CS 473: Algorithms

Ruta Mehta

University of Illinois, Urbana-Champaign

Spring 2018

Review session

Lecture 99

May 9, 2018

Topics

Included topics:

Dynamic Programming.

Shortest paths in graphs including negative lengths and negative cycle detection (Bellman Ford).

Basics of randomization.

Network flows and applications to mincuts, matching, assignment problems, disjoint paths.

Basics of LP, modeling, writing a dual of an LP.

Reductions and NP-Completeness.

Basics of approximation.

Omitted topics:

FFT and applications.

Advanced topics in randomization including hashing, streaming, finger printing, string matching.

We will review

- Basics of LP, modeling, writing a dual of an LP
- Reductions and NP-Completeness.
- Basics of approximation.

Part I

Linear Programming

Linear Programs

Problem

Find a vector $x \in \mathbb{R}^d$ that

$$\begin{array}{lll} \text{maximize/minimize} & \sum_{j=1}^{d} c_j x_j & \\ \text{subject to} & \sum_{j=1}^{d} a_{ij} x_j \leq b_i & \text{for } i = 1 \ldots p \\ & \sum_{j=1}^{d} a_{ij} x_j = b_i & \text{for } i = p+1 \ldots q \\ & \sum_{j=1}^{d} a_{ij} x_j \geq b_i & \text{for } i = q+1 \ldots n \end{array}$$

Linear Programs

Problem

Find a vector $x \in \mathbb{R}^d$ that

$$\begin{array}{ll} \text{maximize/minimize} & \sum_{j=1}^{d} c_j x_j \\ \text{subject to} & \sum_{j=1}^{d} a_{ij} x_j \leq b_i \quad \text{for } i = 1 \ldots p \\ & \sum_{j=1}^{d} a_{ij} x_j = b_i \quad \text{for } i = p+1 \ldots q \\ & \sum_{j=1}^{d} a_{ij} x_j \geq b_i \quad \text{for } i = q+1 \ldots n \end{array}$$

Input is matrix $A = (a_{ij}) \in \mathbb{R}^{n \times d}$, column vector $b = (b_i) \in \mathbb{R}^n$, and row vector $c = (c_j) \in \mathbb{R}^d$

Canonical Form of Linear Programs

Canonical Form

A linear program is in canonical form if it has the following structure

$$\begin{array}{ll} \text{maximize} & \sum_{j=1}^{d} c_j x_j \\ \text{subject to} & \sum_{j=1}^{d} a_{ij} x_j \leq b_i \quad \text{for } i = 1 \ldots n \end{array}$$

Canonical Form of Linear Programs

Canonical Form

A linear program is in canonical form if it has the following structure

$$\begin{array}{ll} \text{maximize} & \sum_{j=1}^{d} c_j x_j \\ \text{subject to} & \sum_{j=1}^{d} a_{ij} x_j \leq b_i \quad \text{for } i = 1 \ldots n \end{array}$$

Conversion to Canonical Form

1. Replace $\sum_j a_{ij} x_j = b_i$ by

$$\sum_j a_{ij} x_j \leq b_i \quad \text{and} \quad -\sum_j a_{ij} x_j \leq -b_i$$

2. Replace $\sum_j a_{ij} x_j \geq b_i$ by $-\sum_j a_{ij} x_j \leq -b_i$

Matrix Representation of Linear Programs

A linear program in canonical form can be written as

$$\begin{array}{ll} \text{maximize} & \boldsymbol{c} \cdot \boldsymbol{x} \\ \text{subject to} & \boldsymbol{A x} \leq \boldsymbol{b} \end{array}$$

where $\boldsymbol{A} = (\boldsymbol{a_{ij}}) \in \mathbb{R}^{\boldsymbol{n} \times \boldsymbol{d}}$, column vector $\boldsymbol{b} = (\boldsymbol{b_i}) \in \mathbb{R}^{\boldsymbol{n}}$, row vector $\boldsymbol{c} = (\boldsymbol{c_j}) \in \mathbb{R}^{\boldsymbol{d}}$, and column vector $\boldsymbol{x} = (\boldsymbol{x_j}) \in \mathbb{R}^{\boldsymbol{d}}$

1. Number of variable is $\boldsymbol{d}$
2. Number of constraints is $\boldsymbol{n}$

Feasible Region and Convexity

Canonical Form

Given $A \in R^{n \times d}$, $b \in R^{n \times 1}$ and $c \in R^{1 \times d}$, find $x \in R^{d \times 1}$

$$\begin{array}{ll} \mathbf{max:} & c \cdot x \\ s.t. & Ax \leq b \end{array}$$

1. Each linear constraint defines a **halfspace**, a convex set.
2. Feasible region, which is an intersection of halfspaces, is a convex **polyhedron**.
3. Optimal value attained at a vertex of the polyhedron.
4. **Simplex method:** starting at a vertex, moves to a neighbor where objective improves. Stops if no such neighbor.

Dual Linear Program

Given a linear program Π in canonical form

$$\begin{array}{ll} \text{maximize} & \sum_{j=1}^{d} c_j x_j \\ \text{subject to} & \sum_{j=1}^{d} a_{ij} x_j \leq b_i \quad i = 1, 2, \ldots n \end{array}$$

the dual $\mathrm{Dual}(\Pi)$ is given by

$$\begin{array}{lll} \text{minimize} & \sum_{i=1}^{n} b_i y_i & \\ \text{subject to} & \sum_{i=1}^{n} y_i a_{ij} = c_j & j = 1, 2, \ldots d \\ & y_i \geq 0 & i = 1, 2, \ldots n \end{array}$$

Dual Linear Program

Given a linear program Π in canonical form

$$\begin{array}{ll} \text{maximize} & \sum_{j=1}^{d} c_j x_j \\ \text{subject to} & \sum_{j=1}^{d} a_{ij} x_j \leq b_i \quad i = 1, 2, \ldots n \end{array}$$

the dual $\mathrm{Dual}(\Pi)$ is given by

$$\begin{array}{ll} \text{minimize} & \sum_{i=1}^{n} b_i y_i \\ \text{subject to} & \sum_{i=1}^{n} y_i a_{ij} = c_j \quad j = 1, 2, \ldots d \\ & y_i \geq 0 \qquad\qquad\quad i = 1, 2, \ldots n \end{array}$$

Proposition

$\mathrm{Dual}(\mathrm{Dual}(\Pi))$ *is equivalent to* Π

Dual Linear Program

Succinct representation..

Given a $A \in \mathbb{R}^{n \times d}$, $b \in \mathbb{R}^n$ and $c \in \mathbb{R}^d$, linear program Π

$$\begin{array}{ll} \text{maximize} & c \cdot x \\ \text{subject to} & Ax \leq b \end{array}$$

the dual $\mathbf{Dual}(\Pi)$ is given by

$$\begin{array}{ll} \text{minimize} & y \cdot b \\ \text{subject to} & yA = c \\ & y \geq 0 \end{array}$$

Proposition

$\mathbf{Dual}(\mathbf{Dual}(\Pi))$ *is equivalent to* Π

Duality Theorem

Theorem (Weak Duality)

If x *is a feasible solution to* Π *and* y *is a feasible solution to* $\mathbf{Dual}(\Pi)$ *then* $c \cdot x \leq y \cdot b$.

Duality Theorem

Theorem (Weak Duality)

If x *is a feasible solution to* Π *and* y *is a feasible solution to* $\mathrm{Dual}(\Pi)$ *then* $c \cdot x \leq y \cdot b$.

Theorem (Strong Duality)

If x^* *is an optimal solution to* Π *and* y^* *is an optimal solution to* $\mathrm{Dual}(\Pi)$ *then* $c \cdot x^* = y^* \cdot b$.

Many applications! Maxflow-Mincut theorem can be deduced from duality.

Strong Duality and Complementary Slackness

Definition (Complementary Slackness)

x feasible in Π and y feasible in $\mathrm{Dual}(\Pi)$, s.t.,

$$\forall i = 1..n, \quad y_i > 0 \Rightarrow (Ax)_i = b_i$$

Theorem

(x^*, y^*) *satisfies complementary Slackness if and only if strong duality holds, i.e.,* $c \cdot x^* = y^* \cdot b$.

Strong Duality and Complementary Slackness

Definition (Complementary Slackness)

x feasible in Π and y feasible in $\mathrm{Dual}(\Pi)$, s.t.,

$$\forall i = 1..n, \quad y_i > 0 \;\Rightarrow\; (Ax)_i = b_i$$

Theorem

(x^*, y^*) satisfies complementary Slackness if and only if strong duality holds, i.e., $c \cdot x^* = y^* \cdot b$.

Proof using Farka's Lemma: Given a set of vectors $A_1, \ldots, A_n$, and a vector c, either c is inside the $cone(A_1, \ldots, A_n)$ or outside it.

Either $\exists y \geq 0$ such that $y^T A = c$ or $\exists x$ such that $Ax \leq 0$ and $c \cdot x > 0$.

Example

Given a graph $G = (V, E)$, write an LP and its dual to find a minimum perfect matching.

Example

Consider the load balancing problem: The input consists of n jobs $J_1, \ldots, J_n$ and an integer m denoting the number of machines. The size of J_i is a non-negative number s_i. The goal is to assign the jobs to machines to minimize the makespan (the largest load of any machine).

- Describe an integer programming formulation for the problem.

Example Contd.

Describe the dual of the LP relaxation of the integer program.

Part II

NP-Completeness

Types of Problems

Decision, Search, and Optimization

1. **Decision problem**. Example: given n, is n prime?.
2. **Search problem**. Example: given n, find a factor of n if it exists.
3. **Optimization problem**. Example: find the smallest prime factor of n.

We focus on **Decision Problems**.

Polynomial Time Reduction

Karp reduction

$X \leq_P Y$: *algorithm* $\mathcal{A}$ reduces problem X to problem Y in polynomial-time:

1. given an instance I_X of X, $\mathcal{A}$ produces an instance I_Y of Y
2. $\mathcal{A}$ runs in time $poly(|I_X|) \Rightarrow |I_Y| = poly(|I_X|)$
3. Answer to I_X YES *iff* answer to I_Y is YES.

Polynomial Time Reduction

Karp reduction

$X \leq_P Y$: *algorithm* $\mathcal{A}$ reduces problem X to problem Y in polynomial-time:

1. given an instance I_X of X, $\mathcal{A}$ produces an instance I_Y of Y
2. $\mathcal{A}$ runs in time $poly(|I_X|) \Rightarrow |I_Y| = poly(|I_X|)$
3. Answer to I_X YES *iff* answer to I_Y is YES.

Consequences:

- poly-time algorithm for $Y \Rightarrow$ poly-time algorithm for X.
- X is "hard" $\Rightarrow$ Y is "hard".

Polynomial Time Reduction

Karp reduction

$X \leq_P Y$: *algorithm* $\mathcal{A}$ reduces problem X to problem Y in polynomial-time:

1. given an instance I_X of X, $\mathcal{A}$ produces an instance I_Y of Y
2. $\mathcal{A}$ runs in time $poly(|I_X|) \Rightarrow |I_Y| = poly(|I_X|)$
3. Answer to I_X YES *iff* answer to I_Y is YES.

Consequences:

- poly-time algorithm for $Y \Rightarrow$ poly-time algorithm for X.
- X is "hard" $\Rightarrow$ Y is "hard".

Note. $X \leq_P Y \quad \not\Longrightarrow \quad Y \leq_P X$

Problems with no known polynomial time algorithms

Problems

1. **Independent Set**
2. **Vertex Cover**
3. **Set Cover**
4. **SAT**
5. **3SAT**

There are of course undecidable problems (no algorithm at all!) but many problems that we want to solve are of similar flavor to the above.

Question: What is common to above problems?

Efficient Checkability

Above problems share the following feature:

Checkability

For any YES instance I_X *of* X *there is a proof/certificate/solution that is of length poly*($|I_X|$) *such that given a proof one can efficiently check that* I_X *is indeed a YES instance.*

Efficient Checkability

Above problems share the following feature:

Checkability

For any YES instance I_X *of* X *there is a proof/certificate/solution that is of length poly(*$|I_X|$*) such that given a proof one can efficiently check that* I_X *is indeed a YES instance.*

Examples:

1. **SAT** formula φ: proof is a satisfying assignment.
2. **Independent Set** in graph G and k:

Efficient Checkability

Above problems share the following feature:

Checkability

For any YES instance I_X *of* X *there is a proof/certificate/solution that is of length poly(*$|I_X|$*) such that given a proof one can efficiently check that* I_X *is indeed a YES instance.*

Examples:

1. **SAT** formula φ: proof is a satisfying assignment.
2. **Independent Set** in graph G and k: a subset S of vertices.

Certifiers

Definition

An algorithm $C(\cdot, \cdot)$ is a **certifier** for problem X if for every $I_x \in X$ there is some string t such that $C(I_x, t) =$ "yes", and conversely, if for some I_x and t, $C(I_x, t) =$ "yes" then $I_x \in X$.
The string t is called a certificate or proof for s.

Certifiers

Definition

An algorithm $C(\cdot, \cdot)$ is a **certifier** for problem X if for every $I_x \in X$ there is some string t such that $C(I_x, t) =$ "yes", and conversely, if for some I_x and t, $C(I_x, t) =$ "yes" then $I_x \in X$.

The string t is called a certificate or proof for s.

Definition (Efficient Certifier.)

A certifier C is an **efficient certifier** for problem X if there is a polynomial $p(\cdot)$ such that for every string s, we have that

- ⋆ $I_x \in X$ if and only if
- ⋆ there is a string t:
  1. $|t| \leq p(|I_x|)$,
  2. $C(I_x, t) =$ "yes",
  3. and C runs in polynomial time in $|I_x|$.

Example: Independent Set

1. Problem: Does $G = (V, E)$ have an independent set of size $\geq k$?
   1. Certificate: Set $S \subseteq V$.
   2. Certifier: Check $|S| \geq k$ and no pair of vertices in S is connected by an edge.

Class NP

NP: languages/problems that have polynomial time certifiers/verifiers

A problem X is **NP-Complete** iff

- X is in **NP**
- X is **NP-Hard**.

X is **NP-Hard** if for every Y in **NP**, $Y \leq_P X$.

Theorem (Cook-Levin)

SAT *is* **NP-Complete**.

Class NP contd

Theorem (Cook-Levin)

SAT *is* **NP-Complete**.

Establish **NP-Complete**ness via reductions:

1. **SAT** is **NP-Complete**.
2. **SAT** $\leq_P$ **3-SAT** and hence 3-SAT is **NP-Complete**.
3. **3-SAT** $\leq_P$ **Independent Set** (which is in **NP**) and hence **Independent Set** is **NP-Complete**.
4. **Clique** is **NP-Complete**
5. **Vertex Cover** is **NP-Complete**
6. **Set Cover** is **NP-Complete**
7. **Hamilton Cycle** and **Hamiltonian Path** are **NP-Complete**
8. **3-Color** is **NP-Complete**

Solving NP-Complete Problems

Proposition

Suppose X *is* **NP-Complete**. *Then* X *can be solved in polynomial time if and only if* $\mathbf{P} = \mathbf{NP}$.

Consequence of proving NP-Completeness

If X is **NP-Complete**

1. Since we believe $\mathbf{P} \neq \mathbf{NP}$,
2. and solving X implies $\mathbf{P} = \mathbf{NP}$.

X is unlikely to be efficiently solvable.

Solving **NP-Complete** Problems

Proposition

Suppose X is **NP-Complete***. Then X can be solved in polynomial time if and only if* $\mathbf{P} = \mathbf{NP}$.

Consequence of proving NP-Completeness

If X is **NP-Complete**

1. Since we believe $\mathbf{P} \neq \mathbf{NP}$,
2. and solving X implies $\mathbf{P} = \mathbf{NP}$.

X is unlikely to be efficiently solvable.

Solving **NP-Complete** Problems

Proposition

Suppose X *is* **NP-Complete**. *Then* X *can be solved in polynomial time if and only if* $\mathbf{P} = \mathbf{NP}$.

Consequence of proving NP-Completeness

If X is **NP-Complete**

1. Since we believe $\mathbf{P} \neq \mathbf{NP}$,
2. and solving X implies $\mathbf{P} = \mathbf{NP}$.

X is unlikely to be efficiently solvable.

At the very least, many smart people before you have failed to find an efficient algorithm for X.
(This is proof by mob opinion — take with a grain of salt.)

3SAT $\leq_P$ Independent Set

The reduction 3SAT $\leq_P$ Independent Set

Input: Given a $3\mathrm{CNF}$ formula φ
Goal: Construct a graph G_φ and number k such that G_φ has an independent set of size k if and only if φ is satisfiable.

3SAT $\leq_P$ Independent Set

The reduction 3SAT $\leq_P$ Independent Set

Input: Given a $3\mathrm{CNF}$ formula φ
Goal: Construct a graph G_φ and number k such that G_φ has an independent set of size k if and only if φ is satisfiable.
G_φ should be constructable in time polynomial in size of φ

3SAT $\leq_P$ Independent Set

The reduction 3SAT $\leq_P$ Independent Set

Input: Given a $3\mathrm{CNF}$ formula φ

Goal: Construct a graph G_φ and number k such that G_φ has an independent set of size k if and only if φ is satisfiable.

G_φ should be constructable in time polynomial in size of φ

Importance of reduction: Although **3SAT** is much more expressive, it can be reduced to a seemingly specialized Independent Set problem.

Interpreting **3SAT**

There are two ways to think about **3SAT**

Interpreting **3SAT**

There are two ways to think about **3SAT**

1. Find a way to assign 0/1 (false/true) to the variables such that the formula evaluates to true, that is each clause evaluates to true.

Interpreting **3SAT**

There are two ways to think about **3SAT**

1. Find a way to assign 0/1 (false/true) to the variables such that the formula evaluates to true, that is each clause evaluates to true.
2. Pick a literal from each clause and find a truth assignment to make all of them true

Interpreting **3SAT**

There are two ways to think about **3SAT**

1. Find a way to assign 0/1 (false/true) to the variables such that the formula evaluates to true, that is each clause evaluates to true.
2. Pick a literal from each clause and find a truth assignment to make all of them true. You will fail if two of the literals you pick are in conflict, i.e., you pick x_i and $\neg x_i$

We will take the second view of **3SAT** to construct the reduction.

The Reduction

1. G_φ will have one vertex for each literal in a clause

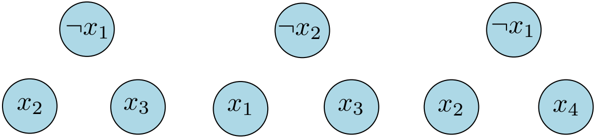


Figure: Graph for
$\varphi = (\neg x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee x_4)$

The Reduction

1. G_φ will have one vertex for each literal in a clause
2. Connect the 3 literals in a clause to form a triangle; the independent set will pick at most one vertex from each clause, which will correspond to the literal to be set to true

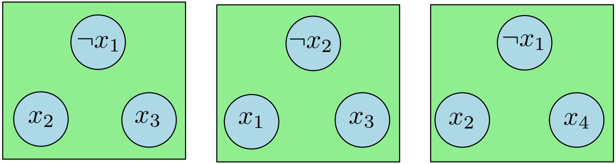


Figure: Graph for
$\varphi = (\neg x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee x_4)$

The Reduction

1. G_φ will have one vertex for each literal in a clause
2. Connect the 3 literals in a clause to form a triangle; the independent set will pick at most one vertex from each clause, which will correspond to the literal to be set to true

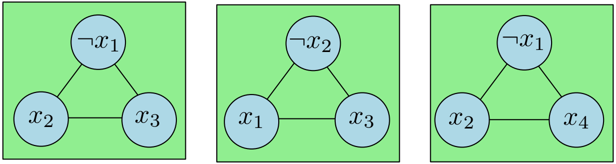


Figure: Graph for
$\varphi = (\neg x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee x_4)$

The Reduction

1. G_φ will have one vertex for each literal in a clause
2. Connect the 3 literals in a clause to form a triangle; the independent set will pick at most one vertex from each clause, which will correspond to the literal to be set to true
3. Connect 2 vertices if they label complementary literals; this ensures that the literals corresponding to the independent set do not have a conflict

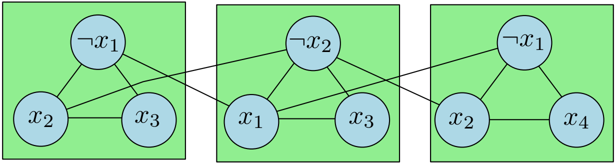


Figure: Graph for
$\varphi = (\neg x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee x_4)$

The Reduction

1. G_φ will have one vertex for each literal in a clause
2. Connect the 3 literals in a clause to form a triangle; the independent set will pick at most one vertex from each clause, which will correspond to the literal to be set to true
3. Connect 2 vertices if they label complementary literals; this ensures that the literals corresponding to the independent set do not have a conflict
4. Take k to be the number of clauses

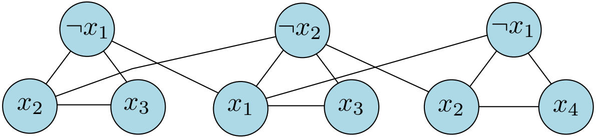


Figure: Graph for
$\varphi = (\neg x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee x_4)$

Correctness

Proposition

φ is satisfiable iff G_φ has an independent set of size k ($=$ number of clauses in φ).

Proof.

$\Rightarrow$ Let a be the truth assignment satisfying φ

Correctness

Proposition

φ is satisfiable iff G_φ has an independent set of size k ($=$ number of clauses in φ).

Proof.

$\Rightarrow$ Let a be the truth assignment satisfying φ

1. Pick one of the vertices, corresponding to true literals under a, from each triangle. This is an independent set of the appropriate size ☐

Correctness (contd)

Proposition

φ is satisfiable iff G_φ has an independent set of size k ($=$ number of clauses in φ).

Proof.

$\Leftarrow$ Let S be an independent set of size k

1. S must contain exactly one vertex from each clause
2. S cannot contain vertices labeled by conflicting clauses
3. Thus, it is possible to obtain a truth assignment that makes in the literals in S true; such an assignment satisfies one literal in every clause ☐

Example – Decision to Computation

Given a black-box to check if a directed graph has a Hamiltonian cycle or not and a graph G, find a Hamiltonian cycle in G.

Example – Decision to Computation

Given a black-box to check if a directed graph has a Hamiltonian cycle or not and a graph G, find a Hamiltonian cycle in G.

Part III

Approximation Algorithms

What is an approximation algorithm?

An algorithm $\mathcal{A}$ for an optimization problem X is an α-approximation algorithm if the following conditions hold:

- for each instance I of X the algorithm $\mathcal{A}$ correctly outputs a valid solution to I
- $\mathcal{A}$ is a polynomial-time algorithm
- Letting $OPT(I)$ and $\mathcal{A}(I)$ denote the values of an optimum solution and the solution output by $\mathcal{A}$ on instances I,
  - If X is a minimization problem: $\mathcal{A}(I)/OPT(I) \leq \alpha$
  - If X is a maximization problem: $OPT(I)/\mathcal{A}(I) \leq \alpha$

Definition ensures that $\alpha \geq 1$

To be formal we need to say $\alpha(n)$ where $n = |I|$ since in some cases the *approximation ratio* depends on the size of the instance.

We saw

- 2 approximation for vertex cover – LP rounding
- $2(1 - 1/m)$ and $3/2$ approximation for the Load Balancing problem, where m is number of machines.
- $\log n$ approximation for setcover
- $3/2$ approximation for undirected TSP
- $\log n$ approximation for directed TSP

Load Balancing

Given n jobs $J_1, J_2, \ldots, J_n$ with sizes $s_1, s_2, \ldots, s_n$ and m identical machines $M_1, \ldots, M_m$ assign jobs to machines to minimize maximum load (also called makespan).

Formally, an assignment is a mapping $f : \{1, 2, \ldots, n\} \to \{1, \ldots, m\}$.

- The load $\ell_f(j)$ of machine M_j under f is $\sum_{i:f(i)=j} s_i$
- Goal is to find f to minimize $\max_j \ell_f(j)$.

Greedy List Scheduling

List-Scheduling

```
    Let $J_1, J_2, \ldots, J_n$ be an ordering of jobs
    for $i = 1$ to $n$ do
        Schedule job $J_i$ on the currently least loaded machine
```

OPT is the optimum load

Lower bounds on OPT:

Greedy List Scheduling

List-Scheduling

```
    Let $J_1, J_2, \ldots, J_n$ be an ordering of jobs
    for $i = 1$ to $n$ do
        Schedule job $J_i$ on the currently least loaded machine
```

OPT is the optimum load

Lower bounds on OPT:

- average load: $OPT \geq \sum_{i=1}^{n} s_i / m$. Why?
- maximum job size: $OPT \geq \max_{i=1}^{n} s_i$. Why?

Analysis of Greedy List Scheduling

Theorem

Let L *be makespan of Greedy List Scheduling on a given instance. Then* $L \leq 2(1 - 1/m)OPT$ *where* OPT *is the optimum makespan for that instance.*

Analysis of Greedy List Scheduling

Theorem

Let L be makespan of Greedy List Scheduling on a given instance. Then $L \leq 2(1 - 1/m)OPT$ where OPT is the optimum makespan for that instance.

- Let M_h be the machine which achieves the load L for Greedy List Scheduling.
- Let J_i be the job that was last scheduled on M_h.
- Why was J_i scheduled on M_h? It means that M_h was the least loaded machine when J_i was considered. Implies all machines had load at least $L - s_i$ at that time.

Analysis continued

Lemma

$L - s_i \leq (\sum_{\ell=1}^{i-1} s_\ell)/m.$

Analysis continued

Lemma

$L - s_i \leq (\sum_{\ell=1}^{i-1} s_\ell)/m$.

But then

$$\begin{aligned} L &\leq (\sum_{\ell=1}^{i-1} s_\ell)/m + s_i \\ &\leq (\sum_{\ell=1}^{n} s_\ell)/m + (1 - \frac{1}{m})s_i \\ &\leq OPT + (1 - \frac{1}{m})OPT \\ &\leq (2 - \frac{2}{m})OPT \end{aligned}$$

Ordering jobs from largest to smallest

Obvious heuristic: Order jobs in decreasing size order and then use Greedy.

$$s_1 \geq s_2 \geq \cdots \geq s_n$$

Ordering jobs from largest to smallest

Obvious heuristic: Order jobs in decreasing size order and then use Greedy.

$$s_1 \geq s_2 \geq \cdots \geq s_n$$

Does it lead to an improved performance in the worst case? How much?

Ordering jobs from largest to smallest

Obvious heuristic: Order jobs in decreasing size order and then use Greedy.

$$s_1 \geq s_2 \geq \cdots \geq s_n$$

Does it lead to an improved performance in the worst case? How much?

Theorem

Greedy List Scheduling with jobs sorted from largest to smallest gives a $\mathbf{3/2}$*-approximation and this is essentially tight.*

Ordering jobs from largest to smallest

Obvious heuristic: Order jobs in decreasing size order and then use Greedy.

$$s_1 \geq s_2 \geq \cdots \geq s_n$$

Does it lead to an improved performance in the worst case? How much?

Theorem

Greedy List Scheduling with jobs sorted from largest to smallest gives a $3/2$*-approximation and this is essentially tight.*

New lower bound: $s_m + s_{m+1} \leq OPT$.

Traveling Salesman/Salesperson Problem (TSP)

Perhaps the most famous discrete optimization problem

Input: A (un)directed complete graph $G = (V, E)$ with edge costs $c : E \to \mathbb{R}_+$.
Goal: Find a Hamiltonian Cycle of minimum total edge cost

Traveling Salesman/Salesperson Problem (TSP)

Perhaps the most famous discrete optimization problem

Input: A (un)directed complete graph $G = (V, E)$ with edge costs $c : E \to \mathbb{R}_+$.
Goal: Find a Hamiltonian Cycle of minimum total edge cost

Observation: Inapproximable to any polynomial factor.

Traveling Salesman/Salesperson Problem (TSP)

Perhaps the most famous discrete optimization problem

Input: A (un)directed complete graph $G = (V, E)$ with edge costs $c : E \to \mathbb{R}_+$.
Goal: Find a Hamiltonian Cycle of minimum total edge cost

Observation: Inapproximable to any polynomial factor.

Metric-TSP: $G = (V, E)$ is a **complete graph** and c defines a metric space. $c(u, v) = c(v, u)$ for all u, v and $c(u, w) \leq c(u, v) + c(v, w)$ for all u, v, w.

Theorem

Metric-TSP is NP-Hard.

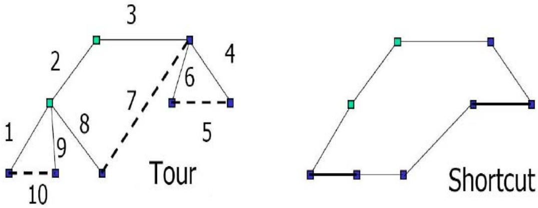
Metric-TSP: closed walk

Another interpretation of Metric-TSP: Given $G = (V, E)$ with edges costs c, find a tour of minimum cost that visits all vertices but can visit a vertex more than once – A closed walk.

Metric-TSP: closed walk

Another interpretation of Metric-TSP: Given $G = (V, E)$ with edges costs c, find a tour of minimum cost that visits all vertices but can visit a vertex more than once – A closed walk.

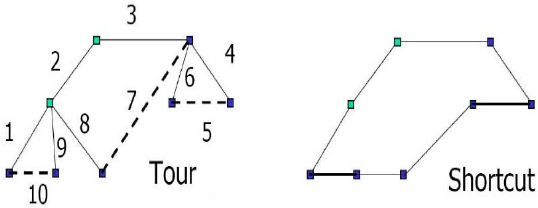
Because, any such tour can be converted in to a simple cycle of smaller cost by adding "short-cuts".



Metric-TSP: closed walk

Another interpretation of Metric-TSP: Given $G = (V, E)$ with edges costs c, find a tour of minimum cost that visits all vertices but can visit a vertex more than once – A closed walk.

Because, any such tour can be converted in to a simple cycle of smaller cost by adding "short-cuts".



Essentially need to find an Eulerian graph.

Christofides Heuristic: 3/2 approximation

```
Christofides-Heuristic($G = (V, E), c$)
    Compute a minimum spanning tree (MST) $T$ in $G$
```

Christofides Heuristic: 3/2 approximation

Christofides-Heuristic$(G = (V, E), c)$

```
    Compute a minimum spanning tree (MST) $T$ in $G$
    Let $S$ be vertices of odd degree in $T$ (Note: $|S|$ is even)
```

Christofides Heuristic: 3/2 approximation

Christofides-Heuristic$(G = (V, E), c)$

```
    Compute a minimum spanning tree (MST) $T$ in $G$
    Let $S$ be vertices of odd degree in $T$ (Note:  $|S|$ is even)
    Find a minimum cost matching $M$ on $S$ in $G$
```

Christofides Heuristic: $3/2$ approximation

Christofides-Heuristic$(G = (V, E), c)$

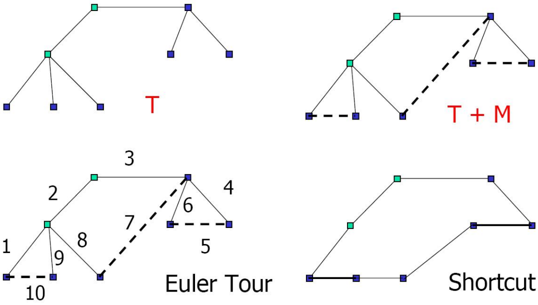
```
    Compute a minimum spanning tree (MST) $T$ in $G$
    Let $S$ be vertices of odd degree in $T$ (Note: $|S|$ is even)
    Find a minimum cost matching $M$ on $S$ in $G$
    Add $M$ to $T$ to obtain Eulerian graph $H$
```

Christofides Heuristic: $3/2$ approximation

```
**Christofides-Heuristic**($G = (V, E), c$)
    Compute a minimum spanning tree (MST) $T$ in $G$
    Let $S$ be vertices of odd degree in $T$ (Note: $|S|$ is even)
    Find a minimum cost matching $M$ on $S$ in $G$
    Add $M$ to $T$ to obtain Eulerian graph $H$
    An Eulerian tour of $H$ gives a tour of $G$
    Obtain Hamiltonian cycle by shortcutting the tour
```

Christofides Heuristic: 3/2 approximation

```
Christofides-Heuristic($G = (V, E), c$)
    Compute a minimum spanning tree (MST) $T$ in $G$
    Let $S$ be vertices of odd degree in $T$ (Note: $|S|$ is even)
    Find a minimum cost matching $M$ on $S$ in $G$
    Add $M$ to $T$ to obtain Eulerian graph $H$
    An Eulerian tour of $H$ gives a tour of $G$
    Obtain Hamiltonian cycle by shortcutting the tour
```



Analysis of Christofides Heuristic

Main lemma:

Lemma

$c(M) \leq OPT/2$.

Analysis of Christofides Heuristic

Main lemma:

Lemma

$c(M) \leq OPT/2$.

Assuming lemma:

Theorem

Christofides heuristic returns a tour of cost at most $3OPT/2$.

Proof.

$c(H) = c(T) + c(M) \leq OPT + OPT/2 \leq 3OPT/2$. Cost of tour is at most cost of H. ☐

Example – Randomized Approximation Scheme

Consider the LP relaxation for Set Cover. Let x_i be the variable in the relaxation for set S_i. Suppose x^* is an optimum solution to the LP relaxation. Define $y_i = \min\{1, 2\ln n \cdot x_i^*\}$ for each set S_i. Pick each set S_i independently with probability y_i.

- Prove that the expected weight of the sets chosen is at most $2\ln n \cdot OPT$.

Contd.

Prove that the probability that any fixed element in the universe is *not* covered by the chosen sets is at most $1/n^2$.

Contd.

Prove that, with probability at least $\mathbf{1 - 1/n}$ all the elements of the universe are covered by the chosen sets. *Hint:* Use union bound.

Contd.

Prove that with probability $1/2 - 1/n$ the algorithm outputs a set cover for the universe whose weight at most $4 \ln n \cdot OPT$ where OPT is the weight of an optimum Set Cover. *Hint:* Use Markov's inequality.