Algorithms for approximation, parameterization, and data streams

Maria Serna

Spring 2024

We will cover

We will cover

- Optimization/decision/on-line problems

We will cover

- Optimization/decision/on-line problems
- Approximation and Parameterized complexity

We will cover

- Optimization/decision/on-line problems
- Approximation and Parameterized complexity
  - Definition
  - Examples

We will cover

- Optimization/decision/on-line problems
- Approximation and Parameterized complexity
  - Definition
  - Examples
- Some techniques for the design of approximation algorithms
- Linear Programming and approximation

We will cover

- Optimization/decision/on-line problems
- Approximation and Parameterized complexity
    - Definition
    - Examples
- Some techniques for the design of approximation algorithms
- Linear Programming and approximation
- Some techniques for the design of FPT algorithms
- Some algorithms for data streams

We will cover

- Optimization/decision/on-line problems
- Approximation and Parameterized complexity
  - Definition
  - Examples
- Some techniques for the design of approximation algorithms
- Linear Programming and approximation
- Some techniques for the design of FPT algorithms
- Some algorithms for data streams

Not necessarily in this order.

References:Approximation

(S. = Springer)

- Garey, Johnson: Computers and intractability a Theory of the NP-completeness, Freeman, 1979
- Ausiello, et al.: Complexity and Approximation S. 1999
- Vazirani: Approximation Algorithms, S. 2001.
- Willliamsom, Shmoys: The Design of Approximation Algorithms. Cambridge University Press, 2011.

References: Parameterization

(S. = Springer)

- Downey and Fellows: Parameterized Complexity, S. 1999
- Flum and Grohe: Parameterized Complexity Theory, S. 2006.
- Niedermeier: Invitation to Fixed-Parameter Algorithms, Oxford University Press, 2006.
- Cygan, Fomin, Kowalik: Parameterized Algorithms. S. 2015

References: Data Streaming

References: Data Streaming

- Aggarwal, Ed.: Data Streams: Models and Algorithms. S. 2007
- Blum, Hopcropft, Kannan: Foundations of data Science, Cambridge UP, 2020
- Chakrabarti: CS49: Data Stream Algorithms Lecture Notes, Dartmouth College. 2014
- Demetrescu, Finocchi: Data Streams: Models and Algorithms in Handbook of Applied Algorithms: Solving Scientific, Engineering and Practical Problems, John Wiley & Sons, Inc. 2008.

Problems

Problems

Examples:

- Given a graph and two vertices, is there a path joining them?

Problems

Examples:

- Given a graph and two vertices, is there a path joining them?
  decison problem

Problems

Examples:

- Given a graph and two vertices, is there a path joining them? decison problem
- Given a graph and two vertices, obtain a path joining them with minimum length.

Problems

Examples:

- Given a graph and two vertices, is there a path joining them?
  decison problem
- Given a graph and two vertices, obtain a path joining them with minimum length.
  optimization problem

Problems

Examples:

- Given a graph and two vertices, is there a path joining them?
  decison problem
- Given a graph and two vertices, obtain a path joining them with minimum length.
  optimization problem
- Given a set of vertices and two of them, obtain a path joining them with minimum length,

Problems

Examples:

- Given a graph and two vertices, is there a path joining them?
  decison problem
- Given a graph and two vertices, obtain a path joining them with minimum length.
  optimization problem
- Given a set of vertices and two of them, obtain a path joining them with minimum length, as time passes, in

Problems

Examples:

- Given a graph and two vertices, is there a path joining them?
  decison problem
- Given a graph and two vertices, obtain a path joining them with minimum length.
  optimization problem
- Given a set of vertices and two of them, obtain a path joining them with minimum length, as time passes, in the graph

Problems

Examples:

- Given a graph and two vertices, is there a path joining them?
  decison problem
- Given a graph and two vertices, obtain a path joining them with minimum length.
  optimization problem
- Given a set of vertices and two of them, obtain a path joining them with minimum length, as time passes, in the graph discovered by accessing the sequence of edges $e_1, e_2, \ldots$.
  The algorithm needs to answer the question at any time step without knowing the future edges in the graph.

Problems

Examples:

- Given a graph and two vertices, is there a path joining them?
  decison problem
- Given a graph and two vertices, obtain a path joining them with minimum length.
  optimization problem
- Given a set of vertices and two of them, obtain a path joining them with minimum length, as time passes, in the graph discovered by accessing the sequence of edges $e_1, e_2, \ldots$.
  The algorithm needs to answer the question at any time step without knowing the future edges in the graph.
  data stream problem

Decision Problems: Complexity classes

Decision Problems: Complexity classes

- P polynomial time
  there is a polynomial time algorithm providing the correct answer. for any input.

Decision Problems: Complexity classes

- P polynomial time
  there is a polynomial time algorithm providing the correct answer. for any input.
- EXP exponential time optimization
  there is an exponential time algorithm providing the correct answer. for any input.

Decision Problems: Complexity classes

- P polynomial time
  there is a polynomial time algorithm providing the correct answer. for any input.
- EXP exponential time optimization
  there is an exponential time algorithm providing the correct answer. for any input.
- NP non-deterministic polynomial time
  Syntactic definition!

You already known about these classes!

Optimization Problems

Optimization Problems

An optimization problem is a structure $\mathcal{P} = (\mathsf{I}, \mathsf{sol}, \mathsf{m}, \mathsf{goal})$, where

Optimization Problems

An optimization problem is a structure $\mathcal{P} = (\mathrm{I}, \mathrm{sol}, \mathrm{m}, \mathrm{goal})$, where

- I is the input set to $\mathcal{P}$;
- $\mathrm{sol}(x)$ is the set of feasible solutions for an input x.
- The objective function m is an integer (rational) measure defined over pairs (x, y), for $x \in I$ and $y \in \mathrm{sol}(x)$.
- goal is the optimization criterium MAX or MIN.

Optimization Problems

An optimization problem is a structure $\mathcal{P} = (\mathrm{I}, \mathrm{sol}, \mathrm{m}, \mathrm{goal})$, where

- I is the input set to $\mathcal{P}$;
- $\mathrm{sol}(x)$ is the set of feasible solutions for an input x.
- The objective function m is an integer (rational) measure defined over pairs (x, y), for $x \in I$ and $y \in \mathrm{sol}(x)$.
- goal is the optimization criterium MAX or MIN.

That is the function problem whose goal, with respect to an instance x, is to find an optimum solution, that is, a feasible solution y such that

$$y = \mathrm{goal}\{(m(x, y') \mid y' \in \mathrm{sol}(x)\}.$$

Optimization Problems: Complexity classes

Optimization Problems: Complexity classes

- PO polynomial time optimization
  there is a polynomial time algorithm computing an optimal solution for any input.

Optimization Problems: Complexity classes

- PO polynomial time optimization
  there is a polynomial time algorithm computing an optimal solution for any input.
- EXPO exponential time optimization
  there is an exponential time algorithm computing an optimal solution for any input.

Optimization Problems: Complexity classes

- PO polynomial time optimization
  there is a polynomial time algorithm computing an optimal solution for any input.
- EXPO exponential time optimization
  there is an exponential time algorithm computing an optimal solution for any input.
- NPO NP optimization
  Syntactic definition (next slide)

The NPO class

The NPO class

An optimization problem $\mathcal{P} = (\mathrm{I}, \mathrm{sol}, \mathrm{m}, \mathrm{goal})$ belongs to NPO iff

The NPO class

An optimization problem $\mathcal{P} = (\mathsf{I}, \mathsf{sol}, \mathsf{m}, \mathsf{goal})$ belongs to NPO iff

- I is recognizable in polynomial time.

The NPO class

An optimization problem $\mathcal{P} = (\mathsf{I}, \mathsf{sol}, \mathsf{m}, \mathsf{goal})$ belongs to NPO iff

- I is recognizable in polynomial time.
- The feasible solutions are short:
  a polynomial p exists such that, for $y \in \mathsf{sol}(x)$, $|y| \leq p(|x|)$.

The NPO class

An optimization problem $\mathcal{P} = (\mathrm{I}, \mathrm{sol}, \mathrm{m}, \mathrm{goal})$ belongs to NPO iff

- I is recognizable in polynomial time.
- The feasible solutions are short:
  a polynomial p exists such that, for $y \in \mathrm{sol}(x)$, $|y| \leq p(|x|)$.
  Moreover, it is decidable in polynomial time whether
  $y \in \mathrm{sol}(x)$, for x, y with $|y| \leq p(|x|)$,

The NPO class

An optimization problem $\mathcal{P} = (\mathrm{I}, \mathrm{sol}, \mathrm{m}, \mathrm{goal})$ belongs to NPO iff

- I is recognizable in polynomial time.
- The feasible solutions are short:
  a polynomial p exists such that, for $y \in \mathrm{sol}(x)$, $|y| \leq p(|x|)$.
  Moreover, it is decidable in polynomial time whether $y \in \mathrm{sol}(x)$, for x, y with $|y| \leq p(|x|)$,
- The objective function m is computable in polynomial time.

The NPO class: Hardness

The NPO class: Hardness

- The bounded version of an optimization problem is the decision problem

The NPO class: Hardness

- The bounded version of an optimization problem is the decision problem
  - minimization $\mathcal{P} = (\mathrm{I}, \mathrm{sol}, \mathrm{m}, \mathrm{min})$ is

  Given $x \in I$ and an integer k
  Is there a solution $y \in sol(x)$ such that $m(x, y) \leqslant k$?

The NPO class: Hardness

- The bounded version of an optimization problem is the decision problem
  - minimization $\mathcal{P} = (\mathrm{I}, \mathrm{sol}, \mathrm{m}, \mathrm{min})$ is

  Given $x \in I$ and an integer k
  Is there a solution $y \in sol(x)$ such that $m(x, y) \leqslant k$?
  - maximization $\mathcal{P} = (\mathrm{I}, \mathrm{sol}, \mathrm{m}, \mathrm{max})$ is

  Given $x \in I$ and an integer k
  Is there a solution $y \in sol(x)$ such that $m(x, y) \geqslant k$?
- A NPO problem is NP-hard if its bounded version is NP-complete.

The NPO class: Justification

The NPO class: Justification

- The bounded version of a NPO problem belongs to NP.

The NPO class: Justification

- The bounded version of a NPO problem belongs to NP.

  Why?

The NPO class: Justification

- The bounded version of a NPO problem belongs to NP.

  Why?

- For a NPO problem the bounded version and the optimization problem are polynomially equivalent in the following sense:

  There is a polynomial time algorithm for the bounded version iff there is a polynomial time algorithm to compute the cost of an optimal solution

The NPO class: Justification

- The bounded version of a NPO problem belongs to NP.

  Why?

- For a NPO problem the bounded version and the optimization problem are polynomially equivalent in the following sense:

  There is a polynomial time algorithm for the bounded version iff there is a polynomial time algorithm to compute the cost of an optimal solution

  Why?

- Can an NPO problem be NP-complete?

Some NPO problems

MIN-BIN PACKING

Given n objects, object i has volume v_i, $0 \leq v_i \leq 1$, compute the minimum number of unit bins needed to pack all the objects.

MAX-SAT

Given a CNF formula F, compute an assignment that satisfies the maximum number of clauses.

MAX-W-SAT

Given a CNF formula F, in which each clause has an assigned weight. Define the value of an assignment as the sum of the weights of the satisfied clauses. The problem consists in computing an assignment with maximum value.

And the bounded literal per clause families MAX-K-SAT.

Some NPO problems: hardness

Which problems in the previous slide are NP-hard? Why?

How to “solve” NP-hard optimization problems in practice

How to "solve" NP-hard optimization problems in practice

We know that an optimization problem, whose decision version is NP-hard, cannot be solved in polynomial time unless P =NP.

How to "solve" NP-hard optimization problems in practice

We know that an optimization problem, whose decision version is NP-hard, cannot be solved in polynomial time unless P =NP.

What to do?

- Exact algorithms: for instances of small size or for restricted classes of instances.

How to “solve” NP-hard optimization problems in practice

We know that an optimization problem, whose decision version is NP-hard, cannot be solved in polynomial time unless P =NP.

What to do?

- Exact algorithms: for instances of small size or for restricted classes of instances.
- Randomization: algorithms that use random bits, either with high probability of succes or with high probability of poly time.

How to “solve” NP-hard optimization problems in practice

We know that an optimization problem, whose decision version is NP-hard, cannot be solved in polynomial time unless P =NP.
What to do?

- Exact algorithms: for instances of small size or for restricted classes of instances.
- Randomization: algorithms that use random bits, either with high probability of succes or with high probability of poly time.
- Heuristic method: a feasible solution with empirical guarantee.

How to "solve" NP-hard optimization problems in practice

We know that an optimization problem, whose decision version is NP-hard, cannot be solved in polynomial time unless P =NP.
What to do?

- Exact algorithms: for instances of small size or for restricted classes of instances.
- Randomization: algorithms that use random bits, either with high probability of succes or with high probability of poly time.
- Heuristic method: a feasible solution with empirical guarantee.
- Approximation algorithm: a feasible solution with a performance guarantee.

How to "solve" NP-hard optimization problems in practice

We know that an optimization problem, whose decision version is NP-hard, cannot be solved in polynomial time unless P =NP.

What to do?

- Exact algorithms: for instances of small size or for restricted classes of instances.
- Randomization: algorithms that use random bits, either with high probability of succes or with high probability of poly time.
- Heuristic method: a feasible solution with empirical guarantee.
- Approximation algorithm: a feasible solution with a performance guarantee.
- Parameterization: solve efficiently some slices of the problem.

Approximation algorithms

Approximation algorithms

- Let $\mathcal{P}$ be an optimization problem.

Approximation algorithms

- Let $\mathcal{P}$ be an optimization problem.
- For any instance x of $\mathcal{P}$ let opt(x) be the cost of an optimal solution.

Approximation algorithms

- Let $\mathcal{P}$ be an optimization problem.
- For any instance x of $\mathcal{P}$ let opt(x) be the cost of an optimal solution.
- Let $\mathcal{A}$ be an algorithm such that for any instance x of $\mathcal{P}$ computes a feasible solution with cost $\mathcal{A}(x)$.

Approximation algorithms

- Let $\mathcal{P}$ be an optimization problem.
- For any instance x of $\mathcal{P}$ let opt(x) be the cost of an optimal solution.
- Let $\mathcal{A}$ be an algorithm such that for any instance x of $\mathcal{P}$ computes a feasible solution with cost $\mathcal{A}(x)$.

$\mathcal{A}$ is an r-approximation for $\mathcal{P}$ ($r \geq 1$) if for any instance x of $\mathcal{P}$

$$\frac{1}{r} \leq \frac{\text{opt}(x)}{\mathcal{A}(x)} \leq r$$

Approximation algorithms

- Let $\mathcal{P}$ be an optimization problem.
- For any instance x of $\mathcal{P}$ let opt(x) be the cost of an optimal solution.
- Let $\mathcal{A}$ be an algorithm such that for any instance x of $\mathcal{P}$ computes a feasible solution with cost $\mathcal{A}(x)$.

$\mathcal{A}$ is an r-approximation for $\mathcal{P}$ ($r \geq 1$) if for any instance x of $\mathcal{P}$

$$\frac{1}{r} \leq \frac{\text{opt}(x)}{\mathcal{A}(x)} \leq r$$

$\mathcal{P}$ is r-approximable in polynomial time if there is a polynomial time computable r-approximation for $\mathcal{P}$.

Be sure about r

$\mathcal{A}$ is an r-approximation for $\mathcal{P}$ ($r \geq 1$) if for any instance x of $\mathcal{P}$

$$\frac{1}{r} \leq \frac{\mathrm{opt}(x)}{\mathcal{A}(x)} \leq r$$

- Why $r \geq 1$?

Be sure about r

$\mathcal{A}$ is an r-approximation for $\mathcal{P}$ ($r \geq 1$) if for any instance x of $\mathcal{P}$

$$\frac{1}{r} \leq \frac{\text{opt}(x)}{\mathcal{A}(x)} \leq r$$

- Why $r \geq 1$?
- In which cases we can have $r = 1$?

Be sure about r

$\mathcal{A}$ is an r-approximation for $\mathcal{P}$ ($r \geq 1$) if for any instance x of $\mathcal{P}$

$$\frac{1}{r} \leq \frac{\mathrm{opt}(x)}{\mathcal{A}(x)} \leq r$$

- Why $r \geq 1$?
- In which cases we can have $r = 1$?
- How would you like r to be?

Be sure about r

$\mathcal{A}$ is an r-approximation for $\mathcal{P}$ ($r \geq 1$) if for any instance x of $\mathcal{P}$

$$\frac{1}{r} \leq \frac{\mathrm{opt}(x)}{\mathcal{A}(x)} \leq r$$

- Why $r \geq 1$?
- In which cases we can have $r = 1$?
- How would you like r to be?
- Is there any trivial condition about r, for maximization problems? for minimization ones?

NPO: approximation classes

NPO: approximation classes

Classification of NPO problems as approximable within a constant r in polynomial time:

NPO: approximation classes

Classification of NPO problems as approximable within a constant r in polynomial time:

APX exists r.

NPO: approximation classes

Classification of NPO problems as approximable
within a constant r in polynomial time:

APX exists r.

PTAS Polynomial Time Approximation Schema
for any r, but time may depend exponentially in $1/(r-1)$.

NPO: approximation classes

Classification of NPO problems as approximable
within a constant r in polynomial time:

- **APX** exists r.
- **PTAS** Polynomial Time Approximation Schema
  for any r, but time may depend exponentially in $1/(r-1)$.
- **FPTAS** Fully Polynomial Time Approximation Schema
  for any r in time polynomial in the input size and $1/(r-1)$.

NPO: approximation classes

Classification of NPO problems as approximable
within a constant r in polynomial time:

- **APX** exists r.
- **PTAS** Polynomial Time Approximation Schema
  for any r, but time may depend exponentially in $1/(r-1)$.
- **FPTAS** Fully Polynomial Time Approximation Schema
  for any r in time polynomial in the input size and $1/(r-1)$.

Further classification can be obtained by considering non-constant r, for example $\log n$ or $\log \log n$ ....

NPO: approximation classes

Classification of NPO problems as approximable
within a constant r in polynomial time:

- **APX** exists r.
- **PTAS** Polynomial Time Approximation Schema
  for any r, but time may depend exponentially in $1/(r-1)$.
- **FPTAS** Fully Polynomial Time Approximation Schema
  for any r in time polynomial in the input size and $1/(r-1)$.

Further classification can be obtained by considering non-constant r, for example $\log n$ or $\log \log n$ . . . .

Negative results through hardness APX-hard etc.

Hard to approximate problems

Hardness levels:

Hard to approximate problems

Hardness levels:

APX-hard:

Hard to approximate problems

Hardness levels:

APX-hard: unless P =NP no PTAS

Hard to approximate problems

Hardness levels:

APX-hard: unless P =NP no PTAS
unless P =NP no constant approximation

Hard to approximate problems

Hardness levels:

APX-hard: unless P =NP no PTAS
unless P =NP no constant approximation
⋮
Non-approximable

Hard to approximate problems

Hardness levels:

APX-hard: unless P =NP no PTAS

unless P =NP no constant approximation

⋮

Non-approximable

unless P =NP, for any r at most a polynomial function of n, there is no polynomial time r-approximation algorithm

First approximation algorithms

- We start analyzing approximation algorithms based in the greedy technique.
- One algorithm for MIN-BIN PACKING
- And two algorithms for a load-balancing problem.

Nex Fit: An approximation algorithm for BinPacking

MIN-BIN PACKING

Given n objects, object i has volume v_i, $0 \leq v_i \leq 1$, compute the minimum number of unit bins needed to pack all the objects.

Nex Fit: An approximation algorithm for BinPacking

Nex Fit: An approximation algorithm for BinPacking

Let us assume that the bins are labeled $B_1, B_2, \ldots, B_n$ and that initially all are closed and during the execution of the algorithm only one bin will be open.

Nex Fit: An approximation algorithm for BinPacking

Let us assume that the bins are labeled $B_1, B_2, \ldots, B_n$ and that initially all are closed and during the execution of the algorithm only one bin will be open.

Next Fit places the n objects, one after the other, as follows:

- Opens B_1 and places the first object in B_1.
- If the i-th object fits in the open box, we put it inside. Otherwise, we close the bin, open the next one and place the object in it.

Nex Fit: An approximation algorithm for BinPacking

Let us assume that the bins are labeled $B_1, B_2, \ldots, B_n$ and that initially all are closed and during the execution of the algorithm only one bin will be open.

Next Fit places the n objects, one after the other, as follows:

- Opens B_1 and places the first object in B_1.
- If the i-th object fits in the open box, we put it inside. Otherwise, we close the bin, open the next one and place the object in it.

For the case of $v_1 = 0.3$, $v_2 = 0.8$, and $v_3 = 0.7$, Next Fit solution needs three bins. But there is a solution with uses only two bins. Not optimal!

Nex Fit: An approximation algorithm for BinPacking

Theorem

Let x be an input to the MIN-BIN PACKING *problem and let $opt(x)$ be the minimum number of bins needed to pack de objects in x. If $NF(x)$ is the number of bins in the solution computed by Next Fit, then $opt(x) \leq NF(x) \leq 2opt(x)$.*

Nex Fit: An approximation algorithm for BinPacking

Theorem

Let $x = (v_1, \dots, v_n$ be an input to the MIN-BIN PACKING. *Let $NF(x)$ be the number of bins in the solution computed by Next Fit, we have $opt(x) \leq NF(x) \leq 2opt(x)$.*

Proof.

The first inequality is always true:

- MIN-BIN PACKING is a minimization problem
- Next Fit provides a feasible solution

Nex Fit: An approximation algorithm for BinPacking

Theorem

Let $x = (v_1, \dots, v_n$ be an input to the MIN-BIN PACKING. *Let $NF(x)$ be the number of bins in the solution computed by Next Fit, we have $opt(x) \leq NF(x) \leq 2opt(x)$.*

Proof.

The first inequality is always true:

- MIN-BIN PACKING is a minimization problem
- Next Fit provides a feasible solution

Let $V = \sum_{i=1}^{n} v_i$, we have $\text{opt}(x) \geq \lceil V \rceil$.

Nex Fit: An approximation algorithm for BinPacking

Proof.

Let us look to two consecutive bins in the Next Fit solution

Nex Fit: An approximation algorithm for BinPacking

Proof.

Let us look to two consecutive bins in the Next Fit solution The total packet size in the two bins must be bigger than 1, otherwise we will never have opened the second bin. So,

$$\mathrm{NF}(x) \leq 2\lceil V \rceil.$$

Nex Fit: An approximation algorithm for BinPacking

Proof.

Let us look to two consecutive bins in the Next Fit solution The total packet size in the two bins must be bigger than 1, otherwise we will never have opened the second bin. So,

$$\mathrm{NF}(x) \leq 2\lceil V\rceil.$$

But we have seen $\mathrm{opt}(x) \geq \lceil V\rceil$, so

Nex Fit: An approximation algorithm for BinPacking

Proof.

Let us look to two consecutive bins in the Next Fit solution The total packet size in the two bins must be bigger than 1, otherwise we will never have opened the second bin. So,

$$\mathrm{NF}(x) \leq 2\lceil V \rceil.$$

But we have seen $\mathrm{opt}(x) \geq \lceil V \rceil$, so

$$\mathrm{NF}(x) \leq 2\lceil V \rceil \leq 2\mathrm{opt}(x).$$

☺

Load Balancing problem

Processing scenario

- We have m identical machines;
  n jobs, job j has processing time t_j
- Job j must run contiguously on one machine.
- A machine can process at most one job at a time.
- We want to assign jobs to machines optimizing the makespan.

Processing scenario

- We have m identical machines;
  n jobs, job j has processing time t_j
- Job j must run contiguously on one machine.
- A machine can process at most one job at a time.
- We want to assign jobs to machines optimizing the makespan.

- Let $J(i)$ be the subset of jobs assigned to machine i.
- The load of machine i is $L_i = \sum_{j \in J(i)} t_j$
- The makespan is the maximum load on any machine, $L = \max_i L_i$.

Load Balancing problem

LBAL

Given n jobs, job j has processing time t_j, assign jobs to m identical machines as to minimize the makespan.

Load Balancing problem

LBAL

Given n jobs, job j has processing time t_j, assign jobs to m identical machines as to minimize the makespan.

- The problem is NP-hard and belongs to NPO.
  Load balancing is hard even if $m = 2$ machines (reduction from Partition).

Load Balancing problem

LBAL

Given n jobs, job j has processing time t_j, assign jobs to m identical machines as to minimize the makespan.

- The problem is NP-hard and belongs to NPO.
  Load balancing is hard even if $m = 2$ machines (reduction from Partition).
- The approximation algorithm we propose is a greedy algorithm called list-scheduling.

List scheduling

List scheduling

For $j = 1, \dots, n$:

Assign job j to the machine having smallest load so far.

List scheduling: Implementation

function LIST SCHEDULING(m, n, T)
  for $i = 1, dots, m$ **do**
    $L[i] = 0$ — load on machine
    $J[i] = \emptyset$ — jobs assigned to
  end for
  for $j = 1, \ldots, n$ **do**
    $i = \text{argmin}_k L_k$ — machine with smallest load
    $J[i] = J[i] \cup \{j\}$
    $L[i] = L[i] + T[j]$
  end for
end function

List scheduling: Implementation

function LIST SCHEDULING(m, n, T)
  for $i = 1, dots, m$ **do**
    $L[i] = 0$ — load on machine
    $J[i] = \emptyset$ — jobs assigned to
  end for
  for $j = 1, \ldots, n$ **do**
    $i = \text{argmin}_k L_k$ — machine with smallest load
    $J[i] = J[i] \cup \{j\}$
    $L[i] = L[i] + T[j]$
  end for
end function

Cost:

List scheduling: Implementation

function LIST SCHEDULING(m, n, T)
  for $i = 1, dots, m$ **do**
    $L[i] = 0$ load on machine
    $J[i] = \emptyset$ jobs assigned to
  end for
  for $j = 1, \dots, n$ **do**
    $i = \text{argmin}_k L_k$ machine with smallest load
    $J[i] = J[i] \cup \{j\}$
    $L[i] = L[i] + T[j]$
  end for
end function

Cost: Using a priority queue to maintain L, the cost is $O(n \log m)$

List scheduling: Approximation rate

Theorem

LIST SCHEDULING *is a polynomial 2-approximation algorithm for* LBAL.

List scheduling: Approximation rate

Theorem

LIST SCHEDULING *is a polynomial 2-approximation algorithm for* LBAL.

Proof.

- Let L^* be the optimum makespan and L the makespan of the solution computed by list scheduling.

List scheduling: Approximation rate

Theorem

LIST SCHEDULING *is a polynomial 2-approximation algorithm for* LBAL.

Proof.

- Let L^* be the optimum makespan and L the makespan of the solution computed by list scheduling.
- $L^* \geq \max_j t_j$ and $L^* \geq \frac{1}{m} \sum_j t_j$.

List scheduling: Approximation rate

Theorem

LIST SCHEDULING *is a polynomial 2-approximation algorithm for* LBAL*.*

Proof.

- Let L^* be the optimum makespan and L the makespan of the solution computed by list scheduling.
- $L^* \geq \max_j t_j$ and $L^* \geq \frac{1}{m}\sum_j t_j$.
- Assume that $L = L_i$. Let j be the last job scheduled in machine i.

List scheduling: Approximation rate

Theorem

LIST SCHEDULING *is a polynomial 2-approximation algorithm for* LBAL.

Proof.

- Let L^* be the optimum makespan and L the makespan of the solution computed by list scheduling.
- $L^* \geq \max_j t_j$ and $L^* \geq \frac{1}{m} \sum_j t_j$.
- Assume that $L = L_i$. Let j be the last job scheduled in machine i.
- When job j was assigned, all the other machines have higher load, so $L_i - t_j \leq L_k$, for $k \neq i$.

List scheduling: Approximation rate

Theorem

LIST SCHEDULING *is a polynomial 2-approximation algorithm for* LBAL.

Proof.

- When job j was assigned, all the other machines have higher load, so $L_i - t_j \leq L_k$, for $1 \leq k \leq m$.

List scheduling: Approximation rate

Theorem

LIST SCHEDULING *is a polynomial 2-approximation algorithm for* LBAL.

Proof.

- When job j was assigned, all the other machines have higher load, so $L_i - t_j \leq L_k$, for $1 \leq k \leq m$. Summing up for all k and dividing by m we get

List scheduling: Approximation rate

Theorem

LIST SCHEDULING *is a polynomial 2-approximation algorithm for* LBAL.

Proof.

- When job j was assigned, all the other machines have higher load, so $L_i - t_j \leq L_k$, for $1 \leq k \leq m$. Summing up for all k and dividing by m we get
- $L_i - t_j \leq \frac{1}{m} \sum_k L_k \leq \frac{1}{m} \sum_j t_j \leq L^*$

List scheduling: Approximation rate

Theorem

LIST SCHEDULING *is a polynomial 2-approximation algorithm for* LBAL.

Proof.

- When job j was assigned, all the other machines have higher load, so $L_i - t_j \leq L_k$, for $1 \leq k \leq m$. Summing up for all k and dividing by m we get
- $L_i - t_j \leq \frac{1}{m} \sum_k L_k \leq \frac{1}{m} \sum_j t_j \leq L^*$
- We have, $L = L_i = (L_i - t_j) + t_j \leq L^* + L^* = 2L^*$

List scheduling: Approximation rate

Theorem

LIST SCHEDULING *is a polynomial 2-approximation algorithm for* LBAL.

Proof.

- When job j was assigned, all the other machines have higher load, so $L_i - t_j \leq L_k$, for $1 \leq k \leq m$. Summing up for all k and dividing by m we get
- $L_i - t_j \leq \frac{1}{m} \sum_k L_k \leq \frac{1}{m} \sum_j t_j \leq L^*$
- We have, $L = L_i = (L_i - t_j) + t_j \leq L^* + L^* = 2L^*$

☺

Approximation rate: tightness

- When we design an approximation algorithm, we wish to approach the best possible approximation ratio.
- Which one is the best for a problem, as usual, requires some complexity consideration and, in some cases, we have answers like
  "this problem cannot be approximated for $r \leq \ldots$
  unless P $=$NP"

Approximation algorithm: tightness

- We can ask a similar tightness question, not for the optimization problem, but about the approximation algorithm at hand.
- In this case the question is about the tightness in the analysis of the approximation ratio. The value of r is correct or can it be reduced further?
- We can show the tightness in the analysis of r by finding an input x so that the rate between $\text{opt}(x)$ and $\mathcal{A}(x)$ rules out any improvement on r.
- The previous step involves computing the optimal solution for a particularly adequate input, not solving the optimization problem.

List scheduling: Tightness?

List scheduling: Tightness?

- m machines, $m(m-1)$ length 1 jobs and 1 job of length m.

List scheduling: Tightness?

- m machines, $m(m-1)$ length 1 jobs and 1 job of length m.
- List scheduling assigns the first m length 1 jobs, to different machines, and

List scheduling: Tightness?

- m machines, $m(m-1)$ length 1 jobs and 1 job of length m.
- List scheduling assigns the first m length 1 jobs, to different machines, and
- after scheduling the unit length jobs, all machines have load $L_i = m - 1$.

List scheduling: Tightness?

- m machines, $m(m-1)$ length 1 jobs and 1 job of length m.
- List scheduling assigns the first m length 1 jobs, to different machines, and
- after scheduling the unit length jobs, all machines have load $L_i = m - 1$.
- The last job is assigned to one machine, giving a makespan $L = m + m - 1 = 2m - 1$.

List scheduling: Tightness?

- m machines, $m(m-1)$ length 1 jobs and 1 job of length m.
- List scheduling assigns the first m length 1 jobs, to different machines, and
- after scheduling the unit length jobs, all machines have load $L_i = m - 1$.
- The last job is assigned to one machine, giving a makespan $L = m + m - 1 = 2m - 1$.
- An optimal solution assigns the big job to one machine and m unit jobs to the other machines, so $L^* = m$.

List scheduling: Tightness?

- m machines, $m(m-1)$ length 1 jobs and 1 job of length m.
- List scheduling assigns the first m length 1 jobs, to different machines, and
- after scheduling the unit length jobs, all machines have load $L_i = m - 1$.
- The last job is assigned to one machine, giving a makespan $L = m + m - 1 = 2m - 1$.
- An optimal solution assigns the big job to one machine and m unit jobs to the other machines, so $L^* = m$.
- The approximation rate is tight ☺

Longest processing first

Longest processing first

- Analizamos una variante,

Longest processing first

- Analizamos una variante,

  LONGEST PROCESSING FIRST:

Longest processing first

- Analizamos una variante,

  LONGEST PROCESSING FIRST:

  Sort jobs in decreasing order of processing time.

Longest processing first

- Analizamos una variante,

  Longest processing first:

  Sort jobs in decreasing order of processing time.

  Run list scheduling.

Longest processing first: Approximation rate

Theorem

LONGEST PROCESSING FIRST *is a polynomial 3/2-approximation algorithm for* LBAL.

Longest processing first: Approximation rate

Theorem

Longest processing first is a polynomial 3/2-approximation algorithm for LBAL.

Proof.

Longest processing first: Approximation rate

Theorem

Longest processing first is a polynomial 3/2-approximation algorithm for LBAL.

Proof.

- If $n \leq m$, $L = t_1 = L^* \leq 3/2L^*$

Longest processing first: Approximation rate

Theorem

LONGEST PROCESSING FIRST is a polynomial 3/2-approximation algorithm for LBAL.

Proof.

- If $n \leq m$, $L = t_1 = L^* \leq 3/2L^*$
- If $n > m$, since there are more jobs than machines, a machine must take two jobs in $(t_1, \ldots, t_{m+1})$.

Longest processing first: Approximation rate

Theorem

LONGEST PROCESSING FIRST *is a polynomial 3/2-approximation algorithm for* LBAL.

Proof.

- If $n \leq m$, $L = t_1 = L^* \leq 3/2L^*$
- If $n > m$, since there are more jobs than machines, a machine must take two jobs in $(t_1, \ldots, t_{m+1})$.
- Therefore, $L^* \geq 2t_{m+1} \geq 2t_j$, being j the last job assigned to the machine i that gives the makespan.

Longest processing first: Approximation rate

Theorem

LONGEST PROCESSING FIRST *is a polynomial 3/2-approximation algorithm for* LBAL.

Proof.

- If $n \leq m$, $L = t_1 = L^* \leq 3/2L^*$
- If $n > m$, since there are more jobs than machines, a machine must take two jobs in $(t_1, \ldots, t_{m+1})$.
- Therefore, $L^* \geq 2t_{m+1} \geq 2t_j$, being j the last job assigned to the machine i that gives the makespan.
- So, $L_i = (L_i - t_j) + t_j \leq L^* + \frac{1}{2}L^* \leq \frac{3}{2}L^*$.

Longest processing first: Approximation rate

Theorem

LONGEST PROCESSING FIRST *is a polynomial 3/2-approximation algorithm for* LBAL.

Proof.

- If $n \leq m$, $L = t_1 = L^* \leq 3/2L^*$
- If $n > m$, since there are more jobs than machines, a machine must take two jobs in $(t_1, \dots, t_{m+1})$.
- Therefore, $L^* \geq 2t_{m+1} \geq 2t_j$, being j the last job assigned to the machine i that gives the makespan.
- So, $L_i = (L_i - t_j) + t_j \leq L^* + \frac{1}{2}L^* \leq \frac{3}{2}L^*$.

☺

Longest processing first: Tightness?

- The 3/2 bound on the approximation rate is not tight
- In fact LONGEST PROCESSING FIRST is a 4/3-approximation algorithm [Graham 1969]
- 4/3 is tight:

Longest processing first: Tightness?

- The $3/2$ bound on the approximation rate is not tight
- In fact LONGEST PROCESSING FIRST is a $4/3$-approximation algorithm [Graham 1969]
- $4/3$ is tight:
  - m machines
  - $n = 2m + 1$ jobs
  - 2 jobs of length $m, m+1, \ldots, 2m-1$ and one more job of lenght m.
  - $L^* = 3m$ and $L = 4m - 1$, which gives a ratio tending to $4/3$.

Longest processing first: Tightness?

- The 3/2 bound on the approximation rate is not tight
- In fact LONGEST PROCESSING FIRST is a 4/3-approximation algorithm [Graham 1969]
- 4/3 is tight:
  - m machines
  - $n = 2m + 1$ jobs
  - 2 jobs of length $m, m+1, \ldots, 2m-1$ and one more job of lenght m.
  - $L^* = 3m$ and $L = 4m - 1$, which gives a ratio tending to 4/3.
  - 4/3 is tight ☺

Max Cut

MAX-CUT

Given a graph $G = (V, A)$ we want to find a partition of V into V_1, V_2 in such a way that

$$\mathrm{cut}(V_1, V_2) = \|\{(u, v) \mid u \in V_1\ v \in V_2\}\|$$

is maximum.

Max Cut

MAX-CUT

Given a graph $G = (V, A)$ we want to find a partition of V into V_1, V_2 in such a way that

$$\text{cut}(V_1, V_2) = \|\{(u, v) \mid u \in V_1\ v \in V_2\}\|$$

is maximum.

- The problem is NP-hard and belongs to NPO.

Max Cut

MAX-CUT

Given a graph $G = (V, A)$ we want to find a partition of V into V_1, V_2 in such a way that

$$\text{cut}(V_1, V_2) = \|\{(u, v) \mid u \in V_1 \; v \in V_2\}\|$$

is maximum.

- The problem is NP-hard and belongs to NPO.
- Let us analyze a local search algorithm using the HillClimbing paradigm.

Local search

Local search

- A neighborhood structure is defined on the set of solutions.
- The algorithm performs an exploration of the neighborhood graph.
- Hill Climbing: It starts at one feasible solution and moves to a better one. It finishes at a local optimum, when no neighbor improves the value of m.
- Many heuristics are local search algorithms performing some kind of random exploration on the neighborhood. Te result of such an exploration is the best seen solution.

Local search

- A neighborhood structure is defined on the set of solutions.
- The algorithm performs an exploration of the neighborhood graph.
- Hill Climbing: It starts at one feasible solution and moves to a better one. It finishes at a local optimum, when no neighbor improves the value of m.
- Many heuristics are local search algorithms performing some kind of random exploration on the neighborhood. Te result of such an exploration is the best seen solution.

A local optimum is a solution such that all its neighbors have equal or worse cost.

Neighborhood for MaxCut

Neighborhood for MaxCut

Given a graph $G = (V, E)$ define $\mathcal{N}(G)$:

Neighborhood for MaxCut

Given a graph $G = (V, E)$ define $\mathcal{N}(G)$:

> *The neighbors of a solution* (V_1, V_2) *are all those partitions that can be obtained by moving either one element from* V_1 *to* V_2 *or one element from* V_2 *to* V_1*.*

Neighborhood for MaxCut

Given a graph $G = (V, E)$ define $\mathcal{N}(G)$:

> *The neighbors of a solution (V_1, V_2) are all those partitions that can be obtained by moving either one element from V_1 to V_2 or one element from V_2 to V_1.*

Using $\mathcal{N}(G)$ we consider the following algorithm:

Neighborhood for MaxCut

Given a graph $G = (V, E)$ define $\mathcal{N}(G)$:

The neighbors of a solution (V_1, V_2) are all those partitions that can be obtained by moving either one element from V_1 to V_2 or one element from V_2 to V_1.

Using $\mathcal{N}(G)$ we consider the following algorithm:

HillClimbing Max-Cut (G:graph, n: integer)
V_1, V_2: set of $[1 \ldots n]$;
$V_1 := \varnothing$; $V_2 :=$ V(G);
while not local-optimum(V_1, V_2) do
$(V_1, V2)$:= a neighboring partition of (V_1, V_2) with improved cost

HillClimbing MaxCut

Observe that the algorithm needs only polynomial time as

HillClimbing MaxCut

Observe that the algorithm needs only polynomial time as

- we can compute $\mathrm{cut}(X, Y)$ in $O(|\,\mathrm{E}(G)|)$ steps.

HillClimbing MaxCut

Observe that the algorithm needs only polynomial time as

- we can compute cut(X, Y) in $O(|E(G)|)$ steps.
  In fact you can recompute faster from the value of a neighbor.

HillClimbing MaxCut

Observe that the algorithm needs only polynomial time as

- we can compute cut(X, Y) in $O(|\mathrm{E}(G)|)$ steps.
  In fact you can recompute faster from the value of a neighbor.
- The number of neighbors of a solution is n.

HillClimbing MaxCut

Observe that the algorithm needs only polynomial time as

- we can compute cut(X, Y) in $O(|\mathrm{E}(G)|)$ steps.
  In fact you can recompute faster from the value of a neighbor.
- The number of neighbors of a solution is n.
- The cost of the initial solution is 0.

HillClimbing MaxCut

Observe that the algorithm needs only polynomial time as

- we can compute cut(X, Y) in $O(|\mathrm{E}(G)|)$ steps.
  In fact you can recompute faster from the value of a neighbor.
- The number of neighbors of a solution is n.
- The cost of the initial solution is 0.
- The maximum partition cut is upper bounded by $|\mathrm{E}(G)|$.

HillClimbing MaxCut

Observe that the algorithm needs only polynomial time as

- we can compute $\text{cut}(X, Y)$ in $O(|\text{E}(G)|)$ steps.
  In fact you can recompute faster from the value of a neighbor.
- The number of neighbors of a solution is n.
- The cost of the initial solution is 0.
- The maximum partition cut is upper bounded by $|\text{E}(G)|$.
- At each step the cut is increased in one unit.

HillClimbing MaxCut

Lemma

Let $G = (V, A)$ be a graph, if (V_1, V_2) is a local optimum of $\mathcal{N}(G)$ then $cut((V_1, V_2)) \leq 2opt(G)$.

HillClimbing MaxCut

Lemma

Let $G = (V, A)$ be a graph, if (V_1, V_2) is a local optimum of $\mathcal{N}(G)$ then $cut((V_1, V_2)) \leq 2opt(G)$.

Theorem

HILLCLIMBING MAX-CUT *is a polynomial 2-approximation algorithm for* MAX-CUT.

The class PLS

- Polynomial Local Search (PLS) is a complexity class that models the difficulty of finding a locally optimal solution to an optimization problem.

The class PLS

- Polynomial Local Search (PLS) is a complexity class that models the difficulty of finding a locally optimal solution to an optimization problem.
- A Local search problem (*LSP*) is an optimization problem together with a neighborhood defined on the set of solutions.

The class PLS

- Polynomial Local Search (PLS) is a complexity class that models the difficulty of finding a locally optimal solution to an optimization problem.
- A Local search problem (*LSP*) is an optimization problem together with a neighborhood defined on the set of solutions.
- A *LSP* problem $\mathcal{P} = (\mathrm{I}, \mathrm{sol}, \mathrm{m}, \mathrm{goal}, \mathcal{N})$ belongs to PLS if

The class PLS

- Polynomial Local Search (PLS) is a complexity class that models the difficulty of finding a locally optimal solution to an optimization problem.
- A Local search problem (*LSP*) is an optimization problem together with a neighborhood defined on the set of solutions.
- A *LSP* problem $\mathcal{P} = (\mathrm{I}, \mathrm{sol}, \mathrm{m}, \mathrm{goal}, \mathcal{N})$ belongs to PLS if
  - $(\mathrm{I}, \mathrm{sol}, \mathrm{m}, \mathrm{goal}) \in \mathrm{NPO}$.
  - Given $x \in \mathrm{I}$, a $y \in \mathrm{sol}$ can be computed in polynomial time.
  - Given $y \in \mathrm{sol}(x)$, there is a polynomial (in $|x|$) algorithm that decides whether y is a local optimum, for m and, if not, outputs a neighbor of y with better cost.

The class PLS

- Polynomial Local Search (PLS) is a complexity class that models the difficulty of finding a locally optimal solution to an optimization problem.
- A Local search problem (LSP) is an optimization problem together with a neighborhood defined on the set of solutions.
- A LSP problem $\mathcal{P} = (\mathrm{I}, \mathrm{sol}, \mathrm{m}, \mathrm{goal}, \mathcal{N})$ belongs to PLS if
  - $(\mathrm{I}, \mathrm{sol}, \mathrm{m}, \mathrm{goal}) \in \mathrm{NPO}$.
  - Given $x \in \mathrm{I}$, a $y \in \mathrm{sol}$ can be computed in polynomial time.
  - Given $y \in \mathrm{sol}(x)$, there is a polynomial (in $|x|$) algorithm that decides whether y is a local optimum, for m and, if not, outputs a neighbor of y with better cost.
- PLS problems always have a solution!

The class PLS

- The conditions guarantee that a navigation step can be performed in polynomial time.
- The size of the solution set do not guarantee that an exploration will end within polynomial time.
- However the computation uses only polynomial space.
- PLS was introduced in
  David S. Johnson, Christos H. Papadimitriou, and Mihalis Yannakakis. "How easy is local search?" In: Journal of computer and system sciences 37.1 (1988), pp. 79–100.
- With associated notions of PLS-reductions and PLS-complete problems.

Approximation Schema

- An approximation scheme is an algorithm $\mathcal{A}$ that takes as input an instance of an optimization problem and a parameter $r \geq 1$ and outputs a solution with cost within r of the optimal solution.

Approximation Schema

- An approximation scheme is an algorithm $\mathcal{A}$ that takes as input an instance of an optimization problem and a parameter $r \geq 1$ and outputs a solution with cost within r of the optimal solution.
- For any r, $\mathcal{A}(x, r)$ is an r-approximation algorithm.

Approximation Schema

- An approximation scheme is an algorithm $\mathcal{A}$ that takes as input an instance of an optimization problem and a parameter $r \geq 1$ and outputs a solution with cost within r of the optimal solution.
- For any r, $\mathcal{A}(x, r)$ is an r-approximation algorithm.
- However, for an NP-hard NPO problem, the time performed by the algorithm should increase as r approaches to 1.

PTAS and FPTAS

An optimization problem belongs to

- Polynomial Time Approximation Scheme (PTAS) if it has an approximation scheme $\mathcal{A}$ that takes time polynomial in $|x|$ independently of the dependency on $\frac{1}{r-1}$.

PTAS and FPTAS

An optimization problem belongs to

- Polynomial Time Approximation Scheme (PTAS) if it has an approximation scheme $\mathcal{A}$ that takes time polynomial in $|x|$ independently of the dependency on $\frac{1}{r-1}$.
  This insures, polynomial time for any constant r.
- Fully Polynomial Time Approximation Scheme (FPTAS) if it has an approximation scheme $\mathcal{A}$ that takes time polynomial in both $p(|x|)$ and $\frac{1}{r-1}$.

PTAS and FPTAS

An optimization problem belongs to

- Polynomial Time Approximation Scheme (PTAS) if it has an approximation scheme $\mathcal{A}$ that takes time polynomial in $|x|$ independently of the dependency on $\frac{1}{r-1}$.
  This insures, polynomial time for any constant r.
- Fully Polynomial Time Approximation Scheme (FPTAS) if it has an approximation scheme $\mathcal{A}$ that takes time polynomial in both $p(|x|)$ and $\frac{1}{r-1}$.
  This insures, polynomial time algorithms even for values of r that are not constant.

PTAS and FPTAS

An optimization problem belongs to

- Polynomial Time Approximation Scheme (PTAS) if it has an approximation scheme $\mathcal{A}$ that takes time polynomial in $|x|$ independently of the dependency on $\frac{1}{r-1}$.
  This insures, polynomial time for any constant r.
- Fully Polynomial Time Approximation Scheme (FPTAS) if it has an approximation scheme $\mathcal{A}$ that takes time polynomial in both $p(|x|)$ and $\frac{1}{r-1}$.
  This insures, polynomial time algorithms even for values of r that are not constant.
- Usually there is no distinction in the name of the complexity class

A problem in FPTAS: 0-1 Knapsack

0-1 KNAPSACK

Given an integer b and a set of n objects, object i has weight w_i and value v_i, compute a selection of objects with total size less than or equal to b and maximum profit.

A problem in FPTAS: 0-1 Knapsack

0-1 KNAPSACK

Given an integer b and a set of n objects, object i has weight w_i and value v_i, compute a selection of objects with total size less than or equal to b and maximum profit.

The problem is NP-hard and belongs to NPO. There is a dynamic programming algorithm that solves 0-1 KNAPSACK in time

$$\sim n \sum_{i=1}^{n} v_i.$$

The algorithm is polynomial for $poly(n)$ values.

Consider the following SCALEDOWN algorithm which has r as input:

SCALEDOWN(w, v, b, r)

$v_{\max} := \max v_i$;

$t := \lfloor \log[\frac{r-1}{r} \frac{v_{\max}}{n}] \rfloor$;

$z :=$ instance obtained by changing profits to $v_i' = v_i/2^t$;

$y :=$ optimal solution for z;

return y

Is SCALEDOWN a polynomial time approximation schema? time? rate of approximation?

Time

Time

The most difficult part is the computation of the optimal solution that takes time

$$n\sum_{i=1}^{n} v_i' = n\frac{\sum_{i=1}^{n} v_i}{2^t} = n^2\frac{v_{\max}}{2^t}$$

Time

The most difficult part is the computation of the optimal solution that takes time

$$n\sum_{i=1}^{n} v_i' = n\frac{\sum_{i=1}^{n} v_i}{2^t} = n^2\frac{v_{\max}}{2^t}$$

But, $t = \lfloor \log \frac{r-1}{r} \frac{v_{\max}}{n} \rfloor$
Thus, for $r \to 1$, $= n^2 \frac{v_{\max}}{\frac{r-1}{r}\frac{v_{\max}}{n}} = \frac{rn^3}{r-1} = O(\frac{n^3}{r-1})$.

polynomial in input size and $1/(r-1)$

Quality of the solution

Quality of the solution

$\text{opt}(x) - A(x, r) \leq n2^t$, and $nv_{\max} \geq \text{opt}(x) \geq v_{\max}$.

Quality of the solution

$\text{opt}(x) - A(x,r) \leq n2^t$, and $nv_{\max} \geq \text{opt}(x) \geq v_{\max}$. Thus

$$\frac{n2^t}{v_{\max}} \geq \frac{\text{opt}(x) - A(x,r)}{\text{opt}(x)} = 1 - \frac{A(x,r)}{\text{opt}(x)}$$

$$\frac{A(x,r)}{\text{opt}(x)} \geq 1 - \frac{n2^t}{v_{\max}} = \frac{v_{\max} - n2^t}{v_{\max}}$$

But $t = \lfloor \log \frac{r-1}{r} \frac{v_{\max}}{n} \rfloor$ and $n2^t = n\frac{r-1}{r}\frac{v_{\max}}{n} = v_{\max}\frac{r-1}{r}$.

$$\text{opt}(x) \leq \frac{v_{\max}}{v_{\max} - n2^t} A(x,r) = \frac{v_{\max}}{v_{\max} - \frac{V_{\max}(r-1)}{r}} A(x,r) \leq rA(x,r).$$

Thus, SCALEDOWN is an r-approximation, we have a FPTAS for KNAPSACK.

Min-TSP with triangle inequality

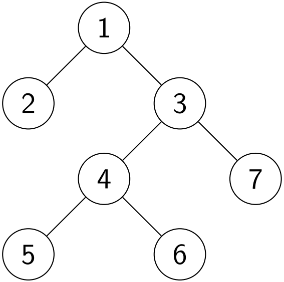
MIN-M-TSP

Given a set of n cities together with distances among any pair of cities, under the assumption that distances verify the triangle inequality, find a shortest tour.

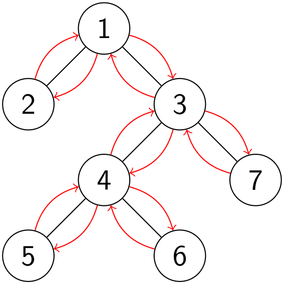
We model the instance by a weighted graph $G = (V, E, d)$.

Algorithm TSP-ST

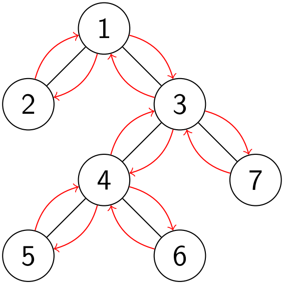
- Compute a minimum spanning tree T of G.
- Find the directed graph T' obtained from T by replacing each edge with two arcs in oposite directions.
- Find an *Eulerian circuit* R of T'.
- Let S be the walk in G directed by R.
- Transform S in a tour C, by removing (in order) all the vertices that have been already visited in S.



MST T

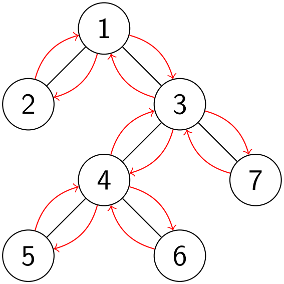


MST T Directed T'



MST T Directed T'

Walk $S = 1213454643731$



MST T Directed T'

Walk $S = 1213454643731$

Tour $S = 12345671$

Theorem

Algorithm TSP-ST *is a polynomial 2-aproximation algorithm for* MIN-M-TSP.

Theorem

Algorithm TSP-ST *is a polynomial 2-aproximation algorithm for* MIN-M-TSP*.*

Proof.

Observe that:

- $\text{opt}(G) \leq c(C) \leq c(S) = c(R)$ due to triangle inequality.
- $c(R) = 2c(T)$ as we use each edge twice.
- Furthermore, any circuit provides a spanning tree, just by removing one of their edges, total cost is below the circuit's distance: $c(T) \leq \text{opt}(G)$.

☺

Christofides' algorithm TSP-CH

Handshaking lemma: every finite undirected graph has an even number of vertices with odd degree

Christofides' algorithm TSP-CH

Handshaking lemma: every finite undirected graph has an even number of vertices with odd degree

- Compute a minimum spanning tree T of G.

Christofides' algorithm TSP-CH

Handshaking lemma: every finite undirected graph has an even number of vertices with odd degree

- Compute a minimum spanning tree T of G.
- Let O be the set of vertices with odd degree in T. By the handshaking lemma, O has an even number of vertices.

Christofides' algorithm TSP-CH

Handshaking lemma: every finite undirected graph has an even number of vertices with odd degree

- Compute a minimum spanning tree T of G.
- Let O be the set of vertices with odd degree in T. By the handshaking lemma, O has an even number of vertices.
- Find a minimum-weight perfect matching M in the induced subgraph given by the vertices from O.

Christofides' algorithm TSP-CH

Handshaking lemma: every finite undirected graph has an even number of vertices with odd degree

- Compute a minimum spanning tree T of G.
- Let O be the set of vertices with odd degree in T. By the handshaking lemma, O has an even number of vertices.
- Find a minimum-weight perfect matching M in the induced subgraph given by the vertices from O.
- Combine the edges of M and T to form a connected multigraph H in which each vertex has even degree.

Christofides' algorithm TSP-CH

Handshaking lemma: every finite undirected graph has an even number of vertices with odd degree

- Compute a minimum spanning tree T of G.
- Let O be the set of vertices with odd degree in T. By the handshaking lemma, O has an even number of vertices.
- Find a minimum-weight perfect matching M in the induced subgraph given by the vertices from O.
- Combine the edges of M and T to form a connected multigraph H in which each vertex has even degree.
- Find an *Eulerian circuit* R of H.

Christofides' algorithm TSP-CH

Handshaking lemma: every finite undirected graph has an even number of vertices with odd degree

- Compute a minimum spanning tree T of G.
- Let O be the set of vertices with odd degree in T. By the handshaking lemma, O has an even number of vertices.
- Find a minimum-weight perfect matching M in the induced subgraph given by the vertices from O.
- Combine the edges of M and T to form a connected multigraph H in which each vertex has even degree.
- Find an *Eulerian circuit* R of H.
- Let S be the walk in G directed by R.

Christofides' algorithm TSP-CH

Handshaking lemma: every finite undirected graph has an even number of vertices with odd degree

- Compute a minimum spanning tree T of G.
- Let O be the set of vertices with odd degree in T. By the handshaking lemma, O has an even number of vertices.
- Find a minimum-weight perfect matching M in the induced subgraph given by the vertices from O.
- Combine the edges of M and T to form a connected multigraph H in which each vertex has even degree.
- Find an *Eulerian circuit* R of H.
- Let S be the walk in G directed by R.
- Transform S in a tour C, by removing (in order) all the vertices that have been already visited in S.

Lemma

$G = (V, E)$ is a graph. Let M be a minimum-weight perfect matching for O. Then $c(M) \leq opt(G)/2$.

Lemma

$G = (V, E)$ is a graph. Let M be a minimum-weight perfect matching for O. Then $c(M) \leq opt(G)/2$.

Proof.

- Take any optimal tour of G, take the shortcuts to make a tour K for O.

Lemma

$G = (V, E)$ is a graph. Let M be a minimum-weight perfect matching for O. Then $c(M) \leq opt(G)/2$.

Proof.

- Take any optimal tour of G, take the shortcuts to make a tour K for O. By the triangle inequality $c(K) \leq \text{opt}(G)$

Lemma

$G = (V, E)$ is a graph. Let M be a minimum-weight perfect matching for O. Then $c(M) \leq opt(G)/2$.

Proof.

- Take any optimal tour of G, take the shortcuts to make a tour K for O. By the triangle inequality $c(K) \leq \text{opt}(G)$
- As O has even number of vertices K can be decomposed into two (alternating) perfect matchings S and S'.

Lemma

$G = (V, E)$ is a graph. Let M be a minimum-weight perfect matching for O. Then $c(M) \leq opt(G)/2$.

Proof.

- Take any optimal tour of G, take the shortcuts to make a tour K for O. By the triangle inequality $c(K) \leq \text{opt}(G)$
- As O has even number of vertices K can be decomposed into two (alternating) perfect matchings S and S'.
  M is a minimum weight perfect matching and
  $2c(M) \leq c(S) + c(S') = c(K) \leq \text{opt}(G)$

Lemma

$G = (V, E)$ is a graph. Let M be a minimum-weight perfect matching for O. Then $c(M) \leq opt(G)/2$.

Proof.

- Take any optimal tour of G, take the shortcuts to make a tour K for O. By the triangle inequality $c(K) \leq \text{opt}(G)$
- As O has even number of vertices K can be decomposed into two (alternating) perfect matchings S and S'.
  M is a minimum weight perfect matching and
  $2c(M) \leq c(S) + c(S') = c(K) \leq \text{opt}(G)$

☺

Theorem

Algorithm TSP-CH *is a polynomial 3/2-aproximation algorithm for* MIN-M-TSP*.*

Theorem

Algorithm TSP-CH *is a polynomial 3/2-aproximation algorithm for* MIN-M-TSP.

Proof.

Theorem

Algorithm TSP-CH *is a polynomial 3/2-aproximation algorithm for* MIN-M-TSP.

Proof.

- By the previous lemma $c(M) \leq \text{opt}(G)/2$.
- We already know that $c(T) \leq \text{opt}(G)$

Theorem

Algorithm TSP-CH *is a polynomial 3/2-aproximation algorithm for* MIN-M-TSP.

Proof.

- By the previous lemma $c(M) \leq \text{opt}(G)/2$.
- We already know that $c(T) \leq \text{opt}(G)$
- Also, $\text{opt}(G) \leq c(C) \leq c(S) = c(R)$ due to triangle inequality.

Theorem

Algorithm TSP-CH *is a polynomial 3/2-aproximation algorithm for* MIN-M-TSP.

Proof.

- By the previous lemma $c(M) \leq \text{opt}(G)/2$.
- We already know that $c(T) \leq \text{opt}(G)$
- Also, $\text{opt}(G) \leq c(C) \leq c(S) = c(R)$ due to triangle inequality.
- By construction, $c(R) \leq c(T) + w(M) \leq \text{opt}(G) + \frac{\text{opt}(G)}{2}$

Theorem

Algorithm TSP-CH *is a polynomial 3/2-aproximation algorithm for* MIN-M-TSP*.*

Proof.

- By the previous lemma $c(M) \leq \text{opt}(G)/2$.
- We already know that $c(T) \leq \text{opt}(G)$
- Also, $\text{opt}(G) \leq c(C) \leq c(S) = c(R)$ due to triangle inequality.
- By construction, $c(R) \leq c(T) + w(M) \leq \text{opt}(G) + \frac{\text{opt}(G)}{2}$

☺

TSP is not approximable

MIN-TSP

Given a set of n cities together with weights among any pair of cities, find a shortest tour.

Theorem

MIN-TSP *is non-approximable.*

Proof

Assume that we have a polynomial time $r(n)$-approximation algorithm $\mathcal{A}$, where $r(n)$ requires polynomial number of bits.

TSP is not approximable

TSP is not approximable

Given a graph G with n vertices consider the instance of Min-TSP on n cities and weights:

$$w(i,j) = \begin{cases} 0 & \text{if } (i,j) \in E \\ r(n)n & \text{otherwise} \end{cases}$$

TSP is not approximable

Given a graph G with n vertices consider the instance of Min-TSP on n cities and weights:

$$w(i,j) = \begin{cases} 0 & \text{if } (i,j) \in E \\ r(n)n & \text{otherwise} \end{cases}$$

- If G has a Hamiltonian Circuit, there is a TSP circuit with weight n, so $\mathcal{A}(G) \leq r(n)n$.

TSP is not approximable

Given a graph G with n vertices consider the instance of Min-TSP on n cities and weights:

$$w(i,j) = \begin{cases} 0 & \text{if } (i,j) \in E \\ r(n)n & \text{otherwise} \end{cases}$$

- If G has a Hamiltonian Circuit, there is a TSP circuit with weight n, so $\mathcal{A}(G) \leq r(n)n$.
- If G has no Hamiltonian Circuit, any TSP circuit has weight $> nr(n)$, so $\mathcal{A}(G) > r(n)n$.

TSP is not approximable

Given a graph G with n vertices consider the instance of Min-TSP on n cities and weights:

$$w(i,j) = \begin{cases} 0 & \text{if } (i,j) \in E \\ r(n)n & \text{otherwise} \end{cases}$$

- If G has a Hamiltonian Circuit, there is a TSP circuit with weight n, so $\mathcal{A}(G) \leq r(n)n$.
- If G has no Hamiltonian Circuit, any TSP circuit has weight $> nr(n)$, so $\mathcal{A}(G) > r(n)n$.

But, as $\mathcal{A}$ is polynomial, P=NP!

TSP is not approximable

Given a graph G with n vertices consider the instance of Min-TSP on n cities and weights:

$$w(i,j) = \begin{cases} 0 & \text{if } (i,j) \in E \\ r(n)n & \text{otherwise} \end{cases}$$

- If G has a Hamiltonian Circuit, there is a TSP circuit with weight n, so $\mathcal{A}(G) \leq r(n)n$.
- If G has no Hamiltonian Circuit, any TSP circuit has weight $> nr(n)$, so $\mathcal{A}(G) > r(n)n$.

But, as $\mathcal{A}$ is polynomial, P=NP!

End Proof