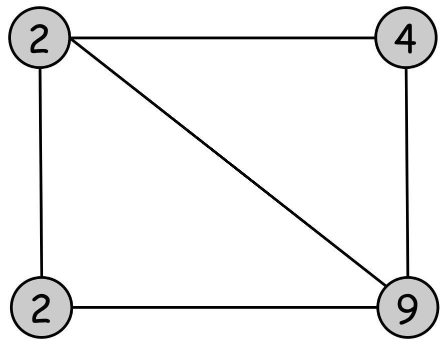
total weight = 55

Pricing Method

Pricing method. Each edge must be covered by some vertex.
Edge $e = (i, j)$ pays price $p_e \geq 0$ to use vertex i and j.

Fairness. Edges incident to vertex i should pay $\leq w_i$ in total.

$$\text{for each vertex } i: \sum_{e=(i,j)} p_e \leq w_i$$



Lemma. For any vertex cover S and any fair prices p_e: $\sum_e p_e \leq w(S)$.

Pf. ▪

$$\sum_{e \in E} p_e \leq \sum_{i \in S} \sum_{e=(i,j)} p_e \leq \sum_{i \in S} w_i = w(S).$$

each edge e covered by at least one node in S

sum fairness inequalities for each node in S

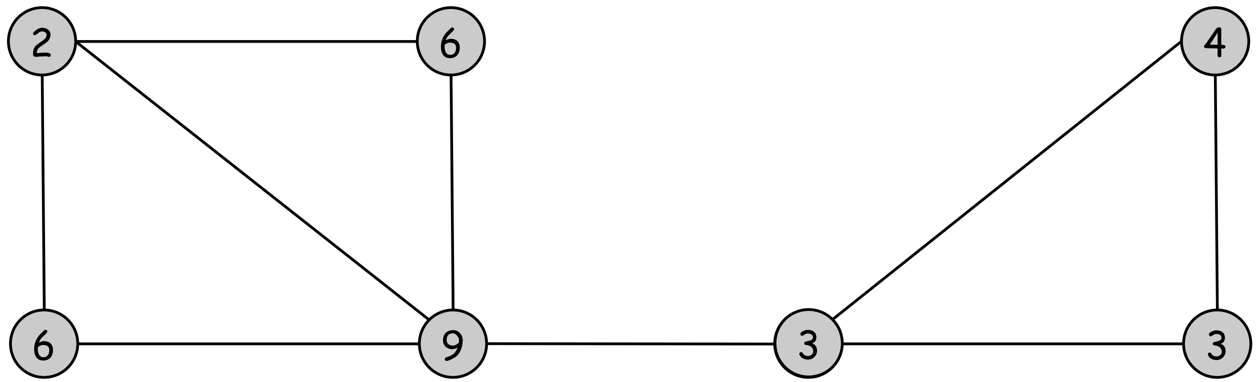
Pricing Method

Pricing method. Set prices and find vertex cover simultaneously.

$$\sum_{e=(i,j)} p_e = w_i$$

```
Weighted-Vertex-Cover-Approx(G, w) {
   foreach e in E
      pe = 0

   while (∃ edge i-j such that neither i nor j are tight)
      select such an edge e
      increase pe as much as possible until i or j tight
   }

   S ← set of all tight nodes
   return S
}
```

Example run



Pricing Method

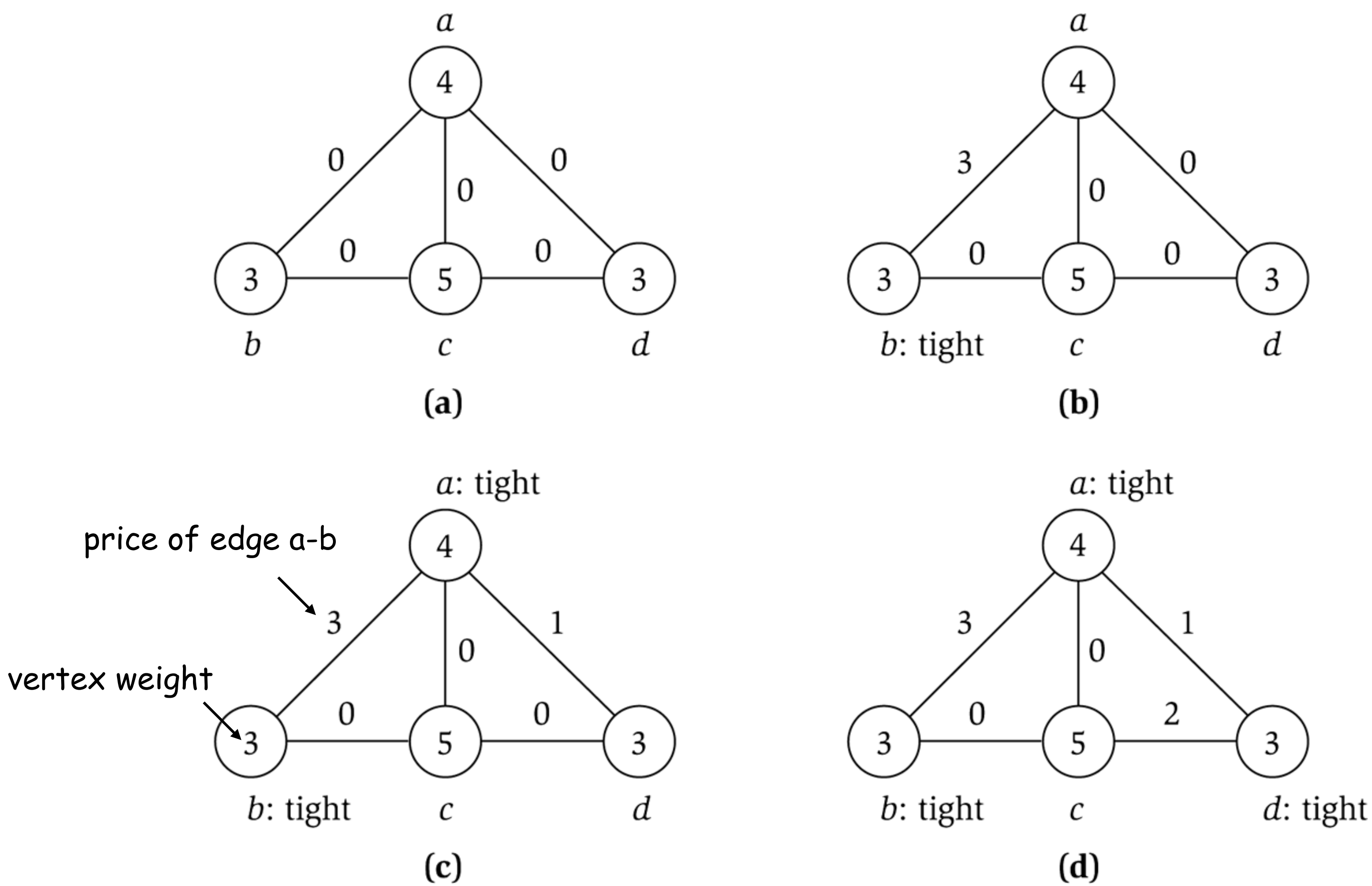


Figure 11.8

Theorem. Pricing method is a 2-approximation.

Pf.

- Algorithm terminates since at least one new node becomes tight after each iteration of while loop.
- Let S = set of all tight nodes upon termination of algorithm. S is a vertex cover: if some edge i-j is uncovered, then neither i nor j is tight. But then while loop would not terminate.
- Let S* be optimal vertex cover. We show w(S) ≤ 2w(S*).

$$w(S) = \sum_{i \in S} w_i \underbrace{=}_{\text{all nodes in S are tight}} \sum_{i \in S} \sum_{e=(i,j)} p_e \underbrace{\leq}_{S \subseteq V,\ \text{prices} \geq 0} \sum_{i \in V} \sum_{e=(i,j)} p_e \underbrace{=}_{\text{each edge counted twice}} 2 \sum_{e \in E} p_e \underbrace{\leq}_{\text{fairness lemma}} 2w(S^*). \quad \blacksquare$$

11.6 LP Rounding: Vertex Cover

Weighted Vertex Cover: IP Formulation

Weighted vertex cover. Given an undirected graph $G = (V, E)$ with vertex weights $w_i \geq 0$, find a minimum weight subset of nodes S such that every edge is incident to at least one vertex in S.

Integer programming formulation.

- Model inclusion of each vertex i using a 0/1 variable x_i.

$$x_i = \begin{cases} 0 & \text{if vertex } i \text{ is not in vertex cover} \\ 1 & \text{if vertex } i \text{ is in vertex cover} \end{cases}$$

  Vertex covers in 1-1 correspondence with 0/1 assignments: $S = \{i \in V : x_i = 1\}$

- Objective function: minimize $\Sigma_i w_i x_i$.

- For each edge i-j, must take either i or j: $x_i + x_j \geq 1$.

Weighted Vertex Cover: IP Formulation

Weighted vertex cover. Integer programming formulation.

$$
\begin{array}{llll}
(ILP) \ \min & \sum\limits_{i \in V} w_i \, x_i & & \\
\text{s. t.} & x_i + x_j & \geq & 1 & (i,j) \in E \\
& x_i & \in & \{0,1\} & i \in V
\end{array}
$$

Observation. If x^* is optimal solution to (ILP), then $S = \{i \in V : x^*_i = 1\}$ is a min weight vertex cover.

Integer Programming

INTEGER-PROGRAMMING. Given integers a_{ij} and b_i, find integers x_j that satisfy:

$$
\begin{array}{rcll}
\sum_{j=1}^{n} a_{ij} x_j & \geq & b_i & 1 \leq i \leq m \\
x_j & \geq & 0 & 1 \leq j \leq n \\
x_j & & \text{integral} & 1 \leq j \leq n
\end{array}
$$

Observation. Vertex cover formulation proves that integer programming is NP-hard search problem.

even if all coefficients are 0/1 and
at most two variables per inequality

Linear Programming

Linear programming. Max/min linear objective function subject to linear inequalities.

- Input: integers c_j, b_i, a_{ij} .
- Output: real numbers x_j.

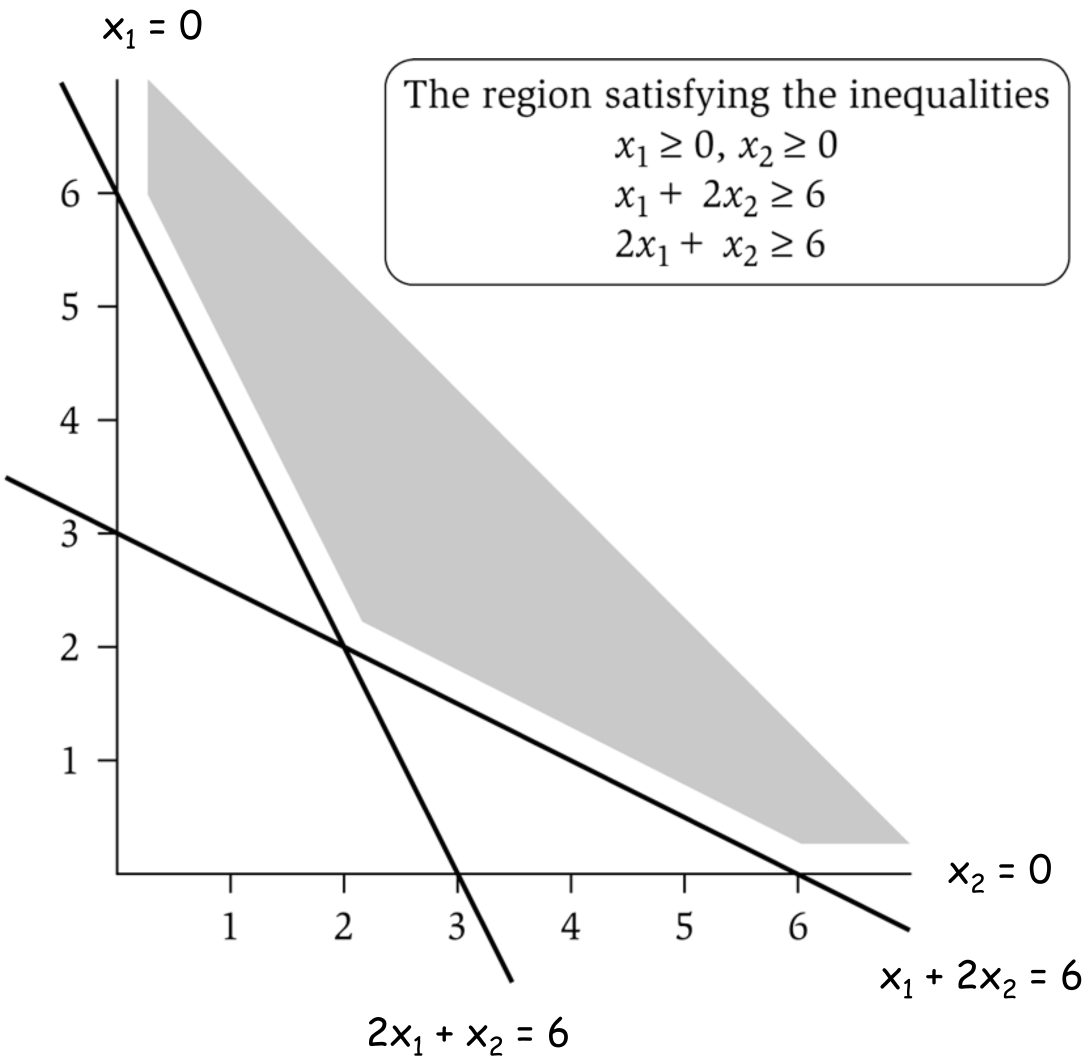
$$
\begin{array}{llrcll}
\text{(P)} & \max & \displaystyle\sum_{j=1}^{n} c_j x_j & & & \\
 & \text{s. t.} & \displaystyle\sum_{j=1}^{n} a_{ij} x_j & \geq & b_i & 1 \leq i \leq m \\
 & & x_j & \geq & 0 & 1 \leq j \leq n
\end{array}
$$

Linear. No x^2, xy, $\arccos(x)$, $x(1-x)$, etc.

Simplex algorithm. [Dantzig 1947] Can solve LP in practice.
Ellipsoid algorithm. [Khachian 1979] Can solve LP in poly-time.

LP Feasible Region

LP geometry in 2D.



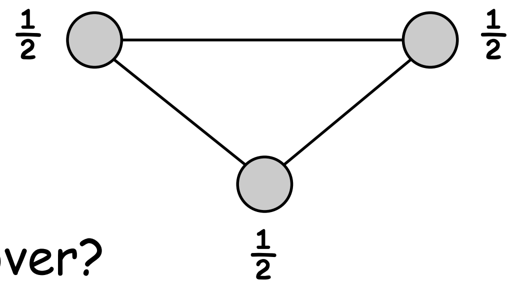
Weighted Vertex Cover: LP Relaxation

Weighted vertex cover. Linear programming formulation.

$$\begin{array}{lllll} (LP) \ \min & \displaystyle\sum_{i \,\in\, V} w_i\, x_i & & & \\ \text{s. t.} & x_i + x_j & \geq & 1 & (i,j) \in E \\ & x_i & \geq & 0 & i \in V \end{array}$$

Observation. Optimal value of (LP) is $\leq$ optimal value of (ILP).
Pf. LP has fewer constraints.

Note. LP is not equivalent to vertex cover.



Q. How can solving LP help us find a small vertex cover?
A. Solve LP and round fractional values.

Weighted Vertex Cover

Theorem. If x^* is optimal solution to (LP), then $S = \{i \in V : x^*_i \geq \frac{1}{2}\}$ is a vertex cover whose weight is at most twice the min possible weight.

Pf. [S is a vertex cover]
- Consider an edge $(i, j) \in E$.
- Since $x^*_i + x^*_j \geq 1$, either $x^*_i \geq \frac{1}{2}$ or $x^*_j \geq \frac{1}{2}$ $\Rightarrow$ (i, j) covered.

Pf. [S has desired cost]
- Let S^* be an optimal vertex cover. Then

$$\underbrace{\sum_{i \in S^*} w_i \geq \sum_{i \in S} w_i x_i^*}_{\text{LP is a relaxation}} \geq \underbrace{\tfrac{1}{2} \sum_{i \in S} w_i}_{x^*_i \geq \frac{1}{2}} = \tfrac{1}{2} w(S)$$

Better approximation? Vertex Cover is discussed at
http://www.nada.kth.se/~viggo/wwwcompendium/node10.html

11.8 Knapsack Problem

Polynomial Time Approximation Scheme

PTAS. $(1 + \varepsilon)$-approximation algorithm for any constant $\varepsilon > 0$.

FPTAS. Running time depends polynomially on $1/\varepsilon$.

- Load balancing. [Hochbaum-Shmoys 1987]
- Euclidean TSP. [Arora 1996]

Consequence. PTAS produces arbitrarily high quality solution, but trades off accuracy for time.

Next: PTAS for knapsack problem via rounding and scaling.

Knapsack Problem

Knapsack problem.

- Given n objects and a "knapsack."
- Item i has value $v_i > 0$ and weighs $w_i > 0$. ← we'll assume $w_i \le W$
- Knapsack can carry weight up to W.
- Goal: fill knapsack so as to maximize total value.

Ex: { 3, 4 } has value 40.

W = 11

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

Claim. SUBSET-SUM $\le_P$ KNAPSACK.
So KNAPSACK is NP-hard.

Knapsack Problem: Dynamic Programming 1

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

- Case 1: OPT does not select item i.
  - OPT selects best of 1, …, i–1 using up to weight limit w
- Case 2: OPT selects item i.
  - new weight limit = $w - w_i$
  - OPT selects best of 1, …, i–1 using up to weight limit $w - w_i$

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

Running time. O(n W).

- W = weight limit.
- Not polynomial in input size!

Def. OPT(i, v) = min weight subset of items 1, …, i that yields value exactly v.

- Case 1: OPT does not select item i.
  - OPT selects best of 1, …, i-1 that achieves exactly value v
- Case 2: OPT selects item i.
  - consumes weight w_i, new value needed = v – v_i
  - OPT selects best of 1, …, i-1 that achieves exactly value v

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

$V^* \le n\, v_{max}$

Running time. $O(n\, V^*) = O(n^2\, v_{max})$.

- V^* = optimal value = maximum v such that OPT(n, v) ≤ W.
- Not polynomial in input size!

Intuition for approximation algorithm.

- Round all values up to lie in smaller range.
- Run dynamic programming algorithm on rounded instance.
- Return optimal items in rounded instance.

Item	Value	Weight
1	934,221	1
2	5,956,342	2
3	17,810,013	5
4	21,217,800	6
5	27,343,199	7

W = 11

original instance

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

W = 11

rounded instance

Knapsack: FPTAS

Knapsack FPTAS. Round up all values: $\bar{v}_i = \left\lceil \frac{v_i}{\theta} \right\rceil \theta, \quad \hat{v}_i = \left\lceil \frac{v_i}{\theta} \right\rceil$

- v_{max} = largest value in original instance
- ε = precision parameter
- θ = scaling factor = $\varepsilon \, v_{max} / n$

Observation. Optimal solution to problems with $\bar{v}$ or $\hat{v}$ are equivalent.

Intuition. $\bar{v}$ close to v so optimal solution using $\bar{v}$ is nearly optimal; $\hat{v}$ small and integral so dynamic programming algorithm is fast.

Running time. $O(n^3 / \varepsilon)$.

- Dynamic program II running time is $O(n^2 \, \hat{v}_{\max})$, where

$$\hat{v}_{\max} = \left\lceil \frac{v_{\max}}{\theta} \right\rceil = \left\lceil \frac{n}{\varepsilon} \right\rceil$$

Knapsack: FPTAS

Knapsack FPTAS. Round up all values: $\bar{v}_i = \left\lceil \frac{v_i}{\theta} \right\rceil \theta$

Theorem. If S is solution found by our algorithm and S* is any other feasible solution then $(1+\varepsilon) \sum_{i \in S} v_i \geq \sum_{i \in S^*} v_i$

Pf. Let S* be any feasible solution satisfying weight constraint.

$$
\begin{aligned}
\sum_{i \in S^*} v_i &\leq \sum_{i \in S^*} \bar{v}_i && \text{always round up} \\
&\leq \sum_{i \in S} \bar{v}_i && \text{solve rounded instance optimally} \\
&\leq \sum_{i \in S} (v_i + \theta) && \text{never round up by more than } \theta \\
&\leq \sum_{i \in S} v_i + n\theta && |S| \leq n \\
&\leq (1+\varepsilon) \sum_{i \in S} v_i && n\theta = \varepsilon\, v_{max},\ \underset{\substack{\uparrow \\ \text{DP alg can take } v_{max}}}{v_{max} \leq \Sigma_{i \in S}\, v_i}
\end{aligned}
$$