Approximation Algorithms (ADM III)

2- Local Search & Greedy Algorithms

Guillaume Sagnol

Technische
Universität
Berlin

Outline

Scheduling

- Scheduling studies the optimal allocation of resources to a set of tasks, or activities

Scheduling

- Scheduling studies the optimal allocation of resources to a set of tasks, or activities
- Those problems have many applications, and are often studied in the context of Approximation Algorithms

Scheduling

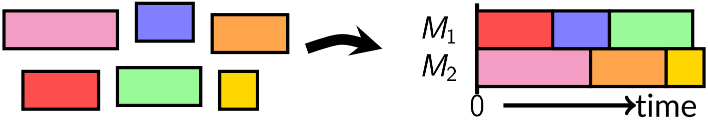
- Scheduling studies the optimal allocation of resources to a set of tasks, or activities
- Those problems have many applications, and are often studied in the context of Approximation Algorithms
- There is a variety of such problems, and we will review many of them in this course

Scheduling

- Scheduling studies the optimal allocation of resources to a set of tasks, or activities
- Those problems have many applications, and are often studied in the context of Approximation Algorithms
- There is a variety of such problems, and we will review many of them in this course
- Standard notation:
    - n tasks, called *jobs*
    - m resources, called *machines*
    - Job j has processing time $p_j \geq 0$. Sometimes, proc. time of job j depends on the machine on which it is executed, in this case $p_{ij} \geq 0$ represents the proc. time of job j on machine i.

Scheduling

- Scheduling studies the optimal allocation of resources to a set of tasks, or activities
- Those problems have many applications, and are often studied in the context of Approximation Algorithms
- There is a variety of such problems, and we will review many of them in this course
- Standard notation:
  - n tasks, called *jobs*
  - m resources, called *machines*
  - Job j has processing time $p_j \geq 0$. Sometimes, proc. time of job j depends on the machine on which it is executed, in this case $p_{ij} \geq 0$ represents the proc. time of job j on machine i.
- Schedules can be represented by Gantt charts



Classification of Scheduling Problems

- A 3-fields notation was introduced by Graham to classify scheduling problems
- Each problem is represented by a triple $\alpha|\beta|\gamma$, where
    - α describes the machine environment
    - β describes special job characteristics
    - γ describes the objective function to minimize

Classification of Scheduling Problems

- A 3-fields notation was introduced by Graham to classify scheduling problems
- Each problem is represented by a triple $\alpha|\beta|\gamma$, where
  - α describes the machine environment
  - β describes special job characteristics
  - γ describes the objective function to minimize
- Machine environment α:
  - $\alpha = 1$: A single machine

Classification of Scheduling Problems

- A 3-fields notation was introduced by Graham to classify scheduling problems
- Each problem is represented by a triple $\alpha|\beta|\gamma$, where
  - α describes the machine environment
  - β describes special job characteristics
  - γ describes the objective function to minimize
- Machine environment α:
  - $\alpha = 1$: A single machine
  - $\alpha = P$: Parallel identical machines. We can also write $\alpha = Pm$ to indicate that the number of machines is fixed, so m is considered as a constant for the analysis of running times.

Classification of Scheduling Problems

- A 3-fields notation was introduced by Graham to classify scheduling problems
- Each problem is represented by a triple $\alpha|\beta|\gamma$, where
  - α describes the machine environment
  - β describes special job characteristics
  - γ describes the objective function to minimize
- Machine environment α:
  - $\alpha = 1$: A single machine
  - $\alpha = P$: Parallel identical machines. We can also write $\alpha = Pm$ to indicate that the number of machines is fixed, so m is considered as a constant for the analysis of running times.
  - $\alpha = R$ or Rm: unrelated parallel machines. The processing time of job j on machine i is p_{ij}.

Classification of Scheduling Problems

- A 3-fields notation was introduced by Graham to classify scheduling problems
- Each problem is represented by a triple $\alpha|\beta|\gamma$, where
  - α describes the machine environment
  - β describes special job characteristics
  - γ describes the objective function to minimize
- Machine environment α:
  - $\alpha = 1$: A single machine
  - $\alpha = P$: Parallel identical machines. We can also write $\alpha = Pm$ to indicate that the number of machines is fixed, so m is considered as a constant for the analysis of running times.
  - $\alpha = R$ or Rm: unrelated parallel machines. The processing time of job j on machine i is p_{ij}.
  - $\alpha = O$: open shop. Each job must be executed on all machines, in any order (job j requires machine i for p_{ij} time units).

Classification of Scheduling Problems

- A 3-fields notation was introduced by Graham to classify scheduling problems
- Each problem is represented by a triple $\alpha|\beta|\gamma$, where
  - α describes the machine environment
  - β describes special job characteristics
  - γ describes the objective function to minimize
- Machine environment α:
  - $\alpha = 1$: A single machine
  - $\alpha = P$: Parallel identical machines. We can also write $\alpha = Pm$ to indicate that the number of machines is fixed, so m is considered as a constant for the analysis of running times.
  - $\alpha = R$ or Rm: unrelated parallel machines. The processing time of job j on machine i is p_{ij}.
  - $\alpha = O$: open shop. Each job must be executed on all machines, in any order (job j requires machine i for p_{ij} time units).
  - Other environments: Q: uniform parallel machines, F: flow shop, J: Job shop,...

Classification of Scheduling Problems

- A 3-fields notation was introduced by Graham to classify scheduling problems
- Each problem is represented by a triple $\alpha|\beta|\gamma$, where
    - α describes the machine environment
    - β describes special job characteristics
    - γ describes the objective function to minimize
- Job characteristics, represented by a string β:
    - $\beta = \{\}$: Default characteristics.

Classification of Scheduling Problems

- A 3-fields notation was introduced by Graham to classify scheduling problems
- Each problem is represented by a triple $\alpha|\beta|\gamma$, where
  - α describes the machine environment
  - β describes special job characteristics
  - γ describes the objective function to minimize

- Job characteristics, represented by a string β:
  - $\beta = \{\}$: Default characteristics.
  - $r_j \in \beta$: Each job has a *release-date* r_j, i.e., job j cannot start before time $t = r_j$.

Classification of Scheduling Problems

- A 3-fields notation was introduced by Graham to classify scheduling problems
- Each problem is represented by a triple $\alpha|\beta|\gamma$, where
  - α describes the machine environment
  - β describes special job characteristics
  - γ describes the objective function to minimize
- Job characteristics, represented by a string β:
  - $\beta = \{\}$: Default characteristics.
  - $r_j \in \beta$: Each job has a *release-date* r_j, i.e., job j cannot start before time $t = r_j$.
  - $\texttt{pmtn} \in \beta$: Preemption is allowed, i.e., job execution can be interrupted and resumed, possibly on another machine.

Classification of Scheduling Problems

- A 3-fields notation was introduced by Graham to classify scheduling problems
- Each problem is represented by a triple $\alpha|\beta|\gamma$, where
  - α describes the machine environment
  - β describes special job characteristics
  - γ describes the objective function to minimize

- Job characteristics, represented by a string β:
  - $\beta = \{\}$: Default characteristics.
  - $r_j \in \beta$: Each job has a *release-date* r_j, i.e., job j cannot start before time $t = r_j$.
  - $\texttt{pmtn} \in \beta$: Preemption is allowed, i.e., job execution can be interrupted and resumed, possibly on another machine.
  - $\texttt{prec} \in \beta$ A list of *precedences* is part of the input, e.g., $i \prec j$ means that job j cannot start before completion of job i.

Classification of Scheduling Problems

- A 3-fields notation was introduced by Graham to classify scheduling problems
- Each problem is represented by a triple $\alpha|\beta|\gamma$, where
  - α describes the machine environment
  - β describes special job characteristics
  - γ describes the objective function to minimize

- Job characteristics, represented by a string β:
  - $\beta = \{\}$: Default characteristics.
  - $r_j \in \beta$: Each job has a *release-date* r_j, i.e., job j cannot start before time $t = r_j$.
  - $\texttt{pmtn} \in \beta$: Preemption is allowed, i.e., job execution can be interrupted and resumed, possibly on another machine.
  - $\texttt{prec} \in \beta$ A list of *precedences* is part of the input, e.g., $i \prec j$ means that job j cannot start before completion of job i.
  - Other obvious specifications (e.g., $p_j = 1$ means that all jobs have unit processing times)

Classification of Scheduling Problems

- A 3-fields notation was introduced by Graham to classify scheduling problems
- Each problem is represented by a triple $\alpha|\beta|\gamma$, where
    - α describes the machine environment
    - β describes special job characteristics
    - γ describes the objective function to minimize
- Objective criterion to minimize γ: A function of the following parameters of the jobs:
    - C_j: completion time of job j; $C_{\max} := \max_j C_j$ denotes the latest completion time and is called the *makespan*.

Classification of Scheduling Problems

- A 3-fields notation was introduced by Graham to classify scheduling problems
- Each problem is represented by a triple $\alpha|\beta|\gamma$, where
  - α describes the machine environment
  - β describes special job characteristics
  - γ describes the objective function to minimize
- Objective criterion to minimize γ: A function of the following parameters of the jobs:
  - C_j: completion time of job j; $C_{\max} := \max_j C_j$ denotes the latest completion time and is called the *makespan*.
  - $L_j = C_j - d_j$: lateness of job j for a given deadline d_j; $L_{\max} := \max_j L_j$ denotes the maximal lateness.

Classification of Scheduling Problems

- A 3-fields notation was introduced by Graham to classify scheduling problems
- Each problem is represented by a triple $\alpha|\beta|\gamma$, where
    - α describes the machine environment
    - β describes special job characteristics
    - γ describes the objective function to minimize

- Objective criterion to minimize γ: A function of the following parameters of the jobs:
    - C_j: completion time of job j; $C_{\max} := \max_j C_j$ denotes the latest completion time and is called the *makespan*.
    - $L_j = C_j - d_j$: lateness of job j for a given deadline d_j; $L_{\max} := \max_j L_j$ denotes the maximal lateness.
    - Many other criterions (flow time $F_j = C_j - r_j$, tardiness $T_j = \max(0, C_j - d_j)$ of job j, unit penalty for tardy jobs $U_j = 1_{C_j > d_j}$, ...)

Classification of Scheduling Problems

Example 2.1

- $1|r_j|C_{\max}$: minimize the makespan on a single machine; jobs have release dates.
  Ex. input: $n = 4$, $p = (1, 3, 2, 2)$, $r = (0, 5, 4, 1)$.

Classification of Scheduling Problems

Example 2.1

- $1|r_j|C_{\max}$: minimize the makespan on a single machine; jobs have release dates.
  Ex. input: $n = 4$, $p = (1, 3, 2, 2)$, $r = (0, 5, 4, 1)$.

- $P||\sum w_j Cj$: minimize the weighted sum of completion times on parallel identical machines.
  Ex. input: $n = 5$, $m = 2$, $p = (1, 2, 2, 4, 8)$, $w = (10, 1, 1, 1, 1)$.

Classification of Scheduling Problems

Example 2.1

- $1|r_j|C_{\max}$: minimize the makespan on a single machine; jobs have release dates.
  Ex. input: $n = 4$, $p = (1, 3, 2, 2)$, $r = (0, 5, 4, 1)$.

- $P||\sum w_j Cj$: minimize the weighted sum of completion times on parallel identical machines.
  Ex. input: $n = 5$, $m = 2$, $p = (1, 2, 2, 4, 8)$, $w = (10, 1, 1, 1, 1)$.

- $P3|\mathtt{prec}, d_j = d|\sum w_j Lj$: minimize the total weighted lateness on 3 parallel identical machines. Jobs must respect precedences and have a common deadline $d_j = d$.
  Ex. input: $n = 6$, $p = (1, 2, 1, 5, 4, 3)$, $d = 2$, $\{1 \prec 2 \prec 4, 1 \prec 3\}$.

Outline

1 Introduction to Scheduling Problems

2 Scheduling Jobs with Due Dates on a Single Machine

3 The *k*-Center Problem

4 Scheduling Jobs on Identical Parallel Machines

5 The Traveling Salesperson Problem (TSP)

6 Greedy Maximization of Submodular Functions

7 Minimum Edge Coloring

Scheduling Jobs with Due Dates on a Single Machine

Given: n jobs $j = 1, \ldots, n$ with processing time $p_j \geq 0$, release date $r_j \geq 0$ and due dates d_j, $j = 1, \ldots, n$.

Scheduling Jobs with Due Dates on a Single Machine

Given: n jobs $j = 1, \dots, n$ with processing time $p_j \geq 0$, release date $r_j \geq 0$ and due dates d_j, $j = 1, \dots, n$.

Task: Schedule each job nonpreemptively for p_j units of time, starting no earlier than time r_j, such that no two jobs overlap.

Scheduling Jobs with Due Dates on a Single Machine

Given: n jobs $j = 1, \ldots, n$ with processing time $p_j \geq 0$, release date $r_j \geq 0$ and due dates d_j, $j = 1, \ldots, n$.

Task: Schedule each job nonpreemptively for p_j units of time, starting no earlier than time r_j, such that no two jobs overlap.

Objective: Minimize the maximum lateness $L_{\max} := \max_{j=1,\ldots,n} L_j$ with $L_j := C_j - d_j$ where C_j denotes the completion time of job j, $j = 1, \ldots, n$.

Scheduling Jobs with Due Dates on a Single Machine

Given: n jobs $j = 1, \ldots, n$ with processing time $p_j \geq 0$, release date $r_j \geq 0$ and due dates d_j, $j = 1, \ldots, n$.

Task: Schedule each job nonpreemptively for p_j units of time, starting no earlier than time r_j, such that no two jobs overlap.

Objective: Minimize the maximum lateness $L_{\max} := \max_{j=1,\ldots,n} L_j$ with $L_j := C_j - d_j$ where C_j denotes the completion time of job j, $j = 1, \ldots, n$.

- In other words, we consider the problem

$$1|r_j|L_{\max}.$$

Scheduling Jobs with Due Dates on a Single Machine

- Minimize the maximum lateness on one machine with release dates: $1|r_j|L_{\max}$.

Theorem 2.2

Deciding whether $L^*_{\max} \leq 0$ is strongly *NP*-complete.

Scheduling Jobs with Due Dates on a Single Machine

- Minimize the maximum lateness on one machine with release dates: $1|r_j|L_{\max}$.

Theorem 2.2

Deciding whether $L^*_{\max} \leq 0$ is strongly *NP*-complete.

Proof: Polynomial transformation of the 3-Partition Problem. ☐

Scheduling Jobs with Due Dates on a Single Machine

- Minimize the maximum lateness on one machine with release dates: $1|r_j|L_{\max}$.

Theorem 2.2

Deciding whether $L^*_{\max} \leq 0$ is strongly NP-complete.

Proof: Polynomial transformation of the 3-Partition Problem. ☐

Corollary 2.3

There is no α-approximation algorithm for the scheduling problem for any α, unless $P = NP$.

Scheduling Jobs with Due Dates on a Single Machine

- Minimize the maximum lateness on one machine with release dates: $1|r_j|L_{\max}$.

Theorem 2.2

Deciding whether $L^*_{\max} \leq 0$ is strongly *NP*-complete.

Proof: Polynomial transformation of the 3-Partition Problem. ☐

Corollary 2.3

There is no α-approximation algorithm for the scheduling problem for any α, unless $P = NP$.

Proof: … ☐

Greedy 2-Approximation Algorithm for Negative Due Dates

For a subset of jobs $S \subseteq \{1, \dots, n\}$ let:

$$r(S) := \min_{j \in S} r_j \qquad p(S) := \sum_{j \in S} p_j \qquad d(S) := \max_{j \in S} d_j$$

Lemma 2.4

Let $L^*_{\max}$ denote the optimal value. For each subset S of jobs

$$L^*_{\max} \geq r(S) + p(S) - d(S) \enspace .$$

Greedy 2-Approximation Algorithm for Negative Due Dates

For a subset of jobs $S \subseteq \{1, \ldots, n\}$ let:

$$r(S) := \min_{j \in S} r_j \qquad p(S) := \sum_{j \in S} p_j \qquad d(S) := \max_{j \in S} d_j$$

Lemma 2.4

Let $L^*_{\max}$ denote the optimal value. For each subset S of jobs

$$L^*_{\max} \geq r(S) + p(S) - d(S) \ .$$

Proof: … ☐

Greedy 2-Approximation Algorithm for Negative Due Dates

Algorithm EDD (earliest due date): Whenever the machine is idle, start to process among all available jobs the one with the earliest due date.

Greedy 2-Approximation Algorithm for Negative Due Dates

Algorithm EDD (earliest due date): Whenever the machine is idle, start to process among all available jobs the one with the earliest due date.

We will see that EDD in an approximation algorithm whenever the due dates are non-positive , i.e., for $1|r_j, d_j \leq 0|L_{\max}$.

Greedy 2-Approximation Algorithm for Negative Due Dates

Algorithm EDD (earliest due date): Whenever the machine is idle, start to process among all available jobs the one with the earliest due date.

We will see that EDD in an approximation algorithm whenever the due dates are non-positive , i.e., for $1|r_j, d_j \leq 0|L_{\max}$.

Theorem 2.5

For the case of non-positive due dates $d_j \leq 0$ for all jobs j, Algorithm EDD is a 2-approximation algorithm.

Greedy 2-Approximation Algorithm for Negative Due Dates

Algorithm EDD (earliest due date): Whenever the machine is idle, start to process among all available jobs the one with the earliest due date.

We will see that EDD in an approximation algorithm whenever the due dates are non-positive , i.e., for $1|r_j, d_j \leq 0|L_{\max}$.

Theorem 2.5

For the case of non-positive due dates $d_j \leq 0$ for all jobs j, Algorithm EDD is a 2-approximation algorithm.

Proof: … ☐

Outline

The k-Center Problem

Given: A finite metric space V with distances d_{ij} for $i, j \in V$ and $k \in \mathbb{N}$.

Task: Find k centers in V, i.e., $S \subseteq V$ with $|S| = k$.

Objective: Minimize $\max_{i \in V} d(i, S)$ where $d(i, S) := \min_{j \in S} d_{ij}$.

The k-Center Problem

Given: A finite metric space V with distances d_{ij} for $i, j \in V$ and $k \in \mathbb{N}$.

Task: Find k centers in V, i.e., $S \subseteq V$ with $|S| = k$.

Objective: Minimize $\max_{i \in V} d(i, S)$ where $d(i, S) := \min_{j \in S} d_{ij}$.

Greedy Algorithm:

1. pick arbitrary $i \in V$ and set $S := \{i\}$;
2. while $|S| < k$ let $j := \arg\max_{\ell \in V} d(\ell, S)$ and set $S := S \cup \{j\}$;

The k-Center Problem

Given: A finite metric space V with distances d_{ij} for $i, j \in V$ and $k \in \mathbb{N}$.

Task: Find k centers in V, i.e., $S \subseteq V$ with $|S| = k$.

Objective: Minimize $\max_{i \in V} d(i, S)$ where $d(i, S) := \min_{j \in S} d_{ij}$.

Greedy Algorithm:

1. pick arbitrary $i \in V$ and set $S := \{i\}$;
2. while $|S| < k$ let $j := \arg\max_{\ell \in V} d(\ell, S)$ and set $S := S \cup \{j\}$;

Theorem 2.6

The algorithm is a 2-approximation algorithm for the k-Center Problem.

The k-Center Problem: hardness of approximation

Theorem 2.7

There is no α-approximation algorithm for the k-center problem for $\alpha < 2$, unless $P = NP$.

The k-Center Problem: hardness of approximation

Theorem 2.7

There is no α-approximation algorithm for the k-center problem for $\alpha < 2$, unless $P = NP$.

Proof: Reduction from Dominating Set Problem…

☐

Outline

Scheduling Jobs on Identical Parallel Machines

Given: n jobs $j = 1, \dots, n$ with processing time $p_j \geq 0$, $j = 1, \dots, n$, and m identical parallel machines.

Task: Process each job j nonpreemptively for p_j units of time on one of the m machines. A machine can process at most one job at a time.

Objective: Minimize the maximum machine load, i.e., the maximum completion time $C_{\max} := \max\limits_{j=1,\dots,n} C_j$ (makespan).

Scheduling Jobs on Identical Parallel Machines

Given: n jobs $j = 1, \dots, n$ with processing time $p_j \geq 0$, $j = 1, \dots, n$, and m identical parallel machines.

Task: Process each job j nonpreemptively for p_j units of time on one of the m machines. A machine can process at most one job at a time.

Objective: Minimize the maximum machine load, i.e., the maximum completion time $C_{\max} := \max\limits_{j=1,\dots,n} C_j$ (makespan).

In other words, we consider $P||C_{\max}$.

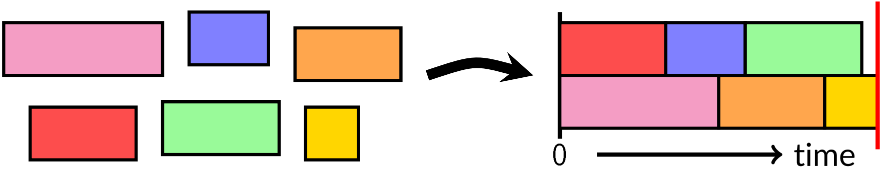
Scheduling Jobs on Identical Parallel Machines

Given: n jobs $j = 1, \ldots, n$ with processing time $p_j \geq 0$, $j = 1, \ldots, n$, and m identical parallel machines.

Task: Process each job j nonpreemptively for p_j units of time on one of the m machines. A machine can process at most one job at a time.

Objective: Minimize the maximum machine load, i.e., the maximum completion time $C_{\max} := \max_{j=1,\ldots,n} C_j$ (makespan).

In other words, we consider $P||C_{\max}$.



Scheduling Jobs on Indentical Parallel Machines

Theorem 2.8

This scheduling problem is strongly *NP*-hard.

Scheduling Jobs on Indentical Parallel Machines

Theorem 2.8

This scheduling problem is strongly *NP*-hard.

Proof: Reduction from 3-Partition…

□

Local Search

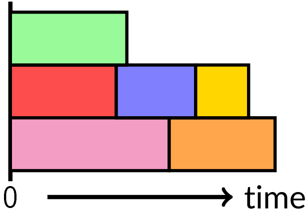
Local Search Algorithm

1. start with an arbitrary schedule;
2. let $j := \arg\max_{j=1,\dots,n} C_j$;
3. if there is a machine i with load $< C_j - p_j$, reassign j to i and goto 2;

Local Search

Local Search Algorithm

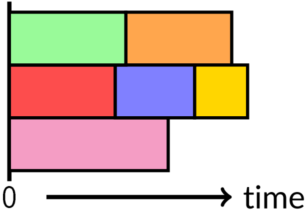
1. start with an arbitrary schedule;
2. let $j := \arg \max_{j=1,\ldots,n} C_j$;
3. if there is a machine i with load $< C_j - p_j$, reassign j to i and goto 2;



Local Search

Local Search Algorithm

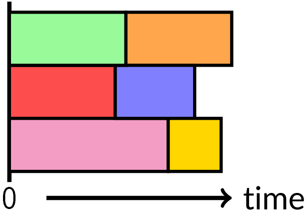
1. start with an arbitrary schedule;
2. let $j := \arg\max_{j=1,\ldots,n} C_j$;
3. if there is a machine i with load $< C_j - p_j$, reassign j to i and goto 2;



Local Search

Local Search Algorithm

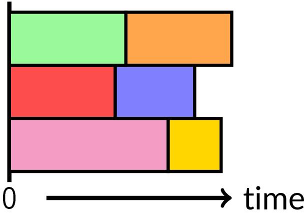
1. start with an arbitrary schedule;
2. let $j := \arg\max_{j=1,\ldots,n} C_j$;
3. if there is a machine i with load $< C_j - p_j$, reassign j to i and goto 2;



Local Search

Local Search Algorithm

1. start with an arbitrary schedule;
2. let $j := \arg\max_{j=1,\dots,n} C_j$;
3. if there is a machine i with load $< C_j - p_j$, reassign j to i and goto 2;



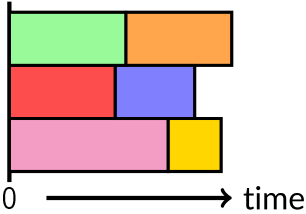
Theorem 2.9

When the algorithm terminates, the makespan of the final solution is at most $2 - \dfrac{1}{m}$ times the optimum makespan.

Local Search

Local Search Algorithm

1. start with an arbitrary schedule;
2. let $j := \arg\max_{j=1,\dots,n} C_j$;
3. if there is a machine i with load $< C_j - p_j$, reassign j to i and goto 2;



Theorem 2.9

When the algorithm terminates, the makespan of the final solution is at most $2 - \dfrac{1}{m}$ times the optimum makespan.

Lemma 2.10

If the Local Search Algorithm always moves job j to a currently least loaded machine, it terminates after at most n iterations.

List Scheduling

List Scheduling Algorithm

1. start with the empty schedule and consider the jobs in arbitrary order;
2. always assign the next job to the currently least loaded machine;

List Scheduling

List Scheduling Algorithm

1. start with the empty schedule and consider the jobs in arbitrary order;
2. always assign the next job to the currently least loaded machine;

Theorem 2.11

The List Scheduling Algorithm is a $(2 - \frac{1}{m})$-approximation algorithm.

List Scheduling

List Scheduling Algorithm

1. start with the empty schedule and consider the jobs in arbitrary order;
2. always assign the next job to the currently least loaded machine;

Theorem 2.11

The List Scheduling Algorithm is a $(2 - \frac{1}{m})$-approximation algorithm.

Proof: …

☐

LPT rule

The variant of list scheduling where jobs are considered in non-increasing order is called the longest processing time (LPT) rule.

LPT rule

The variant of list scheduling where jobs are considered in non-increasing order is called the longest processing time (LPT) rule.

Theorem 2.12

The LPT rule is a $(\frac{4}{3} - \frac{1}{3m})$-approximation algorithm.

LPT rule

The variant of list scheduling where jobs are considered in non-increasing order is called the longest processing time (LPT) rule.

Theorem 2.12

The LPT rule is a $(\frac{4}{3}-\frac{1}{3m})$-approximation algorithm.

Proof: … ☐

LPT rule

The variant of list scheduling where jobs are considered in non-increasing order is called the longest processing time (LPT) rule.

Theorem 2.12

The LPT rule is a $(\frac{4}{3} - \frac{1}{3m})$-approximation algorithm.

Proof: … ☐

Later: There is a PTAS for this scheduling problem!

Outline

Traveling Salesperson Problem (TSP)

Given: Finite set of n points $V = \{1, \ldots, n\}$ with (symmetric) distances $d_{ij} \geq 0$ for $i, j \in V$.

Traveling Salesperson Problem (TSP)

Given: Finite set of n points $V = \{1, \ldots, n\}$ with (symmetric) distances $d_{ij} \geq 0$ for $i, j \in V$.

Task: Find a closed tour that visits every point in V exactly once (i.e., a permutation π of V).

Traveling Salesperson Problem (TSP)

Given: Finite set of n points $V = \{1, \dots, n\}$ with (symmetric) distances $d_{ij} \geq 0$ for $i, j \in V$.

Task: Find a closed tour that visits every point in V exactly once (i.e., a permutation π of V).

Objective: Minimize total length of tour: $d_{\pi(n)\pi(1)} + \sum_{i=1}^{n-1} d_{\pi(i)\pi(i+1)}$

Traveling Salesperson Problem (TSP)

Given: Finite set of n points $V = \{1, \dots, n\}$ with (symmetric) distances $d_{ij} \geq 0$ for $i, j \in V$.

Task: Find a closed tour that visits every point in V exactly once (i.e., a permutation π of V).

Objective: Minimize total length of tour: $d_{\pi(n)\pi(1)} + \sum_{i=1}^{n-1} d_{\pi(i)\pi(i+1)}$

Theorem 2.13

There is no α-approximation algorithm for the TSP for any α (e. g., $\alpha = 2^n$), unless $P = NP$.

Traveling Salesperson Problem (TSP)

Given: Finite set of n points $V = \{1, \dots, n\}$ with (symmetric) distances $d_{ij} \geq 0$ for $i, j \in V$.

Task: Find a closed tour that visits every point in V exactly once (i.e., a permutation π of V).

Objective: Minimize total length of tour: $d_{\pi(n)\pi(1)} + \sum_{i=1}^{n-1} d_{\pi(i)\pi(i+1)}$

Theorem 2.13

There is no α-approximation algorithm for the TSP for any α (e. g., $\alpha = 2^n$), unless $P = NP$.

Proof: Reduction from Hamiltonian Circuit… ☐

Traveling Salesperson Problem (TSP)

Given: Finite set of n points $V = \{1, \dots, n\}$ with (symmetric) distances $d_{ij} \geq 0$ for $i, j \in V$.

Task: Find a closed tour that visits every point in V exactly once (i.e., a permutation π of V).

Objective: Minimize total length of tour: $d_{\pi(n)\pi(1)} + \sum_{i=1}^{n-1} d_{\pi(i)\pi(i+1)}$

Theorem 2.13

There is no α-approximation algorithm for the TSP for any α (e. g., $\alpha = 2^n$), unless $P = NP$.

Proof: Reduction from Hamiltonian Circuit… ☐

In the following we thus consider the metric TSP where distances between cities fulfill the triangle inequalities.

Lower Bounds for TSP

Lemma 2.14

The cost of a minimum spanning tree of the complete graph with nodes V and edge costs d_{ij} is a lower bound on the length of a shortest TSP tour.

Lower Bounds for TSP

Lemma 2.14

The cost of a minimum spanning tree of the complete graph with nodes V and edge costs d_{ij} is a lower bound on the length of a shortest TSP tour.

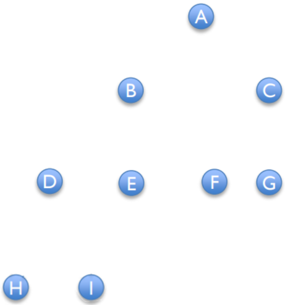
Proof:… □

Lower Bounds for TSP

Lemma 2.14

The cost of a minimum spanning tree of the complete graph with nodes V and edge costs d_{ij} is a lower bound on the length of a shortest TSP tour.

Proof:… □

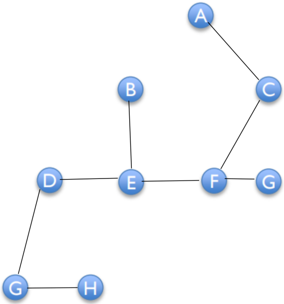


Lower Bounds for TSP

Lemma 2.14

The cost of a minimum spanning tree of the complete graph with nodes V and edge costs d_{ij} is a lower bound on the length of a shortest TSP tour.

Proof:… □

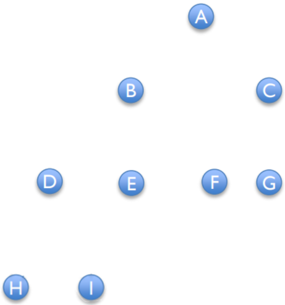


Lower Bounds for TSP

Lemma 2.14

The cost of a minimum spanning tree of the complete graph with nodes V and edge costs d_{ij} is a lower bound on the length of a shortest TSP tour.

Proof:… ☐

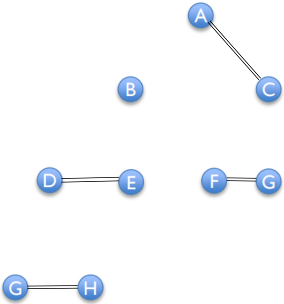


Lower Bounds for TSP

Lemma 2.14

The cost of a minimum spanning tree of the complete graph with nodes V and edge costs d_{ij} is a lower bound on the length of a shortest TSP tour.

Proof:… ☐



Lower Bounds for TSP

Lemma 2.14

The cost of a minimum spanning tree of the complete graph with nodes V and edge costs d_{ij} is a lower bound on the length of a shortest TSP tour.

Proof:… $\square$

Lemma 2.15

Let $S \subseteq V$ with $|S|$ even and consider the complete graph on nodes S and edge costs d_{ij}. Twice the cost of a min-cost perfect matching of S is a lower bound on the length of a shortest TSP tour of V.

Lower Bounds for TSP

Lemma 2.14

The cost of a minimum spanning tree of the complete graph with nodes V and edge costs d_{ij} is a lower bound on the length of a shortest TSP tour.

Proof:… □

Lemma 2.15

Let $S \subseteq V$ with $|S|$ even and consider the complete graph on nodes S and edge costs d_{ij}. Twice the cost of a min-cost perfect matching of S is a lower bound on the length of a shortest TSP tour of V.

Proof:… □

Double-Tree Algorithm for TSP

1. compute a minimum spanning tree T on V;

Double-Tree Algorithm for TSP

1. compute a minimum spanning tree T on V;
2. take two copies of each edge in T;

Double-Tree Algorithm for TSP

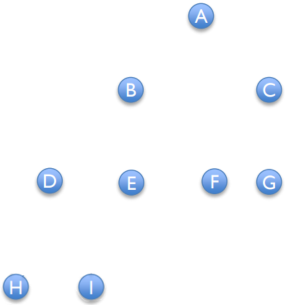
1. compute a minimum spanning tree T on V;
2. take two copies of each edge in T;
3. compute a Eulerian tour on the resulting graph;

Double-Tree Algorithm for TSP

1. compute a minimum spanning tree T on V;
2. take two copies of each edge in T;
3. compute a Eulerian tour on the resulting graph;
4. reduce the Eulerian tour to a TSP tour by shortcutting;

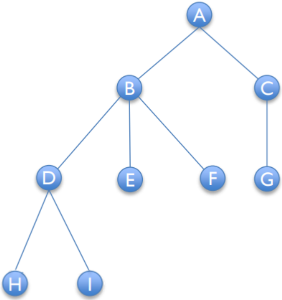
Double-Tree Algorithm for TSP

1. compute a minimum spanning tree T on V;
2. take two copies of each edge in T;
3. compute a Eulerian tour on the resulting graph;
4. reduce the Eulerian tour to a TSP tour by shortcutting;



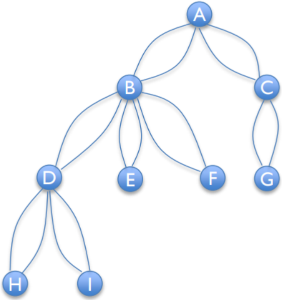
Double-Tree Algorithm for TSP

1. compute a minimum spanning tree T on V;
2. take two copies of each edge in T;
3. compute a Eulerian tour on the resulting graph;
4. reduce the Eulerian tour to a TSP tour by shortcutting;



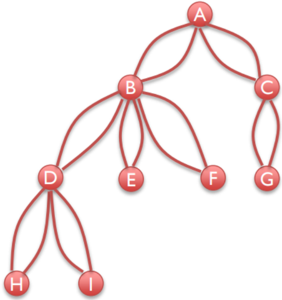
Double-Tree Algorithm for TSP

1. compute a minimum spanning tree T on V;
2. take two copies of each edge in T;
3. compute a Eulerian tour on the resulting graph;
4. reduce the Eulerian tour to a TSP tour by shortcutting;



Double-Tree Algorithm for TSP

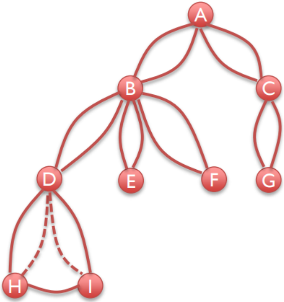
1. compute a minimum spanning tree T on V;
2. take two copies of each edge in T;
3. compute a Eulerian tour on the resulting graph;
4. reduce the Eulerian tour to a TSP tour by shortcutting;



A,B,D,H,D,I,D,B,E,B,F,B,A,C,G,C,A

Double-Tree Algorithm for TSP

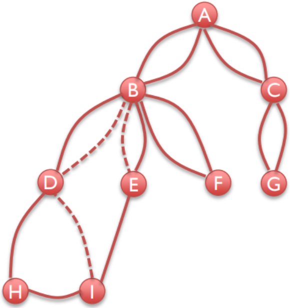
1. compute a minimum spanning tree T on V;
2. take two copies of each edge in T;
3. compute a Eulerian tour on the resulting graph;
4. reduce the Eulerian tour to a TSP tour by shortcutting;



A,B,D,H,~~D~~,I,D,B,E,B,F,B,A,C,G,C,A

Double-Tree Algorithm for TSP

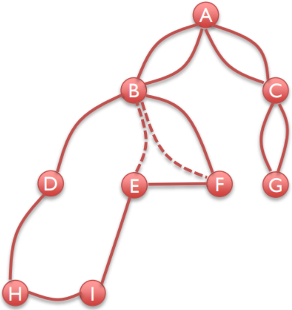
1. compute a minimum spanning tree T on V;
2. take two copies of each edge in T;
3. compute a Eulerian tour on the resulting graph;
4. reduce the Eulerian tour to a TSP tour by shortcutting;



A,B,D,H,~~D~~,I,~~D~~,~~B~~,E,B,F,B,A,C,G,C,A

Double-Tree Algorithm for TSP

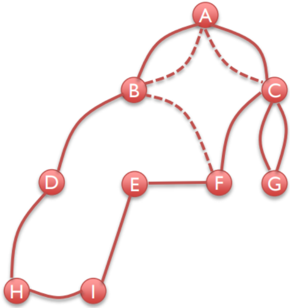
1. compute a minimum spanning tree T on V;
2. take two copies of each edge in T;
3. compute a Eulerian tour on the resulting graph;
4. reduce the Eulerian tour to a TSP tour by shortcutting;



A,B,D,H,~~D~~,I,~~D~~,~~B~~,E,~~B~~,F,B,A,C,G,C,A

Double-Tree Algorithm for TSP

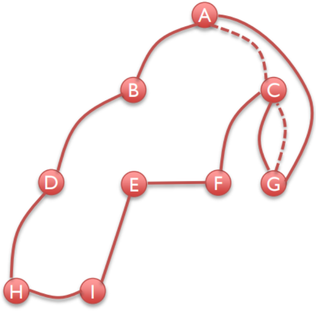
1. compute a minimum spanning tree T on V;
2. take two copies of each edge in T;
3. compute a Eulerian tour on the resulting graph;
4. reduce the Eulerian tour to a TSP tour by shortcutting;



A,B,D,H,~~D~~,I,~~D~~,~~B~~,E,~~B~~,F,~~B~~,~~A~~,C,G,C,A

Double-Tree Algorithm for TSP

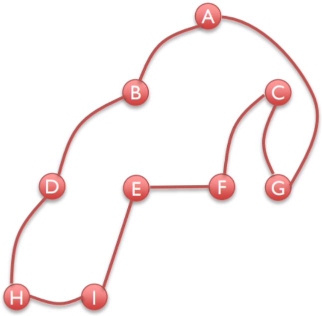
1. compute a minimum spanning tree T on V;
2. take two copies of each edge in T;
3. compute a Eulerian tour on the resulting graph;
4. reduce the Eulerian tour to a TSP tour by shortcutting;



A,B,D,H,~~D~~,I,~~D~~,~~B~~,E,~~B~~,F,~~B~~,~~A~~,C,G,~~C~~,A

Double-Tree Algorithm for TSP

1. compute a minimum spanning tree T on V;
2. take two copies of each edge in T;
3. compute a Eulerian tour on the resulting graph;
4. reduce the Eulerian tour to a TSP tour by shortcutting;



A,B,D,H,~~D~~,I,~~D~~,~~B~~,E,~~B~~,F,~~B~~,~~A~~,C,G,~~C~~,A

Double-Tree Algorithm for TSP

1. compute a minimum spanning tree T on V;
2. take two copies of each edge in T;
3. compute a Eulerian tour on the resulting graph;
4. reduce the Eulerian tour to a TSP tour by shortcutting;

Theorem 2.16

The Double-Tree Algorithm is a 2-approximation algorithm for the TSP.

TSP: Nearest Insertion Algorithm

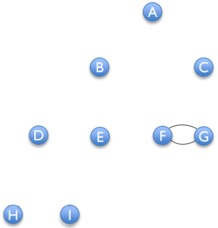
Nearest Insertion Algorithm

1. start with tour through two cities $S := \{i, j\}$ of minimum distance d_{ij};
2. for each uninserted city k, compute the minimum distance $d(k, S)$ between k and a city in the current tour;
3. let $\ell := \arg\min_{k \notin S} d(k, S)$; add ℓ to the tour after its nearest city;
4. set $S := S \cup \{\ell\}$; if $S \neq V$, then goto 2;

TSP: Nearest Insertion Algorithm

Nearest Insertion Algorithm

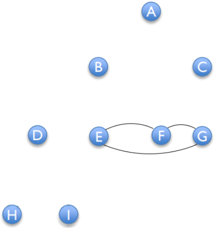
1. start with tour through two cities $S := \{i, j\}$ of minimum distance d_{ij};
2. for each uninserted city k, compute the minimum distance $d(k, S)$ between k and a city in the current tour;
3. let $\ell := \arg\min_{k \notin S} d(k, S)$; add ℓ to the tour after its nearest city;
4. set $S := S \cup \{\ell\}$; if $S \neq V$, then goto 2;



TSP: Nearest Insertion Algorithm

Nearest Insertion Algorithm

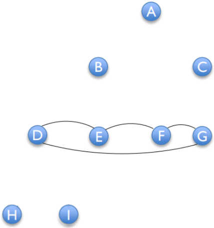
1. start with tour through two cities $S := \{i, j\}$ of minimum distance d_{ij};
2. for each uninserted city k, compute the minimum distance $d(k, S)$ between k and a city in the current tour;
3. let $\ell := \arg\min_{k \notin S} d(k, S)$; add ℓ to the tour after its nearest city;
4. set $S := S \cup \{\ell\}$; if $S \neq V$, then goto 2;



TSP: Nearest Insertion Algorithm

Nearest Insertion Algorithm

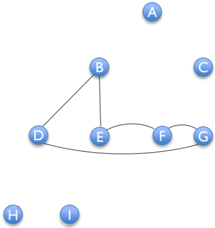
1. start with tour through two cities $S := \{i, j\}$ of minimum distance d_{ij};
2. for each uninserted city k, compute the minimum distance $d(k, S)$ between k and a city in the current tour;
3. let $\ell := \arg\min_{k \notin S} d(k, S)$; add ℓ to the tour after its nearest city;
4. set $S := S \cup \{\ell\}$; if $S \neq V$, then goto 2;



TSP: Nearest Insertion Algorithm

Nearest Insertion Algorithm

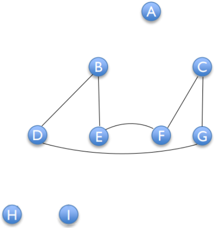
1. start with tour through two cities $S := \{i, j\}$ of minimum distance d_{ij};
2. for each uninserted city k, compute the minimum distance $d(k, S)$ between k and a city in the current tour;
3. let $\ell := \arg\min_{k \notin S} d(k, S)$; add ℓ to the tour after its nearest city;
4. set $S := S \cup \{\ell\}$; if $S \neq V$, then goto 2;



TSP: Nearest Insertion Algorithm

Nearest Insertion Algorithm

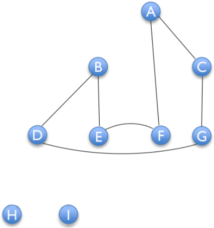
1. start with tour through two cities $S := \{i, j\}$ of minimum distance d_{ij};
2. for each uninserted city k, compute the minimum distance $d(k, S)$ between k and a city in the current tour;
3. let $\ell := \arg\min_{k \notin S} d(k, S)$; add ℓ to the tour after its nearest city;
4. set $S := S \cup \{\ell\}$; if $S \neq V$, then goto 2;



TSP: Nearest Insertion Algorithm

Nearest Insertion Algorithm

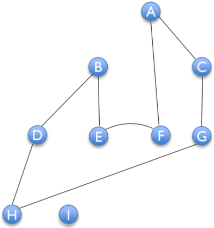
1. start with tour through two cities $S := \{i, j\}$ of minimum distance d_{ij};
2. for each uninserted city k, compute the minimum distance $d(k, S)$ between k and a city in the current tour;
3. let $\ell := \arg\min_{k \notin S} d(k, S)$; add ℓ to the tour after its nearest city;
4. set $S := S \cup \{\ell\}$; if $S \neq V$, then goto 2;



TSP: Nearest Insertion Algorithm

Nearest Insertion Algorithm

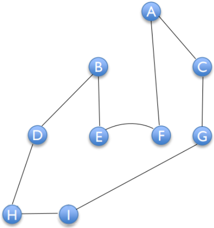
1. start with tour through two cities $S := \{i, j\}$ of minimum distance d_{ij};
2. for each uninserted city k, compute the minimum distance $d(k, S)$ between k and a city in the current tour;
3. let $\ell := \arg\min_{k \notin S} d(k, S)$; add ℓ to the tour after its nearest city;
4. set $S := S \cup \{\ell\}$; if $S \neq V$, then goto 2;



TSP: Nearest Insertion Algorithm

Nearest Insertion Algorithm

1. start with tour through two cities $S := \{i, j\}$ of minimum distance d_{ij};
2. for each uninserted city k, compute the minimum distance $d(k, S)$ between k and a city in the current tour;
3. let $\ell := \arg\min_{k \notin S} d(k, S)$; add ℓ to the tour after its nearest city;
4. set $S := S \cup \{\ell\}$; if $S \neq V$, then goto 2;



TSP: Nearest Insertion Algorithm

Nearest Insertion Algorithm

1. start with tour through two cities $S := \{i, j\}$ of minimum distance d_{ij};
2. for each uninserted city k, compute the minimum distance $d(k, S)$ between k and a city in the current tour;
3. let $\ell := \arg\min_{k \notin S} d(k, S)$; add ℓ to the tour after its nearest city;
4. set $S := S \cup \{\ell\}$; if $S \neq V$, then goto 2;

Theorem 2.17

The Nearest Insertion Algorithm is a 2-approximation algorithm for the metric TSP.

TSP: Nearest Insertion Algorithm

Nearest Insertion Algorithm

1. start with tour through two cities $S := \{i, j\}$ of minimum distance d_{ij};
2. for each uninserted city k, compute the minimum distance $d(k, S)$ between k and a city in the current tour;
3. let $\ell := \arg\min_{k \notin S} d(k, S)$; add ℓ to the tour after its nearest city;
4. set $S := S \cup \{\ell\}$; if $S \neq V$, then goto 2;

Theorem 2.17

The Nearest Insertion Algorithm is a 2-approximation algorithm for the metric TSP.

Notice: The Nearest Insertion Algorithm is closely related to Prim's Algorithm and the Double-Tree Algorithm.

Christofides' Algorithm

1. compute a minimum spanning tree T on V; let S be the subset of nodes with odd degree in T;

Christofides' Algorithm

1. compute a minimum spanning tree T on V; let S be the subset of nodes with odd degree in T;
2. compute a min-cost perfect matching on S and add it to T;

Christofides' Algorithm

1. compute a minimum spanning tree T on V; let S be the subset of nodes with odd degree in T;
2. compute a min-cost perfect matching on S and add it to T;
3. compute a Eulerian tour on the resulting graph;

Christofides' Algorithm

1. compute a minimum spanning tree T on V; let S be the subset of nodes with odd degree in T;
2. compute a min-cost perfect matching on S and add it to T;
3. compute a Eulerian tour on the resulting graph;
4. reduce the Eulerian tour to a TSP tour by shortcutting;

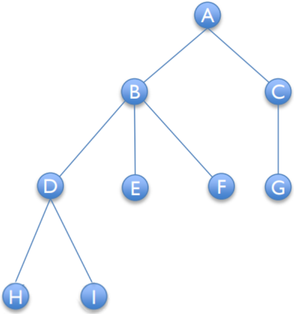
Christofides' Algorithm

1. compute a minimum spanning tree T on V; let S be the subset of nodes with odd degree in T;
2. compute a min-cost perfect matching on S and add it to T;
3. compute a Eulerian tour on the resulting graph;
4. reduce the Eulerian tour to a TSP tour by shortcutting;

Proof:… ☐

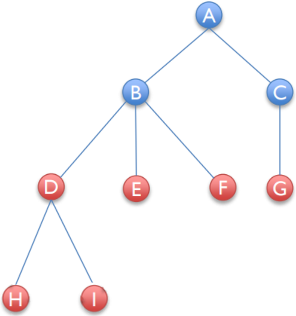
Christofides' Algorithm

1. compute a minimum spanning tree T on V; let S be the subset of nodes with odd degree in T;
2. compute a min-cost perfect matching on S and add it to T;
3. compute a Eulerian tour on the resulting graph;
4. reduce the Eulerian tour to a TSP tour by shortcutting;



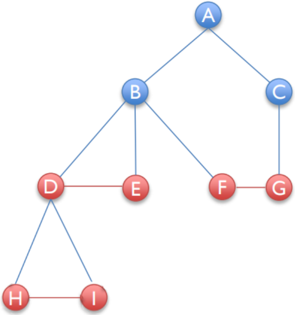
Christofides' Algorithm

1. compute a minimum spanning tree T on V; let S be the subset of nodes with odd degree in T;
2. compute a min-cost perfect matching on S and add it to T;
3. compute a Eulerian tour on the resulting graph;
4. reduce the Eulerian tour to a TSP tour by shortcutting;



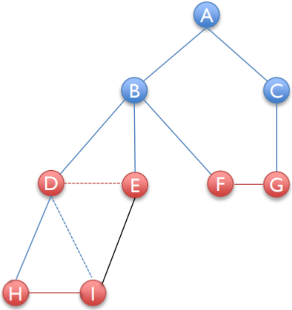
Christofides' Algorithm

1. compute a minimum spanning tree T on V; let S be the subset of nodes with odd degree in T;
2. compute a min-cost perfect matching on S and add it to T;
3. compute a Eulerian tour on the resulting graph;
4. reduce the Eulerian tour to a TSP tour by shortcutting;



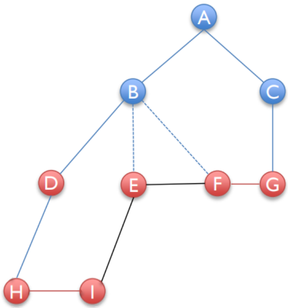
Christofides' Algorithm

1. compute a minimum spanning tree T on V; let S be the subset of nodes with odd degree in T;
2. compute a min-cost perfect matching on S and add it to T;
3. compute a Eulerian tour on the resulting graph;
4. reduce the Eulerian tour to a TSP tour by shortcutting;



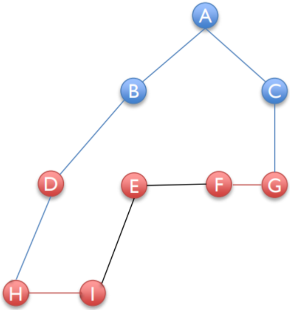
Christofides' Algorithm

1. compute a minimum spanning tree T on V; let S be the subset of nodes with odd degree in T;
2. compute a min-cost perfect matching on S and add it to T;
3. compute a Eulerian tour on the resulting graph;
4. reduce the Eulerian tour to a TSP tour by shortcutting;



Christofides' Algorithm

1. compute a minimum spanning tree T on V; let S be the subset of nodes with odd degree in T;
2. compute a min-cost perfect matching on S and add it to T;
3. compute a Eulerian tour on the resulting graph;
4. reduce the Eulerian tour to a TSP tour by shortcutting;



Christofides' Algorithm

1. compute a minimum spanning tree T on V; let S be the subset of nodes with odd degree in T;
2. compute a min-cost perfect matching on S and add it to T;
3. compute a Eulerian tour on the resulting graph;
4. reduce the Eulerian tour to a TSP tour by shortcutting;

Theorem 2.18

Christofides' Algorithm is a $\frac{3}{2}$-approximation algorithm for the TSP.

Proof:… $\square$

Approximability of Metric TSP

Theorem 2.19 (Papadimitriou & Vempala 2006)

There is no α-approximation algorithm for the metric TSP for $\alpha < 220/219$, unless $P = NP$.

Approximability of Metric TSP

Theorem 2.19 (Papadimitriou & Vempala 2006)

There is no α-approximation algorithm for the metric TSP for $\alpha < 220/219$, unless $P = NP$.

Theorem 2.20 (Karlin, Klein, Gharan 2020)

A $(3/2 - \varepsilon)$-approximation algorithm for some $\varepsilon > 10^{-36}$.

Approximability of Metric TSP

Theorem 2.19 (Papadimitriou & Vempala 2006)

There is no α-approximation algorithm for the metric TSP for $\alpha < 220/219$, unless $P = NP$.

Theorem 2.20 (Karlin, Klein, Gharan 2020)

A $(3/2 - \varepsilon)$-approximation algorithm for some $\varepsilon > 10^{-36}$.

Remark: There is a PTAS for the Euclidean TSP (special case).

Outline

1. Introduction to Scheduling Problems
2. Scheduling Jobs with Due Dates on a Single Machine
3. The k-Center Problem
4. Scheduling Jobs on Identical Parallel Machines
5. The Traveling Salesperson Problem (TSP)
6. **Greedy Maximization of Submodular Functions**
7. Minimum Edge Coloring

Submodular Function

Given a finite ground set V, we consider set functions $f : 2^V \to \mathbb{R}$, i.e. we assign each $S \subseteq V$ a value $f(S) \in \mathbb{R}$.
We assume that $f(S)$ can be evaluated in constant time, for each $S \subseteq V$.

Definition 2.21

A function $f : 2^V \to \mathbb{R}$ is submodular if for every $A \subseteq B \subseteq V$ and $i \in V \setminus B$ it holds

$$f(A \cup \{i\}) - f(A) \geq f(B \cup \{i\}) - f(B).$$

Submodular Function

Given a finite ground set V, we consider set functions $f : 2^V \to \mathbb{R}$, i.e. we assign each $S \subseteq V$ a value $f(S) \in \mathbb{R}$.
We assume that $f(S)$ can be evaluated in constant time, for each $S \subseteq V$.

Definition 2.21

A function $f : 2^V \to \mathbb{R}$ is submodular if for every $A \subseteq B \subseteq V$ and $i \in V \setminus B$ it holds

$$f(A \cup \{i\}) - f(A) \geq f(B \cup \{i\}) - f(B).$$

Equivalently, it can be seen that f is submodular iff

$$f(A \cup B) + f(A \cap B) \leq f(A) + f(B), \quad \forall A, B \subseteq V.$$

In words, f is submodular if it exhibits a *diminishing returns* property.

Submodular Function Maximization

We consider the problem of maximizing a nonnegative monotone submodular function, i.e., a submodular function f such that

$$0 \leq f(A) \leq f(B), \qquad \forall A \subseteq B \subseteq V,$$

subject to a cardinality constraint:

$$\max_{S \subseteq V} \{ f(S) : \ |S| \leq k \}.$$

Submodular Function Maximization

We consider the problem of maximizing a nonnegative monotone submodular function, i.e., a submodular function f such that

$$0 \leq f(A) \leq f(B), \qquad \forall A \subseteq B \subseteq V,$$

subject to a cardinality constraint:

$$\max_{S \subseteq V} \{ f(S) : \ |S| \leq k \}.$$

This problem generalizes many classical problems in discrete mathematics !

Submodular Function Maximization

$$\max_{S\subseteq V}\{f(S):\ |S|\leq k\},\qquad f \text{ nonnegative monotone submodular.}$$

Special cases

- Linear (aka modular) functions: $f(S)=\sum_{i\in S} w_i$ for some $w_i \geq 0$

Submodular Function Maximization

$$\max_{S\subseteq V}\{f(S):\ |S|\leq k\},\qquad f \text{ nonnegative monotone submodular.}$$

Special cases

- Linear (aka modular) functions: $f(S)=\sum_{i\in S} w_i$ for some $w_i \geq 0$
- The rank function of a matroid

$$f(S)=\max\{|U|:\ U\subseteq S,\ U \text{ independent}\}.$$

Submodular Function Maximization

$$\max_{S\subseteq V}\{f(S):\ |S|\leq k\},\qquad f \text{ nonnegative monotone submodular.}$$

Special cases

- Linear (aka modular) functions: $f(S)=\sum_{i\in S} w_i$ for some $w_i\geq 0$
- The rank function of a matroid
- Weighted coverage functions: Given a collection of subsets $A_1,\ldots,A_n$ of a finite universe U and some weigths $w_u\geq 0$ $(\forall u\in U)$ and a set $S\subseteq\{1,\ldots,n\}$, $f(S)$ is the weight of elements covered by the union of the A_k's with $k\in S$: $f(S)=\sum_{u\in\bigcup_{k\in S}A_k} w_u$

Submodular Function Maximization

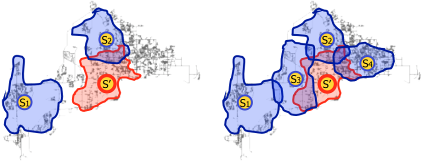
$$\max_{S\subseteq V}\{f(S):\ |S|\leq k\},\qquad f \text{ nonnegative monotone submodular.}$$

Special cases

- Linear (aka modular) functions: $f(S)=\sum_{i\in S} w_i$ for some $w_i\geq 0$
- The rank function of a matroid
- Weighted coverage functions: Given a collection of subsets $A_1,\dots,A_n$ of a finite universe U and some weigths $w_u\geq 0$ ($\forall u\in U$) and a set $S\subseteq\{1,\dots,n\}$, $f(S)$ is the weight of elements covered by the union of the A_k's with $k\in S$: $f(S)=\sum_{u\in\bigcup_{k\in S}A_k} w_u$

Applications

- Sensor location
- Antenna selection



Submodular Function Maximization

$$\max_{S\subseteq V}\{f(S):\ |S|\leq k\},\qquad f \text{ nonnegative monotone submodular.}$$

Special cases

- Linear (aka modular) functions: $f(S)=\sum_{i\in S} w_i$ for some $w_i\geq 0$
- The rank function of a matroid
- Weighted coverage functions
- Facility location:

  There is a collection of n potential locations to open facilities to serve m customers. Opening a facility at location j provides service of value $M_{i,j}\geq 0$ to customer i. If we open the subset of facilities $S\subseteq\{1,\ldots,n\}$ and each customer selects the opened facility with highest value, the total value provided to all customers is

  $$f(S)=\sum_{i\in[m]}\max_{j\in S} M_{ij}$$

Greedy Algorithm for Submodular Optimization

$$\max_{S\subseteq V}\{f(S):\ |S|\leq k\},\qquad f\text{ nonnegative monotone submodular.}$$

Greedy Algorithm

```
$S_0 \leftarrow \emptyset$
For $i = 1, \dots k$:
    $e_i \leftarrow \operatorname{argmax}_{e\in V}\ f(S_{i-1}\cup\{e\}) - f(S_{i-1})$
    $S_i \leftarrow S_{i-1}\cup\{e_i\}$
return $S_k$
```

Greedy Algorithm for Submodular Optimization

$$\max_{S\subseteq V}\{f(S):\ |S|\leq k\},\qquad f \text{ nonnegative monotone submodular.}$$

Greedy Algorithm

```
$S_0 \leftarrow \emptyset$
For $i = 1, \dots k$:
    $e_i \leftarrow \operatorname{argmax}_{e\in V} \; f(S_{i-1}\cup\{e\}) - f(S_{i-1})$
    $S_i \leftarrow S_{i-1}\cup\{e_i\}$
return $S_k$
```

Theorem 2.22 (Nemhauser, Wolsey 1978)

The greedy algorithm is a $(1-\frac{1}{e})$- approximation algorithm for the problem of maximizing a nonnegative monotone submodular function subject to a cardinality constraint.

Greedy Algorithm for Submodular Optimization

$$\max_{S\subseteq V}\{f(S):\ |S|\leq k\},\qquad f \text{ nonnegative monotone submodular.}$$

Greedy Algorithm

```
$S_0 \leftarrow \emptyset$
For $i = 1, \ldots k$:
        $e_i \leftarrow \operatorname{argmax}_{e\in V}\ f(S_{i-1}\cup\{e\}) - f(S_{i-1})$
        $S_i \leftarrow S_{i-1}\cup\{e_i\}$
return $S_k$
```

Theorem 2.22 (Nemhauser, Wolsey 1978)

The greedy algorithm is a $(1-\frac{1}{e})$- approximation algorithm for the problem of maximizing a nonnegative monotone submodular function subject to a cardinality constraint.

Proof

Submodular function maximization: further results

Theorem 2.23 (Sviridenko 2004)

A modified greedy algorithm achieves an approximation ratio of $(1-1/e)$ maximizing monotone submodular functions subject to a *knapsack constraint* $\sum_{i \in S} w_i \leq B$.

Submodular function maximization: further results

Theorem 2.23 (Sviridenko 2004)

A modified greedy algorithm achieves an approximation ratio of $(1 - 1/e)$ maximizing monotone submodular functions subject to a *knapsack constraint* $\sum_{i \in S} w_i \leq B$.

Theorem 2.24 (Fisher, Nemhauser, Wolsey 1978)

The greedy algorithm is a $\frac{1}{2}-$approximation algorithm for the problem of maximizing a ≥ 0 monotone submodular function over a matroid.

Submodular function maximization: further results

Theorem 2.23 (Sviridenko 2004)

A modified greedy algorithm achieves an approximation ratio of $(1-1/e)$ maximizing monotone submodular functions subject to a *knapsack constraint* $\sum_{i\in S} w_i \leq B$.

Theorem 2.24 (Fisher, Nemhauser, Wolsey 1978)

The greedy algorithm is a $\frac{1}{2}-$approximation algorithm for the problem of maximizing a ≥ 0 monotone submodular function over a matroid.

Theorem 2.25 (Calinescu, Chekuri, Pál, Vondrák 2009)

A *continuous greedy algorithm* used with a technique called *pipage rounding* finds a $(1-1/e)$-approximate solution for the above problem.

Submodular function maximization: further results

Theorem 2.23 (Sviridenko 2004)

A modified greedy algorithm achieves an approximation ratio of $(1-1/e)$ maximizing monotone submodular functions subject to a *knapsack constraint* $\sum_{i\in S} w_i \leq B$.

Theorem 2.24 (Fisher, Nemhauser, Wolsey 1978)

The greedy algorithm is a $\frac{1}{2}-$approximation algorithm for the problem of maximizing a ≥ 0 monotone submodular function over a matroid.

Theorem 2.25 (Calinescu, Chekuri, Pál, Vondrák 2009)

A *continuous greedy algorithm* used with a technique called *pipage rounding* finds a $(1-1/e)$-approximate solution for the above problem.

Theorem 2.26 (Buchbinder, Feldman, Naor, Schwartz 2014)

A $1/e-$approx. algo for maximizing non-monotone ≥ 0 submodular funct.

Outline

Minimum Edge Coloring

Given: Graph $G = (V, E)$ without parallel edges

Task: Find a coloring $c : E \to \{1, \dots, C\}$ such that no two incident edges get the same color.

Objective: Use a minimum number of colors $C = \chi'(G)$.

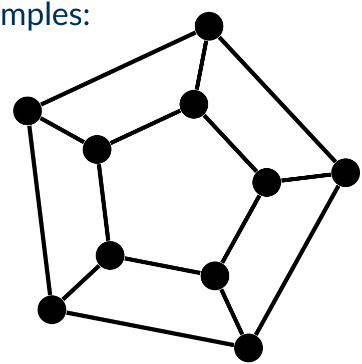
Minimum Edge Coloring

Given: Graph $G = (V, E)$ without parallel edges

Task: Find a coloring $c : E \to \{1, \ldots, C\}$ such that no two incident edges get the same color.

Objective: Use a minimum number of colors $C = \chi'(G)$.

Examples:



$\Delta(G) = 3$

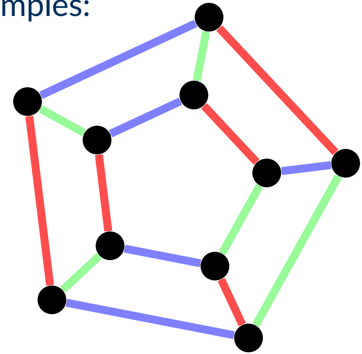
Minimum Edge Coloring

Given: Graph $G = (V, E)$ without parallel edges

Task: Find a coloring $c : E \to \{1, \dots, C\}$ such that no two incident edges get the same color.

Objective: Use a minimum number of colors $C = \chi'(G)$.

Examples:



$\Delta(G) = 3 = \chi'(G)$

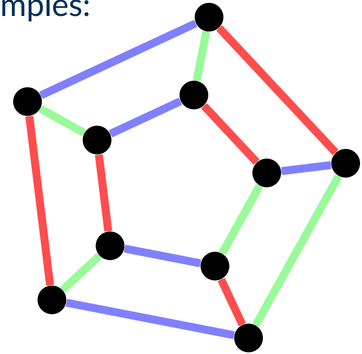
Minimum Edge Coloring

Given: Graph $G = (V, E)$ without parallel edges

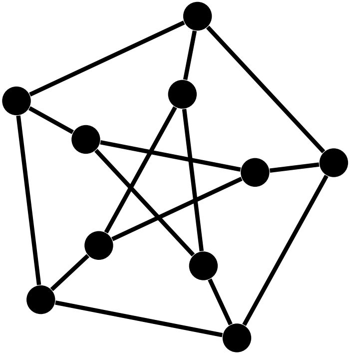
Task: Find a coloring $c : E \rightarrow \{1, \ldots, C\}$ such that no two incident edges get the same color.

Objective: Use a minimum number of colors $C = \chi'(G)$.

Examples:



$\Delta(G) = 3 = \chi'(G)$



Petersen-Graph: $\Delta(G) = 3$

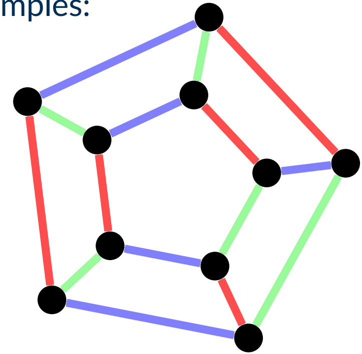
Minimum Edge Coloring

Given: Graph $G = (V, E)$ without parallel edges

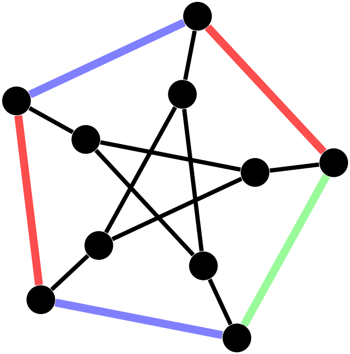
Task: Find a coloring $c : E \to \{1, \ldots, C\}$ such that no two incident edges get the same color.

Objective: Use a minimum number of colors $C = \chi'(G)$.

Examples:



$\Delta(G) = 3 = \chi'(G)$



Petersen-Graph: $\Delta(G) = 3$

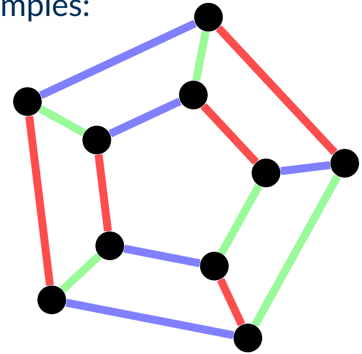
Minimum Edge Coloring

Given: Graph $G = (V, E)$ without parallel edges

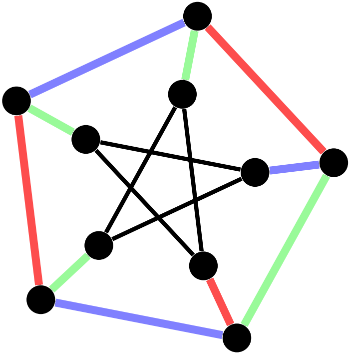
Task: Find a coloring $c : E \to \{1, \dots, C\}$ such that no two incident edges get the same color.

Objective: Use a minimum number of colors $C = \chi'(G)$.

Examples:



$\Delta(G) = 3 = \chi'(G)$



Petersen-Graph: $\Delta(G) = 3$

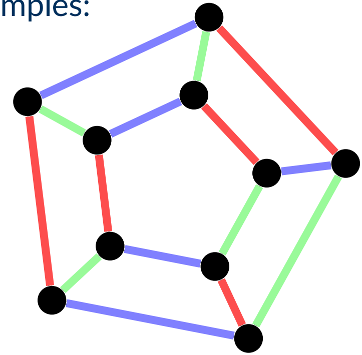
Minimum Edge Coloring

Given: Graph $G = (V, E)$ without parallel edges

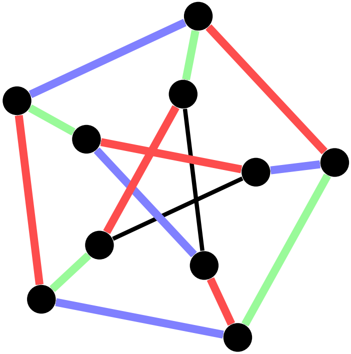
Task: Find a coloring $c : E \rightarrow \{1, \ldots, C\}$ such that no two incident edges get the same color.

Objective: Use a minimum number of colors $C = \chi'(G)$.

Examples:



$\Delta(G) = 3 = \chi'(G)$



Petersen-Graph: $\Delta(G) = 3$

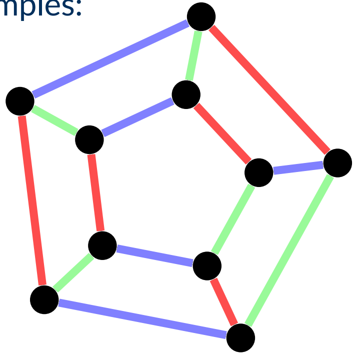
Minimum Edge Coloring

Given: Graph $G = (V, E)$ without parallel edges

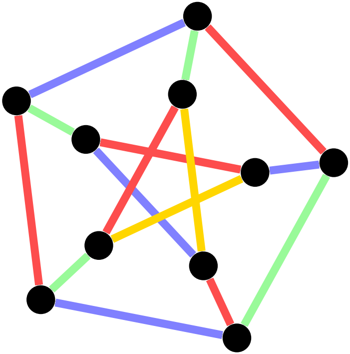
Task: Find a coloring $c : E \to \{1, \ldots, C\}$ such that no two incident edges get the same color.

Objective: Use a minimum number of colors $C = \chi'(G)$.

Examples:



$\Delta(G) = 3 = \chi'(G)$



Petersen-Graph: $\Delta(G) = 3 < 4 = \chi'(G)$

Approximate Minimum Edge Colorings

Observation.

The maximum node degree $\Delta(G)$ is a lower bound on the required number of colors, i.e.,

$$\Delta(G) \leq \chi'(G) \quad .$$

Approximate Minimum Edge Colorings

Observation.

The maximum node degree $\Delta(G)$ is a lower bound on the required number of colors, i.e.,

$$\Delta(G) \le \chi'(G) \quad .$$

Theorem 2.27

For graphs with $\Delta = 3$, it is NP-complete to decide whether the graph is 3-edge-colorable or not. $\square$

Approximate Minimum Edge Colorings

Observation.

The maximum node degree $\Delta(G)$ is a lower bound on the required number of colors, i.e.,

$$\Delta(G) \leq \chi'(G) \ .$$

Theorem 2.27

For graphs with $\Delta = 3$, it is NP-complete to decide whether the graph is 3-edge-colorable or not. $\square$

Theorem 2.28 (Vizing 1964)

There is a polynomial-time algorithm (Vizing's Algorithm) that finds a $(\Delta + 1)$-edge-coloring of a graph. In particular,

$$\chi'(G) \in \{\Delta(G), \Delta(G) + 1\} \ .$$

Outline of Vizing's Algorithm

Input: Undirected graph $G = (V, E)$ without parallel edges.

Output: A $(\Delta(G) + 1)$-edge-coloring.

Main idea: Color one new edge in each iteration;
always maintain a feasible partial $(\Delta + 1)$-edge-coloring.

Outline of Vizing's Algorithm

Input: Undirected graph $G = (V, E)$ without parallel edges.

Output: A $(\Delta(G) + 1)$-edge-coloring.

Main idea: Color one new edge in each iteration;
always maintain a feasible partial $(\Delta + 1)$-edge-coloring.

- Start with an uncolored graph.

Outline of Vizing's Algorithm

Input: Undirected graph $G = (V, E)$ without parallel edges.

Output: A $(\Delta(G) + 1)$-edge-coloring.

Main idea: Color one new edge in each iteration;
always maintain a feasible partial $(\Delta + 1)$-edge-coloring.

- Start with an uncolored graph.
- In every iteration, pick a currently uncolored edge and color it.

Outline of Vizing's Algorithm

Input: Undirected graph $G = (V, E)$ without parallel edges.

Output: A $(\Delta(G) + 1)$-edge-coloring.

Main idea: Color one new edge in each iteration;
always maintain a feasible partial $(\Delta + 1)$-edge-coloring.

- Start with an uncolored graph.
- In every iteration, pick a currently uncolored edge and color it.
- In the process, some other edges might have to be re-colored.

Outline of Vizing's Algorithm

Input: Undirected graph $G = (V, E)$ without parallel edges.

Output: A $(\Delta(G) + 1)$-edge-coloring.

Main idea: Color one new edge in each iteration;
always maintain a feasible partial $(\Delta + 1)$-edge-coloring.

- Start with an uncolored graph.
- In every iteration, pick a currently uncolored edge and color it.
- In the process, some other edges might have to be re-colored.

Useful fact: For any node $v \in V$ there is always a color c that is currently not being used by its incident edges. We say that "v lacks color c."