CS 473: Algorithms

Chandra Chekuri
chekuri@cs.illinois.edu
3228 Siebel Center

University of Illinois, Urbana-Champaign

Fall 2009

Part I

Complementation and Self-Reduction

The class P

- A language L (equivalently decision problem) is in the class P if there is a polynomial time algorithm A for deciding L; that is given a string x, A correctly decides if $x \in L$ and running time of A on x is polynomial in $|x|$, the length of x.

The class *NP*

Two equivalent definitions:

- Language L is in *NP* if there is a non-deterministic polynomial time algorithm A (Turing Machine) that decides L.
  - For $x \in L$, A has some non-deterministic choice of moves that will make A accept x
  - For $x \notin L$, no choice of moves will make A accept x
- L has an efficient certifier $C(\cdot, \cdot)$.
  - C is a polynomial time deterministic algorithm
  - For $x \in L$ there is a string y (proof) of length polynomial in $|x|$ such that $C(x, y)$ accepts
  - For $x \notin L$, no string y will make $C(x, y)$ accept

Complementation

Definition

Given a decision problem X, its complement $\bar{X}$ is the collection of all instances s such that $s \notin X$

Equivalently, in terms of languages:

Definition

Given a language L over alphabet Σ, its complement $\bar{L}$ is the language $\Sigma^* - L$.

Examples

- $PRIME = \{n \mid n \text{ is an integer and } n \text{ is prime}\}$
  $\bar{PRIME} = \{n \mid n \text{ is an integer and } n \text{ is not a prime}\}$
  $\bar{PRIME} = COMPOSITE$
- $SAT = \{\varphi \mid \varphi \text{ is a SAT formula and } \varphi \text{ is satisfiable}\}$
  $\bar{SAT} = \{\varphi \mid \varphi \text{ is a SAT formula and } \varphi \text{ is not satisfiable}\}$
  $\bar{SAT} = UnSAT$

Examples

- $PRIME = \{n \mid n \text{ is an integer and } n \text{ is prime}\}$
  $\bar{PRIME} = \{n \mid n \text{ is an integer and } n \text{ is not a prime}\}$
  $\bar{PRIME} = COMPOSITE$
- $SAT = \{\varphi \mid \varphi \text{ is a SAT formula and } \varphi \text{ is satisfiable}\}$
  $\bar{SAT} = \{\varphi \mid \varphi \text{ is a SAT formula and } \varphi \text{ is not satisfiable}\}$
  $\bar{SAT} = UnSAT$

Technicality: $\bar{SAT}$ also includes strings that do not encode any valid SAT formula. Typically we ignore those strings because they are not interesting. In all problems of interest, we assume that it is "easy" to check whether a given string is a valid instance or not.

P is closed under complementation

Proposition

Decision problem X is in P if and only if $\bar{X}$ is in P.

P is closed under complementation

Proposition

Decision problem X is in P if and only if $\bar{X}$ is in P.

Proof.

- If X is in P let A be a polynomial time algorithm for X.
- Construct polynomial time algorithm A' for $\bar{X}$ as follows: given input x, A' runs A on x and if A accepts x, A' rejects x and if A rejects x then A' accepts x.
- Only if direction is essentially the same argument.

Asymmetry of *NP*

Definition

Nondeterministic Polynomial Time (denoted by *NP*) is the class of all problems that have efficient certifiers.

Observation

To show that a problem is in *NP* we only need short, efficiently checkable certificates for "yes"-instances. What about "no"-instances?

Asymmetry of *NP*

Definition

Nondeterministic Polynomial Time (denoted by *NP*) is the class of all problems that have efficient certifiers.

Observation

To show that a problem is in *NP* we only need short, efficiently checkable certificates for "yes"-instances. What about "no"-instances?

given CNF formulat φ, is φ unsatisfiable?

Asymmetry of *NP*

Definition

Nondeterministic Polynomial Time (denoted by *NP*) is the class of all problems that have efficient certifiers.

Observation

To show that a problem is in *NP* we only need short, efficiently checkable certificates for "yes"-instances. What about "no"-instances?

given CNF formulat φ, is φ unsatisfiable?

Easy to give a proof that φ is satisfiable (an assignment) but no easy (known) proof to show that φ is unsatisfiable!

Examples

Some languages

- UnSAT: CNF formulas φ that are not satisfiable
- No-Hamilton-Cycle: graphs G that do not have a Hamilton cycle
- No-3-Color: graphs G that are not 3-colorable

Examples

Some languages

- UnSAT: CNF formulas φ that are not satisfiable
- No-Hamilton-Cycle: graphs G that do not have a Hamilton cycle
- No-3-Color: graphs G that are not 3-colorable

Above problems are complements of known NP problems (viewed as languages).

NP and co-*NP*

_NP_

Decision problems with a polynomial certifier. Examples, SAT, Hamiltonian Cycle, 3-Colorability

NP and co-*NP*

NP

Decision problems with a polynomial certifier. Examples, SAT, Hamiltonian Cycle, 3-Colorability

Definition

co-*NP* is the class of all decision problems X such that $\bar{X} \in NP$. Examples, UnSAT, No-Hamiltonian-Cycle, No-3-Colorable.

co-NP

L is a language in co-NP implies that there is a polynomial time certifier/verifier $C(\cdot, \cdot)$ such that

- for $s \notin L$ there is a proof t of size polynomial in $|s|$ such that $C(s, t)$ correctly says NO
- for $s \in L$ there is no proof t for which $C(s, t)$ will say NO

co-NP

L is a language in co-NP implies that there is a polynomial time certifier/verifier $C(\cdot, \cdot)$ such that

- for $s \notin L$ there is a proof t of size polynomial in $|s|$ such that $C(s, t)$ correctly says NO
- for $s \in L$ there is no proof t for which $C(s, t)$ will say NO

co-NP has checkable proofs for strings NOT in the language.

Natural Problems in co-NP

- Tautology: given a Boolean formula (not necessarily in CNF form), is it true for *all* possible assignments to the variables?
- Graph expansion: given a graph G, is it an *expander*? A graph $G = (V, E)$ is an expander iff for each $S \subset V$ with $|S| \leq |V|/2$, $|N(S)| \geq |S|$. Expanders are very important graphs in theoretical computer science and mathematics.

P, NP, co-NP

co-P: complement of P. Language X is in co-P iff $\bar{X} \in P$

P, NP, co-NP

co-P: complement of P. Language X is in co-P iff $\bar{X} \in P$

Proposition

$P = co\text{-}P$.

P, NP, co-NP

co-P: complement of P. Language X is in co-P iff $\bar{X} \in P$

Proposition

$P = co\text{-}P$.

Proposition

$P \subseteq NP \cap co\text{-}NP$.

P, NP, co-NP

co-P: complement of P. Language X is in co-P iff $\bar{X} \in P$

Proposition

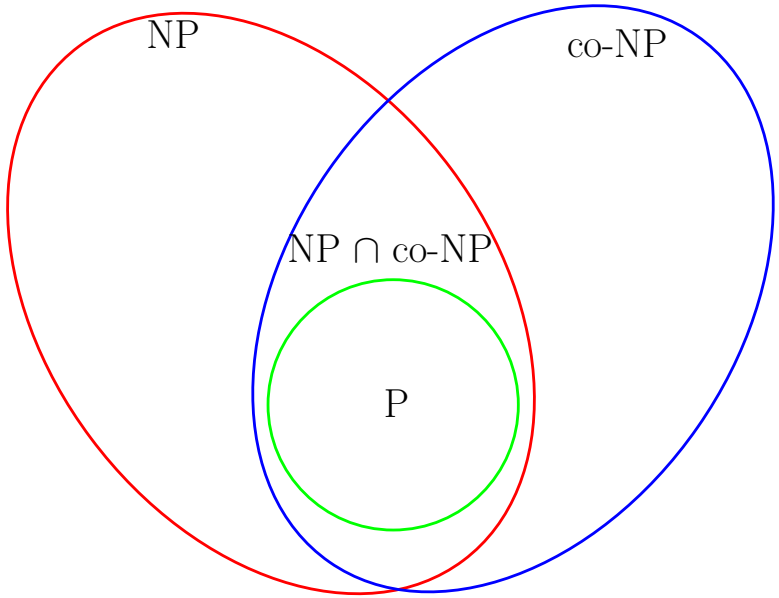
$P = co\text{-}P$.

Proposition

$P \subseteq NP \cap co\text{-}NP$.

Saw that $P \subseteq NP$. Same proof shows $P \subseteq$ co-NP

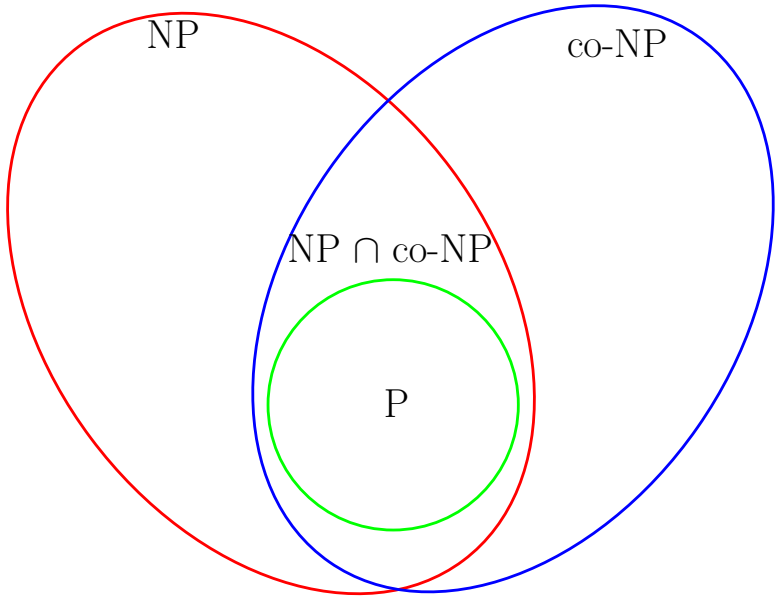
P, *NP*, and co-*NP*



Open Problems:

- Does $NP = \text{co-}NP$?

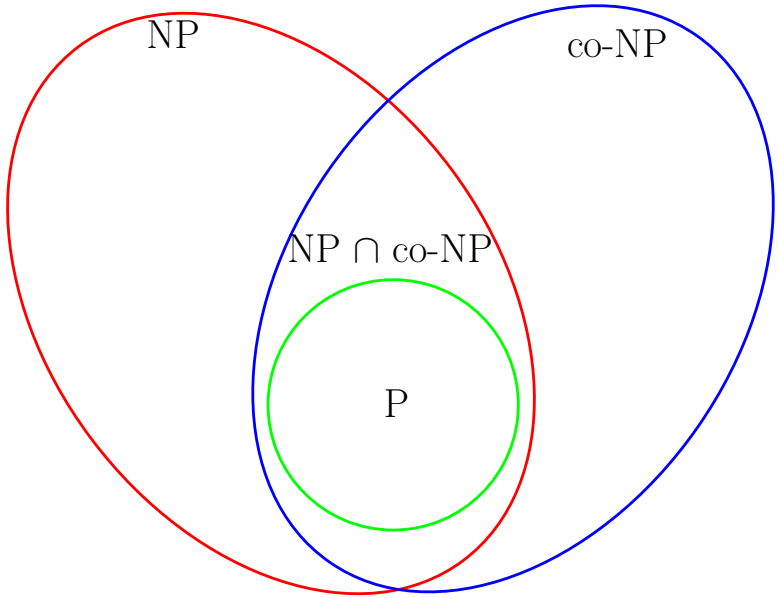
P, *NP*, and co-*NP*



Open Problems:

- Does $NP = \text{co-}NP$? Consensus opinion: No
- Is $P = NP \cap \text{co-}NP$?

P, *NP*, and co-*NP*



Open Problems:

- Does $NP = \text{co-}NP$? Consensus opinion: No
- Is $P = NP \cap \text{co-}NP$? No real consensus

P, *NP*, and co-*NP*

Proposition

If $P = NP$ then $NP =$ co-NP.

P, NP, and co-NP

Proposition

If $P = NP$ then $NP =$ co-NP.

Proof.

$P =$ co-P

If $P = NP$ then co-$NP =$ co-$P = P$. □

P, *NP*, and co-*NP*

Proposition

If $P = NP$ then $NP =$ co-NP.

Proof.

$P =$ co-P

If $P = NP$ then co-$NP =$ co-$P = P$. □

Corollary

If $NP \neq$ co-NP then $P \neq NP$.

P, NP, and co-NP

Proposition

If $P = NP$ then $NP =$ co-NP.

Proof.

$P =$ co-P

If $P = NP$ then co-$NP =$ co-$P = P$. □

Corollary

If $NP \neq$ co-NP then $P \neq NP$.

Importance of corollary: try to prove $P \neq NP$ by proving that $NP \neq$ co-NP.

$NP \cap$ co-NP

Complexity Class $NP \cap$ co-NP

Problems in this class have

- Efficient certifiers for yes-instances
- Efficient disqualifiers for no-instances

Problems have a good characterization property, since for both yes and no instances we have short efficiently checkable proofs

$NP \cap$ co-NP: Example

Example

Bipartite Matching: Given bipartite graph $G = (U \cup V, E)$, does G have a perfect matching?

Bipartite Matching $\in NP \cap$ co-NP

$NP \cap$ co-NP: Example

Example

Bipartite Matching: Given bipartite graph $G = (U \cup V, E)$, does G have a perfect matching?

Bipartite Matching $\in NP \cap$ co-NP

- If G is a yes-instance, then proof is just the perfect matching

$NP \cap$ co-NP: Example

Example

Bipartite Matching: Given bipartite graph $G = (U \cup V, E)$, does G have a perfect matching?

Bipartite Matching $\in NP \cap$ co-NP

- If G is a yes-instance, then proof is just the perfect matching
- If G is a no-instance, then by Hall's Theorem, there is a subset of vertices $A \subseteq U$ such that $|N(A)| < |A|$

Good Characterization $\stackrel{?}{=}$ Efficient Solution

- Bipartite Matching has a polynomial time algorithm
- Do all problems in $NP \cap$ co-NP have polynomial time algorithms? That is, is $P = NP \cap$ co-NP?

Good Characterization $\stackrel{?}{=}$ Efficient Solution

- Bipartite Matching has a polynomial time algorithm
- Do all problems in $NP \cap$ co-NP have polynomial time algorithms? That is, is $P = NP \cap$ co-NP?
  Problems in $NP \cap$ co-NP have been proved to be in P many years later
  - Linear programming (Khachiyan 1979)
    - Duality easily shows that it is in $NP \cap$ co-NP
  - Primality Testing (Agarwal-Kayal-Saxena 2002)
    - Easy to see that PRIME is in co-NP (why?)
    - PRIME is in NP - not easy to show! (Vaughan Pratt 1975)

$P \stackrel{?}{=} NP \cap$ co-NP (contd)

- Some problems in $NP \cap$ co-NP still cannot be proved to have polynomial time algorithms

$P \stackrel{?}{=} NP \cap$ co-NP (contd)

- Some problems in $NP \cap$ co-NP still cannot be proved to have polynomial time algorithms
  - Parity Games

$P \stackrel{?}{=} NP \cap$ co-NP (contd)

- Some problems in $NP \cap$ co-NP still cannot be proved to have polynomial time algorithms
  - Parity Games
  - Other more specialized problems

co-NP Completeness

Definition

A problem X is said to be *co-NP*-complete if

- $X \in$ *co-NP*
- (Hardness) For any $Y \in$ *co-NP*, $Y \leq_P X$

co-NP Completeness

Definition

A problem X is said to be *co-NP*-complete if

- $X \in$ *co-NP*
- (Hardness) For any $Y \in$ *co-NP*, $Y \leq_P X$

co-NP-Complete problems are the hardest problems in *co-NP*.

co-NP Completeness

Definition

A problem X is said to be *co-NP*-complete if

- $X \in$ *co-NP*
- (Hardness) For any $Y \in$ *co-NP*, $Y \leq_P X$

co-NP-Complete problems are the hardest problems in *co-NP*.

Lemma

X is co-NP-Complete if and only if $\bar{X}$ is NP-Complete.

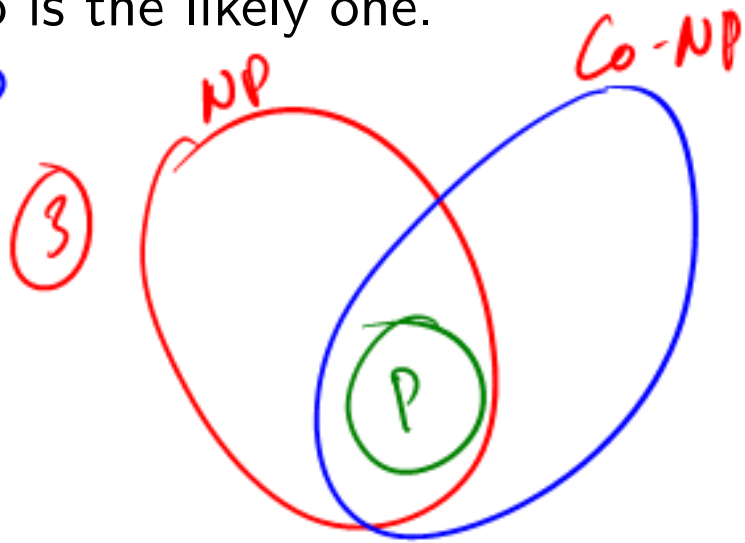
Proof left as an exercise.

P, *NP* and *co-NP*

Possible scenarios:

- $P = NP$. Then $P = NP = co\text{-}NP$.
- $NP = co\text{-}NP$ and $P \neq NP$ (and hence also $P \neq co\text{-}NP$).
- $NP \neq co\text{-}NP$. Then $P \neq NP$ and also $P \neq co\text{-}NP$.

Most people believe that the last scenario is the likely one.



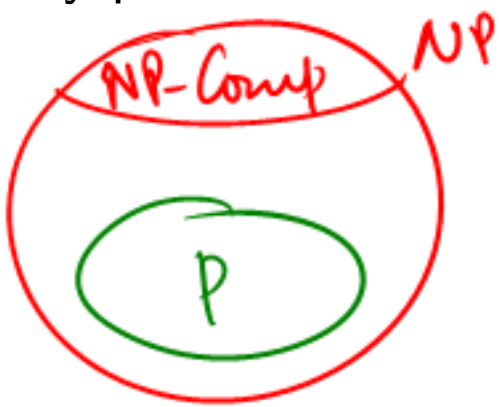
P, *NP* and *co-NP*

Possible scenarios:

- $P = NP$. Then $P = NP = co\text{-}NP$.
- $NP = co\text{-}NP$ and $P \neq NP$ (and hence also $P \neq co\text{-}NP$).
- $NP \neq co\text{-}NP$. Then $P \neq NP$ and also $P \neq co\text{-}NP$.

Most people believe that the last scenario is the likely one.

Question: Suppose $P \neq NP$. Is every problem in $NP - P$ *NP*-Complete?



P, *NP* and *co-NP*

Possible scenarios:

- $P = NP$. Then $P = NP = co\text{-}NP$.
- $NP = co\text{-}NP$ and $P \neq NP$ (and hence also $P \neq co\text{-}NP$).
- $NP \neq co\text{-}NP$. Then $P \neq NP$ and also $P \neq co\text{-}NP$.

Most people believe that the last scenario is the likely one.

Question: Suppose $P \neq NP$. Is every problem in $NP - P$ NP-Complete?

Theorem (Ladner)

If $P \neq NP$ then there is a problem/language $X \in NP - P$ such that X is not NP-Complete.

Back to Decision versus Search

- Recall, decision problems are those with yes/no answers, while search problems require an explicit solution for a yes instance

Back to Decision versus Search

- Recall, decision problems are those with yes/no answers, while search problems require an explicit solution for a yes instance

Example

Back to Decision versus Search

- Recall, decision problems are those with yes/no answers, while search problems require an explicit solution for a yes instance

Example

- Satisfiability

Back to Decision versus Search

- Recall, decision problems are those with yes/no answers, while search problems require an explicit solution for a yes instance

Example

- Satisfiability
  - Decision: Is the formula φ satisfiable?

Back to Decision versus Search

- Recall, decision problems are those with yes/no answers, while search problems require an explicit solution for a yes instance

Example

- Satisfiability
  - Decision: Is the formula φ satisfiable?
  - Search: Find assignment that satisfies φ

Back to Decision versus Search

- Recall, decision problems are those with yes/no answers, while search problems require an explicit solution for a yes instance

Example

- Satisfiability
  - Decision: Is the formula φ satisfiable?
  - Search: Find assignment that satisfies φ
- Graph coloring

Back to Decision versus Search

- Recall, decision problems are those with yes/no answers, while search problems require an explicit solution for a yes instance

Example

- Satisfiability
  - Decision: Is the formula φ satisfiable?
  - Search: Find assignment that satisfies φ
- Graph coloring
  - Decision: Is graph G 3-colorable?

Back to Decision versus Search

- Recall, decision problems are those with yes/no answers, while search problems require an explicit solution for a yes instance

Example

- Satisfiability
  - Decision: Is the formula φ satisfiable?
  - Search: Find assignment that satisfies φ
- Graph coloring
  - Decision: Is graph G 3-colorable?
  - Search: Find a 3-coloring of the vertices of G

Decision "reduces to" Search

- Efficient algorithm for search implies efficient algorithm for decision

Decision "reduces to" Search

- Efficient algorithm for search implies efficient algorithm for decision
- If decision problem is difficult then search problem is also difficult

Decision "reduces to" Search

- Efficient algorithm for search implies efficient algorithm for decision
- If decision problem is difficult then search problem is also difficult
- Can an efficient algorithm for decision imply an efficient algorithm for search?

Decision "reduces to" Search

- Efficient algorithm for search implies efficient algorithm for decision
- If decision problem is difficult then search problem is also difficult
- Can an efficient algorithm for decision imply an efficient algorithm for search? Yes, for all the problems we have seen. In fact for all NP-Complete Problems.

Self Reduction

Definition

A problem is said to be self reducible if the search problem reduces (by Cook reduction) in polynomial time to decision problem. In other words, there is an algorithm to solve the search problem that has polynomially many steps, where each step is either

- A conventional computational step, or
- A call to subroutine solving the decision problem

Back to SAT

Proposition

SAT is self reducible

In other words, there is a polynomial time algorithm to find the satisfying assignment if one can periodically check if some formula is satisfiable

Search Algorithm for SAT from a Decision Algorithm for SAT

Input: SAT formula φ with n variables $x_1, x_2, \ldots, x_n$.

- set $x_1 = 0$ in φ and get new formula φ_1. check if φ_1 is satisfiable using decision algorithm. if φ_1 is satisfiable, recursively find assignment to $x_2, x_3, \ldots, x_n$ that satisfy φ_1 and output $x_1 = 0$ along with the assignment to $x_2, \ldots, x_n$.
- if φ_1 is not satisfiable then set $x_1 = 1$ in φ to get formula φ_2. if φ_2 is satisfiable , recursively find assignment to $x_2, x_3, \ldots, x_n$ that satisfy φ_2 and output $x_1 = 1$ along with the assignment to $x_2, \ldots, x_n$.
- if φ_1 and φ_2 are both not satisfiable then φ is not satisfiable.

Algorithm runs in polynomial time if the decision algorithm for SAT runs in polynomial time. At most $2n$ calls to decision algorithm.

$$\varphi = (x_1 \vee \bar{x}_2)(\bar{x}_2 \vee x_3)(\bar{x}_1 \vee \bar{x}_3)$$

Sd. $x_1 = 1$

Gd. $\varphi' = (\bar{x}_2 \vee x_3)(\bar{x}_3)$

Self-Reduction for NP-Complete Problems

Theorem

Every NP-Complete problem/language L is self-reducible.

Proof out of scope.

Note that proof is only for complete languages, not for all languages in NP. Otherwise Factoring would be in polynomial time and we would not rely on it for our current security protocols.

Easy and instructive to prove self-reducibility for specific NP-Complete problems such as Independent Set, Vertex Cover, Hamiltonian Cycle etc. See HBS problems and finals prep.

Part II

Coping with Intractability

Intractability

Many important and useful problems are hard (NP-hard, co-NP-hard, undecidable, ...). No efficient algorithm possible or known.

Intractability

Many important and useful problems are hard (NP-hard, co-NP-hard, undecidable, ...). No efficient algorithm possible or known.

Nevertheless, need to address problems.

Example Problems:

- build tools to check for bugs in programs and hardware (undecidable in general)
- assign frequencies to cell phone towers (NP-hard via coloring)
- solve integer programs to minimize cost of manufacturing goods (NP-hard)

Coping with Intractability

Compromise!

Some general things that people do.

- Consider special cases of the problem which may be tractable.

Coping with Intractability

Compromise!

Some general things that people do.

- Consider special cases of the problem which may be tractable.
- Run inefficient algorithms (for example exponential time algorithms for NP-hard problems) augmented with (very) clever heuristics

Coping with Intractability

Compromise!

Some general things that people do.

- Consider special cases of the problem which may be tractable.
- Run inefficient algorithms (for example exponential time algorithms for NP-hard problems) augmented with (very) clever heuristics
  - stop algorithm when time/resources run out

Coping with Intractability

Compromise!

Some general things that people do.

- Consider special cases of the problem which may be tractable.
- Run inefficient algorithms (for example exponential time algorithms for NP-hard problems) augmented with (very) clever heuristics
  - stop algorithm when time/resources run out
  - use massive computational power

Coping with Intractability

Compromise!

Some general things that people do.

- Consider special cases of the problem which may be tractable.
- Run inefficient algorithms (for example exponential time algorithms for NP-hard problems) augmented with (very) clever heuristics
  - stop algorithm when time/resources run out
  - use massive computational power
- Exploit properties of instances that arise in practice which may be much easier. Give up on hard instances, which is ok.

Coping with Intractability

Compromise!

Some general things that people do.

- Consider special cases of the problem which may be tractable.
- Run inefficient algorithms (for example exponential time algorithms for NP-hard problems) augmented with (very) clever heuristics
  - stop algorithm when time/resources run out
  - use massive computational power
- Exploit properties of instances that arise in practice which may be much easier. Give up on hard instances, which is ok.
- Settle for sub-optimal solutions, especially for optimization problems

Heuristics

heuristic:

1. Of or relating to a usually speculative formulation serving as a guide in the investigation or solution of a problem.
2. Of or constituting an educational method in which learning takes place through discoveries that result from investigations made by the student.
3. Computer Science. Relating to or using a problem-solving technique in which the most appropriate solution of several found by alternative methods is selected at successive stages of a program for use in the next step of the program.

Heuristics

- Heuristics work very well for many problems in practice! For example there are algorithms for SAT (SAT-solvers) that can solve restricted classes of SAT formulas even with thousands of variables and clauses!
- Heuristics need to be developed based on a good understanding of the underlying problem.
- Heuristics for some expressive problems such as SAT and Integer Programming are extensively studied because many other problems can be "naturally" reduced to them.
- Some generic heuristic methods:
  - Intelligent exhaustive search methods via backtracking: Branch-and-Bound and Branch-and-Cut etc
  - Local search and variants: Simulated Annealing, Random starts etc.
  - Tabu serach, genetic algorithms, ...

Part III

Approximation Algorithms

Approximation Algorithms

Approximation algorithms are heuristics with guarantees on their performance.

Definition

Let X is an optimization problem such that $\mathrm{opt}(s)$ is the optimum value on input instance s. An algorithm A is a ρ-approximation algorithm for X if

Approximation Algorithms

Approximation algorithms are heuristics with guarantees on their performance.

Definition

Let X is an optimization problem such that $\mathrm{opt}(s)$ is the optimum value on input instance s. An algorithm A is a ρ-approximation algorithm for X if

1. On *all* inputs s, $A(s)$ runs in polynomial time

Approximation Algorithms

Approximation algorithms are heuristics with guarantees on their performance.

Definition

Let X is an optimization problem such that $\mathrm{opt}(s)$ is the optimum value on input instance s. An algorithm A is a ρ-approximation algorithm for X if

1. On *all* inputs s, $A(s)$ runs in polynomial time
2. On *all* inputs s, the output $A(s)$ is within a ρ-ratio of the optimal value $\mathrm{opt}(s)$

Approximation Algorithms

Approximation algorithms are heuristics with guarantees on their performance.

Definition

Let X is an optimization problem such that $\mathrm{opt}(s)$ is the optimum value on input instance s. An algorithm A is a ρ-approximation algorithm for X if

1. On *all* inputs s, $A(s)$ runs in polynomial time
2. On *all* inputs s, the output $A(s)$ is within a ρ-ratio of the optimal value $\mathrm{opt}(s)$, i.e., if X is a maximization problem then $A(s) \geq \mathrm{opt}(s)/\rho$ and if X is a minimization problem then $A(s) \leq \rho \cdot \mathrm{opt}(s)$

Approximation Algorithms

Approximation algorithms are heuristics with guarantees on their performance.

Definition

Let X is an optimization problem such that $\mathrm{opt}(s)$ is the optimum value on input instance s. An algorithm A is a ρ-approximation algorithm for X if

1. On *all* inputs s, $A(s)$ runs in polynomial time
2. On *all* inputs s, the output $A(s)$ is within a ρ-ratio of the optimal value $\mathrm{opt}(s)$, i.e., if X is a maximization problem then $A(s) \geq \mathrm{opt}(s)/\rho$ and if X is a minimization problem then $A(s) \leq \rho \cdot \mathrm{opt}(s)$

Thus, on *all* inputs A runs in polynomial time, and gives an answer that is guaranteed to be close to the optimal value.

Load Balancing

Problem

Input: m identical machines, n jobs with ith job having processing time t_i

Goal: Schedule jobs to computers such that

- Jobs run contiguously on a machine

Load Balancing

Problem

Input: m identical machines, n jobs with ith job having processing time t_i

Goal: Schedule jobs to computers such that

- Jobs run contiguously on a machine
- A machine processes only one job a time

Load Balancing

Problem

Input: m identical machines, n jobs with ith job having processing time t_i

Goal: Schedule jobs to computers such that

- Jobs run contiguously on a machine
- A machine processes only one job a time
- Makespan or maximum load on any machine is minimized

Load Balancing

Problem

Input: m identical machines, n jobs with ith job having processing time t_i

Goal: Schedule jobs to computers such that

- Jobs run contiguously on a machine
- A machine processes only one job a time
- Makespan or maximum load on any machine is minimized

Definition

Let $A(i)$ be the set of jobs assigned to machine i. The load on i is $T_i = \sum_{j \in A(i)} t_j$.

Load Balancing

Problem

Input: m identical machines, n jobs with ith job having processing time t_i

Goal: Schedule jobs to computers such that

- Jobs run contiguously on a machine
- A machine processes only one job a time
- Makespan or maximum load on any machine is minimized

Definition

Let $A(i)$ be the set of jobs assigned to machine i. The load on i is $T_i = \sum_{j \in A(i)} t_j$.

The makespan of A is $T = \max_i T_i$

Load Balancing: Example

Example

Consider 6 jobs whose processing times is given as follows

Jobs	1	2	3	4	5	6
t_i	2	3	4	6	2	2

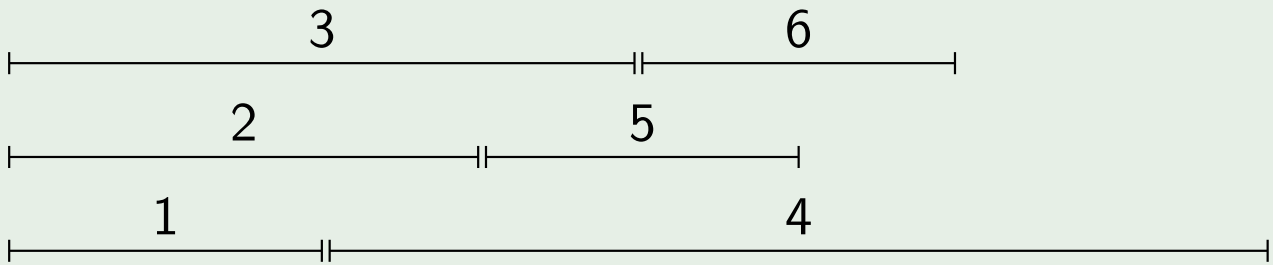
Load Balancing: Example

Example

Consider 6 jobs whose processing times is given as follows

Jobs	1	2	3	4	5	6
t_i	2	3	4	6	2	2

Consider the following schedule on 3 machines



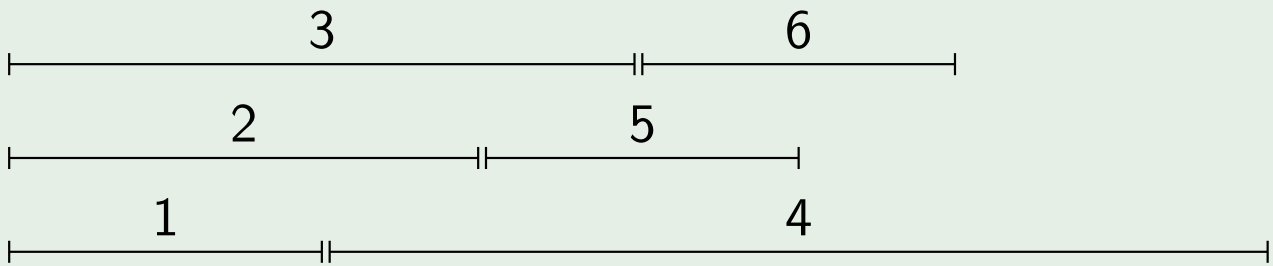
Load Balancing: Example

Example

Consider 6 jobs whose processing times is given as follows

Jobs	1	2	3	4	5	6
t_i	2	3	4	6	2	2

Consider the following schedule on 3 machines



The loads are: $T_1 = 8$, $T_2 = 5$, and $T_3 = 6$. So makespan of schedule is 8

Load Balancing is NP-Complete

Decision version: given n, m and $t_1, t_2, \ldots, t_n$ and a target T, is there a schedule with makespan at most T?

Problem is NP-Complete: reduce from Subset Sum.

Greedy Algorithm

1. Consider the jobs in some fixed order
2. Assign job j to the machine with lowest load so far

Example

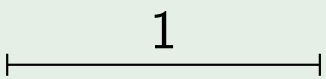
Jobs	1	2	3	4	5	6
t_i	2	3	4	6	2	2

Greedy Algorithm

1. Consider the jobs in some fixed order
2. Assign job j to the machine with lowest load so far

Example

Jobs	1	2	3	4	5	6
t_i	2	3	4	6	2	2

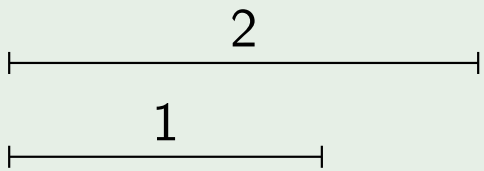


Greedy Algorithm

1. Consider the jobs in some fixed order
2. Assign job j to the machine with lowest load so far

Example

Jobs	1	2	3	4	5	6
t_i	2	3	4	6	2	2

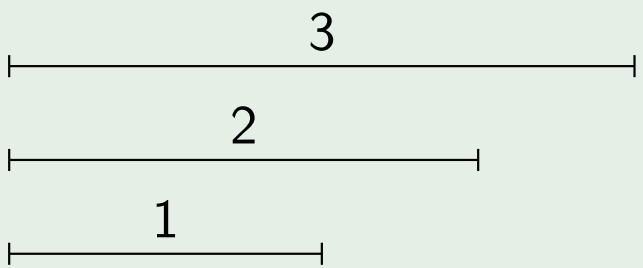


Greedy Algorithm

1. Consider the jobs in some fixed order
2. Assign job j to the machine with lowest load so far

Example

Jobs	1	2	3	4	5	6
t_i	2	3	4	6	2	2

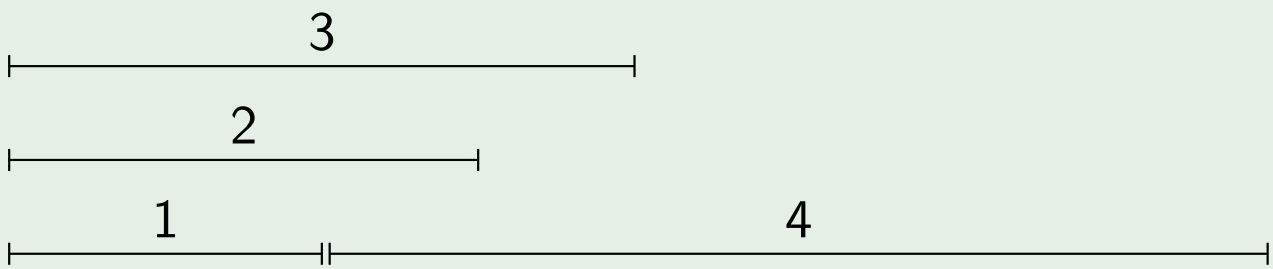


Greedy Algorithm

1. Consider the jobs in some fixed order
2. Assign job j to the machine with lowest load so far

Example

Jobs	1	2	3	4	5	6
t_i	2	3	4	6	2	2

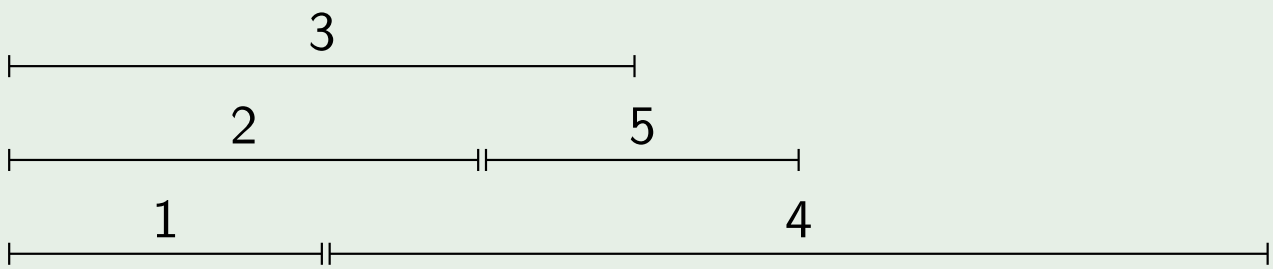


Greedy Algorithm

1. Consider the jobs in some fixed order
2. Assign job j to the machine with lowest load so far

Example

Jobs	1	2	3	4	5	6
t_i	2	3	4	6	2	2

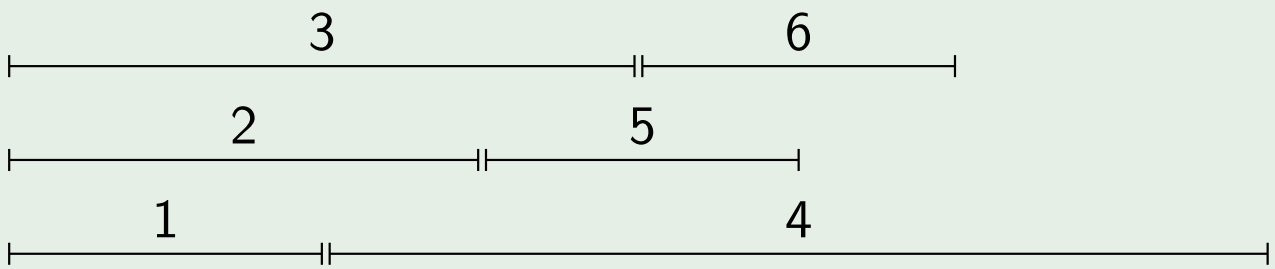


Greedy Algorithm

1. Consider the jobs in some fixed order
2. Assign job j to the machine with lowest load so far

Example

Jobs	1	2	3	4	5	6
t_i	2	3	4	6	2	2



Putting it together

```
for each machine i
    T_i = 0 (* initially no load *)
    A(i) = ∅ (* initially no jobs *)
for each job j
    Let i be machine with smallest load
    A(i) = A(i) ∪ {j} (* schedule j on i *)
    T_i = T_i + t_j (* compute new load *)
```

Running Time

Putting it together

```
for each machine i
    T_i = 0 (* initially no load *)
    A(i) = ∅ (* initially no jobs *)
for each job j
    Let i be machine with smallest load
    A(i) = A(i) ∪ {j} (* schedule j on i *)
    T_i = T_i + t_j (* compute new load *)
```

Running Time

- First loop takes $O(m)$ time

Putting it together

```
for each machine i
    T_i = 0 (* initially no load *)
    A(i) = ∅ (* initially no jobs *)
for each job j
    Let i be machine with smallest load
    A(i) = A(i) ∪ {j} (* schedule j on i *)
    T_i = T_i + t_j (* compute new load *)
```

Running Time

- First loop takes $O(m)$ time
- Second loop has $O(n)$ iterations

Putting it together

```
for each machine i
    T_i = 0 (* initially no load *)
    A(i) = ∅ (* initially no jobs *)
for each job j
    Let i be machine with smallest load
    A(i) = A(i) ∪ {j} (* schedule j on i *)
    T_i = T_i + t_j (* compute new load *)
```

Running Time

- First loop takes $O(m)$ time
- Second loop has $O(n)$ iterations

Putting it together

```
for each machine i
    T_i = 0 (* initially no load *)
    A(i) = ∅ (* initially no jobs *)
for each job j
    Let i be machine with smallest load
    A(i) = A(i) ∪ {j} (* schedule j on i *)
    T_i = T_i + t_j (* compute new load *)
```

Running Time

- First loop takes $O(m)$ time
- Second loop has $O(n)$ iterations
- Body of loop takes $O(\log m)$ time using a priority heap

Putting it together

```
for each machine i
    T_i = 0 (* initially no load *)
    A(i) = ∅ (* initially no jobs *)
for each job j
    Let i be machine with smallest load
    A(i) = A(i) ∪ {j} (* schedule j on i *)
    T_i = T_i + t_j (* compute new load *)
```

Running Time

- First loop takes $O(m)$ time
- Second loop has $O(n)$ iterations
- Body of loop takes $O(\log m)$ time using a priority heap
- Total time is $O(n \log m + m)$

Optimality

Problem

Is the greedy algorithm optimal?

Optimality

Problem

Is the greedy algorithm optimal? No! For example, on

Jobs	1	2	3	4	5	6
t_i	2	3	4	6	2	2

the greedy algorithm gives schedule with makespan 8, but optimal is 7

Optimality

Problem

Is the greedy algorithm optimal? No! For example, on

Jobs	1	2	3	4	5	6
t_i	2	3	4	6	2	2

the greedy algorithm gives schedule with makespan 8, but optimal is 7

In fact, the load balancing problem is *NP*-complete.

Quality of Solution

Quality of Solution

Theorem (Graham 1966)

The makespan of the schedule output by the greedy algorithm is at most 2 times the optimal make span. In other words, the greedy algorithm is a 2-approximation.

Challenge

How do we compare the output of the greedy algorithm with the optimal? How do we get the value of the optimal solution?

Quality of Solution

Theorem (Graham 1966)

The makespan of the schedule output by the greedy algorithm is at most 2 times the optimal make span. In other words, the greedy algorithm is a 2-approximation.

Challenge

How do we compare the output of the greedy algorithm with the optimal? How do we get the value of the optimal solution?

- We will obtain bounds on the optimal value

Bounding the Optimal Value

Lemma

$T^* \geq \max_j t_j$, where T^* is the optimal makespan

Bounding the Optimal Value

Lemma

$T^* \geq \max_j t_j$, *where* T^* *is the optimal makespan*

Proof.

Some machine will run the job with maximum processing time □

Bounding the Optimal Value

Lemma

$T^* \geq \max_j t_j$, where T^* is the optimal makespan

Proof.

Some machine will run the job with maximum processing time □

Lemma

$T^* \geq \frac{1}{m} \sum_j t_j$, where T^* is the optimal makespan

Bounding the Optimal Value

Lemma

$T^* \geq \max_j t_j$, *where* T^* *is the optimal makespan*

Proof.

Some machine will run the job with maximum processing time □

Lemma

$T^* \geq \frac{1}{m} \sum_j t_j$, *where* T^* *is the optimal makespan*

Proof.

Bounding the Optimal Value

Lemma

$T^* \geq \max_j t_j$, where T^* is the optimal makespan

Proof.

Some machine will run the job with maximum processing time □

Lemma

$T^* \geq \frac{1}{m} \sum_j t_j$, where T^* is the optimal makespan

Proof.

- Total processing time is $\sum_j t_j$

Bounding the Optimal Value

Lemma

$T^* \geq \max_j t_j$, where T^* is the optimal makespan

Proof.

Some machine will run the job with maximum processing time □

Lemma

$T^* \geq \frac{1}{m} \sum_j t_j$, where T^* is the optimal makespan

Proof.

- Total processing time is $\sum_j t_j$
- Some machine must do at least $\frac{1}{m}$ (or average) of the total work

□

Analysis of Greedy Algorithm

Theorem

The greedy algorithm is a 2-approximation

Analysis of Greedy Algorithm

Theorem

The greedy algorithm is a 2-approximation

Proof.

Let machine i have the maximum load T_i, and let j be the last job scheduled on machine i

Analysis of Greedy Algorithm

Theorem

The greedy algorithm is a 2-approximation

Proof.

Let machine i have the maximum load T_i, and let j be the last job scheduled on machine i

- At the time j was scheduled, machine i must have had the least load

Analysis of Greedy Algorithm

Theorem

The greedy algorithm is a 2-approximation

Proof.

Let machine i have the maximum load T_i, and let j be the last job scheduled on machine i

- At the time j was scheduled, machine i must have had the least load; load on i before assigning job j is $T_i - t_j$

Analysis of Greedy Algorithm

Theorem

The greedy algorithm is a 2-approximation

Proof.

Let machine i have the maximum load T_i, and let j be the last job scheduled on machine i

- At the time j was scheduled, machine i must have had the least load; load on i before assigning job j is $T_i - t_j$
- Since i has the least load, we know $T_i - t_j \leq T_k$, for all k. Thus, $m(T_i - t_j) \leq \sum_k T_k$

Analysis of Greedy Algorithm

Theorem

The greedy algorithm is a 2-approximation

Proof.

Let machine i have the maximum load T_i, and let j be the last job scheduled on machine i

- At the time j was scheduled, machine i must have had the least load; load on i before assigning job j is $T_i - t_j$
- Since i has the least load, we know $T_i - t_j \leq T_k$, for all k. Thus, $m(T_i - t_j) \leq \sum_k T_k$
- But $\sum_k T_k = \sum_\ell t_\ell$. So $T_i - t_j \leq \frac{1}{m} \sum_k T_k = \frac{1}{m} \sum_\ell t_\ell \leq T^*$

Analysis of Greedy Algorithm

Theorem

The greedy algorithm is a 2-approximation

Proof.

Let machine i have the maximum load T_i, and let j be the last job scheduled on machine i

- At the time j was scheduled, machine i must have had the least load; load on i before assigning job j is $T_i - t_j$
- Since i has the least load, we know $T_i - t_j \leq T_k$, for all k. Thus, $m(T_i - t_j) \leq \sum_k T_k$
- But $\sum_k T_k = \sum_\ell t_\ell$. So $T_i - t_j \leq \frac{1}{m} \sum_k T_k = \frac{1}{m} \sum_\ell t_\ell \leq T^*$
- Finally, $T_i = (T_i - t_j) + t_j \leq T^* + T^* = 2T^*$ □

Tightness of Analysis

Proposition

The analysis of the greedy algorithm is tight, i.e., there is an example on which the greedy schedule has twice the optimal makespan

Tightness of Analysis

Proposition

The analysis of the greedy algorithm is tight, i.e., there is an example on which the greedy schedule has twice the optimal makespan

Proof.

Consider problem of $m(m-1)$ jobs with processing time 1 and the last job with processing time m.

Tightness of Analysis

Proposition

The analysis of the greedy algorithm is tight, i.e., there is an example on which the greedy schedule has twice the optimal makespan

Proof.

Consider problem of $m(m-1)$ jobs with processing time 1 and the last job with processing time m.
Greedy schedule: distribute first the $m(m-1)$ jobs equally among the m machines, and then schedule the last job on machine 1, giving a makespan of $(m-1)+m = 2m-1$

Tightness of Analysis

Proposition

The analysis of the greedy algorithm is tight, i.e., there is an example on which the greedy schedule has twice the optimal makespan

Proof.

Consider problem of $m(m-1)$ jobs with processing time 1 and the last job with processing time m.
Greedy schedule: distribute first the $m(m-1)$ jobs equally among the m machines, and then schedule the last job on machine 1, giving a makespan of $(m-1)+m = 2m-1$
The optimal schedule: run last job on machine 1, and then distribute remaining jobs equally among $m-1$ machines, giving a makespan of m □

Improved Greedy Algorithm

Modified Greedy

Sort the jobs in descending order of processing time, and process jobs using greedy algorithm

Improved Greedy Algorithm

Modified Greedy

Sort the jobs in descending order of processing time, and process jobs using greedy algorithm

```
for each machine i
    T_i = 0 (* initially no load *)
    A(i) = ∅ (* initially no jobs *)
for each job j in descending order of processing time
    Let i be machine with smallest load
    A(i) = A(i) ∪ {j} (* schedule j on i *)
    T_i = T_i + t_j (* compute new load *)
```

Technical Lemma

Lemma

If there are more than m jobs then $T^* \geq 2t_{m+1}$

Technical Lemma

Lemma

If there are more than m jobs then $T^* \geq 2t_{m+1}$

Proof.

Consider the first $m + 1$ jobs

Technical Lemma

Lemma

If there are more than m jobs then $T^* \geq 2t_{m+1}$

Proof.

Consider the first $m + 1$ jobs

- Two of them must be scheduled on same machine, by pigeon-hole principle

Technical Lemma

Lemma

If there are more than m jobs then $T^* \geq 2t_{m+1}$

Proof.

Consider the first $m+1$ jobs

- Two of them must be scheduled on same machine, by pigeon-hole principle
- Both jobs have processing time at least t_{m+1}, since we consider jobs according to processing time, and this proves the lemma □

Analysis of Modified Greedy

Theorem

The modified greedy algorithm is a 3/2-approximation

Analysis of Modified Greedy

Theorem

The modified greedy algorithm is a 3/2-approximation

Proof.

Once again let i be the machine with highest load, and let j be the last job scheduled

Analysis of Modified Greedy

Theorem

The modified greedy algorithm is a 3/2-approximation

Proof.

Once again let i be the machine with highest load, and let j be the last job scheduled

- If machine i has only one job then schedule is optimal

Analysis of Modified Greedy

Theorem

The modified greedy algorithm is a 3/2-approximation

Proof.

Once again let i be the machine with highest load, and let j be the last job scheduled

- If machine i has only one job then schedule is optimal
- If i has at least 2 jobs, then it must be the case that $j \geq m+1$. This means $t_j \leq t_{m+1} \leq \frac{1}{2} T^*$

Analysis of Modified Greedy

Theorem

The modified greedy algorithm is a 3/2-approximation

Proof.

Once again let i be the machine with highest load, and let j be the last job scheduled

- If machine i has only one job then schedule is optimal
- If i has at least 2 jobs, then it must be the case that $j \geq m+1$. This means $t_j \leq t_{m+1} \leq \frac{1}{2} T^*$
- Thus, $T_i = (T_i - t_j) + t_j \leq T^* + \frac{1}{2} T^*$ □

Tightness of Analysis

Theorem (Graham)

Modified greedy is a 4/3-approximation

The 4/3-analysis is tight.