Approximation Algorithms: the concept

Ways to deal with NP-hard problems:

Approximation Algorithms: the concept

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.

Approximation Algorithms: the concept

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 in the literature as *fixed-parmameter tractable* algorithms.

Approximation Algorithms: the concept

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 in the literature as *fixed-parmameter tractable* algorithms.
- Poly-time algorithms for NP-hard problems in special cases

Approximation Algorithms: the concept

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 in the literature as *fixed-parmameter tractable* algorithms.
- 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.

Approximation Algorithms: the concept

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 in the literature as *fixed-parmameter tractable* algorithms.
- 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.
- But how do we justify heuristic algorithms? How do we compare one heuristic with another?

Approximation Algorithms: the concept

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 in the literature as *fixed-parmameter tractable* algorithms.
- 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.
- But how do we justify heuristic algorithms? How do we compare one heuristic with another?
- One particular framework that inherits the worst-case analysis we have done so far: show that an algorithm's output on *any* instance is not *far* from the optimal.

Approximation Ratios

Measure the distance of an algorithm's output from the optimal solution, for optimization problems: look at the ratio between the two quantities.

Approximation Ratios

Measure the distance of an algorithm's output from the optimal solution, for optimization problems: look at the ratio between the two quantities.

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, $\alpha \cdot \mathrm{ALG} \geq \mathrm{OPT}$, where ALG is the objective value of $\mathcal{A}$'s output (on this instance), and OPT the value of the optimal solution.

Approximation Ratios

Measure the distance of an algorithm's output from the optimal solution, for optimization problems: look at the ratio between the two quantities.

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, $\alpha \cdot \mathrm{ALG} \geq \mathrm{OPT}$, where ALG is the objective value of $\mathcal{A}$'s output (on this instance), and OPT the value of the optimal solution.

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

Approximation Ratios

Measure the distance of an algorithm's output from the optimal solution, for optimization problems: look at the ratio between the two quantities.

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, $\alpha \cdot \mathrm{ALG} \geq \mathrm{OPT}$, where ALG is the objective value of $\mathcal{A}$'s output (on this instance), and OPT the value of the optimal solution.

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

Approximation Ratios

Measure the distance of an algorithm's output from the optimal solution, for optimization problems: look at the ratio between the two quantities.

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, $\alpha \cdot \mathrm{ALG} \geq \mathrm{OPT}$, where ALG is the objective value of $\mathcal{A}$'s output (on this instance), and OPT the value of the optimal solution.

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

Example

Independent set: pick an arbitrary node and stop. This is an *n*-approximation.

Approximation Ratios

Measure the distance of an algorithm's output from the optimal solution, for optimization problems: look at the ratio between the two quantities.

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, $\alpha \cdot \mathrm{ALG} \geq \mathrm{OPT}$, where ALG is the objective value of $\mathcal{A}$'s output (on this instance), and OPT the value of the optimal solution.

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

Example

Independent set: pick an arbitrary node and stop. This 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.)

First (serious) examle: 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.

First (serious) examle: 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.

First (serious) examle: 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 $\max_i \sum_{j \in S_i} t_j$.

First (serious) examle: 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.

First (serious) examle: 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?)

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.

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.

The above analysis is tight: for any m, there is an instance with m machines for which the approximation ratio of the greedy algorithm is at least $2 - \frac{1}{m}$.

Proof of the approximation ratio

General proof strategy:

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

Proof of the approximation ratio

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 of the approximation ratio

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 of the approximation ratio

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.

Let's lower bound OPT, the optimal makespan:

Proof of the approximation ratio

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.

Let's lower bound OPT, the optimal makespan:

Proposition (Makespan no less than longest job)

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

Proof of the approximation ratio

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.

Let's lower bound OPT, the optimal makespan:

Proposition (Makespan no less than longest job)

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

Proposition (Makespan no less than average lengths)

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

Final Analysis

- 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.

Final Analysis

- 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:
    - $t_j \leq$ OPT by Proposition 2.

Final Analysis

- 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.

Final Analysis

- 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.

Final Analysis

- 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.
  - Hence $\sum_{k \in S_i - \{j\}} t_k \leq \text{OPT}$.
- Therefore $\sum_{k \in S_i} t_k \leq 2\,\text{OPT}$ for all i.

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.

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.

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