Learning Goals

- Define approximation algorithms and their approximation ratios
- Understand the 2-approximation for Vertex Cover
- Define makespan
- Analyze the 2-approximation algorithm for Load Balancing
- Analyze the 1.5-approximation algorithm for Load Balancing
- Design simple greedy approximation algorithms

Ways to deal with NP-hard problems

- Relatively fast exponential-time algorithms
  - Typically with a running time that has an exponential dependence on some *parameter* of the problem
  - Practical when this parameter is small.

Ways to deal with NP-hard problems

- Relatively fast exponential-time algorithms
  - Typically with a running time that has an exponential dependence on some *parameter* of the problem
  - Practical when this parameter is small.
  - Known as *fixed-parmameter tractable* algorithms (Chapter 10)

Ways to deal with NP-hard problems

- Relatively fast exponential-time algorithms
    - Typically with a running time that has an exponential dependence on some *parameter* of the problem
    - Practical when this parameter is small.
    - Known as *fixed-parmameter tractable* algorithms (Chapter 10)
- Poly-time algorithms for NP-hard problems in special cases

Ways to deal with NP-hard problems

- Relatively fast exponential-time algorithms
  - Typically with a running time that has an exponential dependence on some *parameter* of the problem
  - Practical when this parameter is small.
  - Known as *fixed-parmameter tractable* algorithms (Chapter 10)
- Poly-time algorithms for NP-hard problems in special cases
- In general we cannot hope to get optimal solutions in practically acceptable time, and have to run *heuristic* algorithms.

Motivating Approximation Algorithms

- How do we justify heuristic algorithms? How do we compare one heuristic with another?

Motivating Approximation Algorithms

- How do we justify heuristic algorithms? How do we compare one heuristic with another?
- A worst-case analysis framework: show that an algorithm's output on *any* instance is not far from the *optimal*.

Motivating Approximation Algorithms

- How do we justify heuristic algorithms? How do we compare one heuristic with another?
- A worst-case analysis framework: show that an algorithm's output on *any* instance is not far from the *optimal*.
- How do we measure how much worse an output is compared with an optimal solution?

Motivating Approximation Algorithms

- How do we justify heuristic algorithms? How do we compare one heuristic with another?
- A worst-case analysis framework: show that an algorithm's output on *any* instance is not far from the *optimal*.
- How do we measure how much worse an output is compared with an optimal solution?
  - Multiplicative ratio between the objectives

Definition of Approximation Algorithms

Definition

For a maximization problem Q that asks to maximize the value of an objective, an algorithm $\mathcal{A}$ is said to be an *α-approximation* algorithm if, on any instance of Q, we have $\alpha \cdot \mathrm{ALG} \geq \mathrm{OPT}$, where ALG is the objective value of $\mathcal{A}$'s output (on this instance), and OPT the objective value of an optimal solution.

Definition of Approximation Algorithms

Definition

For a maximization problem Q that asks to maximize the value of an objective, an algorithm $\mathcal{A}$ is said to be an *α-approximation* algorithm if, on any instance of Q, we have $\alpha \cdot \text{ALG} \geq \text{OPT}$, where ALG is the objective value of $\mathcal{A}$'s output (on this instance), and OPT the objective value of an optimal solution.

In this definition, $\alpha \geq 1$ is called the *approximation ratio* of $\mathcal{A}$.

Definition of Approximation Algorithms

Definition

For a maximization problem Q that asks to maximize the value of an objective, an algorithm $\mathcal{A}$ is said to be an *α-approximation* algorithm if, on any instance of Q, we have $\alpha \cdot \text{ALG} \geq \text{OPT}$, where ALG is the objective value of $\mathcal{A}$'s output (on this instance), and OPT the objective value of an optimal solution.

In this definition, $\alpha \geq 1$ is called the *approximation ratio* of $\mathcal{A}$.
We also say that algorithm $\mathcal{A}$ *α-approximates* the objective.

Simple Example 1: Independent Set

Example (Independent Set)

Input: Given an undirected simple graph $G = (V, E)$.
Output: The size of an independent set of maximum cardinality.

Simple Example 1: Independent Set

Example (Independent Set)

Input: Given an undirected simple graph $G = (V, E)$.
Output: The size of an independent set of maximum cardinality.

Algorithm: Pick an arbitrary node. This singleton set is an n-approximation.

Simple Example 1: Independent Set

Example (Independent Set)

Input: Given an undirected simple graph $G = (V, E)$.
Output: The size of an independent set of maximum cardinality.

Algorithm: Pick an arbitrary node. This singleton set is an n-approximation.
(Asymptotically this is in fact the best possible unless $\mathrm{P} = \mathrm{NP}$. Showing this is way beyond the scope of this class.)

Simple Example 2: Vertex Cover

Example

Input: An undirected simple graph $G = (V, E)$.
Output: The size of a vertex cover of minimum cardinality.

Simple Example 2: Vertex Cover

Example

Input: An undirected simple graph $G = (V, E)$.
Output: The size of a vertex cover of minimum cardinality.

Algorithm: Initialize $S = \emptyset$. Pick an arbitrary edge and add both its endpoints to the cover. Then remove from the graph the two vertices and all edges incident to either. Repeat until there is no edge left. Return $|S|$.

Simple Example 2: Vertex Cover

Example

Input: An undirected simple graph $G = (V, E)$.
Output: The size of a vertex cover of minimum cardinality.

Algorithm: Initialize $S = \emptyset$. Pick an arbitrary edge and add both its endpoints to the cover. Then remove from the graph the two vertices and all edges incident to either. Repeat until there is no edge left. Return $|S|$.

Claim

This algorithm gives a 2-approximation for Vertex Cover.

Proof of Vertex Cover 2-Approximation

Claim

This algorithm gives a 2-approximation for Vertex Cover.

Proof.

By the end of the algorithm, S is a vertex cover, because every edge in G was removed because it is incident to some node in S.

Proof of Vertex Cover 2-Approximation

Claim

This algorithm gives a 2-approximation for Vertex Cover.

Proof.

By the end of the algorithm, S is a vertex cover, because every edge in G was removed because it is incident to some node in S.
The set of $\frac{|S|}{2}$ edges picked by the algorithm is a matching. Therefore any vertex cover has size at least $\frac{|S|}{2}$.

Proof of Vertex Cover 2-Approximation

Claim

This algorithm gives a 2-approximation for Vertex Cover.

Proof.

By the end of the algorithm, S is a vertex cover, because every edge in G was removed because it is incident to some node in S.

The set of $\frac{|S|}{2}$ edges picked by the algorithm is a matching. Therefore any vertex cover has size at least $\frac{|S|}{2}$.

Formally, $\mathrm{OPT} \geq \frac{|S|}{2} \Rightarrow |S| \leq 2\,\mathrm{OPT}$. $\square$

Serious example: Load balancing

- We have m machines and n tasks. Each task has a processing time t_j. We need to assign tasks to machines. The machines work in parallel.

Serious example: Load balancing

- We have m machines and n tasks. Each task has a processing time t_j. We need to assign tasks to machines. The machines work in parallel.
- The *makespan* is the amount of time that elapses from the start of work to the end, i.e. till all machines finish the jobs assigned to them.

Serious example: Load balancing

- We have m machines and n tasks. Each task has a processing time t_j. We need to assign tasks to machines. The machines work in parallel.
- The *makespan* is the amount of time that elapses from the start of work to the end, i.e. till all machines finish the jobs assigned to them.
- Formally, let S_i be the set of jobs assigned to machine i, then the makespan is $\max_i \sum_{j \in S_i} t_j$.

Serious example: Load balancing

- We have m machines and n tasks. Each task has a processing time t_j. We need to assign tasks to machines. The machines work in parallel.
- The *makespan* is the amount of time that elapses from the start of work to the end, i.e. till all machines finish the jobs assigned to them.
- Formally, let S_i be the set of jobs assigned to machine i, then the makespan is $\max_i \sum_{j \in S_i} t_j$.
- We need to assign jobs to the machines to minimize the makespan.

Serious example: Load balancing

- We have m machines and n tasks. Each task has a processing time t_j. We need to assign tasks to machines. The machines work in parallel.
- The *makespan* is the amount of time that elapses from the start of work to the end, i.e. till all machines finish the jobs assigned to them.
- Formally, let S_i be the set of jobs assigned to machine i, then the makespan is $\max_i \sum_{j \in S_i} t_j$.
- We need to assign jobs to the machines to minimize the makespan.
- The problem is NP-hard. (Reduction?)

Serious example: Load balancing

- We have m machines and n tasks. Each task has a processing time t_j. We need to assign tasks to machines. The machines work in parallel.
- The *makespan* is the amount of time that elapses from the start of work to the end, i.e. till all machines finish the jobs assigned to them.
- Formally, let S_i be the set of jobs assigned to machine i, then the makespan is $\max_i \sum_{j \in S_i} t_j$.
- We need to assign jobs to the machines to minimize the makespan.
- The problem is NP-hard. (Reduction?)
  - Number Partition can be solved by an oracle that solves the two machine case.

Greedy algorithm

- A natural algorithm: consider the jobs one by one in an arbitrary order.

Greedy algorithm

- A natural algorithm: consider the jobs one by one in an arbitrary order.
- For task j, if jobs assigned to machine i take least time to process, assign task j to machine i.

Greedy algorithm

- A natural algorithm: consider the jobs one by one in an arbitrary order.
- For task j, if jobs assigned to machine i take least time to process, assign task j to machine i.
- Running time obviously polynomial.

Greedy algorithm

- A natural algorithm: consider the jobs one by one in an arbitrary order.
- For task j, if jobs assigned to machine i take least time to process, assign task j to machine i.
- Running time obviously polynomial.

Theorem

The above greey algorithm gives a 2-approximation to the makespan.

Proof Strategy (for all approximation algorithms)

General proof strategy:

- In order to compare with the optimal, we need to know something about the optimal solution.

Proof Strategy (for all approximation algorithms)

General proof strategy:

- In order to compare with the optimal, we need to know something about the optimal solution.
- For NP-hard problems, we in general don't have a clean characterization of the optimal solution.

Proof Strategy (for all approximation algorithms)

General proof strategy:

- In order to compare with the optimal, we need to know something about the optimal solution.
- For NP-hard problems, we in general don't have a clean characterization of the optimal solution.
- But we can *bound* the optimal, either using given information or using steps from the algorithm.

Proof for the Greedy Algorithm

We need to lower bound OPT, the optimal makespan:

Proof for the Greedy Algorithm

We need to lower bound OPT, the optimal makespan:

Proposition (Makespan no less than longest job)

$\mathrm{OPT} \geq \max_j t_j$.

Proof for the Greedy Algorithm

We need to lower bound OPT, the optimal makespan:

Proposition (Makespan no less than longest job)

OPT $\geq \max_j t_j$.

Proposition (Makespan no less than average lengths)

For any subset S of jobs, OPT $\geq \frac{1}{m} \sum_{j \in S} t_j$.

Putting things together

- Let S_i be the set of tasks assigned to machine i by our algorithm.
- If $|S_i| = 1$, its execution time is no more than OPT by Proposition 1.

Putting things together

- Let S_i be the set of tasks assigned to machine i by our algorithm.
- If $|S_i| = 1$, its execution time is no more than OPT by Proposition 1.
- If $|S_i| \geq 2$, suppose the last job added in is j:
    - $t_j \leq \text{OPT}$ by Proposition 2.

Putting things together

- Let S_i be the set of tasks assigned to machine i by our algorithm.
- If $|S_i| = 1$, its execution time is no more than OPT by Proposition 1.
- If $|S_i| \geq 2$, suppose the last job added in is j:
  - $t_j \leq \text{OPT}$ by Proposition 2.
  - $\sum_{k \in S_i - \{j\}} t_k$ was the smallest when task j was added.

Putting things together

- Let S_i be the set of tasks assigned to machine i by our algorithm.
- If $|S_i| = 1$, its execution time is no more than OPT by Proposition 1.
- If $|S_i| \geq 2$, suppose the last job added in is j:
  - $t_j \leq \text{OPT}$ by Proposition 2.
  - $\sum_{k \in S_i - \{j\}} t_k$ was the smallest when task j was added.
  - Then $\sum_{k \in S_i - \{j\}} t_k$ was no more than the "machine average" over the jobs that have been assigned when the algorithm considered task j.

Putting things together

- Let S_i be the set of tasks assigned to machine i by our algorithm.
- If $|S_i| = 1$, its execution time is no more than OPT by Proposition 1.
- If $|S_i| \geq 2$, suppose the last job added in is j:
  - $t_j \leq \mathrm{OPT}$ by Proposition 2.
  - $\sum_{k \in S_i - \{j\}} t_k$ was the smallest when task j was added.
  - Then $\sum_{k \in S_i - \{j\}} t_k$ was no more than the "machine average" over the jobs that have been assigned when the algorithm considered task j.
  - Hence $\sum_{k \in S_i - \{j\}} t_k \leq \mathrm{OPT}$.
- Therefore $\sum_{k \in S_i} t_k \leq 2\,\mathrm{OPT}$ for all i.

Tightness of the analysis

The above analysis is tight. Intuitively, a good algorithm should be holistic, i.e., should consider the weights of all jobs before making individual decisions.

Tightness of the analysis

The above analysis is tight. Intuitively, a good algorithm should be holistic, i.e., should consider the weights of all jobs before making individual decisions.

- For two machines, if the jobs have weight $\frac{1}{2}, \frac{1}{2}, 1$, then the optimal makespan is 1 but the algorithm's makespan is $\frac{3}{2}$.
- For any number of machines $m > 0$, if we have $m(m-1)$ jobs with weight 1, and one with weight m, which is considered the last.
    - The algorithm's makespan is $m - 1 + m = 2m - 1$.

Tightness of the analysis

The above analysis is tight. Intuitively, a good algorithm should be holistic, i.e., should consider the weights of all jobs before making individual decisions.

- For two machines, if the jobs have weight $\frac{1}{2}, \frac{1}{2}, 1$, then the optimal makespan is 1 but the algorithm's makespan is $\frac{3}{2}$.
- For any number of machines $m > 0$, if we have $m(m-1)$ jobs with weight 1, and one with weight m, which is considered the last.
  - The algorithm's makespan is $m - 1 + m = 2m - 1$.
  - The optimal has makespan m.

Tightness of the analysis

The above analysis is tight. Intuitively, a good algorithm should be holistic, i.e., should consider the weights of all jobs before making individual decisions.

- For two machines, if the jobs have weight $\frac{1}{2}, \frac{1}{2}, 1$, then the optimal makespan is 1 but the algorithm's makespan is $\frac{3}{2}$.
- For any number of machines $m > 0$, if we have $m(m-1)$ jobs with weight 1, and one with weight m, which is considered the last.
  - The algorithm's makespan is $m - 1 + m = 2m - 1$.
  - The optimal has makespan m.
  - Hence approximation ratio is no better than $2 - \frac{1}{m} \to 2 (m \to \infty)$.

Improving the Greedy Algorithm

- In the tight example, the greedy algorithm did badly because it doesn't foresee a large task coming at last.

Improving the Greedy Algorithm

- In the tight example, the greedy algorithm did badly because it doesn't foresee a large task coming at last.
- This motivates considering larger jobs first: run the greedy algorithm just as before, but consider the tasks in decreasing lengths.

Improving the Greedy Algorithm

- In the tight example, the greedy algorithm did badly because it doesn't foresee a large task coming at last.
- This motivates considering larger jobs first: run the greedy algorithm just as before, but consider the tasks in decreasing lengths.

Theorem

The improved greedy algorithm $\frac{3}{2}$-approximates the makespan.

Theorem

The improved greedy algorithm $\frac{3}{2}$-approximates the makespan.

Proof.

Proof: Have a tighter bound on OPT: say $t_1 \geq t_2 \geq \ldots \geq t_n$, then $\text{OPT} \geq 2t_{m+1}$.

Theorem

The improved greedy algorithm $\frac{3}{2}$-approximates the makespan.

Proof.

Proof: Have a tighter bound on OPT: say $t_1 \geq t_2 \geq \ldots \geq t_n$, then $\mathrm{OPT} \geq 2t_{m+1}$.

If machine i finishes the last in the algorithm's solution:

- If i has only one job, then the makespan is equal to optimal.
- If i has at least two jobs, the last job has an index at least $m+1$. Therefore its weight is at most $\frac{1}{2}$ OPT. (Previously we bounded this is using OPT.)

Theorem

The improved greedy algorithm $\frac{3}{2}$-approximates the makespan.

Proof.

Proof: Have a tighter bound on OPT: say $t_1 \geq t_2 \geq \ldots \geq t_n$, then $\mathrm{OPT} \geq 2t_{m+1}$.

If machine i finishes the last in the algorithm's solution:

- If i has only one job, then the makespan is equal to optimal.
- If i has at least two jobs, the last job has an index at least $m+1$. Therefore its weight is at most $\frac{1}{2}$ OPT. (Previously we bounded this is using OPT.)
- The rest of the jobs can be bounded the same as before.
- Putting things together, the makespan is at most $\frac{3}{2}$ OPT.

□