# Chapter 6: Approximation Algorithms

## Advanced Algorithms

Approximation ratio | Traveling salesman (TSP) | Nearest neighbor and MST | 2-opt local search | Job scheduling | Vertex cover | Knapsack | Power iteration

Sercan KÜLCÜ · Giresun University · Computer Engineering

© Sercan Külcü. All rights reserved. · Interactive HTML version of these slides:
https://sercankulcu.github.io/files/advanced_algorithms/slides/Chapter_06_Approximation_Algorithms.html

# Learning Outcomes

### Concepts

Explain what an approximation algorithm is, the **approximation ratio**, and how it differs from a heuristic.

### TSP

Apply nearest neighbor, the MST-based 2-approximation and 2-opt local search to the traveling salesman problem step by step.

### Scheduling and cover

Write and trace machine scheduling (list scheduling, LPT) and the vertex cover 2-approximation in Java; prove their ratios.

### Numerical convergence

Compute how power iteration converges to the largest eigenvalue and its eigenvector.

# Approximation Algorithms

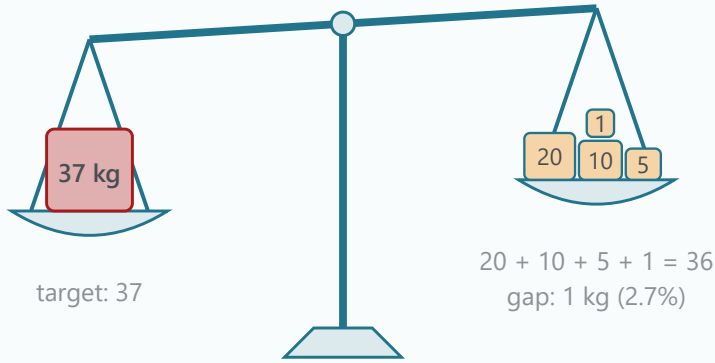
- For some problems finding the **exact (optimal)** solution is very hard or **impossible** in practice: TSP, vertex cover, machine scheduling and knapsack are **NP-hard**; if P ≠ NP they have no polynomial-time exact algorithms.
- Instead of looking for the **perfect solution**, approximation algorithms produce solutions **as close as possible** to the optimum.
- They run in **polynomial time**: hard problems can be solved in reasonable time.
- It can often be **proven** that the solution is within **a fixed ratio or percentage** of the optimum.
- Preferred when computing resources (time, memory) are **limited**.

The trade-off: give up a little **optimality**, gain **efficiency** and a **guarantee** on solution quality.

INTRODUCTION

# The Balance Scale Analogy

- When a weight is placed on one pan of a scale, it can be hard to put **exactly the same** weight on the other pan.
- Instead, we use the **small weights** at hand to balance the large one **as closely as possible**.
- If we only have 20, 10, 5 and 1 kg, each usable once, the closest balance to 37 kg is 36 kg.




An approximation algorithm works the same way: instead of searching for the exact balance (the optimum), it settles for one that is found **quickly** and whose error is **bounded**.

INTRODUCTION

# Approximation Ratio

C: cost of the solution the algorithm finds, C*: cost of the optimal solution. If for every input the algorithm satisfies

$$\max( C / C^*, C^* / C ) \le \rho(n)$$

it is a **ρ(n)-approximation algorithm** (ρ ≥ 1; ρ = 1 means the algorithm is optimal).

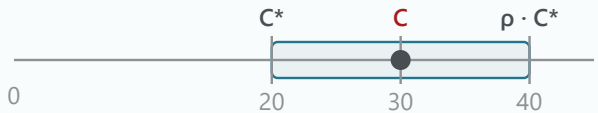
### Minimization

C ≤ ρ · C*. Example: a 2-approximation for vertex cover picks at most **twice** as many vertices as the optimum.

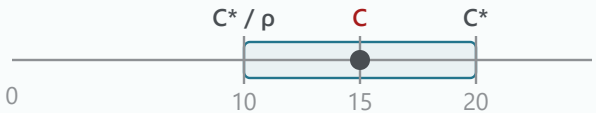
### Maximization

C ≥ C* / ρ. Example: a 2-approximation for knapsack guarantees at least **half** the optimal value.

Minimization (ρ = 2): C* ≤ C ≤ ρ · C*




Maximization (ρ = 2): C* / ρ ≤ C ≤ C*




The ratio is a **worst-case** guarantee; in practice solutions are often much closer to the optimum.

# Approximation Algorithms, Heuristics and Approximation Schemes

| Approach | Guarantee | Example |
|---|---|---|
| **Exact algorithm** | Optimal, but exponential time on NP-hard problems | Brute-force TSP: $(n-1)!/2$ tours |
| **Approximation algorithm** | Polynomial time + a proven ratio $\rho$ | Vertex cover: $\rho = 2$ |
| **PTAS** | Ratio $(1 + \varepsilon)$ for every fixed $\varepsilon > 0$, polynomial in $n$ (may be exponential in $\varepsilon$) | Euclidean TSP (Arora, 1998) |
| **FPTAS** | Ratio $(1 + \varepsilon)$, polynomial in both $n$ and $1/\varepsilon$ | Knapsack: $O(n^3/\varepsilon)$ |
| **Heuristic** | No guarantee; usually good in practice | 2-opt, nearest neighbor |
| **Metaheuristic** | No guarantee; a problem-independent search framework | Genetic algorithms, ant colony |

There are also randomized approximation algorithms such as randomized rounding; they guarantee a ratio in expectation.

# Applications of Approximation Algorithms

**Traveling salesman (TSP)**

Choosing the route that visits every customer and returns with the shortest total length: logistics, parcel delivery, circuit-board drilling.

**Knapsack**

Choosing the items that fit and give the most value: budget, cargo and portfolio planning.

**Job scheduling**

Running several jobs or processes on machines/processors as efficiently as possible.

**Vertex cover and numerical methods**

Placing the fewest monitoring points in a network; converging iteratively to an eigenvalue of a huge matrix (PageRank).

# Kinds of Approximation Algorithms

### Simplification

Solve a **simpler version** of the problem (e.g. relax integrality: LP relaxation, an MST lower bound for TSP).

### Greedy

At each step take the option that **looks best right now** (nearest neighbor, LPT, ratio-sorted knapsack).

### Local search

Improve the current solution with **small changes**; stop when no better neighbor remains (2-opt).

### Primal–dual

Grow primal and dual LP solutions together; common for covering problems.

### Randomized

Randomized rounding or sampling; a guarantee on the expected ratio.

### Metaheuristic

Simulated annealing, genetic algorithms, ant colony.

# The Traveling Salesman Problem (TSP)

- **Optimization problem:** a salesman wants to visit each of a set of cities once with the **shortest total distance** and return to the starting city.
- **Combinatorial problem:** a solution is the **order in which the cities are visited** (a permutation).
- **NP-hard:** finding the best route for many cities is hard in practice. The shortest path between two vertices (Dijkstra), however, is solvable in polynomial time; do not confuse the two problems.
- **Input:** n cities (0, 1, 2, … or A, B, C, …) and $\mathbf{d_{ij}}$: the distance between cities i and j.
- Symmetric TSP: $d_{ij} = d_{ji}$; in graph terms: the lightest **Hamiltonian cycle** in a complete graph.

| d | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| **0** | 0 | 10 | 15 | 20 | 25 |
| **1** | 10 | 0 | 35 | 25 | 30 |
| **2** | 15 | 35 | 0 | 30 | 35 |
| **3** | 20 | 25 | 30 | 0 | 40 |
| **4** | 25 | 30 | 35 | 40 | 0 |

Example: the distance matrix of 5 cities.

# Solving the Problem

### Exact solution

Check every possible route (brute force) or use dynamic programming (Held–Karp, $O(n^2 \cdot 2^n)$).

### Approximate solutions

Heuristic algorithms: nearest neighbor, the MST-based 2-approximation, Christofides, genetic algorithms.

### Optimization tools

Local search (2-opt), simulated annealing, ant colony optimization.

| n | (n−1)!/2 distinct tours |
|---|---|
| 5 | 12 |
| 10 | 181,440 |
| 15 | $\approx 4.4 \cdot 10^{10}$ |
| 20 | $\approx 6.1 \cdot 10^{16}$ |
| 25 | $\approx 3.1 \cdot 10^{23}$ |

The start city is fixed and both directions count once. Even at $10^9$ tours per second, n = 25 would take $\approx 10^7$ years.

# The Nearest Neighbor Heuristic

1. Pick a start city and mark it visited.
2. From the current city go to the **nearest unvisited** city (greedy choice).
3. Repeat step 2 until every city is visited.
4. **Return to the start city** from the last one to close the tour.

### Pros

+ Very simple; **$O(n^2)$** time.
+ Usually gives a tour about 25% above the optimum.
+ A good **initial tour** for local search.

### Cons

– There is **no** constant approximation ratio; even for metric TSP the ratio can be $\Theta(\log n)$.
– The final return edge can be very long.

# Nearest Neighbor: Code Trace

```
1  int[] nearestNeighbor(int[][] d, int start) {
▶    int n = d.length, total = 0, cur = start;
▶    boolean[] visited = new boolean[n];
▶    int[] tour = new int[n + 1];
▶    tour[0] = start; visited[start] = true;
6    for (int k = 1; k < n; k++) {
7      int nearest = -1, minD = Integer.MAX_VALUE;
8      for (int j = 0; j < n; j++)
9        if (!visited[j] && d[cur][j] < minD) {
10         minD = d[cur][j]; nearest = j; }
11     // go to the nearest city
12     tour[k] = nearest; visited[nearest] = true;
13     total += minD; cur = nearest;
14   }
15   // return to the start
16   tour[n] = start; total += d[cur][start];
17   return tour;
18 }
```

| | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 0 | 0 | 10 | 15 | 20 | 25 |
| 1 | 10 | 0 | 35 | 25 | 30 |
| 2 | 15 | 35 | 0 | 30 | 35 |
| 3 | 20 | 25 | 30 | 0 | 40 |
| 4 | 25 | 30 | 35 | 40 | 0 |

tour: 0

Start city 0: `visited[0] = true`, `tour = [0]`, `total = 0`.

Step 1 / 11

**VARIABLES**

`k = –` `cur = 0` `nearest = –` `minD = –` `total = 0` `tour = [0]`

# Nearest Neighbor: Code Trace

```
1  int[] nearestNeighbor(int[][] d, int start) {
2    int n = d.length, total = 0, cur = start;
3    boolean[] visited = new boolean[n];
4    int[] tour = new int[n + 1];
5    tour[0] = start; visited[start] = true;
6    for (int k = 1; k < n; k++) {
▶      int nearest = -1, minD = Integer.MAX_VALUE;
▶      for (int j = 0; j < n; j++)
▶        if (!visited[j] && d[cur][j] < minD) {
▶          minD = d[cur][j]; nearest = j; }
11     // go to the nearest city
12     tour[k] = nearest; visited[nearest] = true;
13     total += minD; cur = nearest;
14   }
15   // return to the start
16   tour[n] = start; total += d[cur][start];
17   return tour;
18 }
```

| | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 0 | 0 | 10 | 15 | 20 | 25 |
| 1 | 10 | 0 | 35 | 25 | 30 |
| 2 | 15 | 35 | 0 | 30 | 35 |
| 3 | 20 | 25 | 30 | 0 | 40 |
| 4 | 25 | 30 | 35 | 40 | 0 |

**tour: 0**

Scan row 0, unvisited: d[0][1] = 10, d[0][2] = 15, d[0][3] = 20, d[0][4] = 25 → the smallest is **1** (10).

Step 2 / 11

VARIABLES

`k = 1` `cur = 0` `nearest = 1` `minD = 10` `total = 0` `tour = [0]`

# Nearest Neighbor: Code Trace

```
1  int[] nearestNeighbor(int[][] d, int start) {
2    int n = d.length, total = 0, cur = start;
3    boolean[] visited = new boolean[n];
4    int[] tour = new int[n + 1];
5    tour[0] = start; visited[start] = true;
6    for (int k = 1; k < n; k++) {
7      int nearest = -1, minD = Integer.MAX_VALUE;
8      for (int j = 0; j < n; j++)
9        if (!visited[j] && d[cur][j] < minD) {
10         minD = d[cur][j]; nearest = j; }
11     // go to the nearest city
▶      tour[k] = nearest; visited[nearest] = true;
▶      total += minD; cur = nearest;
14   }
15   // return to the start
16   tour[n] = start; total += d[cur][start];
17   return tour;
18 }
```

| | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 0 | 0 | 10 | 15 | 20 | 25 |
| 1 | 10 | 0 | 35 | 25 | 30 |
| 2 | 15 | 35 | 0 | 30 | 35 |
| 3 | 20 | 25 | 30 | 0 | 40 |
| 4 | 25 | 30 | 35 | 40 | 0 |

tour: 0 → 1

`tour[1] = 1`, `total += 10` → 10; now `cur = 1`.

Step 3 / 11

VARIABLES

`k = 1` `cur = 1` `nearest = 1` `minD = 10` `total = 10` `tour = [0, 1]`

# Nearest Neighbor: Code Trace

```
1  int[] nearestNeighbor(int[][] d, int start) {
2    int n = d.length, total = 0, cur = start;
3    boolean[] visited = new boolean[n];
4    int[] tour = new int[n + 1];
5    tour[0] = start; visited[start] = true;
6    for (int k = 1; k < n; k++) {
▶      int nearest = -1, minD = Integer.MAX_VALUE;
▶      for (int j = 0; j < n; j++)
▶        if (!visited[j] && d[cur][j] < minD) {
▶          minD = d[cur][j]; nearest = j; }
11     // go to the nearest city
12     tour[k] = nearest; visited[nearest] = true;
13     total += minD; cur = nearest;
14   }
15   // return to the start
16   tour[n] = start; total += d[cur][start];
17   return tour;
18 }
```

| | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 0 | 0 | 10 | 15 | 20 | 25 |
| 1 | 10 | 0 | 35 | 25 | 30 |
| 2 | 15 | 35 | 0 | 30 | 35 |
| 3 | 20 | 25 | 30 | 0 | 40 |
| 4 | 25 | 30 | 35 | 40 | 0 |

**tour: 0 → 1**

Scan row 1, unvisited: d[1][2] = 35, d[1][3] = 25, d[1][4] = 30 → the smallest is **3** (25).

Step 4 / 11

VARIABLES

`k = 2` `cur = 1` `nearest = 3` `minD = 25` `total = 10` `tour = [0, 1]`

# Nearest Neighbor: Code Trace

```
1  int[] nearestNeighbor(int[][] d, int start) {
2    int n = d.length, total = 0, cur = start;
3    boolean[] visited = new boolean[n];
4    int[] tour = new int[n + 1];
5    tour[0] = start; visited[start] = true;
6    for (int k = 1; k < n; k++) {
7      int nearest = -1, minD = Integer.MAX_VALUE;
8      for (int j = 0; j < n; j++)
9        if (!visited[j] && d[cur][j] < minD) {
10         minD = d[cur][j]; nearest = j; }
11     // go to the nearest city
▶      tour[k] = nearest; visited[nearest] = true;
▶      total += minD; cur = nearest;
14   }
15   // return to the start
16   tour[n] = start; total += d[cur][start];
17   return tour;
18 }
```

| | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 0 | 0 | 10 | 15 | 20 | 25 |
| 1 | 10 | 0 | 35 | 25 | 30 |
| 2 | 15 | 35 | 0 | 30 | 35 |
| 3 | 20 | 25 | 30 | 0 | 40 |
| 4 | 25 | 30 | 35 | 40 | 0 |

tour: 0 → 1 → 3

`tour[2] = 3`, `total += 25` → 35; now `cur = 3`.

Step 5 / 11

VARIABLES

`k = 2` `cur = 3` `nearest = 3` `minD = 25` `total = 35` `tour = [0, 1, 3]`

# Nearest Neighbor: Code Trace

```
1  int[] nearestNeighbor(int[][] d, int start) {
2    int n = d.length, total = 0, cur = start;
3    boolean[] visited = new boolean[n];
4    int[] tour = new int[n + 1];
5    tour[0] = start; visited[start] = true;
6    for (int k = 1; k < n; k++) {
▶      int nearest = -1, minD = Integer.MAX_VALUE;
▶      for (int j = 0; j < n; j++)
▶        if (!visited[j] && d[cur][j] < minD) {
▶          minD = d[cur][j]; nearest = j; }
11     // go to the nearest city
12     tour[k] = nearest; visited[nearest] = true;
13     total += minD; cur = nearest;
14   }
15   // return to the start
16   tour[n] = start; total += d[cur][start];
17   return tour;
18 }
```

| | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 0 | 0 | 10 | 15 | 20 | 25 |
| 1 | 10 | 0 | 35 | 25 | 30 |
| 2 | 15 | 35 | 0 | 30 | 35 |
| 3 | 20 | 25 | 30 | 0 | 40 |
| 4 | 25 | 30 | 35 | 40 | 0 |

**tour: 0 → 1 → 3**

Scan row 3, unvisited: d[3][2] = 30, d[3][4] = 40 → the smallest is **2** (30).

Step 6 / 11

VARIABLES

`k = 3` `cur = 3` `nearest = 2` `minD = 30` `total = 35` `tour = [0, 1, 3]`

# Nearest Neighbor: Code Trace

```
1  int[] nearestNeighbor(int[][] d, int start) {
2    int n = d.length, total = 0, cur = start;
3    boolean[] visited = new boolean[n];
4    int[] tour = new int[n + 1];
5    tour[0] = start; visited[start] = true;
6    for (int k = 1; k < n; k++) {
7      int nearest = -1, minD = Integer.MAX_VALUE;
8      for (int j = 0; j < n; j++)
9        if (!visited[j] && d[cur][j] < minD) {
10         minD = d[cur][j]; nearest = j; }
11     // go to the nearest city
▶      tour[k] = nearest; visited[nearest] = true;
▶      total += minD; cur = nearest;
14   }
15   // return to the start
16   tour[n] = start; total += d[cur][start];
17   return tour;
18 }
```

| | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 0 | 0 | 10 | 15 | 20 | 25 |
| 1 | 10 | 0 | 35 | 25 | 30 |
| 2 | 15 | 35 | 0 | 30 | 35 |
| 3 | 20 | 25 | 30 | 0 | 40 |
| 4 | 25 | 30 | 35 | 40 | 0 |

tour: 0 → 1 → 3 → 2

`tour[3] = 2`, `total += 30` → 65; now `cur = 2`.

Step 7 / 11

VARIABLES

`k = 3` `cur = 2` `nearest = 2` `minD = 30`

`total = 65` `tour = [0, 1, 3, 2]`

# Nearest Neighbor: Code Trace

```
1  int[] nearestNeighbor(int[][] d, int start) {
2    int n = d.length, total = 0, cur = start;
3    boolean[] visited = new boolean[n];
4    int[] tour = new int[n + 1];
5    tour[0] = start; visited[start] = true;
6    for (int k = 1; k < n; k++) {
▶      int nearest = -1, minD = Integer.MAX_VALUE;
▶      for (int j = 0; j < n; j++)
▶        if (!visited[j] && d[cur][j] < minD) {
▶          minD = d[cur][j]; nearest = j; }
11     // go to the nearest city
12     tour[k] = nearest; visited[nearest] = true;
13     total += minD; cur = nearest;
14   }
15   // return to the start
16   tour[n] = start; total += d[cur][start];
17   return tour;
18 }
```

| | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 0 | 0 | 10 | 15 | 20 | 25 |
| 1 | 10 | 0 | 35 | 25 | 30 |
| 2 | 15 | 35 | 0 | 30 | 35 |
| 3 | 20 | 25 | 30 | 0 | 40 |
| 4 | 25 | 30 | 35 | 40 | 0 |

**tour: 0 → 1 → 3 → 2**

Scan row 2, unvisited: d[2][4] = 35 → the smallest is **4** (35).

Step 8 / 11

VARIABLES

`k = 4` `cur = 2` `nearest = 4` `minD = 35` `total = 65` `tour = [0, 1, 3, 2]`

# Nearest Neighbor: Code Trace

```
1  int[] nearestNeighbor(int[][] d, int start) {
2    int n = d.length, total = 0, cur = start;
3    boolean[] visited = new boolean[n];
4    int[] tour = new int[n + 1];
5    tour[0] = start; visited[start] = true;
6    for (int k = 1; k < n; k++) {
7      int nearest = -1, minD = Integer.MAX_VALUE;
8      for (int j = 0; j < n; j++)
9        if (!visited[j] && d[cur][j] < minD) {
10         minD = d[cur][j]; nearest = j; }
11     // go to the nearest city
▶      tour[k] = nearest; visited[nearest] = true;
▶      total += minD; cur = nearest;
14   }
15   // return to the start
16   tour[n] = start; total += d[cur][start];
17   return tour;
18 }
```

| | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 0 | 0 | 10 | 15 | 20 | 25 |
| 1 | 10 | 0 | 35 | 25 | 30 |
| 2 | 15 | 35 | 0 | 30 | 35 |
| 3 | 20 | 25 | 30 | 0 | 40 |
| 4 | 25 | 30 | 35 | 40 | 0 |

tour: 0 → 1 → 3 → 2 → 4

`tour[4] = 4`, `total += 35` → 100; now `cur = 4`.

Step 9 / 11

VARIABLES

`k = 4` `cur = 4` `nearest = 4` `minD = 35`

`total = 100` `tour = [0, 1, 3, 2, 4]`

# Nearest Neighbor: Code Trace

```
1  int[] nearestNeighbor(int[][] d, int start) {
2    int n = d.length, total = 0, cur = start;
3    boolean[] visited = new boolean[n];
4    int[] tour = new int[n + 1];
5    tour[0] = start; visited[start] = true;
6    for (int k = 1; k < n; k++) {
7      int nearest = -1, minD = Integer.MAX_VALUE;
8      for (int j = 0; j < n; j++)
9        if (!visited[j] && d[cur][j] < minD) {
10         minD = d[cur][j]; nearest = j; }
11     // go to the nearest city
12     tour[k] = nearest; visited[nearest] = true;
13     total += minD; cur = nearest;
14   }
15   // return to the start
▶    tour[n] = start; total += d[cur][start];
17   return tour;
18 }
```

| | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 0 | 0 | 10 | 15 | 20 | 25 |
| 1 | 10 | 0 | 35 | 25 | 30 |
| 2 | 15 | 35 | 0 | 30 | 35 |
| 3 | 20 | 25 | 30 | 0 | 40 |
| 4 | 25 | 30 | 35 | 40 | 0 |

**tour: 0 → 1 → 3 → 2 → 4 → 0**

Every city is visited; return to the start: `total += d[4][0]` = 25 → **125**.

Step 10 / 11

VARIABLES

`k = 5` `cur = 4` `nearest = –` `minD = –`

`total = 125` `tour = [0, 1, 3, 2, 4, 0]`

# Nearest Neighbor: Code Trace

```
1  int[] nearestNeighbor(int[][] d, int start) {
2    int n = d.length, total = 0, cur = start;
3    boolean[] visited = new boolean[n];
4    int[] tour = new int[n + 1];
5    tour[0] = start; visited[start] = true;
6    for (int k = 1; k < n; k++) {
7      int nearest = -1, minD = Integer.MAX_VALUE;
8      for (int j = 0; j < n; j++)
9        if (!visited[j] && d[cur][j] < minD) {
10         minD = d[cur][j]; nearest = j; }
11     // go to the nearest city
12     tour[k] = nearest; visited[nearest] = true;
13     total += minD; cur = nearest;
14   }
15   // return to the start
16   tour[n] = start; total += d[cur][start];
▶    return tour;
18 }
```

| | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 0 | 0 | 10 | 15 | 20 | 25 |
| 1 | 10 | 0 | 35 | 25 | 30 |
| 2 | 15 | 35 | 0 | 30 | 35 |
| 3 | 20 | 25 | 30 | 0 | 40 |
| 4 | 25 | 30 | 35 | 40 | 0 |

**tour: 0 → 1 → 3 → 2 → 4 → 0**

Tour 0 → 1 → 3 → 2 → 4 → 0, length **125**. (Checked by brute force: optimal for this example.)

Step 11 / 11

VARIABLES

`k = 5` `cur = 4` `nearest = –` `minD = –`

`total = 125` `tour = [0, 1, 3, 2, 4, 0]`

# Comparison with Brute Force: 12 Distinct Tours

| Tour | Length |
|---|---|
| 0 → 1 → 3 → 2 → 4 → 0 | 125 |
| 0 → 1 → 3 → 4 → 2 → 0 | 125 |
| 0 → 1 → 4 → 2 → 3 → 0 | 125 |
| 0 → 1 → 4 → 3 → 2 → 0 | 125 |
| 0 → 2 → 3 → 1 → 4 → 0 | 125 |
| 0 → 2 → 4 → 1 → 3 → 0 | 125 |

| Tour | Length |
|---|---|
| 0 → 1 → 2 → 3 → 4 → 0 | 140 |
| 0 → 1 → 2 → 4 → 3 → 0 | 140 |
| 0 → 2 → 1 → 3 → 4 → 0 | 140 |
| 0 → 2 → 1 → 4 → 3 → 0 | 140 |
| 0 → 3 → 1 → 2 → 4 → 0 | 140 |
| 0 → 3 → 2 → 1 → 4 → 0 | 140 |

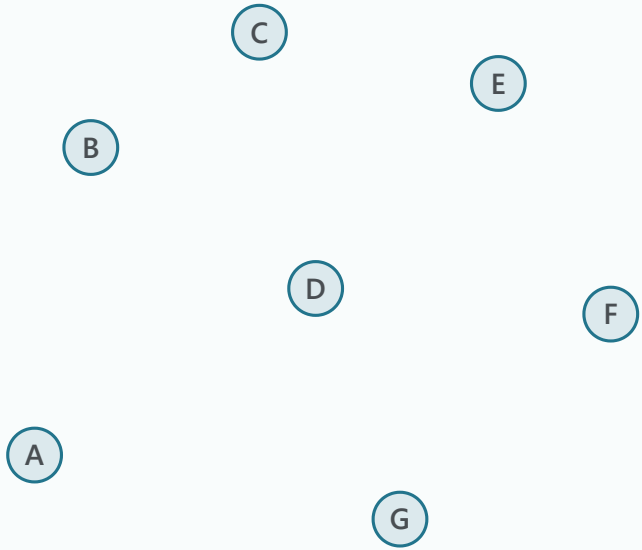
- $(5-1)!/2 = 12$ tours tried: the optimum is **125**; nearest neighbor also found 125 (a lucky case for this example).
- This matrix violates the **triangle inequality**: $d_{12} = 35 > d_{10} + d_{02} = 10 + 15$. So the example is **not metric**.

# The Triangle Inequality and Approximation Limits

Metric TSP: $d_{ij} \leq d_{ik} + d_{kj}$ for all i, j, k (going directly is never longer)

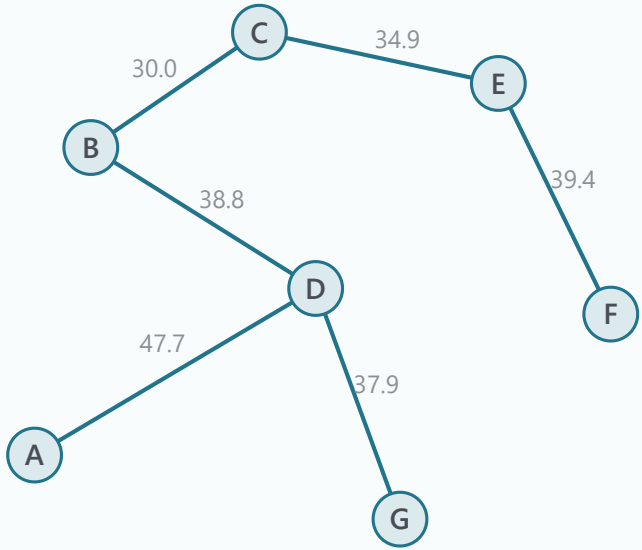
| Case | Ratio | Description |
|---|---|---|
| General TSP | none | If P ≠ NP, no polynomial-time ρ-approximation exists for any constant ρ: such an algorithm would solve the Hamiltonian cycle problem. |
| Metric TSP, MST-based | 2 | MST + preorder walk + shortcuts; $O(n^2)$. |
| Metric TSP, Christofides | 1.5 | MST + minimum perfect matching on odd-degree vertices + Euler tour (1976). |
| Euclidean TSP | 1 + ε | A PTAS exists for points in the plane (Arora, Mitchell). |
| Nearest neighbor | Θ(log n) | No constant ratio is guaranteed, even in the metric case. |

# MST-Based 2-Approximation for Metric TSP



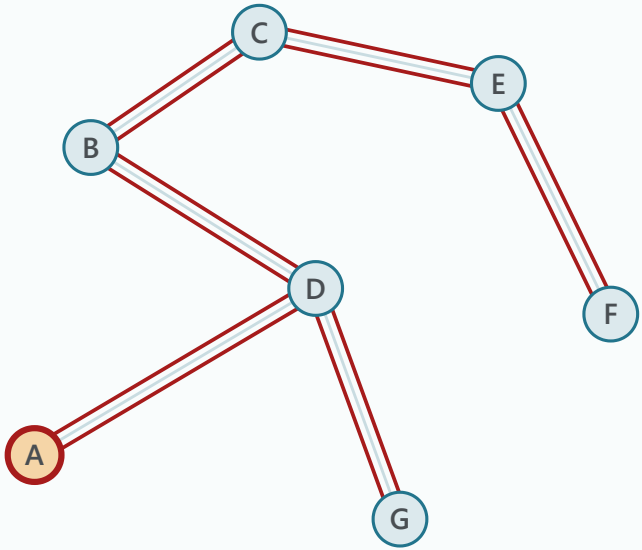

A metric example: 7 cities in the plane; distances are Euclidean (the triangle inequality holds).

# MST-Based 2-Approximation for Metric TSP



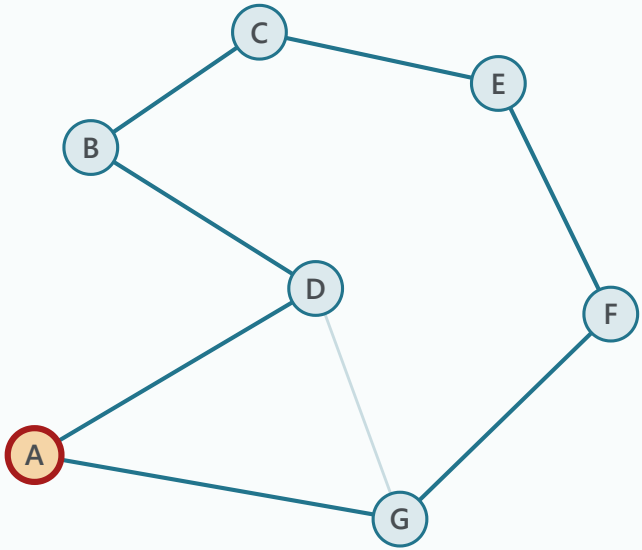

1) Build a **minimum spanning tree** (MST) with Prim: total weight w(T) = 228.8. Deleting one edge of a tour leaves a spanning tree → **w(T) ≤ OPT**.

# MST-Based 2-Approximation for Metric TSP



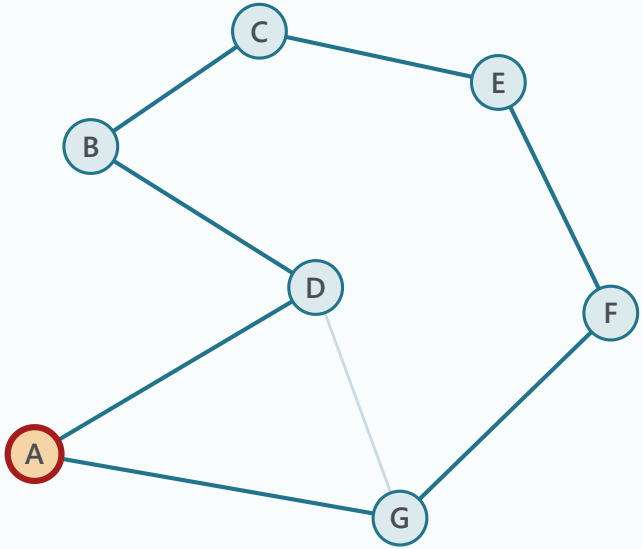

2) Walk the tree depth-first from A, **twice** (each edge once down, once back): A → D → B → C → E → F → E → C → B → D → G → D → A. Length $2 \cdot w(T) = 457.6$.

# MST-Based 2-Approximation for Metric TSP



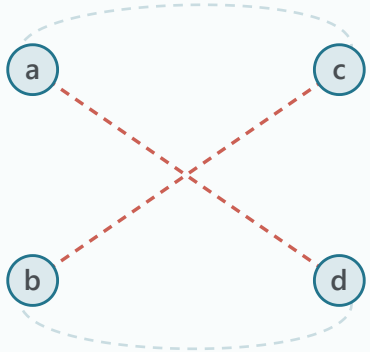

3) Skip repeated cities (**shortcut**): preorder A → D → B → C → E → F → G → A. Thanks to the triangle inequality, shortcuts never lengthen the path: tour = 287.7.

# MST-Based 2-Approximation for Metric TSP

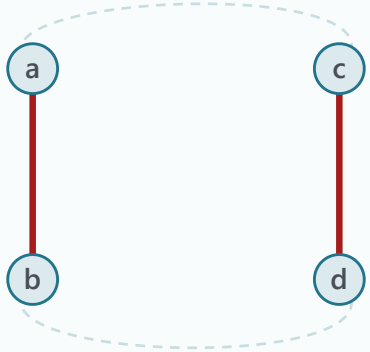



Result: tour ≤ 2 · w(T) ≤ 2 · OPT. Here tour = 287.7, optimum (brute force, 360 tours) = 282.5 → ratio 1.02 ≤ 2.

# Local Search: The 2-opt Move



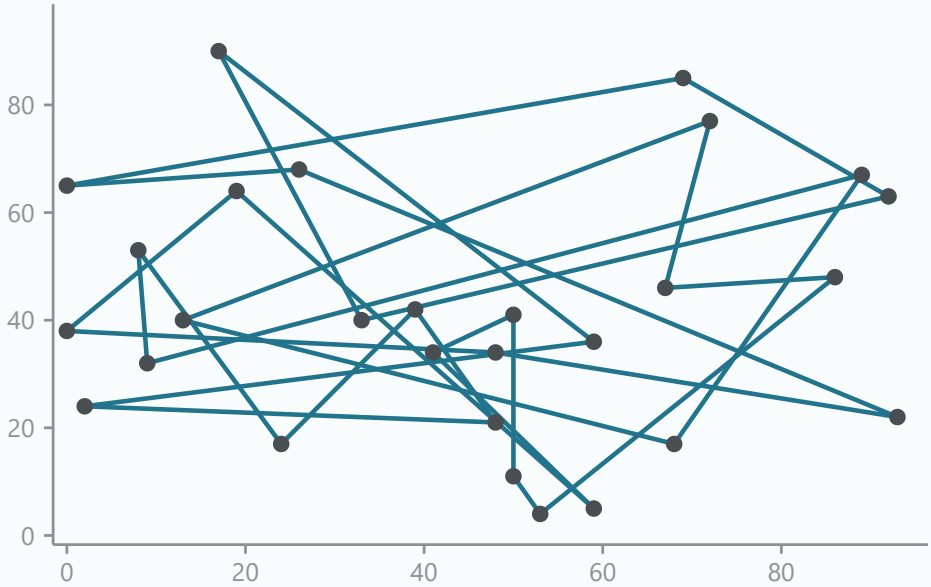

Before: edges (a, b) and (c, d) cross.




After: (a, c) and (b, d); the path between them is reversed.

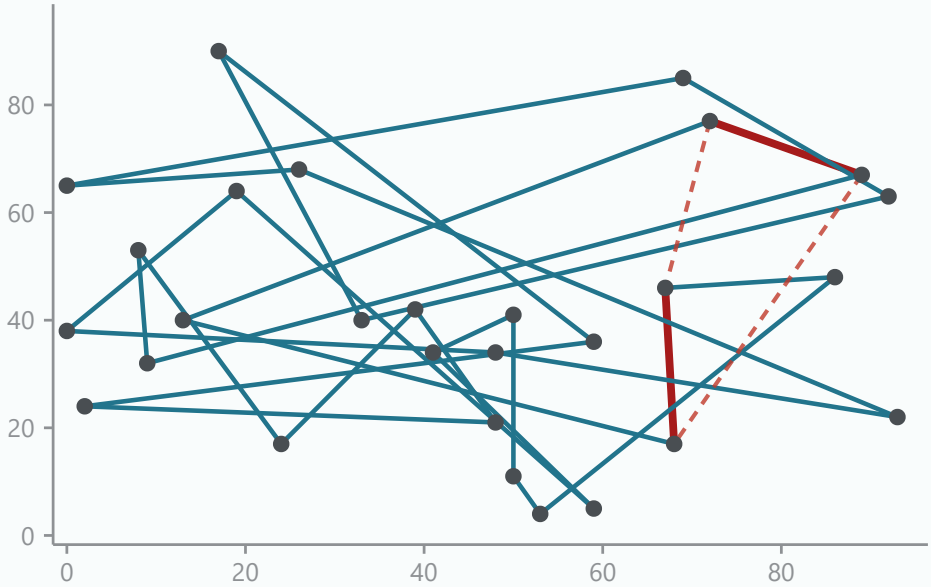
- Remove two edges of the tour and reconnect the ends **crosswise**: the subpath from b to c is **reversed**.
- Make the move if the gain $\Delta = d(a,c) + d(b,d) - d(a,b) - d(c,d) < 0$. In the Euclidean plane every **crossing** can be removed this way.
- When no improving move remains, the tour is a **2-opt local optimum** (not necessarily the global optimum).

# Improving a Tour with 2-opt



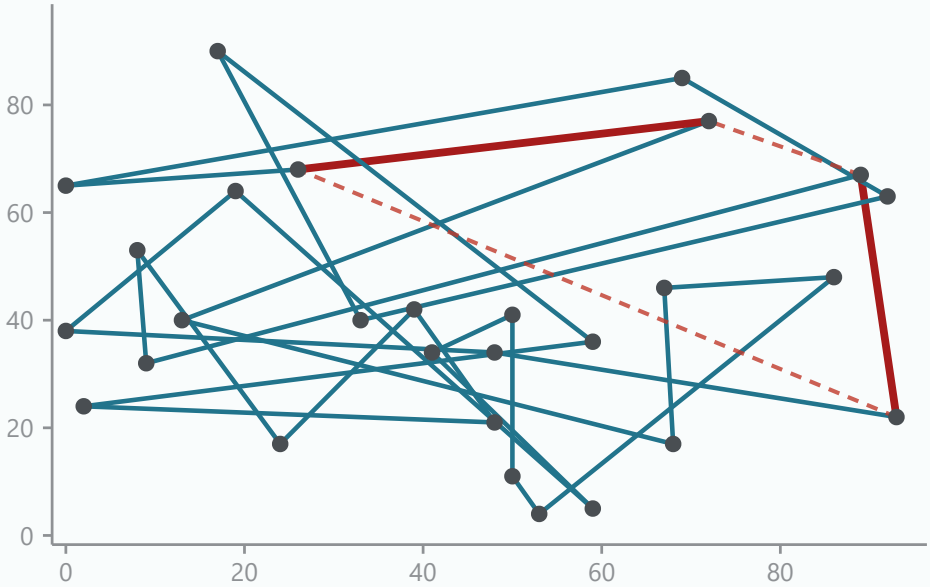

28 cities, a **random** initial tour: length **1269.8**. Edges keep crossing each other.

# Improving a Tour with 2-opt



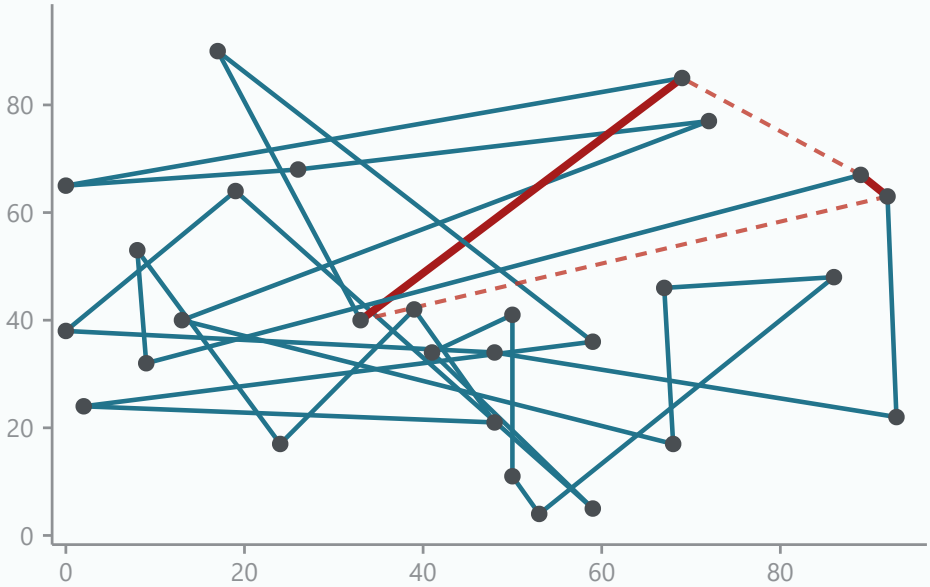

After improving move 1: (a,b), (c,d) removed (dashed), (a,c), (b,d) added (bold). Δ = −36.9 → length **1232.9**.

# Improving a Tour with 2-opt



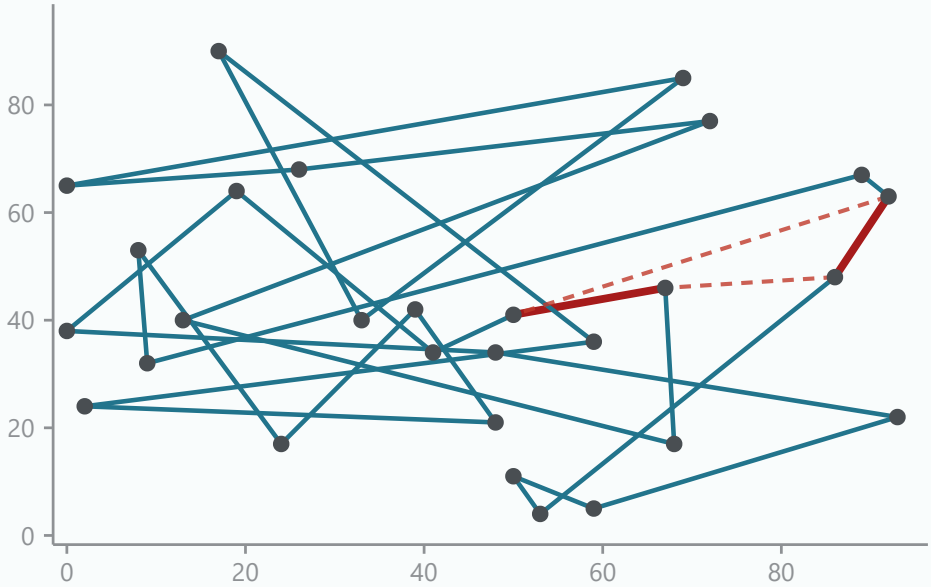

After improving move 2: (a,b), (c,d) removed (dashed), (a,c), (b,d) added (bold). Δ = −8.9 → length **1224.0**.

# Improving a Tour with 2-opt



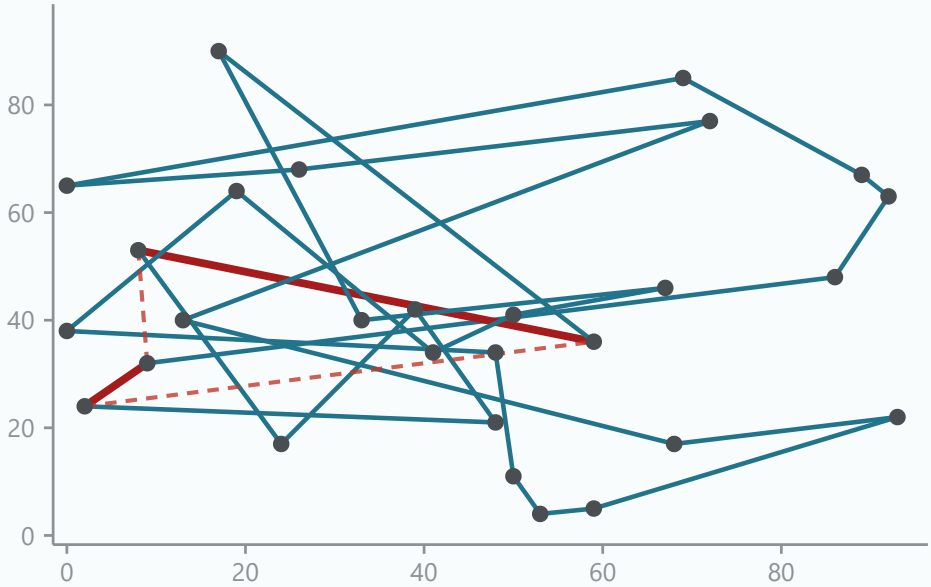

After improving move 4: (a,b), (c,d) removed (dashed), (a,c), (b,d) added (bold). Δ = −27.6 → length **1187.3**.

# Improving a Tour with 2-opt



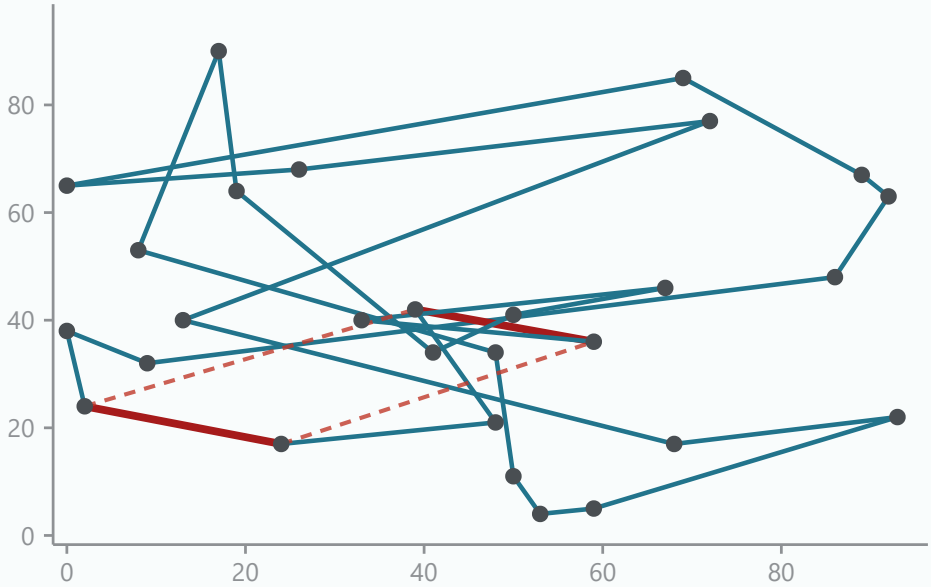

After improving move 8: (a,b), (c,d) removed (dashed), (a,c), (b,d) added (bold). Δ = −32.6 → length **1111.7**.

# Improving a Tour with 2-opt



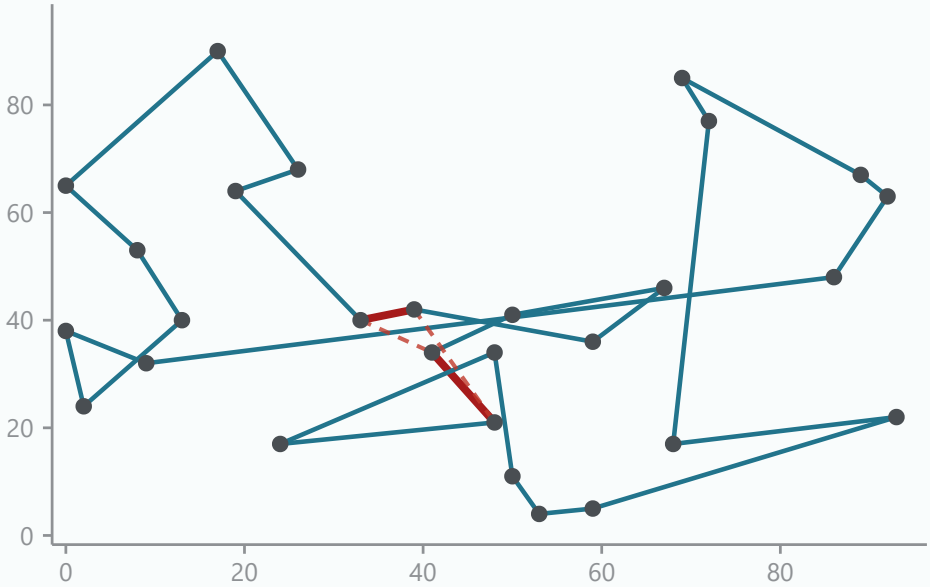

After improving move 14: (a,b), (c,d) removed (dashed), (a,c), (b,d) added (bold). Δ = −14.9 → length **1005.2**.

# Improving a Tour with 2-opt



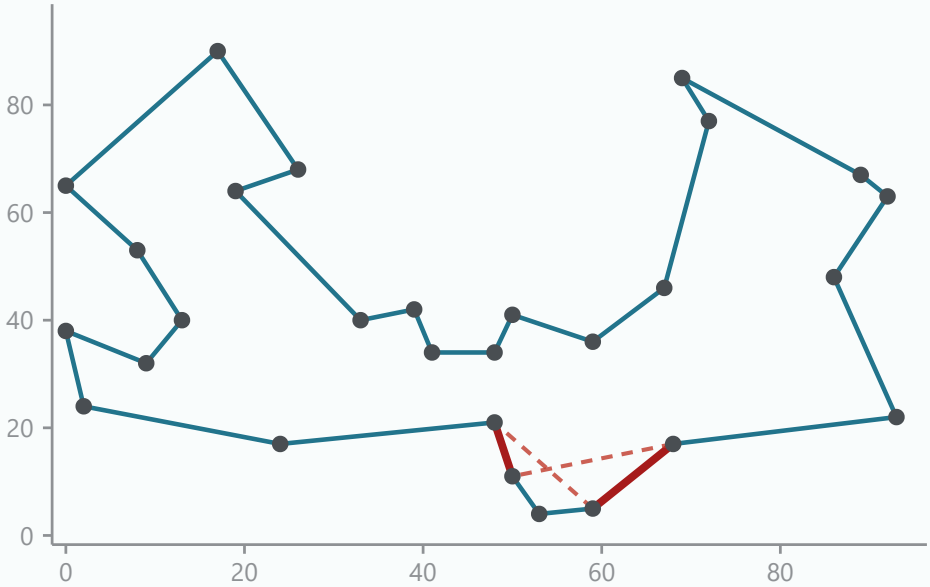

After improving move 22: (a,b), (c,d) removed (dashed), (a,c), (b,d) added (bold). Δ = −37.0 → length **852.8**.

# Improving a Tour with 2-opt




After improving move 32: (a,b), (c,d) removed (dashed), (a,c), (b,d) added (bold). Δ = −11.8 → length **595.9**.

# Improving a Tour with 2-opt




Move 48, the last: no 2-opt move shortens the tour any more → **local optimum**, length **430.8** (34% of the start). No crossing edges remain.

# 2-opt Local Search: Java

```java
void twoOpt(int[] tour, double[][] d) {
  int n = tour.length;
  boolean improved = true;
  while (improved) {
    improved = false;
    for (int i = 0; i < n - 1; i++)
      for (int j = i + 2; j < n; j++) {
        int a = tour[i], b = tour[i + 1];
        int c = tour[j], e = tour[(j + 1) % n];
        double delta = d[a][c] + d[b][e]
                     - d[a][b] - d[c][e];
        if (delta < -1e-9) {  // shorter
          reverse(tour, i + 1, j);
          improved = true;
        }
      }
  }
}
```

### One pass over the tour

$O(n^2)$ candidate moves; each is evaluated in O(1), a reversal costs O(n).

### Local optimum

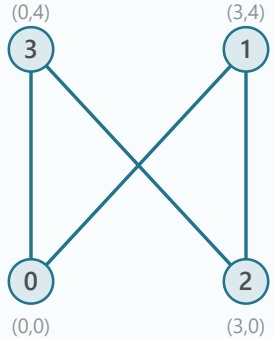
The loop ends when no move gives a gain. The number of passes can be exponential in the worst case but is small in practice.

### A good start

Starting from a nearest-neighbor tour instead of a random one needs fewer moves. Generalizations: 3-opt, Lin–Kernighan.

# 2-opt: Code Trace

```
1  void twoOpt(int[] tour, double[][] d) {
▶    int n = tour.length;
▶    boolean improved = true;
4    while (improved) {
5      improved = false;
6      for (int i = 0; i < n - 1; i++)
7        for (int j = i + 2; j < n; j++) {
8          int a = tour[i], b = tour[i + 1];
9          int c = tour[j], e = tour[(j + 1) %
   n];
10         double delta = d[a][c] + d[b][e]
11                      - d[a][b] - d[c][e];
12         if (delta < -1e-9) {  // shorter
13           reverse(tour, i + 1, j);
14           improved = true;
15         }
16       }
17   }
18 }
```




tour: 0→1→2→3→0

length = 18

Initial tour 0 → 1 → 2 → 3 → 0, length **18**. Edges (0, 1) and (2, 3) cross.

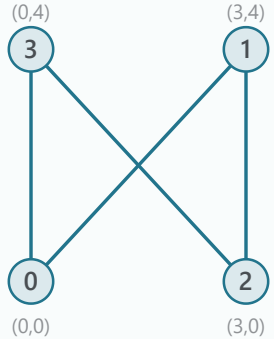
Step 1 / 11

**VARIABLES**

`n = 4` `improved = true` `i = –` `j = –` `delta = –` `tour = [0, 1, 2, 3]`

# 2-opt: Code Trace

```
1  void twoOpt(int[] tour, double[][] d) {
2    int n = tour.length;
3    boolean improved = true;
▶    while (improved) {
▶      improved = false;
6      for (int i = 0; i < n - 1; i++)
7        for (int j = i + 2; j < n; j++) {
8          int a = tour[i], b = tour[i + 1];
9          int c = tour[j], e = tour[(j + 1) %
   n];
10         double delta = d[a][c] + d[b][e]
11                      - d[a][b] - d[c][e];
12         if (delta < -1e-9) {  // shorter
13           reverse(tour, i + 1, j);
14           improved = true;
15         }
16       }
17   }
18 }
```




tour: 0→1→2→3→0

length = 18

pass 1

Pass 1 starts: `improved = false`. If no move is made in this pass, the loop ends.

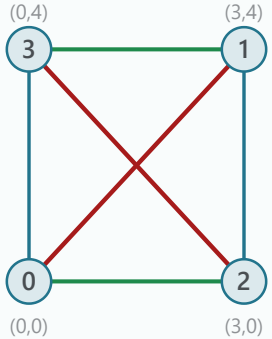
Step 2 / 11

VARIABLES

`n = 4` `improved = false` `i = –` `j = –` `delta = –` `tour = [0, 1, 2, 3]`

# 2-opt: Code Trace

```
1  void twoOpt(int[] tour, double[][] d) {
2    int n = tour.length;
3    boolean improved = true;
4    while (improved) {
5      improved = false;
6      for (int i = 0; i < n - 1; i++)
7        for (int j = i + 2; j < n; j++) {
▶          int a = tour[i], b = tour[i + 1];
▶          int c = tour[j], e = tour[(j + 1) %
   n];
▶          double delta = d[a][c] + d[b][e]
▶                       - d[a][b] - d[c][e];
▶          if (delta < -1e-9) {  // shorter
13           reverse(tour, i + 1, j);
14           improved = true;
15         }
16       }
17   }
18 }
```




tour: 0→1→2→3→0

length = 18

out: 5 + 5

in: 3 + 3

delta = −4

i = 0, j = 2: replace (0,1) and (2,3) with (0,2) and (1,3). delta = 3 + 3 − 5 − 5 = **−4** < 0 → the tour gets shorter.

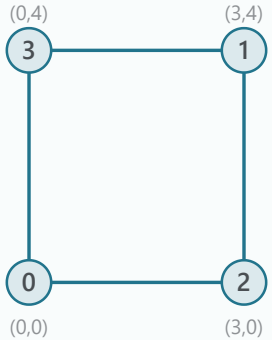
Step 3 / 11

VARIABLES

`n = 4` `improved = false` `i = 0` `j = 2` `a = 0` `b = 1` `c = 2` `e = 3` `delta = -4` `tour = [0, 1, 2, 3]`

# 2-opt: Code Trace

```
1  void twoOpt(int[] tour, double[][] d) {
2    int n = tour.length;
3    boolean improved = true;
4    while (improved) {
5      improved = false;
6      for (int i = 0; i < n - 1; i++)
7        for (int j = i + 2; j < n; j++) {
8          int a = tour[i], b = tour[i + 1];
9          int c = tour[j], e = tour[(j + 1) %
   n];
10         double delta = d[a][c] + d[b][e]
11                      - d[a][b] - d[c][e];
12         if (delta < -1e-9) {  // shorter
▶            reverse(tour, i + 1, j);
▶            improved = true;
15         }
16       }
17   }
18 }
```




tour: 0→2→1→3→0

length = 14

pass 1

`reverse(tour, 1, 2)` : the path in between is reversed → `[0, 2, 1, 3]` , length **14**.
`improved = true` .

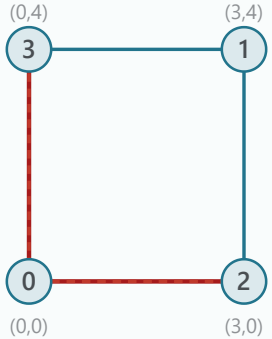
Step 4 / 11

VARIABLES

`n = 4` `improved = true` `i = 0` `j = 2`
`a = 0` `b = 1` `c = 2` `e = 3`
`delta = -4` `tour = [0, 2, 1, 3]`

# 2-opt: Code Trace

```
1  void twoOpt(int[] tour, double[][] d) {
2    int n = tour.length;
3    boolean improved = true;
4    while (improved) {
5      improved = false;
6      for (int i = 0; i < n - 1; i++)
7        for (int j = i + 2; j < n; j++) {
▶          int a = tour[i], b = tour[i + 1];
▶          int c = tour[j], e = tour[(j + 1) %
   n];
▶          double delta = d[a][c] + d[b][e]
▶                       - d[a][b] - d[c][e];
▶          if (delta < -1e-9) {  // shorter
13           reverse(tour, i + 1, j);
14           improved = true;
15         }
16       }
17   }
18 }
```




tour: 0→2→1→3→0

length = 14

out: 3 + 4

in: 4 + 3

delta = 0

i = 0, j = 3: delta = 4 + 3 − 3 − 4 = **0** (the two edges are adjacent: e = a) → no move.

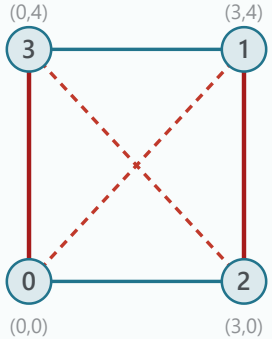
Step 5 / 11

VARIABLES

`n = 4` `improved = true` `i = 0` `j = 3` `a = 0` `b = 2` `c = 3` `e = 0` `delta = 0` `tour = [0, 2, 1, 3]`

# 2-opt: Code Trace

```
1  void twoOpt(int[] tour, double[][] d) {
2    int n = tour.length;
3    boolean improved = true;
4    while (improved) {
5      improved = false;
6      for (int i = 0; i < n - 1; i++)
7        for (int j = i + 2; j < n; j++) {
▶          int a = tour[i], b = tour[i + 1];
▶          int c = tour[j], e = tour[(j + 1) %
   n];
▶          double delta = d[a][c] + d[b][e]
▶                       - d[a][b] - d[c][e];
▶          if (delta < -1e-9) {  // shorter
13           reverse(tour, i + 1, j);
14           improved = true;
15         }
16       }
17   }
18 }
```




tour: 0→2→1→3→0

length = 14

out: 4 + 4

in: 5 + 5

delta = 2

i = 1, j = 3: delta = 5 + 5 − 4 − 4 = **2** → no move.

Step 6 / 11

**VARIABLES**

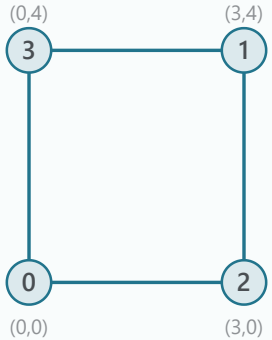
`n = 4` `improved = true` `i = 1` `j = 3`

`a = 2` `b = 1` `c = 3` `e = 0`

`delta = 2` `tour = [0, 2, 1, 3]`

# 2-opt: Code Trace

```
1  void twoOpt(int[] tour, double[][] d) {
2    int n = tour.length;
3    boolean improved = true;
▶    while (improved) {
▶      improved = false;
6      for (int i = 0; i < n - 1; i++)
7        for (int j = i + 2; j < n; j++) {
8          int a = tour[i], b = tour[i + 1];
9          int c = tour[j], e = tour[(j + 1) %
   n];
10         double delta = d[a][c] + d[b][e]
11                      - d[a][b] - d[c][e];
12         if (delta < -1e-9) {  // shorter
13           reverse(tour, i + 1, j);
14           improved = true;
15         }
16       }
17   }
18 }
```




tour: 0→2→1→3→0

length = 14

pass 2

A move was made in the previous pass ( `improved = true` ), so the pass repeats from the start.

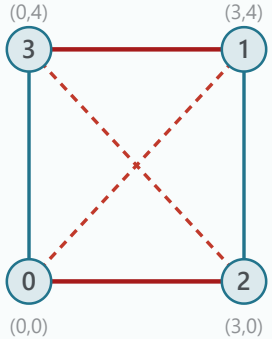
Step 7 / 11

**VARIABLES**

`n = 4` `improved = false` `i = –` `j = –` `delta = –` `tour = [0, 2, 1, 3]`

# 2-opt: Code Trace

```
1  void twoOpt(int[] tour, double[][] d) {
2    int n = tour.length;
3    boolean improved = true;
4    while (improved) {
5      improved = false;
6      for (int i = 0; i < n - 1; i++)
7        for (int j = i + 2; j < n; j++) {
▶          int a = tour[i], b = tour[i + 1];
▶          int c = tour[j], e = tour[(j + 1) %
   n];
▶          double delta = d[a][c] + d[b][e]
▶                       - d[a][b] - d[c][e];
▶          if (delta < -1e-9) {  // shorter
13           reverse(tour, i + 1, j);
14           improved = true;
15         }
16       }
17   }
18 }
```




tour: 0→2→1→3→0

length = 14

out: 3 + 3

in: 5 + 5

delta = 4

i = 0, j = 2: delta = 5 + 5 − 3 − 3 = **4** → no move.

Step 8 / 11

**VARIABLES**

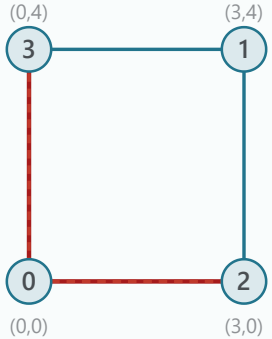
`n = 4` `improved = false` `i = 0` `j = 2`

`a = 0` `b = 2` `c = 1` `e = 3`

`delta = 4` `tour = [0, 2, 1, 3]`

# 2-opt: Code Trace

```
1  void twoOpt(int[] tour, double[][] d) {
2    int n = tour.length;
3    boolean improved = true;
4    while (improved) {
5      improved = false;
6      for (int i = 0; i < n - 1; i++)
7        for (int j = i + 2; j < n; j++) {
▶          int a = tour[i], b = tour[i + 1];
▶          int c = tour[j], e = tour[(j + 1) %
   n];
▶          double delta = d[a][c] + d[b][e]
▶                       - d[a][b] - d[c][e];
▶          if (delta < -1e-9) {  // shorter
13           reverse(tour, i + 1, j);
14           improved = true;
15         }
16       }
17   }
18 }
```




tour: 0→2→1→3→0

length = 14

out: 3 + 4

in: 4 + 3

delta = 0

i = 0, j = 3: delta = 4 + 3 − 3 − 4 = **0** (the two edges are adjacent: e = a) → no move.

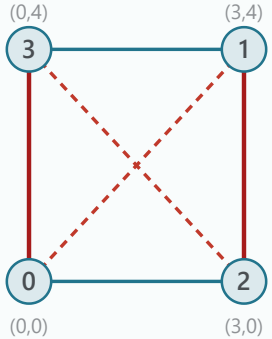
Step 9 / 11

VARIABLES

`n = 4` `improved = false` `i = 0` `j = 3` `a = 0` `b = 2` `c = 3` `e = 0` `delta = 0` `tour = [0, 2, 1, 3]`

# 2-opt: Code Trace

```
1  void twoOpt(int[] tour, double[][] d) {
2    int n = tour.length;
3    boolean improved = true;
4    while (improved) {
5      improved = false;
6      for (int i = 0; i < n - 1; i++)
7        for (int j = i + 2; j < n; j++) {
▶          int a = tour[i], b = tour[i + 1];
▶          int c = tour[j], e = tour[(j + 1) %
   n];
▶          double delta = d[a][c] + d[b][e]
▶                       - d[a][b] - d[c][e];
▶          if (delta < -1e-9) {  // shorter
13           reverse(tour, i + 1, j);
14           improved = true;
15         }
16       }
17   }
18 }
```




tour: 0→2→1→3→0

length = 14

out: 4 + 4

in: 5 + 5

delta = 2

i = 1, j = 3: delta = 5 + 5 − 4 − 4 = **2** → no move.

Step 10 / 11

**VARIABLES**

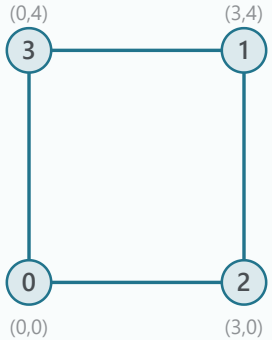
`n = 4` `improved = false` `i = 1` `j = 3`

`a = 2` `b = 1` `c = 3` `e = 0`

`delta = 2` `tour = [0, 2, 1, 3]`

# 2-opt: Code Trace

```
1  void twoOpt(int[] tour, double[][] d) {
2    int n = tour.length;
3    boolean improved = true;
▶    while (improved) {
5      improved = false;
6      for (int i = 0; i < n - 1; i++)
7        for (int j = i + 2; j < n; j++) {
8          int a = tour[i], b = tour[i + 1];
9          int c = tour[j], e = tour[(j + 1) %
   n];
10         double delta = d[a][c] + d[b][e]
11                      - d[a][b] - d[c][e];
12         if (delta < -1e-9) {  // shorter
13           reverse(tour, i + 1, j);
14           improved = true;
15         }
16       }
17   }
18 }
```




tour: 0→2→1→3→0

length = 14

local optimum

No shortening found in the whole pass: the loop ends. Tour `[0, 2, 1, 3]`, length 14: a 2-opt local optimum (and for these 4 points also the optimum).

Step 11 / 11

VARIABLES

`n = 4` `improved = false` `i = –` `j = –` `delta = –` `tour = [0, 2, 1, 3]`

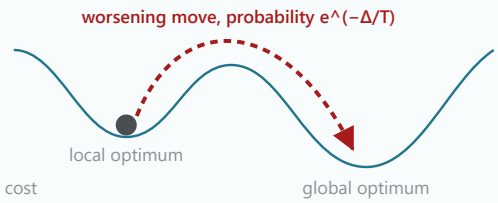
# Escaping Local Optima: Metaheuristics

**Simulated annealing**
A worsening move is also accepted with probability $e^{-\Delta/T}$; the "temperature" T is lowered slowly. It can escape local optima.

**Genetic algorithm**
A **population** of tours improved over generations by selection, crossover and mutation.

**Ant colony**
Ants leave more **pheromone** on the edges of short tours; later ants prefer edges with dense pheromone.

- All three are common in TSP, vehicle routing, job scheduling and assignment problems.
- None of them **guarantees** an approximation ratio; the result depends on the initial solution and the parameters. Typical settings: $T_0 = 100$, $T \leftarrow 0.997 \cdot T$ at each step; a population of 100, crossover at a random cut point, mutation probability 0.05.




Local search that accepts only improving moves stays in the left valley. Simulated annealing can climb over the hill at high T; as T drops, moves become pickier.

# Job Scheduling

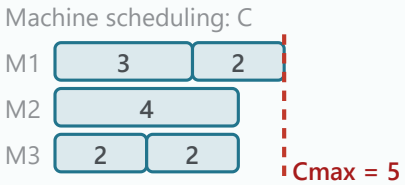
- **Optimization problem:** planning jobs on given resources (workers, machines, processors) **as efficiently as possible**.
- **Goal:** minimize the completion time of the jobs, or optimize other criteria (profit, lateness, deadlines).

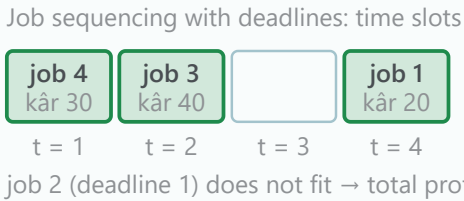
### Machine scheduling (makespan)

n jobs, m identical machines; job j takes $p_j$. Minimize the time the last job finishes, $\mathbf{C_{max}}$ (the makespan). NP-hard for m ≥ 2.

### Job sequencing with deadlines

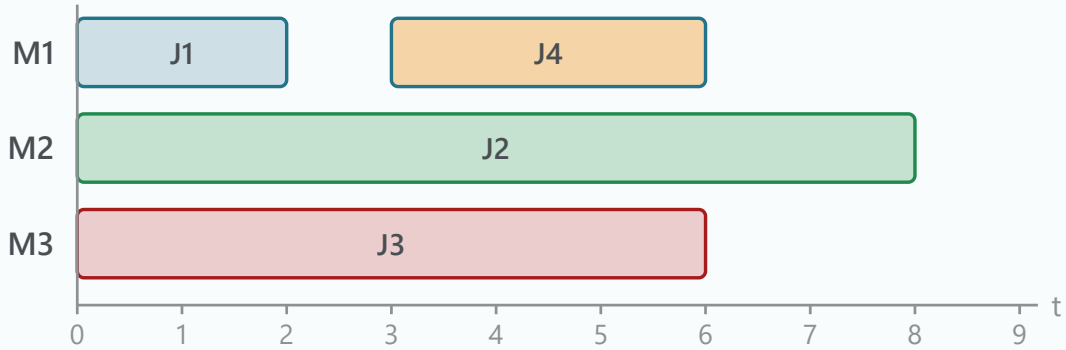
Each job takes 1 unit and has a **profit** and a **deadline**. Maximize the total profit of the jobs finished by their deadlines.



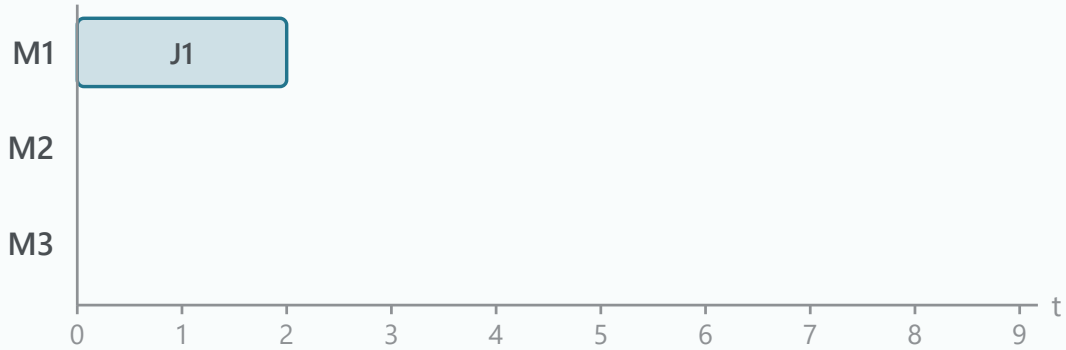

The first problem has greedy **approximation** algorithms (list scheduling, LPT); the second is solved **optimally** by a greedy algorithm.

# Assigning Jobs to Machines: List Scheduling



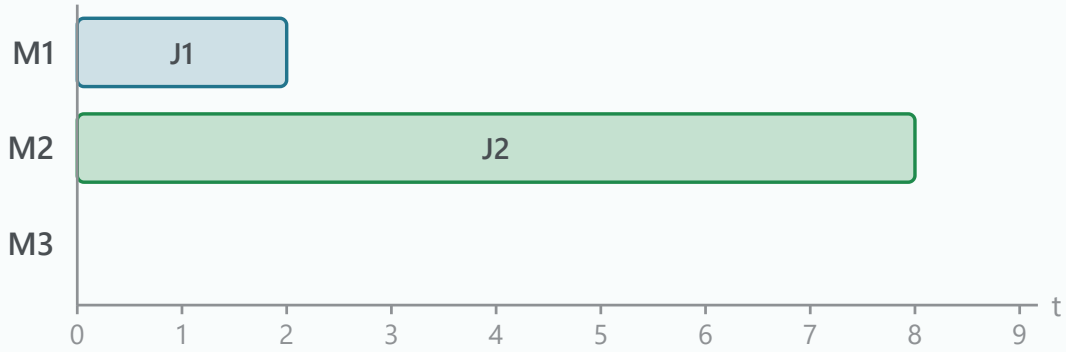

Three identical machines M1–M3. Jobs arrive in order: J1 (2), J2 (8), J3 (6) ready at t = 0; J4 (3) arrives at t = 3. **List scheduling** (Graham): the next job goes to the machine that **becomes free first**.

# Assigning Jobs to Machines: List Scheduling



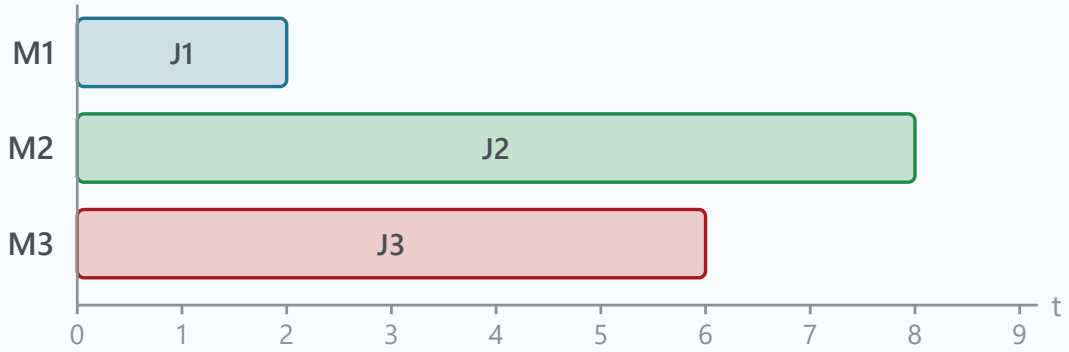

J1 (time 2) → **M1**, runs in [0, 2] (all machines are free; the first one is chosen).

# Assigning Jobs to Machines: List Scheduling



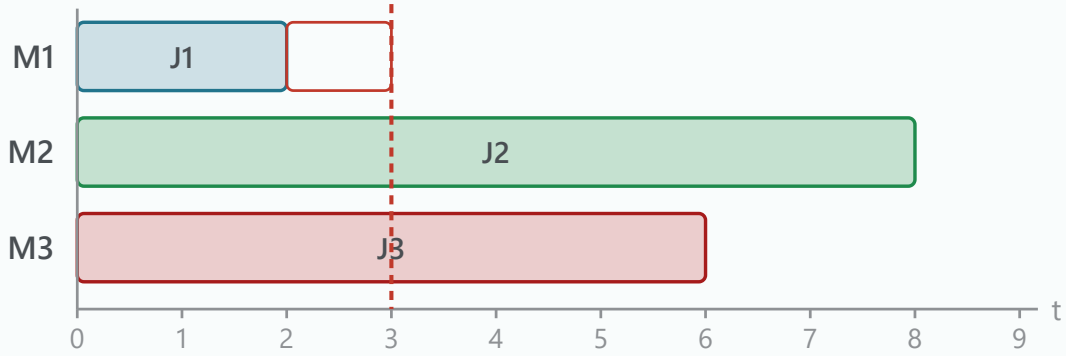

J2 (time 8) → **M2**, runs in [0, 8] (M1 is busy until t = 2).

# Assigning Jobs to Machines: List Scheduling



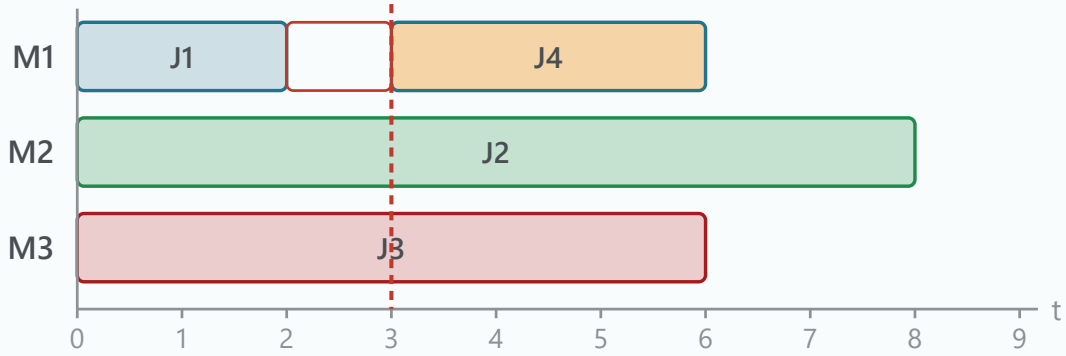

J3 (time 6) → **M3**, runs in [0, 6] (M1 is busy until t = 2; M2 until t = 8).

# Assigning Jobs to Machines: List Scheduling



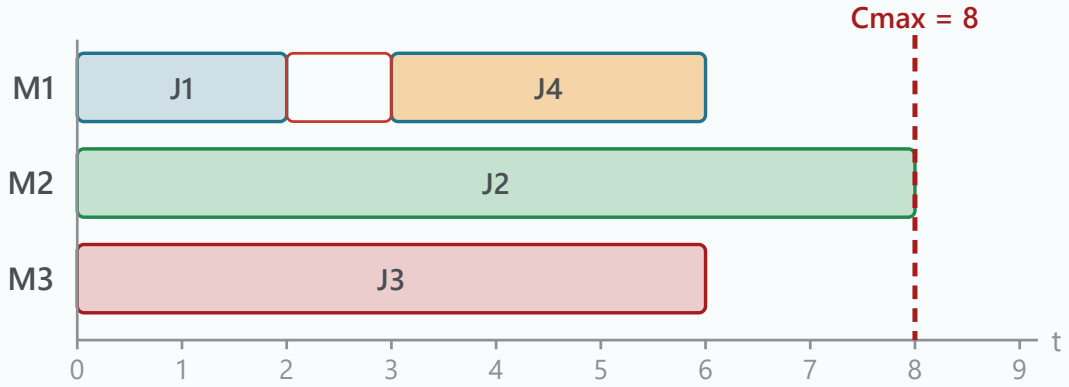

t = 2: M1 is free but no job is waiting; the machine stays **idle** until t = 3 (hatched area).

# Assigning Jobs to Machines: List Scheduling




J4 (time 3) → **M1**, runs in [3, 6] (M2 is busy until t = 8; M3 until t = 6).

# Assigning Jobs to Machines: List Scheduling




All jobs assigned: $\mathbf{C_{max} = 8}$ (M2). Lower bound: OPT ≥ the longest job = 8 → this schedule is **optimal**. In general, though, list scheduling only guarantees a ratio of (2 − 1/m).

# List Scheduling: A (2 − 1/m)-Approximation

**Two lower bounds:** OPT ≥ $\Sigma p_j / m$ (the work must be shared) and OPT ≥ max $p_j$.

1. Let job j, the last to finish, start at time $s_j$.
2. Before $s_j$ **all machines were busy** (otherwise j would have started earlier) → $s_j \le (\Sigma p - p_j) / m$.
3. $C_{max} = s_j + p_j \le \Sigma p / m + (1 - 1/m) \cdot p_j \le OPT + (1 - 1/m) \cdot OPT$.

$C_{max} \le (2 - 1/m) \cdot OPT$; time $O(n \log m)$ (the earliest-free machine is kept in a heap).

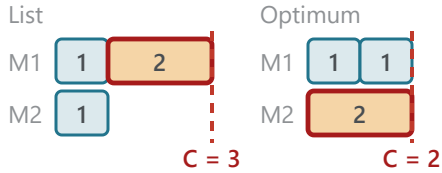
### The bound is tight

m = 2, jobs in the order 1, 1, 2:

• List: M1 = 1 + 2 = 3, M2 = 1 → $C_{max} = 3$

• Optimum: {1, 1} and {2} → 2

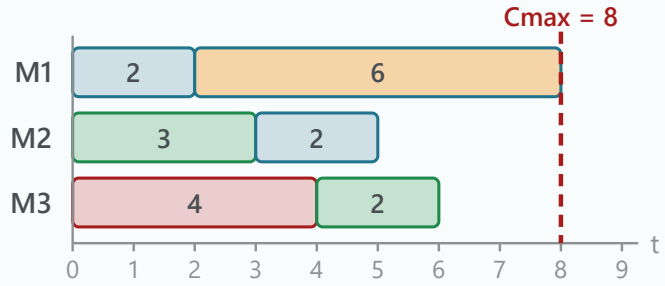
Ratio 3/2 = 2 − 1/2.



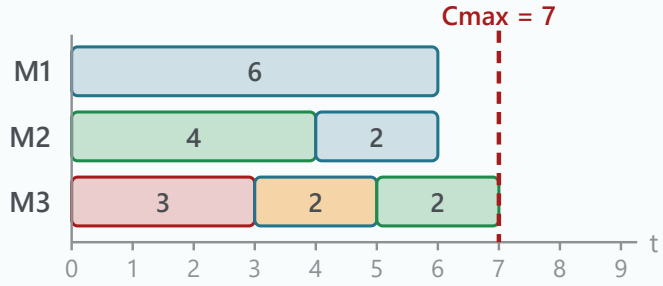

In general, m(m − 1) jobs of 1 unit followed by a single job of m units reach the ratio 2 − 1/m.

# Longest Processing Time First (LPT): A 4/3-Approximation

LPT (Longest Processing Time first): sort the jobs by **decreasing time**, then apply list scheduling. Jobs: 2, 3, 4, 6, 2, 2 and m = 3.




Given order: $C_{max} = 8$

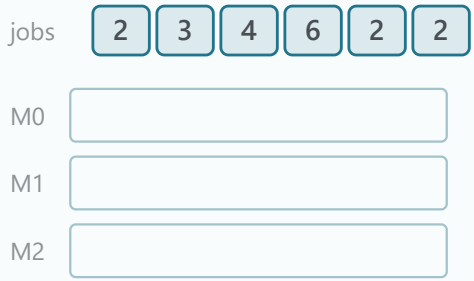



LPT order 6, 4, 3, 2, 2, 2: $C_{max} = 7 = OPT$ ($\geq \lceil 19/3 \rceil = 7$)

- With long jobs first, the short jobs left at the end even out the imbalance: $C_{max} \leq (4/3 - 1/(3m)) \cdot OPT$ (Graham, 1969). Time O(n log n).
- Tight example (m = 3): 5, 5, 4, 4, 3, 3, 3 → LPT 11, optimum 9 ({5, 4}, {5, 4}, {3, 3, 3}).

# LPT Scheduling: Code Trace

```
▶ int lpt(Integer[] jobs, int m) {
2   Arrays.sort(jobs, Comparator.reverseOrder());
3   int[] load = new int[m];
4   for (int p : jobs) {
5     int k = 0;              // least loaded
6     for (int i = 1; i < m; i++)
7       if (load[i] < load[k]) k = i;
8     load[k] += p;
9   }
10  return Arrays.stream(load).max().getAsInt();
11 }
```




Jobs 2, 3, 4, 6, 2, 2 and m = 3 machines.
List scheduling in this order gives $C_{max}$ = 8.

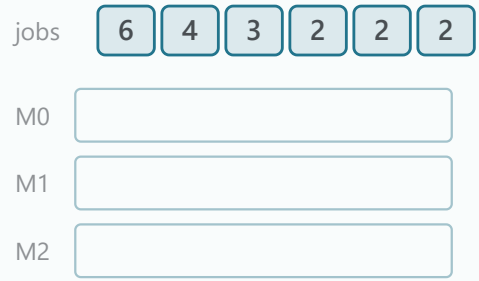
Step 1 / 10

**VARIABLES**

`jobs = [2, 3, 4, 6, 2, 2]` `m = 3`

# LPT Scheduling: Code Trace

```
1  int lpt(Integer[] jobs, int m) {
2    Arrays.sort(jobs, Comparator.reverseOrder());
3    int[] load = new int[m];
4    for (int p : jobs) {
5      int k = 0;                // least loaded
6      for (int i = 1; i < m; i++)
7        if (load[i] < load[k]) k = i;
8      load[k] += p;
9    }
10   return Arrays.stream(load).max().getAsInt();
11 }
```




LPT: sort the jobs by time in **decreasing** order → 6, 4, 3, 2, 2, 2.

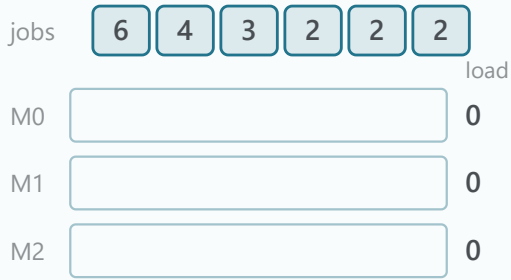
Step 2 / 10

VARIABLES

`jobs = [6, 4, 3, 2, 2, 2]` `m = 3`

# LPT Scheduling: Code Trace

```
1  int lpt(Integer[] jobs, int m) {
2    Arrays.sort(jobs, Comparator.reverseOrder());
▶    int[] load = new int[m];
4    for (int p : jobs) {
5      int k = 0;               // least loaded
6      for (int i = 1; i < m; i++)
7        if (load[i] < load[k]) k = i;
8      load[k] += p;
9    }
10   return Arrays.stream(load).max().getAsInt();
11 }
```




All three machines have load 0. Each job goes to the **least loaded** machine at that moment.

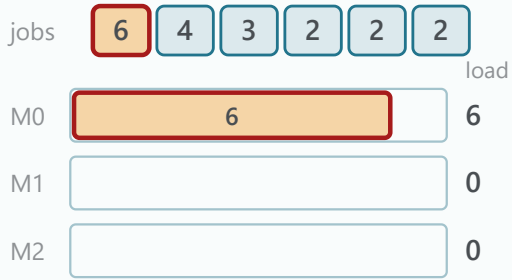
Step 3 / 10

VARIABLES

`load = [0, 0, 0]` `p = –` `k = –`

# LPT Scheduling: Code Trace

```
1  int lpt(Integer[] jobs, int m) {
2    Arrays.sort(jobs, Comparator.reverseOrder());
3    int[] load = new int[m];
4    for (int p : jobs) {
▶      int k = 0;              // least loaded
▶      for (int i = 1; i < m; i++)
▶        if (load[i] < load[k]) k = i;
▶      load[k] += p;
9    }
10   return Arrays.stream(load).max().getAsInt();
11 }
```




p = 6: the least loaded machine is k = 0 (load 0) (on ties, `<` keeps the first machine) → `load = [6, 0, 0]`.

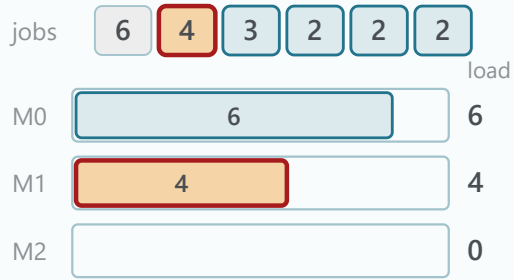
Step 4 / 10

**VARIABLES**

`load = [6, 0, 0]` `p = 6` `k = 0`

# LPT Scheduling: Code Trace

```
1  int lpt(Integer[] jobs, int m) {
2    Arrays.sort(jobs, Comparator.reverseOrder());
3    int[] load = new int[m];
4    for (int p : jobs) {
▶      int k = 0;              // least loaded
▶      for (int i = 1; i < m; i++)
▶        if (load[i] < load[k]) k = i;
▶      load[k] += p;
9    }
10   return Arrays.stream(load).max().getAsInt();
11 }
```




p = 4: the least loaded machine is k = 1 (load 0) → `load = [6, 4, 0]`.

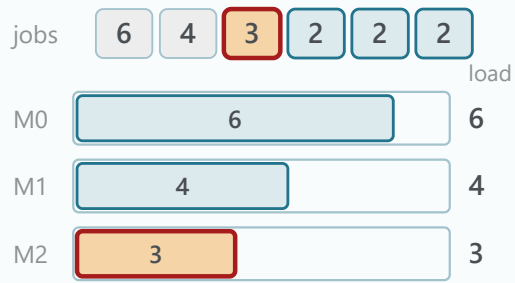
Step 5 / 10

**VARIABLES**

`load = [6, 4, 0]` `p = 4` `k = 1`

# LPT Scheduling: Code Trace

```
1  int lpt(Integer[] jobs, int m) {
2    Arrays.sort(jobs, Comparator.reverseOrder());
3    int[] load = new int[m];
4    for (int p : jobs) {
▶      int k = 0;              // least loaded
▶      for (int i = 1; i < m; i++)
▶        if (load[i] < load[k]) k = i;
▶      load[k] += p;
9    }
10   return Arrays.stream(load).max().getAsInt();
11 }
```




p = 3: the least loaded machine is k = 2 (load 0) → `load = [6, 4, 3]`.

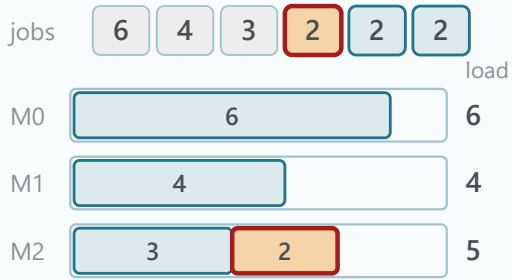
Step 6 / 10

**VARIABLES**

`load = [6, 4, 3]` `p = 3` `k = 2`

# LPT Scheduling: Code Trace

```
1  int lpt(Integer[] jobs, int m) {
2    Arrays.sort(jobs, Comparator.reverseOrder());
3    int[] load = new int[m];
4    for (int p : jobs) {
▶      int k = 0;              // least loaded
▶      for (int i = 1; i < m; i++)
▶        if (load[i] < load[k]) k = i;
▶      load[k] += p;
9    }
10   return Arrays.stream(load).max().getAsInt();
11 }
```




p = 2: the least loaded machine is k = 2 (load 3) → `load = [6, 4, 5]`.

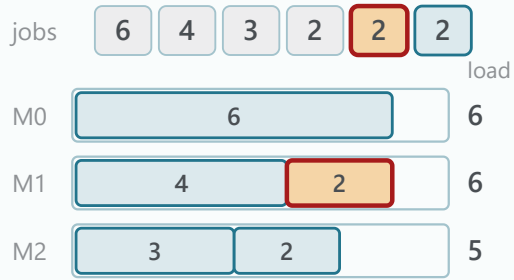
Step 7 / 10

**VARIABLES**

`load = [6, 4, 5]` `p = 2` `k = 2`

# LPT Scheduling: Code Trace

```
1  int lpt(Integer[] jobs, int m) {
2    Arrays.sort(jobs, Comparator.reverseOrder());
3    int[] load = new int[m];
4    for (int p : jobs) {
▶      int k = 0;              // least loaded
▶      for (int i = 1; i < m; i++)
▶        if (load[i] < load[k]) k = i;
▶      load[k] += p;
9    }
10   return Arrays.stream(load).max().getAsInt();
11 }
```




p = 2: the least loaded machine is k = 1 (load 4) → `load = [6, 6, 5]`.

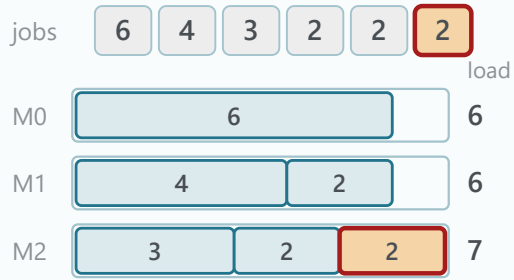
Step 8 / 10

**VARIABLES**

`load = [6, 6, 5]` `p = 2` `k = 1`

# LPT Scheduling: Code Trace

```
1  int lpt(Integer[] jobs, int m) {
2    Arrays.sort(jobs, Comparator.reverseOrder());
3    int[] load = new int[m];
4    for (int p : jobs) {
▶      int k = 0;              // least loaded
▶      for (int i = 1; i < m; i++)
▶        if (load[i] < load[k]) k = i;
▶      load[k] += p;
9    }
10   return Arrays.stream(load).max().getAsInt();
11 }
```




p = 2: the least loaded machine is k = 2 (load 5) → `load = [6, 6, 7]`.

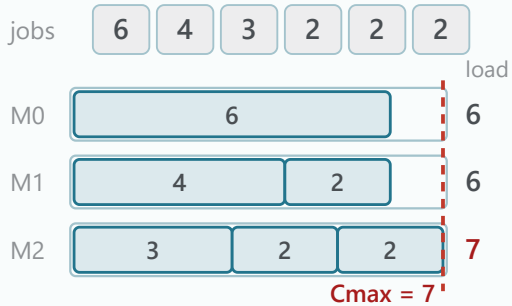
Step 9 / 10

VARIABLES

`load = [6, 6, 7]` `p = 2` `k = 2`

# LPT Scheduling: Code Trace

```
1  int lpt(Integer[] jobs, int m) {
2    Arrays.sort(jobs, Comparator.reverseOrder());
3    int[] load = new int[m];
4    for (int p : jobs) {
5      int k = 0;              // least loaded
6      for (int i = 1; i < m; i++)
7        if (load[i] < load[k]) k = i;
8      load[k] += p;
9    }
▶    return Arrays.stream(load).max().getAsInt();
11 }
```




Makespan $C_{max}$ = **7**. Since the lower bound is $\lceil 19/3 \rceil = 7$, this schedule is optimal.

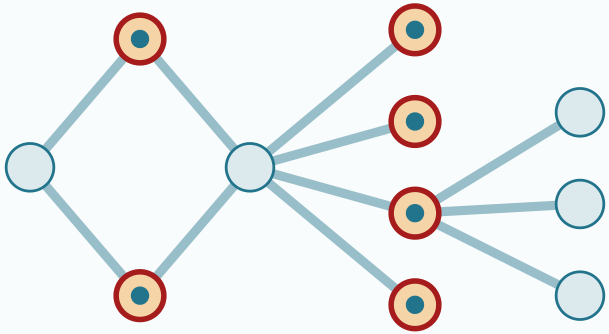
Step 10 / 10

**VARIABLES**

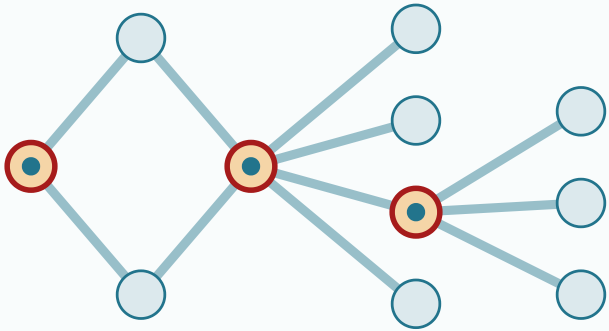
`load = [6, 6, 7]` `returns = 7`

# Vertex Cover

- **Covering problem:** choose the fewest vertices that cover **every edge** of a graph. An edge is covered if **at least one** of its endpoints is chosen.
- **Goal:** find a set C ⊆ V covering all edges with as few vertices as possible. The decision version ("is there a cover of k vertices?") is **NP-complete**.



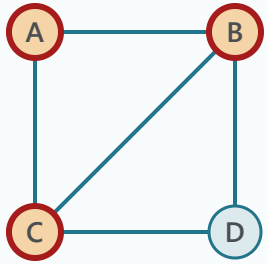
A valid cover of size 6 (every edge is watched by a "guard")



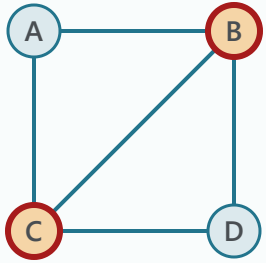
A minimum cover of the same graph, size 3

# Example: A Four-Vertex Graph

- **Vertices:** A, B, C, D
- **Edges:** (A, B), (A, C), (B, C), (B, D), (C, D)
- {A, B, C} is a vertex cover: every edge has at least one endpoint in the set.
- But it is not the **smallest**: {B, C} also covers all five edges.
- A single vertex is not enough (B does not cover (A, C)) → the optimum is **2**.




{A, B, C}: valid, size 3




{B, C}: minimum, size 2

# Vertex Cover: Solution Approaches

### Exact solution

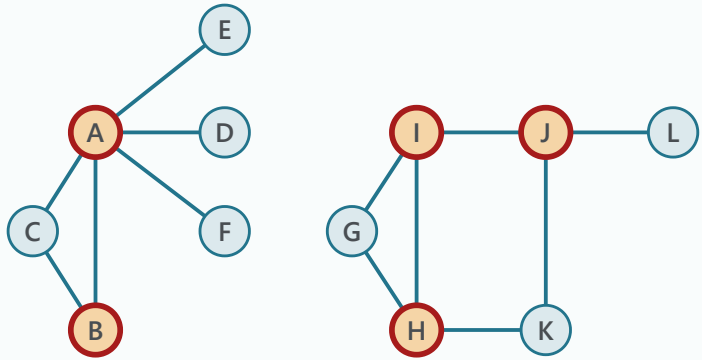
Check every possible vertex set: $2^n$ subsets × O(m) checks.

### Approximate solutions

Greedy algorithms (highest-degree vertex), the edge-picking 2-approximation.

### Optimization tools

Linear programming (LP relaxation + rounding), branch and bound.




Minimum cover {A, B, H, I, J} (5 vertices)

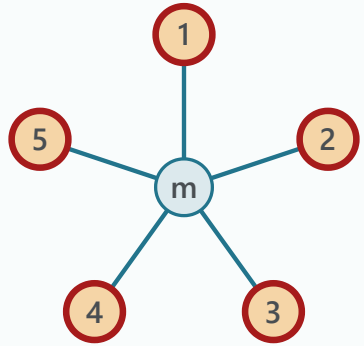
# Vertex Cover Algorithm

**Naive idea:**

1. Start with an empty cover.
2. Pick an uncovered edge and add one of its endpoints to the set.
3. Repeat until every edge is covered.

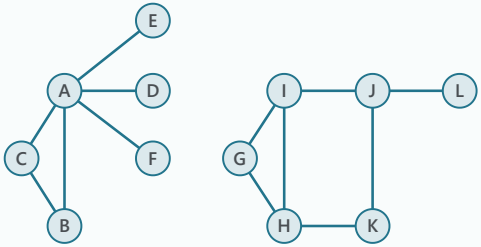
Adding **only one** endpoint gives no guarantee. The correct **2-approximation**:

1. Pick an uncovered edge (u, v) and add **both u and v**.
2. Every edge touching u or v is now covered; repeat until no edges remain.




If the leaf end of every edge is chosen: 5 vertices; the optimum is {m}, 1 vertex. On a star with n leaves the ratio is n.

# Vertex Cover 2-Approximation: Code Trace




The cover is empty. The edges are visited in order: (A,B), (A,C), ..., (J,L).

Step 1 / 15

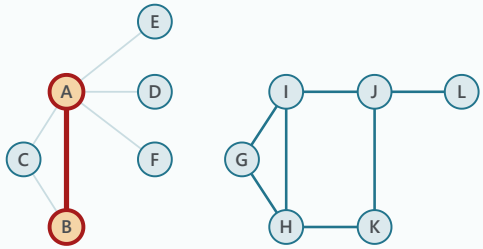
```
1 Set<Character> vertexCover(char[][] edges) {
▶   Set<Character> cover = new HashSet<>();
3   for (char[] e : edges) {
4     char u = e[0], v = e[1];
5     if (!cover.contains(u) && !cover.contains(v)) {
6       cover.add(u); cover.add(v); }  // both endpoints
7   }
8   return cover;
9 }
```

VARIABLES

e = –

cover = {}

# Vertex Cover 2-Approximation: Code Trace




(A, B): neither endpoint is in the cover → add **A and B**. These edges form a **matching**.

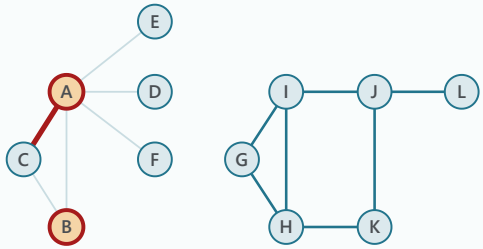
Step 2 / 15

```
1 Set<Character> vertexCover(char[][] edges) {
2   Set<Character> cover = new HashSet<>();
3   for (char[] e : edges) {
▶     char u = e[0], v = e[1];
▶     if (!cover.contains(u) && !cover.contains(v)) {
▶       cover.add(u); cover.add(v); }  // both endpoints
7   }
8   return cover;
9 }
```

VARIABLES

`e = (A,B)` `cover = {A, B}`

# Vertex Cover 2-Approximation: Code Trace




(A, C): A is already in the cover → the edge is covered, skip.

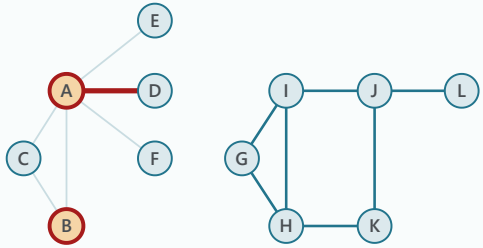
Step 3 / 15

```
1 Set<Character> vertexCover(char[][] edges) {
2   Set<Character> cover = new HashSet<>();
3   for (char[] e : edges) {
▶     char u = e[0], v = e[1];
▶     if (!cover.contains(u) && !cover.contains(v)) {
6       cover.add(u); cover.add(v); }  // both endpoints
7   }
8   return cover;
9 }
```

VARIABLES

e = (A,C)  cover = {A, B}

# Vertex Cover 2-Approximation: Code Trace




(A, D): A is already in the cover → the edge is covered, skip.

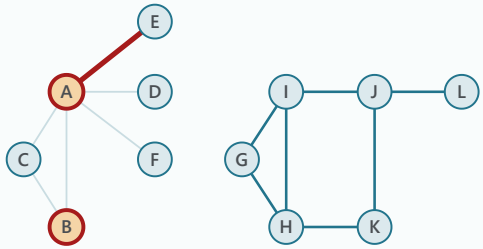
Step 4 / 15

```
1 Set<Character> vertexCover(char[][] edges) {
2   Set<Character> cover = new HashSet<>();
3   for (char[] e : edges) {
▶     char u = e[0], v = e[1];
▶     if (!cover.contains(u) && !cover.contains(v)) {
6       cover.add(u); cover.add(v); }  // both endpoints
7   }
8   return cover;
9 }
```

VARIABLES

e = (A,D)  cover = {A, B}

# Vertex Cover 2-Approximation: Code Trace




(A, E): A is already in the cover → the edge is covered, skip.

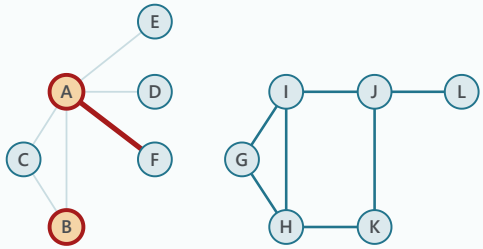
Step 5 / 15

```
1 Set<Character> vertexCover(char[][] edges) {
2   Set<Character> cover = new HashSet<>();
3   for (char[] e : edges) {
▶     char u = e[0], v = e[1];
▶     if (!cover.contains(u) && !cover.contains(v)) {
6       cover.add(u); cover.add(v); }  // both endpoints
7   }
8   return cover;
9 }
```

VARIABLES

`e = {A,E}` `cover = {A, B}`

# Vertex Cover 2-Approximation: Code Trace




(A, F): A is already in the cover → the edge is covered, skip.

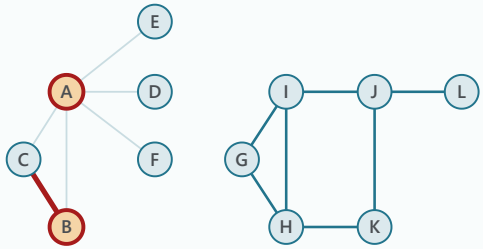
Step 6 / 15

```
1 Set<Character> vertexCover(char[][] edges) {
2   Set<Character> cover = new HashSet<>();
3   for (char[] e : edges) {
▶     char u = e[0], v = e[1];
▶     if (!cover.contains(u) && !cover.contains(v)) {
6       cover.add(u); cover.add(v); }  // both endpoints
7   }
8   return cover;
9 }
```

VARIABLES

e = {A,F}  cover = {A, B}

# Vertex Cover 2-Approximation: Code Trace




(B, C): B is already in the cover → the edge is covered, skip.

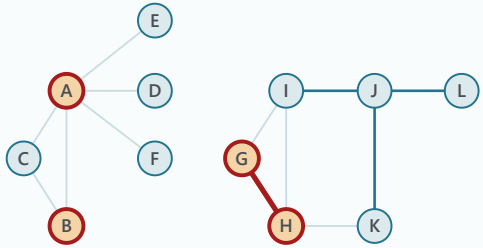
Step 7 / 15

```
1 Set<Character> vertexCover(char[][] edges) {
2   Set<Character> cover = new HashSet<>();
3   for (char[] e : edges) {
▶     char u = e[0], v = e[1];
▶     if (!cover.contains(u) && !cover.contains(v)) {
6       cover.add(u); cover.add(v); }  // both endpoints
7   }
8   return cover;
9 }
```

VARIABLES

e = (B,C)  cover = {A, B}

# Vertex Cover 2-Approximation: Code Trace




(G, H): neither endpoint is in the cover → add **G and H**. These edges form a **matching**.



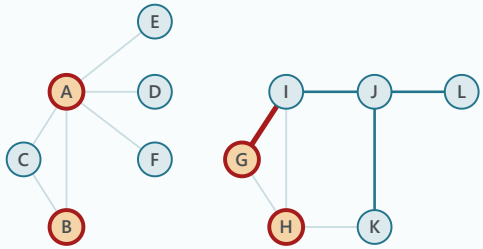
```
1 Set<Character> vertexCover(char[][] edges) {
2   Set<Character> cover = new HashSet<>();
3   for (char[] e : edges) {
▶     char u = e[0], v = e[1];
▶     if (!cover.contains(u) && !cover.contains(v)) {
▶       cover.add(u); cover.add(v); }  // both endpoints
7   }
8   return cover;
9 }
```

VARIABLES

e = (G,H)

cover = {A, B, G, H}

# Vertex Cover 2-Approximation: Code Trace




(G, I): G is already in the cover → the edge is covered, skip.

Step 9 / 15

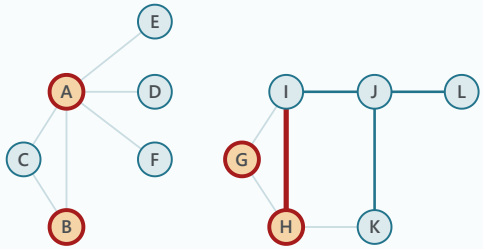
```
1 Set<Character> vertexCover(char[][] edges) {
2   Set<Character> cover = new HashSet<>();
3   for (char[] e : edges) {
▶     char u = e[0], v = e[1];
▶     if (!cover.contains(u) && !cover.contains(v)) {
6       cover.add(u); cover.add(v); }  // both endpoints
7   }
8   return cover;
9 }
```

VARIABLES

e = (G,I)

cover = {A, B, G, H}

# Vertex Cover 2-Approximation: Code Trace




(H, I): H is already in the cover → the edge is covered, skip.

Step 10 / 15

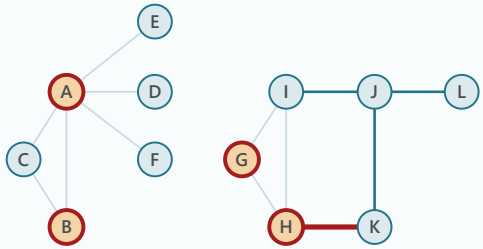
```
1 Set<Character> vertexCover(char[][] edges) {
2   Set<Character> cover = new HashSet<>();
3   for (char[] e : edges) {
▶     char u = e[0], v = e[1];
▶     if (!cover.contains(u) && !cover.contains(v)) {
6       cover.add(u); cover.add(v); }  // both endpoints
7   }
8   return cover;
9 }
```

**VARIABLES**

`e = (H,I)`

`cover = {A, B, G, H}`

# Vertex Cover 2-Approximation: Code Trace




(H, K): H is already in the cover → the edge is covered, skip.

Step 11 / 15

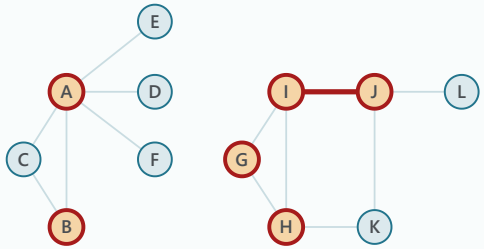
```
1 Set<Character> vertexCover(char[][] edges) {
2   Set<Character> cover = new HashSet<>();
3   for (char[] e : edges) {
▶     char u = e[0], v = e[1];
▶     if (!cover.contains(u) && !cover.contains(v)) {
6       cover.add(u); cover.add(v); }  // both endpoints
7   }
8   return cover;
9 }
```

VARIABLES

e = (H,K)

cover = {A, B, G, H}

# Vertex Cover 2-Approximation: Code Trace




(I, J): neither endpoint is in the cover → add **I and J**. These edges form a **matching**.

Step 12 / 15

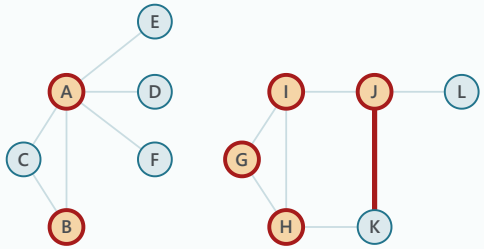
```
1 Set<Character> vertexCover(char[][] edges) {
2   Set<Character> cover = new HashSet<>();
3   for (char[] e : edges) {
▶     char u = e[0], v = e[1];
▶     if (!cover.contains(u) && !cover.contains(v)) {
▶       cover.add(u); cover.add(v); }  // both endpoints
7   }
8   return cover;
9 }
```

VARIABLES

e = (I,J)

cover = {A, B, G, H, I, J}

# Vertex Cover 2-Approximation: Code Trace




(J, K): J is already in the cover → the edge is covered, skip.

Step 13 / 15

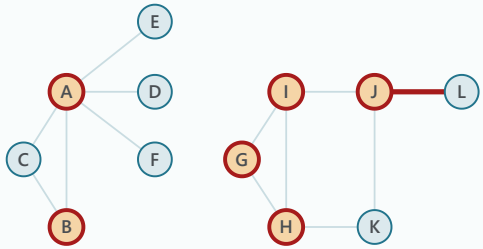
```
1 Set<Character> vertexCover(char[][] edges) {
2   Set<Character> cover = new HashSet<>();
3   for (char[] e : edges) {
▶     char u = e[0], v = e[1];
▶     if (!cover.contains(u) && !cover.contains(v)) {
6       cover.add(u); cover.add(v); }  // both endpoints
7   }
8   return cover;
9 }
```

VARIABLES

e = (J,K)

cover = {A, B, G, H, I, J}

# Vertex Cover 2-Approximation: Code Trace




(J, L): J is already in the cover → the edge is covered, skip.

Step 14 / 15

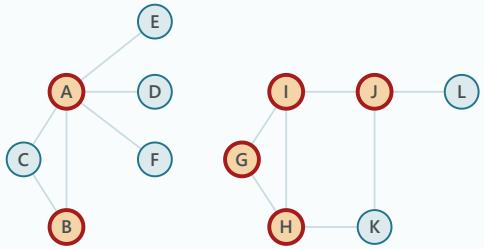
```
1 Set<Character> vertexCover(char[][] edges) {
2   Set<Character> cover = new HashSet<>();
3   for (char[] e : edges) {
▶     char u = e[0], v = e[1];
▶     if (!cover.contains(u) && !cover.contains(v)) {
6       cover.add(u); cover.add(v); }  // both endpoints
7   }
8   return cover;
9 }
```

VARIABLES

e = (J,L)

cover = {A, B, G, H, I, J}

# Vertex Cover 2-Approximation: Code Trace




Result {A, B, G, H, I, J}: 6 vertices. The optimum is 5 ({A, B, H, I, J}) → ratio 6/5 = 1.2 ≤ 2.

Step 15 / 15

```
1 Set<Character> vertexCover(char[][] edges) {
2   Set<Character> cover = new HashSet<>();
3   for (char[] e : edges) {
4     char u = e[0], v = e[1];
5     if (!cover.contains(u) && !cover.contains(v)) {
6       cover.add(u); cover.add(v); }  // both endpoints
7   }
▶   return cover;
9 }
```

VARIABLES

e = –

cover = {A, B, G, H, I, J}

# Why At Most 2 × Optimum?

1. Let M be the set of edges the algorithm picks. Both endpoints of each picked edge are added, so the edges of M **share no endpoints** (M is a matching).
2. Any cover must contain a **distinct** vertex for each edge of M → **OPT ≥ |M|**.
3. The algorithm picks exactly **|C| = 2 |M|** vertices → |C| ≤ 2 · OPT.

### Time

O(V + E): each edge is examined once.

### Greedy by degree

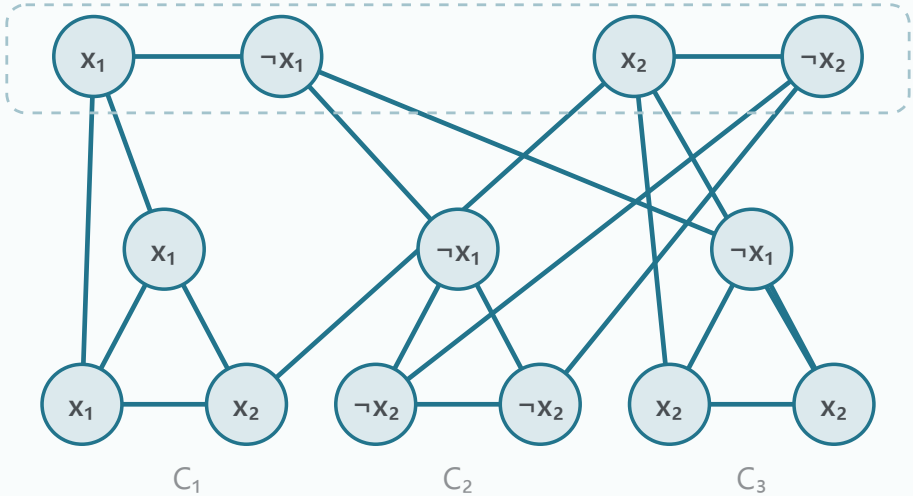
Picking the highest-degree vertex is intuitive, but its ratio is only O(log n).

### LP rounding

Solve the relaxation $x_u + x_v \geq 1$ and pick the vertices with $x_v \geq 1/2$ → again a 2-approximation.

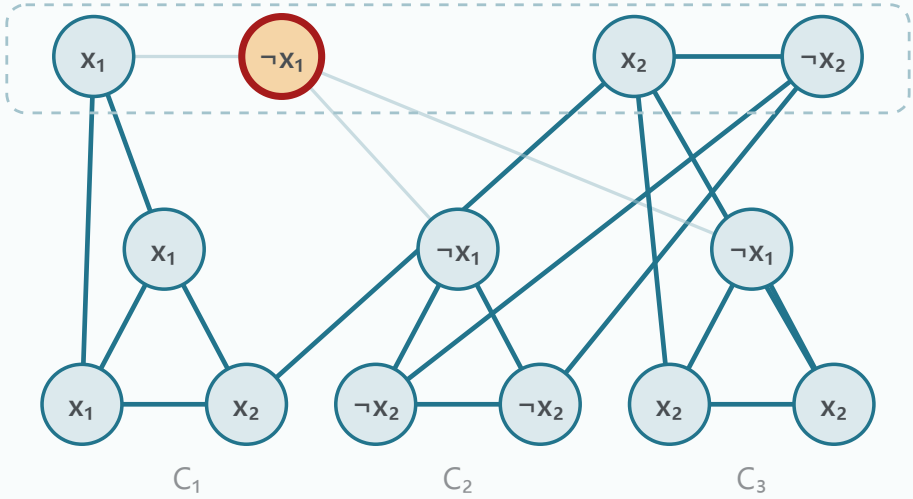
No constant ratio noticeably better than 2 is known; a ratio of 2 − ε has been shown impossible (under the Unique Games Conjecture).

# NP-Completeness: Reducing 3-SAT to Vertex Cover



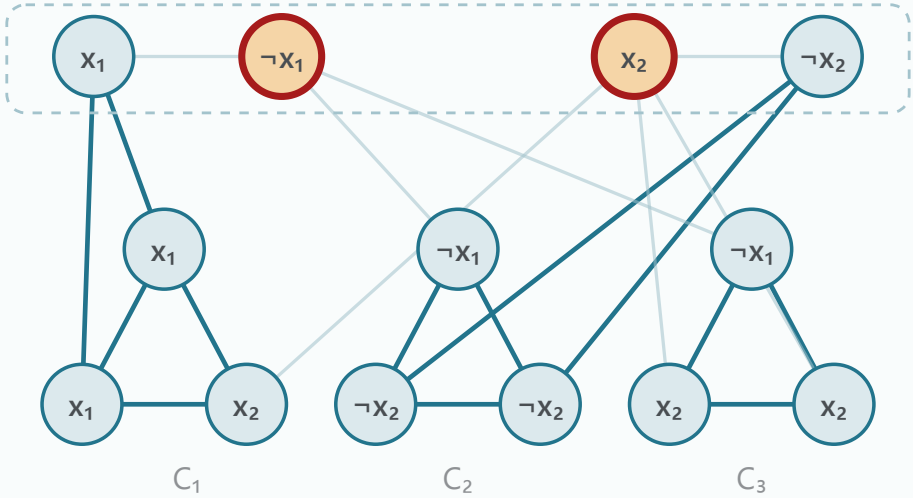

$\varphi = (x_1 \vee x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_2) \wedge (\neg x_1 \vee x_2 \vee x_2)$. One edge per variable (top boxes), one triangle per clause; each literal in a clause is joined to the variable vertex with the same name. $l = 2$ variables, $m = 3$ clauses → $k = l + 2m =$ **8**.

# NP-Completeness: Reducing 3-SAT to Vertex Cover



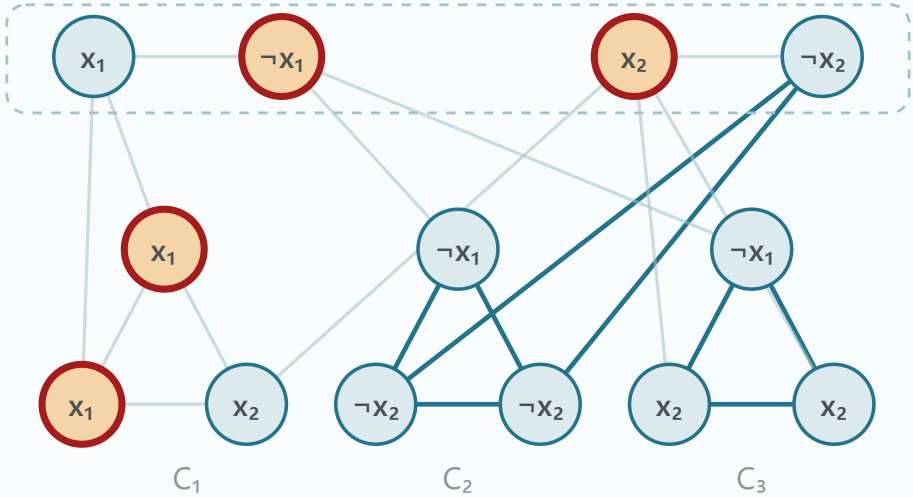

Assignment $x_1$ = false: pick **$\neg x_1$** from the variable edge. Covered edges fade.

# NP-Completeness: Reducing 3-SAT to Vertex Cover



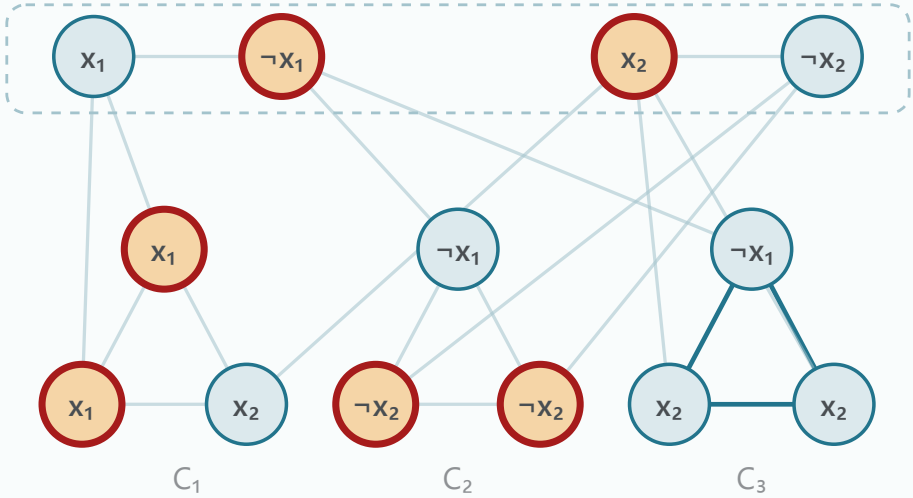

Assignment $x_2$ = true: pick $\mathbf{x_2}$. Each variable edge is covered by exactly one vertex (l = 2 vertices).

# NP-Completeness: Reducing 3-SAT to Vertex Cover



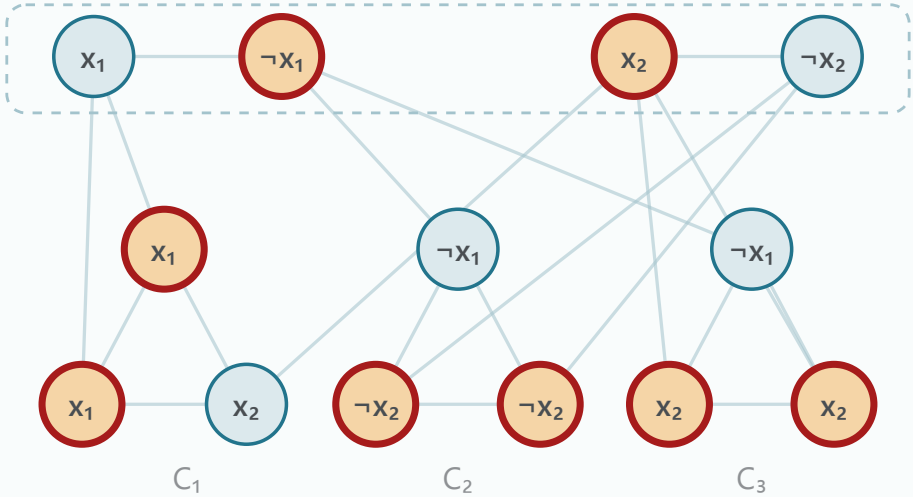

In $C_1$ the true literal $x_2$ (its link edge is already covered) is left out; the other two vertices ($x_1$, $x_1$) are added.

# NP-Completeness: Reducing 3-SAT to Vertex Cover




In $C_2$ the true literal $\neg x_1$ is left out; the vertices $\neg x_2$ and $\neg x_2$ are added.

# NP-Completeness: Reducing 3-SAT to Vertex Cover




In $C_3$ the true literal $\neg x_1$ is left out; $x_2$ and $x_2$ are added. **8 vertices, all 20 edges covered**: $\varphi$ is satisfiable $\Leftrightarrow$ a cover of size $k = l + 2m$ exists.

# Knapsack: Greedy by Ratio

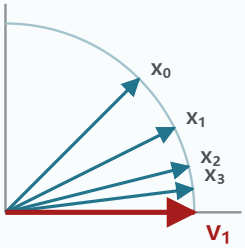
| Item | Value | Weight | Ratio | Capacity 10 |
|---|---|---|---|---|
| 6 | 20 | 7 | 2.86 | ✓ taken (weight 7) |
| 3 | 8 | 4 | 2.00 | ✗ does not fit |
| 7 | 15 | 8 | 1.88 | ✗ does not fit |
| 2 | 5 | 3 | 1.67 | ✓ taken (weight 10) |
| 5 | 10 | 6 | 1.67 | ✗ does not fit |
| 1 | 3 | 2 | 1.50 | ✗ does not fit |
| 4 | 6 | 5 | 1.20 | ✗ does not fit |

Greedy total value 25; the DP optimum is also 25 (items 2 + 6).

- Sort the items by **value/weight** ratio in decreasing order; take each one that fits. O(n log n).
- For the fractional knapsack this rule is **optimal**; for the 0/1 version there is **no guarantee**.
- Bad example: (value 2, weight 1) and (value W, weight W). Greedy takes 2, the optimum is W: ratio 5 for W = 10, W/2 in general; it grows without bound as W grows.
- **Improved rule:** max(greedy value, the most valuable single item that fits) → a **2-approximation** (at least half the optimum).
- Better still: an **FPTAS** that scales the values and runs DP, $(1 - \varepsilon) \cdot OPT$, $O(n^3/\varepsilon)$.

# The Power Iteration Algorithm

- An **iterative** method that approximates a matrix's **largest** (dominant, largest in absolute value) **eigenvalue** $\lambda_1$ and a corresponding **eigenvector** $v_1$.
- Instead of computing all eigenvalues exactly ($O(n^3)$), it gives only an **estimate** of the dominant eigenvalue and eigenvector.
- Each step is just one **matrix–vector product** ($O(n^2)$, or $O(\text{nonzeros})$ for a sparse matrix): effective on **very large** matrices.




$A = [[2, 0], [0, 1]]$, $x_0 = (1, 1)$: $x_k \propto (2^k, 1)$. Each product doubles the $v_1$ component and leaves the other unchanged; the normalized vector turns step by step toward $v_1 = (1, 0)$ (error $\propto (1/2)^k$).

Two ideas meet here: the result is an **approximation** (an estimate) rather than an exact value, and the iterates **converge** to the true value. Google's PageRank is the most famous application of the method.

# Power Iteration: Steps and Why It Works

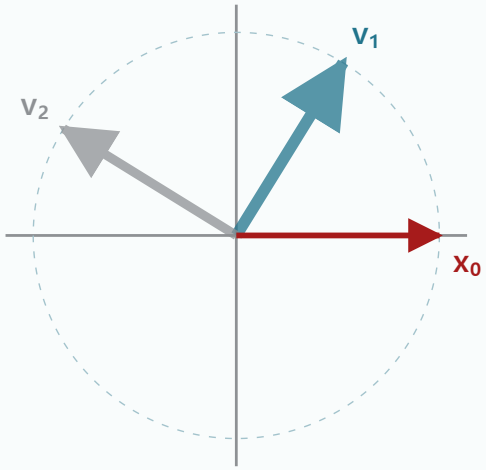
1. Fix a matrix A and choose a **random** (nonzero) start vector $x_0$.
2. Multiply the vector by the matrix to get a new vector: $y = A \cdot x_k$.
3. Normalize the new vector to **norm** (length) 1: $x_{k+1} = y / \|y\|$.
4. Repeat for the desired number of steps or until the change is small.
5. $x_k \to v_1$; the eigenvalue estimate is the **Rayleigh quotient** $\lambda \approx x_k^T A x_k$.

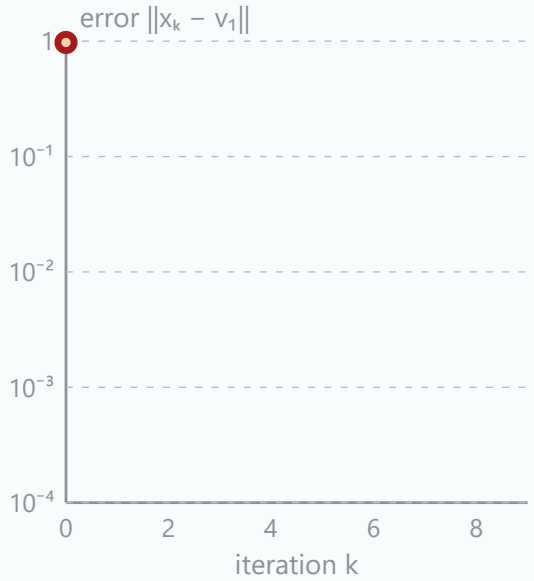
**Why?** If $x_0 = c_1 v_1 + c_2 v_2 + \dots$ then

$$A^k x_0 = \lambda_1^k \left[ c_1 v_1 + c_2 (\lambda_2/\lambda_1)^k v_2 + \dots \right]$$

- Since $|\lambda_2/\lambda_1| < 1$, the other components die out **geometrically**.
- The error shrinks by a factor of $\approx |\lambda_2/\lambda_1|$ per step: if the ratio is close to 1, convergence is slow.
- Condition: $c_1 \neq 0$ ($x_0$ must not be orthogonal to $v_1$) and $|\lambda_1| > |\lambda_2|$.

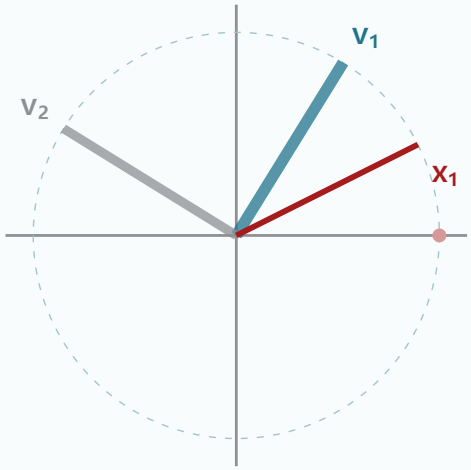
# Power Iteration Step by Step




A = [[2, 1], [1, 3]], eigenvalues $\lambda_1 = (5 + \sqrt{5})/2 \approx 3.618$ and $\lambda_2 \approx 1.382$. Start $x_0 = (1, 0)$; error $\|x_0 - v_1\| = 0.974$.

# Power Iteration Step by Step

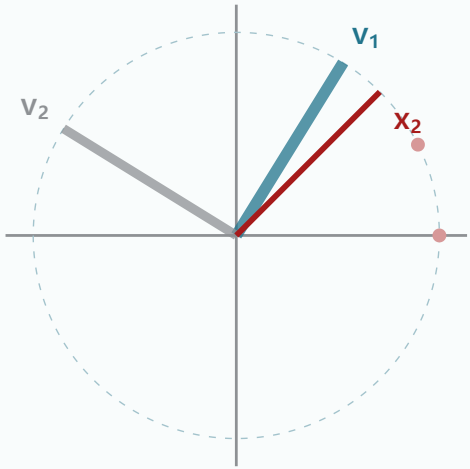


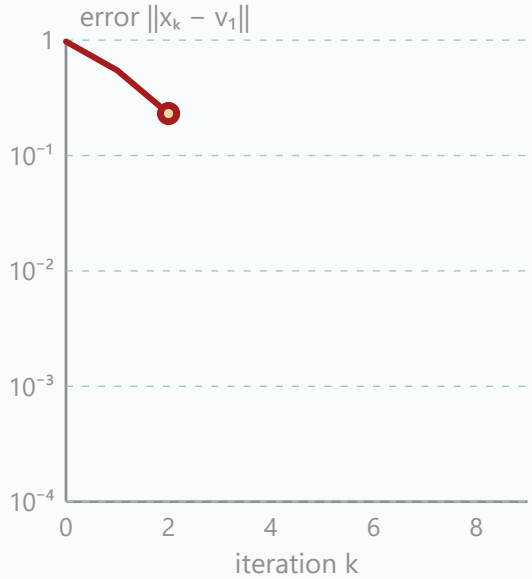




k = 1: A·x = (2.000, 1.000), normalized x = (0.894, 0.447); λ ≈ 3.000, error 0.5465.

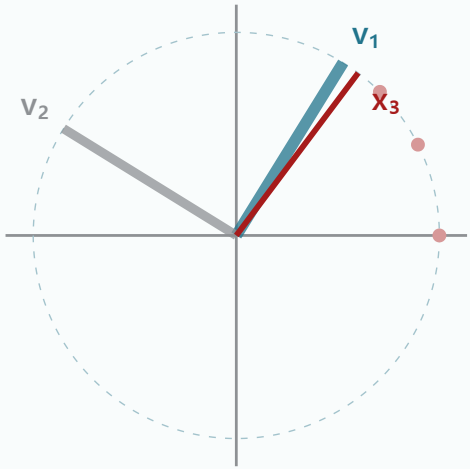
# Power Iteration Step by Step

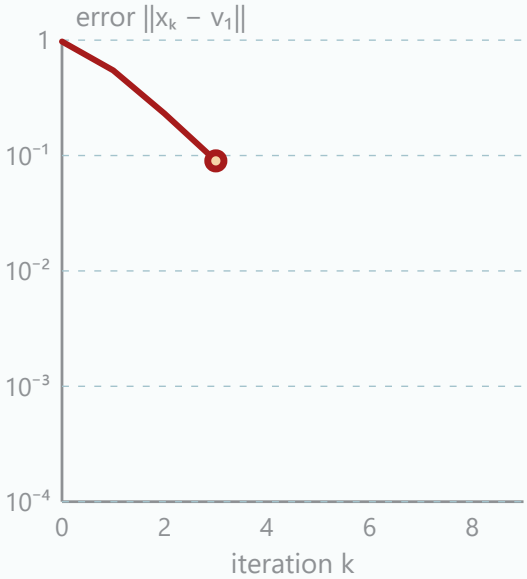



k = 2: A·x = (2.236, 2.236), normalized x = (0.707, 0.707); λ ≈ 3.500, error 0.2313.

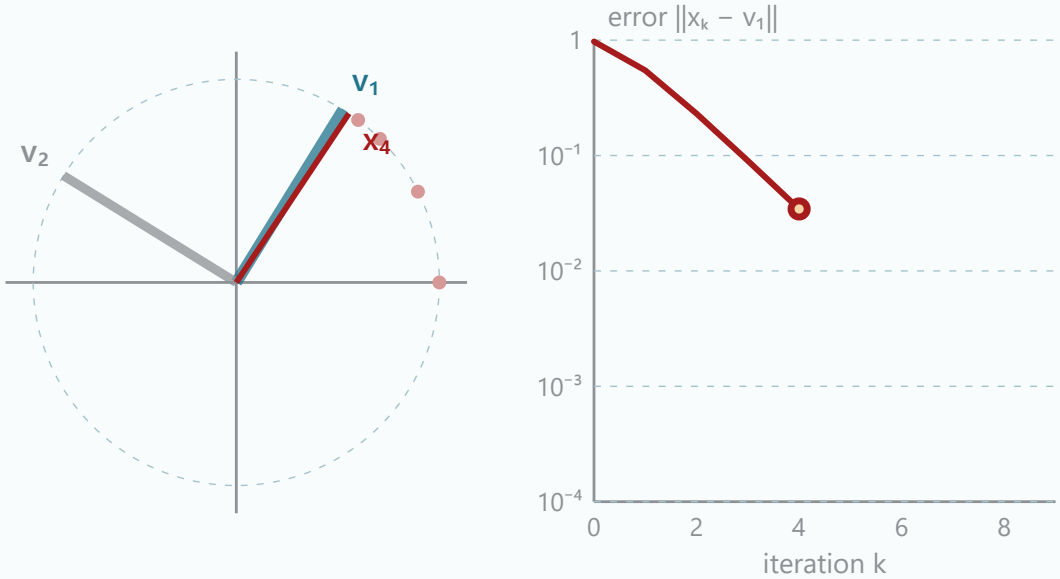
# Power Iteration Step by Step



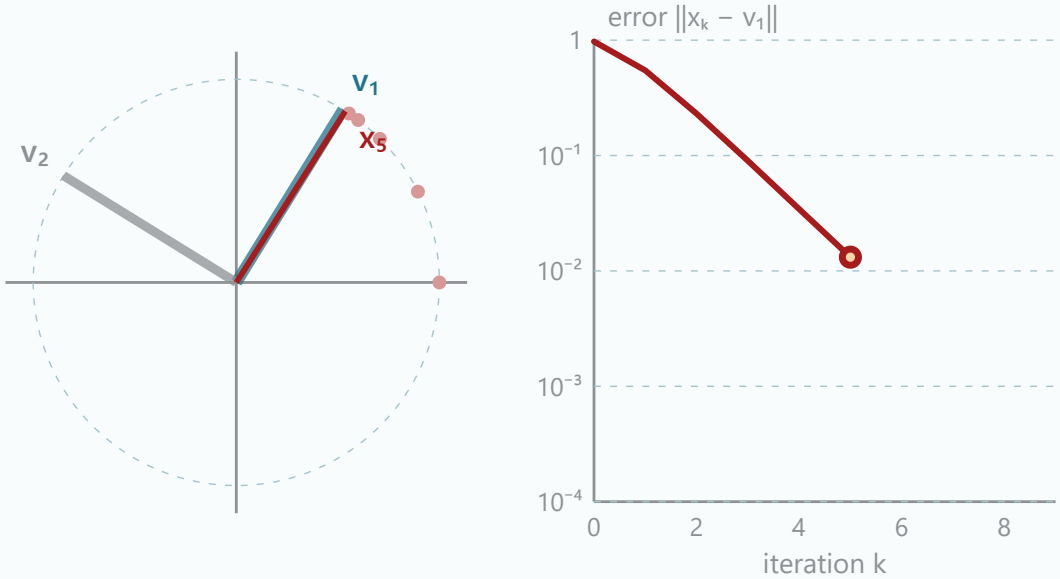

k = 3: A·x = (2.121, 2.828), normalized x = (0.600, 0.800); λ ≈ 3.600, error 0.0899.

# Power Iteration Step by Step



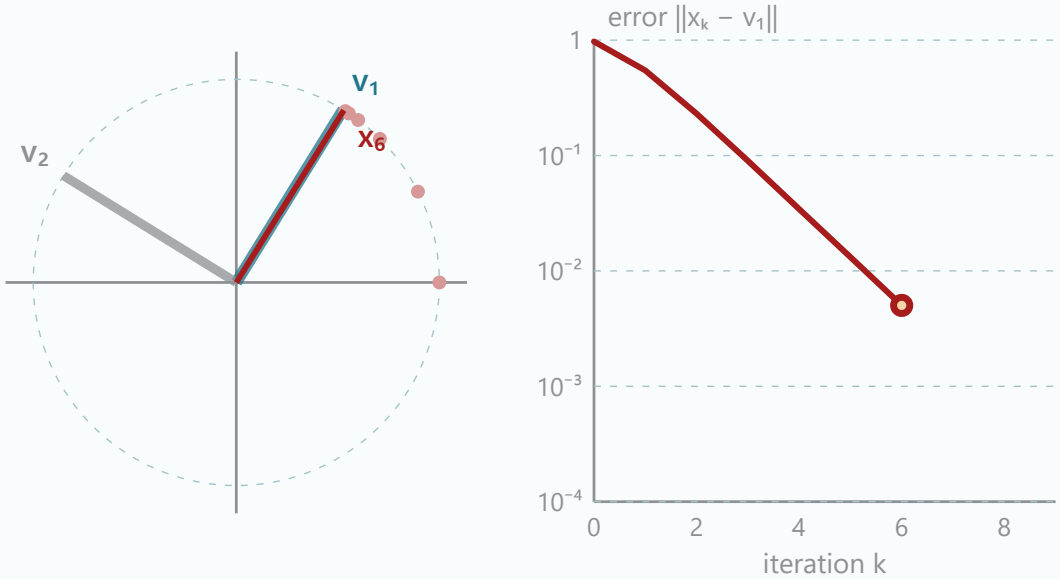

k = 4: x = (0.5547, 0.8321); λ ≈ 3.61538; error $3.4 \cdot 10^{-2}$. Each step shrinks it by ≈ $\lambda_2/\lambda_1$ ≈ 0.38.

# Power Iteration Step by Step



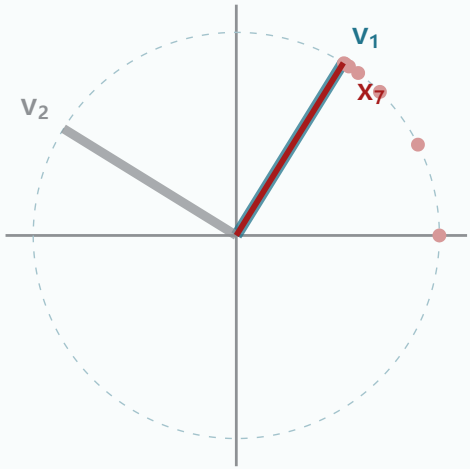

k = 5: x = (0.5369, 0.8437); λ ≈ 3.61765; error $1.3 \cdot 10^{-2}$. Each step shrinks it by ≈ $\lambda_2/\lambda_1$ ≈ 0.38.

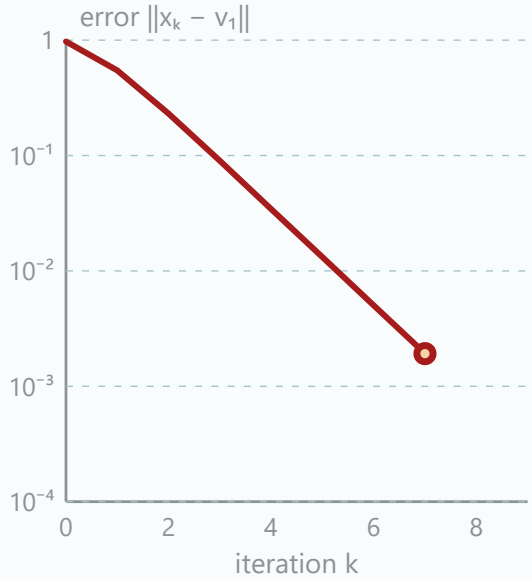
# Power Iteration Step by Step




k = 6: x = (0.5300, 0.8480); λ ≈ 3.61798; error $5.0 \cdot 10^{-3}$. Each step shrinks it by ≈ $\lambda_2/\lambda_1$ ≈ 0.38.

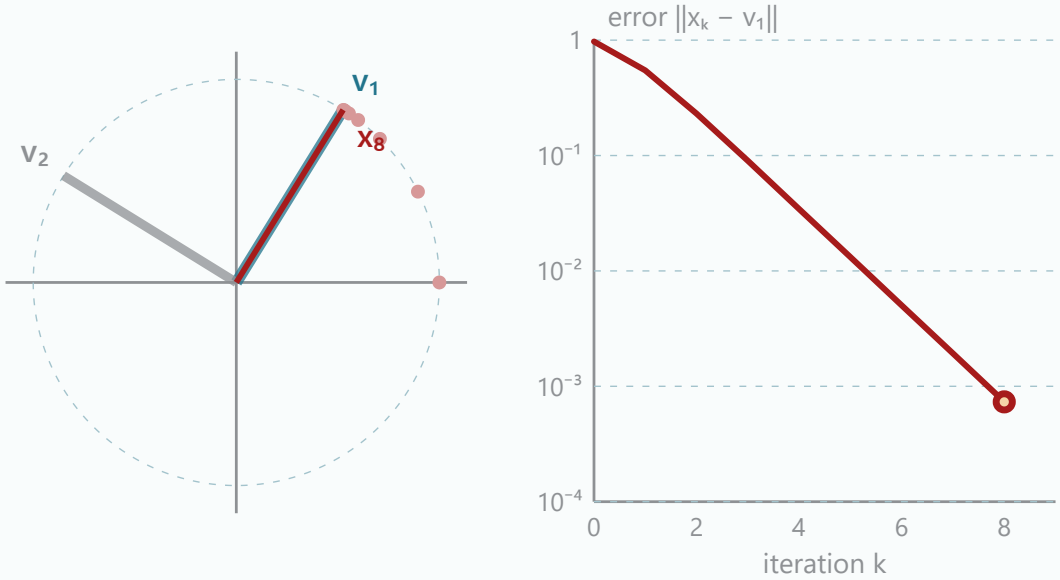
# Power Iteration Step by Step



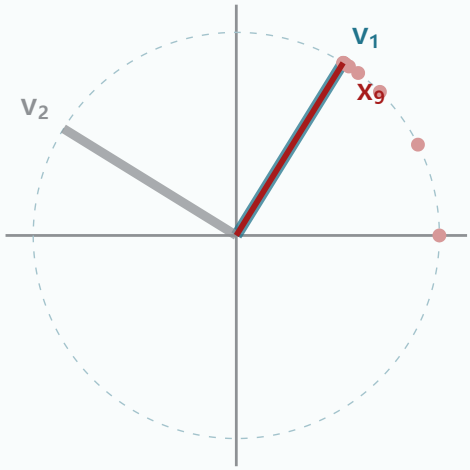

k = 7: x = (0.5274, 0.8496); λ ≈ 3.61803; error $1.9 \cdot 10^{-3}$. Each step shrinks it by ≈ $\lambda_2/\lambda_1$ ≈ 0.38.

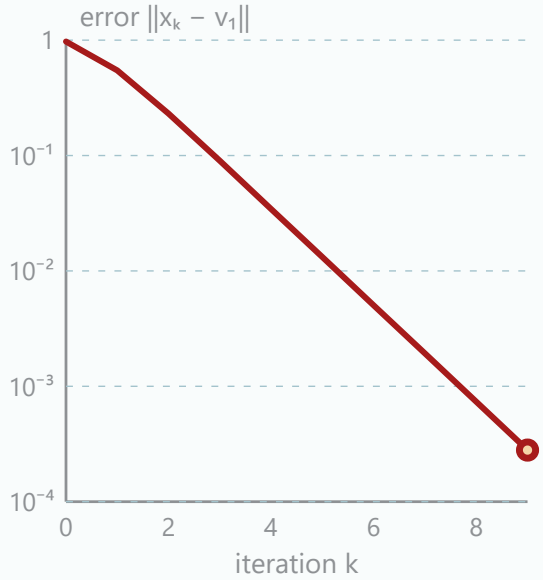
# Power Iteration Step by Step




k = 8: x = (0.5264, 0.8503); λ ≈ 3.61803; error $7.3 \cdot 10^{-4}$. Each step shrinks it by ≈ $\lambda_2/\lambda_1$ ≈ 0.38.

# Power Iteration Step by Step




$k = 9$: $x \approx (0.5260, 0.8505) \approx v_1 = (0.5257, 0.8507)$, $\lambda \approx 3.618034$; error $\approx 2.8 \cdot 10^{-4}$. On a log scale the error plot is a straight line: **linear (geometric) convergence**.

# Power Iteration: Java and a 3 × 3 Example

```java
double[] power(double[][] a, int steps) {
  int n = a.length;
  double[] x = new double[n];
  Arrays.fill(x, 1.0); // x₀ ≠ 0, not ⊥ v₁
  for (int k = 0; k < steps; k++) {
    double[] y = new double[n];
    for (int i = 0; i < n; i++)  // y = A·x
      for (int j = 0; j < n; j++)
        y[i] += a[i][j] * x[j];
    double norm = 0;
    for (double yi : y) norm += yi * yi;
    norm = Math.sqrt(norm);
    if (norm == 0) break; // A·x = 0
    for (int i = 0; i < n; i++)  // ‖x‖ = 1
      x[i] = y[i] / norm;
  }
  return x;
}
```

Example matrix (rows): (0.511 0.32 0.12) · (0.211 0.62 0.12) · (0.311 0.12 0.16); $x_0$ = (1, 1, 1)

| k | $x_1$ | $x_2$ | $x_3$ | $\|A\cdot x\|$ |
|---|---|---|---|---|
| 1 | 0.6474 | 0.6474 | 0.4023 | 1.4690 |
| 2 | 0.6533 | 0.6533 | 0.3827 | 0.8974 |
| 3 | 0.6539 | 0.6539 | 0.3807 | 0.9005 |
| 4 | 0.6539 | 0.6539 | 0.3805 | 0.9008 |
| 5 | 0.6539 | 0.6539 | 0.3805 | 0.9008 |
| 6 | 0.6539 | 0.6539 | 0.3805 | 0.9008 |
| 8 | 0.6539 | 0.6539 | 0.3805 | 0.9008 |

$\|A\cdot x\| \to \lambda_1 = 0.9008$; $v_1 \propto (0.6539, 0.6539, 0.3805)$. $\lambda_2/\lambda_1 \approx 0.33$ → fast convergence.

# Power Iteration: Code Trace

```
1  double[] power(double[][] a, int steps) {
2    int n = a.length;
3    double[] x = new double[n];
▶    Arrays.fill(x, 1.0); // x₀ ≠ 0, not orthogonal to v₁
5    for (int k = 0; k < steps; k++) {
6      double[] y = new double[n];
7      for (int i = 0; i < n; i++)  // y = A·x
8        for (int j = 0; j < n; j++)
9          y[i] += a[i][j] * x[j];
10     double norm = 0;
11     for (double yi : y) norm += yi * yi;
12     norm = Math.sqrt(norm);
13     if (norm == 0) break; // A·x = 0
14     for (int i = 0; i < n; i++)  // ‖x‖ = 1
15       x[i] = y[i] / norm;
16   }
17   return x;
18 }
```

start

A

| | | |
|---|---|---|
| 0.511 | 0.32 | 0.12 |
| 0.211 | 0.62 | 0.12 |
| 0.311 | 0.12 | 0.16 |

·

x

| |
|---|
| 1.0000 |
| 1.0000 |
| 1.0000 |

=

y

| |
|---|
| |
| |
| |

Start vector $x_0$ = (1, 1, 1). Any nonzero vector not orthogonal to $v_1$ will do.

Step 1 / 11

**VARIABLES**

`k = –`

`x = [1.0000, 1.0000, 1.0000]`

# Power Iteration: Code Trace

```
1  double[] power(double[][] a, int steps) {
2    int n = a.length;
3    double[] x = new double[n];
4    Arrays.fill(x, 1.0); // x₀ ≠ 0, not
     orthogonal to v₁
5    for (int k = 0; k < steps; k++) {
6      double[] y = new double[n];
▶      for (int i = 0; i < n; i++)  // y = A·x
▶        for (int j = 0; j < n; j++)
▶          y[i] += a[i][j] * x[j];
10     double norm = 0;
11     for (double yi : y) norm += yi * yi;
12     norm = Math.sqrt(norm);
13     if (norm == 0) break; // A·x = 0
14     for (int i = 0; i < n; i++)  // ‖x‖ = 1
15       x[i] = y[i] / norm;
16   }
17   return x;
18 }
```

k = 0

| A | | | · | x | = | y |
|---|---|---|---|---|---|---|
| 0.511 | 0.32 | 0.12 | | 1.0000 | | 0.9510 |
| 0.211 | 0.62 | 0.12 | | 1.0000 | | |
| 0.311 | 0.12 | 0.16 | | 1.0000 | | |

y[0] = row 0 of A · x = 0.511·1 + 0.32·1 + 0.12·1 = **0.9510**.

Step 2 / 11

VARIABLES

`k = 0` `i = 0` `y = [0.9510]`

# Power Iteration: Code Trace

```
1  double[] power(double[][] a, int steps) {
2    int n = a.length;
3    double[] x = new double[n];
4    Arrays.fill(x, 1.0); // x₀ ≠ 0, not
   orthogonal to v₁
5    for (int k = 0; k < steps; k++) {
6      double[] y = new double[n];
▶      for (int i = 0; i < n; i++)  // y = A·x
▶        for (int j = 0; j < n; j++)
▶          y[i] += a[i][j] * x[j];
10     double norm = 0;
11     for (double yi : y) norm += yi * yi;
12     norm = Math.sqrt(norm);
13     if (norm == 0) break; // A·x = 0
14     for (int i = 0; i < n; i++)  // ‖x‖ = 1
15       x[i] = y[i] / norm;
16   }
17   return x;
18 }
```

k = 0

| A | | | | x | | y |
|---|---|---|---|---|---|---|
| 0.511 | 0.32 | 0.12 | | 1.0000 | | 0.9510 |
| 0.211 | 0.62 | 0.12 | · | 1.0000 | = | 0.9510 |
| 0.311 | 0.12 | 0.16 | | 1.0000 | | |

y[1] = row 1 of A · x = 0.211·1 + 0.62·1 + 0.12·1 = **0.9510**.

Step 3 / 11

VARIABLES

`k = 0` `i = 1` `y = [0.9510, 0.9510]`

# Power Iteration: Code Trace

```
1  double[] power(double[][] a, int steps) {
2    int n = a.length;
3    double[] x = new double[n];
4    Arrays.fill(x, 1.0); // x₀ ≠ 0, not
   orthogonal to v₁
5    for (int k = 0; k < steps; k++) {
6      double[] y = new double[n];
▶      for (int i = 0; i < n; i++)  // y = A·x
▶        for (int j = 0; j < n; j++)
▶          y[i] += a[i][j] * x[j];
10     double norm = 0;
11     for (double yi : y) norm += yi * yi;
12     norm = Math.sqrt(norm);
13     if (norm == 0) break; // A·x = 0
14     for (int i = 0; i < n; i++)  // ‖x‖ = 1
15       x[i] = y[i] / norm;
16   }
17   return x;
18 }
```

k = 0

| | A | | | x | | y |
|---|---|---|---|---|---|---|
| 0.511 | 0.32 | 0.12 | | 1.0000 | | 0.9510 |
| 0.211 | 0.62 | 0.12 | · | 1.0000 | = | 0.9510 |
| 0.311 | 0.12 | 0.16 | | 1.0000 | | 0.5910 |

y[2] = row 2 of A · x = 0.311·1 + 0.12·1 + 0.16·1 = **0.5910**.

Step 4 / 11

VARIABLES

`k = 0` `i = 2`

`y = [0.9510, 0.9510, 0.5910]`

# Power Iteration: Code Trace

```
1  double[] power(double[][] a, int steps) {
2    int n = a.length;
3    double[] x = new double[n];
4    Arrays.fill(x, 1.0); // x₀ ≠ 0, not
   orthogonal to v₁
5    for (int k = 0; k < steps; k++) {
6      double[] y = new double[n];
7      for (int i = 0; i < n; i++)  // y = A·x
8        for (int j = 0; j < n; j++)
9          y[i] += a[i][j] * x[j];
▶      double norm = 0;
▶      for (double yi : y) norm += yi * yi;
▶      norm = Math.sqrt(norm);
13     if (norm == 0) break; // A·x = 0
14     for (int i = 0; i < n; i++)  // ‖x‖ = 1
15       x[i] = y[i] / norm;
16   }
17   return x;
18 }
```

k = 0

| A | | | | x | | y |
|---|---|---|---|---|---|---|
| 0.511 | 0.32 | 0.12 | | 1.0000 | | 0.9510 |
| 0.211 | 0.62 | 0.12 | · | 1.0000 | = | 0.9510 |
| 0.311 | 0.12 | 0.16 | | 1.0000 | | 0.5910 |

**‖y‖ = 1.4690**

‖y‖ = √(0.9510² + 0.9510² + 0.5910²) = **1.4690**.

Step 5 / 11

VARIABLES

`k = 0` `norm = 1.4690`

`x = [1.0000, 1.0000, 1.0000]`

# Power Iteration: Code Trace

```
1  double[] power(double[][] a, int steps) {
2    int n = a.length;
3    double[] x = new double[n];
4    Arrays.fill(x, 1.0); // x₀ ≠ 0, not
   orthogonal to v₁
5    for (int k = 0; k < steps; k++) {
6      double[] y = new double[n];
7      for (int i = 0; i < n; i++)  // y = A·x
8        for (int j = 0; j < n; j++)
9          y[i] += a[i][j] * x[j];
10     double norm = 0;
11     for (double yi : y) norm += yi * yi;
12     norm = Math.sqrt(norm);
▶      if (norm == 0) break; // A·x = 0
▶      for (int i = 0; i < n; i++)  // ‖x‖ = 1
▶        x[i] = y[i] / norm;
16   }
17   return x;
18 }
```

k = 0

| A | | | | x | | y |
|---|---|---|---|---|---|---|
| 0.511 | 0.32 | 0.12 | · | 0.6474 | = | 0.9510 |
| 0.211 | 0.62 | 0.12 | | 0.6474 | | 0.9510 |
| 0.311 | 0.12 | 0.16 | | 0.4023 | | 0.5910 |

$\|y\| = 1.4690$ $x \leftarrow y / \|y\|$

x = y / ||y|| = (0.6474, 0.6474, 0.4023): a new direction of length 1; the numbers neither blow up nor vanish.

Step 6 / 11

VARIABLES

k = 0 norm = 1.4690

x = [0.6474, 0.6474, 0.4023]

# Power Iteration: Code Trace

```
1  double[] power(double[][] a, int steps) {
2    int n = a.length;
3    double[] x = new double[n];
4    Arrays.fill(x, 1.0); // x₀ ≠ 0, not
   orthogonal to v₁
5    for (int k = 0; k < steps; k++) {
▶      double[] y = new double[n];
▶      for (int i = 0; i < n; i++)  // y = A·x
▶        for (int j = 0; j < n; j++)
▶          y[i] += a[i][j] * x[j];
10     double norm = 0;
11     for (double yi : y) norm += yi * yi;
12     norm = Math.sqrt(norm);
13     if (norm == 0) break; // A·x = 0
14     for (int i = 0; i < n; i++)  // ‖x‖ = 1
15       x[i] = y[i] / norm;
16   }
17   return x;
18 }
```

k = 1

| A | | | · | x | = | y |
|---|---|---|---|---|---|---|
| 0.511 | 0.32 | 0.12 | | 0.6474 | | 0.5862 |
| 0.211 | 0.62 | 0.12 | | 0.6474 | | 0.5862 |
| 0.311 | 0.12 | 0.16 | | 0.4023 | | 0.3434 |

k = 1: y = A·x = (0.5862, 0.5862, 0.3434). The ratios of the components barely changed.

Step 7 / 11

**VARIABLES**

k = 1

x = [0.6474, 0.6474, 0.4023]

y = [0.5862, 0.5862, 0.3434]

# Power Iteration: Code Trace

```
1  double[] power(double[][] a, int steps) {
2    int n = a.length;
3    double[] x = new double[n];
4    Arrays.fill(x, 1.0); // x₀ ≠ 0, not
   orthogonal to v₁
5    for (int k = 0; k < steps; k++) {
6      double[] y = new double[n];
7      for (int i = 0; i < n; i++)  // y = A·x
8        for (int j = 0; j < n; j++)
9          y[i] += a[i][j] * x[j];
▶      double norm = 0;
▶      for (double yi : y) norm += yi * yi;
▶      norm = Math.sqrt(norm);
▶      if (norm == 0) break; // A·x = 0
▶      for (int i = 0; i < n; i++)  // ‖x‖ = 1
▶        x[i] = y[i] / norm;
16   }
17   return x;
18 }
```

k = 1

| A | | | | x | | y |
|---|---|---|---|---|---|---|
| 0.511 | 0.32 | 0.12 | · | 0.6533 | = | 0.5862 |
| 0.211 | 0.62 | 0.12 | | 0.6533 | | 0.5862 |
| 0.311 | 0.12 | 0.16 | | 0.3827 | | 0.3434 |

$\|y\| = 0.8974$ $x \leftarrow y / \|y\|$

$\|y\| = 0.8974 \rightarrow x = (0.6533, 0.6533, 0.3827)$. The direction changed very little.

Step 8 / 11

VARIABLES

`k = 1` `norm = 0.8974`

`x = [0.6533, 0.6533, 0.3827]`

# Power Iteration: Code Trace

```
1  double[] power(double[][] a, int steps) {
2    int n = a.length;
3    double[] x = new double[n];
4    Arrays.fill(x, 1.0); // x₀ ≠ 0, not
   orthogonal to v₁
5    for (int k = 0; k < steps; k++) {
▶      double[] y = new double[n];
▶      for (int i = 0; i < n; i++)  // y = A·x
▶        for (int j = 0; j < n; j++)
▶          y[i] += a[i][j] * x[j];
10     double norm = 0;
11     for (double yi : y) norm += yi * yi;
12     norm = Math.sqrt(norm);
13     if (norm == 0) break; // A·x = 0
14     for (int i = 0; i < n; i++)  // ‖x‖ = 1
15       x[i] = y[i] / norm;
16   }
17   return x;
18 }
```

k = 2

| A | | | | x | | y |
|---|---|---|---|---|---|---|
| 0.511 | 0.32 | 0.12 | | 0.6533 | | 0.5888 |
| 0.211 | 0.62 | 0.12 | · | 0.6533 | = | 0.5888 |
| 0.311 | 0.12 | 0.16 | | 0.3827 | | 0.3428 |

k = 2: y = A·x = (0.5888, 0.5888, 0.3428). The ratios of the components barely changed.

Step 9 / 11

**VARIABLES**

- k = 2
- x = [0.6533, 0.6533, 0.3827]
- y = [0.5888, 0.5888, 0.3428]

# Power Iteration: Code Trace

```
1  double[] power(double[][] a, int steps) {
2    int n = a.length;
3    double[] x = new double[n];
4    Arrays.fill(x, 1.0); // x₀ ≠ 0, not
   orthogonal to v₁
5    for (int k = 0; k < steps; k++) {
6      double[] y = new double[n];
7      for (int i = 0; i < n; i++)  // y = A·x
8        for (int j = 0; j < n; j++)
9          y[i] += a[i][j] * x[j];
▶      double norm = 0;
▶      for (double yi : y) norm += yi * yi;
▶      norm = Math.sqrt(norm);
▶      if (norm == 0) break; // A·x = 0
▶      for (int i = 0; i < n; i++)  // ‖x‖ = 1
▶        x[i] = y[i] / norm;
16   }
17   return x;
18 }
```

k = 2

| A | | | | x | | y |
|---|---|---|---|---|---|---|
| 0.511 | 0.32 | 0.12 | · | 0.6539 | = | 0.5888 |
| 0.211 | 0.62 | 0.12 | | 0.6539 | | 0.5888 |
| 0.311 | 0.12 | 0.16 | | 0.3807 | | 0.3428 |

$\|y\| = 0.9005$ $x \leftarrow y / \|y\|$

$\|y\| = 0.9005 \rightarrow x = (0.6539, 0.6539, 0.3807)$.
$\|y\|$ is approaching $\lambda_1 = 0.9008$.

Step 10 / 11

VARIABLES

`k = 2` `norm = 0.9005`

`x = [0.6539, 0.6539, 0.3807]`

# Power Iteration: Code Trace

```
1  double[] power(double[][] a, int steps) {
2    int n = a.length;
3    double[] x = new double[n];
4    Arrays.fill(x, 1.0); // x₀ ≠ 0, not
   orthogonal to v₁
5    for (int k = 0; k < steps; k++) {
6      double[] y = new double[n];
7      for (int i = 0; i < n; i++)  // y = A·x
8        for (int j = 0; j < n; j++)
9          y[i] += a[i][j] * x[j];
10     double norm = 0;
11     for (double yi : y) norm += yi * yi;
12     norm = Math.sqrt(norm);
13     if (norm == 0) break; // A·x = 0
14     for (int i = 0; i < n; i++)  // ‖x‖ = 1
15       x[i] = y[i] / norm;
16   }
▶  return x;
18 }
```

result A x y

| | | | | | |
|---|---|---|---|---|---|
| 0.511 | 0.32 | 0.12 | | 0.6539 | |
| 0.211 | 0.62 | 0.12 | · | 0.6539 | = |
| 0.311 | 0.12 | 0.16 | | 0.3807 | |

`steps = 3` iterations done: x = (0.6539, 0.6539, 0.3807) ≈ $v_1$. In the next iteration the last component becomes 0.3805; after that the four digits no longer change.

Step 11 / 11

**VARIABLES**

`x = [0.6539, 0.6539, 0.3807]`

# Power Iteration: Limits and Applications

### Strengths

+ Very simple; only matrix–vector products.
+ Works on sparse, **huge** matrices (billions of rows).
+ Memory O(n): the matrix is not modified.

### Limits

– Finds only the **dominant** eigenvalue.
– Very slow if $|\lambda_2| \approx |\lambda_1|$; may not converge if $|\lambda_1| = |\lambda_2|$.
– Fails in theory if $x_0$ is orthogonal to $v_1$ ($c_1 = 0$).

- **PageRank:** the dominant eigenvector of the web link matrix gives the importance of pages.
- **Variants:** inverse iteration (smallest eigenvalue), shifted iteration, Lanczos/Arnoldi; the first principal component in PCA.

# Summary

| Problem | Algorithm | Ratio | Time |
|---|---|---|---|
| TSP (general) | nearest neighbor (heuristic) | no guarantee | $O(n^2)$ |
| Metric TSP | MST + preorder + shortcuts / Christofides | 2 / 1.5 | $O(n^2)$ / $O(n^3)$ |
| TSP | 2-opt local search | no guarantee | $O(n^2)$ / pass |
| Machine scheduling | list scheduling / LPT | $2 - 1/m$ / $4/3 - 1/(3m)$ | $O(n \log m)$ / $O(n \log n)$ |
| Vertex cover | pick an edge, add both endpoints | 2 | $O(V + E)$ |
| 0/1 knapsack | max(ratio-sorted, best single item) / FPTAS | 2 / $1 + \varepsilon$ | $O(n \log n)$ / $O(n^3/\varepsilon)$ |
| Dominant eigenvalue | power iteration | error $\propto \lvert\lambda_2/\lambda_1\rvert^k$ | $O(n^2)$ / step |

SUMMARY

# Common Mistakes

### Taking a heuristic for an approximation algorithm

Nearest neighbor and 2-opt give good results but have **no ratio guarantee**; an approximation algorithm needs a proven ρ.

### Writing the ratio backwards

For minimization $C \le \rho \cdot C^*$, for maximization $C \ge C^*/\rho$. ρ is always ≥ 1.

### Adding one endpoint in vertex cover

Adding only one endpoint of an edge can be n times worse on a star graph; the 2-approximation adds **both endpoints**.

### Forgetting the triangle inequality

The shortcut step of the MST-based 2-approximation keeps the path from growing only for **metric** TSP.

### Treating metaheuristics as exact

Simulated annealing and genetic algorithms **do not guarantee** convergence to the optimum; run them several times with different seeds and parameters and keep the best result.

### Skipping normalization

If x is not normalized at every step of power iteration, the numbers overflow ($|\lambda_1| > 1$) or shrink to zero.

SUMMARY

# Sample Problems

1. A minimization algorithm finds 30; the optimum is 20. What is the ratio on this input?
2. Nearest neighbor from city **2** on the 5-city TSP example: tour and length?
3. Path 1–2–3–4–5–6, edges taken in order (1,2), …, (5,6): how many vertices does the 2-approximation pick? The optimum?
4. $m = 2$, jobs 3, 3, 2, 2, 2: what are $C_{max}$ for list scheduling and LPT, and the optimum?
5. Simulated annealing: probability of accepting a move with $\Delta = 5$ at $T = 10$ and at $T = 1$?
6. $W = 10$, items (value, weight) (2, 1) and (10, 10): what do ratio greedy and the improved rule find?

▶ **Show answers**

# Exercises

1. a(0,0), b(4,3), c(4,0), d(0,3), and the tour has edges (a,b), (c,d). What is the 2-opt gain $\Delta$?
2. Build a job list for which list scheduling reaches the ratio 5/3 with $m = 3$.
3. How many vertices does the 2-approximation pick on the complete graph $K_5$? What is the optimum?
4. $A = [[2, 0], [0, 1]]$, $x_0 = (1, 1)$. After how many steps is the second component of $x_k$ below $10^{-3}$ times the first?
5. In simulated annealing with $T_0 = 100$ and $T \leftarrow 0.997 \cdot T$ at each step, after how many steps does T drop below 1?
6. Write the 2-approximation in Java so that it also returns the chosen matching.

▶ **Answers / hints**

# References


- Cormen, Leiserson, Rivest and Stein, *Introduction to Algorithms*, 4th ed., 2022 (Chapter 35: Approximation Algorithms).
- V. V. Vazirani, *Approximation Algorithms*, Springer, 2001.
- D. P. Williamson and D. B. Shmoys, *The Design of Approximation Algorithms*, Cambridge University Press, 2011.
- R. L. Graham, "Bounds on Multiprocessing Timing Anomalies", *SIAM Journal on Applied Mathematics* 17(2), 1969.
- N. Christofides, "Worst-Case Analysis of a New Heuristic for the Travelling Salesman Problem", Report 388, Carnegie Mellon University, 1976.
- M. Sipser, *Introduction to the Theory of Computation*, 3rd ed., 2012 (the 3-SAT → vertex cover reduction).
- G. H. Golub and C. F. Van Loan, *Matrix Computations*, 4th ed., 2013 (power iteration).

# The End

## Chapter 6: Approximation Algorithms

Previous: Chapter 5: Branch and Bound · Next: Chapter 7: Randomized Algorithms