DM811 – Autumn 2011

Heuristics for Combinatorial Optimization

Compendium

Basic Concepts in Algorithmics

Marco Chiarandini

Department of Mathematics & Computer Science
University of Southern Denmark

Outline

Outline

Outline

1. Basic Concepts from Previous Courses
   - Graphs
   - Notation and runtime
   - Machine model
   - Pseudo-code
   - Computational Complexity
   - Analysis of Algorithms

Graphs

Graphs are combinatorial structures useful to model several applications

Terminology:

- $G = (V, E)$, $E \subseteq V \times V$, vertices, edges, $n = |V|$, $m = |E|$, digraphs, undirected graphs, subgraph, induced subgraph

Graphs

Graphs are combinatorial structures useful to model several applications

Terminology:

- $G = (V, E)$, $E \subseteq V \times V$, vertices, edges, $n = |V|, m = |E|$, digraphs, undirected graphs, subgraph, induced subgraph
- $e = (u, v) \in E$, e incident on u and v; u, v adjacent, edge weight or cost

Graphs

Graphs are combinatorial structures useful to model several applications

Terminology:

- $G = (V, E)$, $E \subseteq V \times V$, vertices, edges, $n = |V|, m = |E|$, digraphs, undirected graphs, subgraph, induced subgraph
- $e = (u, v) \in E$, e incident on u and v; u, v adjacent, edge weight or cost
- particular cases often omitted: self-loops, multiple parallel edges

Graphs

Graphs are combinatorial structures useful to model several applications

Terminology:

- $G = (V, E)$, $E \subseteq V \times V$, vertices, edges, $n = |V|, m = |E|$, digraphs, undirected graphs, subgraph, induced subgraph
- $e = (u, v) \in E$, e incident on u and v; u, v adjacent, edge weight or cost
- particular cases often omitted: self-loops, multiple parallel edges
- degree, δ, Δ, outdegree, indegree

Graphs

Graphs are combinatorial structures useful to model several applications

Terminology:

- $G = (V, E)$, $E \subseteq V \times V$, vertices, edges, $n = |V|, m = |E|$, digraphs, undirected graphs, subgraph, induced subgraph
- $e = (u, v) \in E$, e incident on u and v; u, v adjacent, edge weight or cost
- particular cases often omitted: self-loops, multiple parallel edges
- degree, δ, Δ, outdegree, indegree
- path $P =< v_0, v_1, \ldots, v_k >$, $(v_0, v_1) \in E, \ldots, (v_{k-1}, v_k) \in E$, $< v_0, v_1 >$ has length 2, $< v_0, v_1, v_2, v_0 >$ cycle, walk, path

Graphs

Graphs are combinatorial structures useful to model several applications

Terminology:

- $G = (V, E)$, $E \subseteq V \times V$, vertices, edges, $n = |V|, m = |E|$, digraphs, undirected graphs, subgraph, induced subgraph
- $e = (u, v) \in E$, e incident on u and v; u, v adjacent, edge weight or cost
- particular cases often omitted: self-loops, multiple parallel edges
- degree, δ, Δ, outdegree, indegree
- path $P =< v_0, v_1, \ldots, v_k >$, $(v_0, v_1) \in E, \ldots, (v_{k-1}, v_k) \in E$, $< v_0, v_1 >$ has length 2, $< v_0, v_1, v_2, v_0 >$ cycle, walk, path
- directed acyclic digraph

Graphs

Graphs are combinatorial structures useful to model several applications

Terminology:

- $G = (V, E)$, $E \subseteq V \times V$, vertices, edges, $n = |V|, m = |E|$, digraphs, undirected graphs, subgraph, induced subgraph
- $e = (u, v) \in E$, e incident on u and v; u, v adjacent, edge weight or cost
- particular cases often omitted: self-loops, multiple parallel edges
- degree, δ, Δ, outdegree, indegree
- path $P =< v_0, v_1, \ldots, v_k >$, $(v_0, v_1) \in E, \ldots, (v_{k-1}, v_k) \in E$, $< v_0, v_1 >$ has length 2, $< v_0, v_1, v_2, v_0 >$ cycle, walk, path
- directed acyclic digraph
- digraph strongly connected ($\forall u, v \;\; \exists (uv)$-path), strongly connected components

Graphs

Graphs are combinatorial structures useful to model several applications

Terminology:

- $G = (V, E)$, $E \subseteq V \times V$, vertices, edges, $n = |V|, m = |E|$, digraphs, undirected graphs, subgraph, induced subgraph
- $e = (u, v) \in E$, e incident on u and v; u, v adjacent, edge weight or cost
- particular cases often omitted: self-loops, multiple parallel edges
- degree, δ, Δ, outdegree, indegree
- path $P =< v_0, v_1, \ldots, v_k >$, $(v_0, v_1) \in E, \ldots, (v_{k-1}, v_k) \in E$, $< v_0, v_1 >$ has length 2, $< v_0, v_1, v_2, v_0 >$ cycle, walk, path
- directed acyclic digraph
- digraph strongly connected ($\forall u, v \ \exists (uv)$-path), strongly connected components
- G is a tree ($\exists$ path between any two vertices) $\iff$ G is connected and has $n - 1$ edges $\iff$ G is connected and contains no cycles.

Graphs

Graphs are combinatorial structures useful to model several applications

Terminology:

- $G = (V, E)$, $E \subseteq V \times V$, vertices, edges, $n = |V|, m = |E|$, digraphs, undirected graphs, subgraph, induced subgraph
- $e = (u, v) \in E$, e incident on u and v; u, v adjacent, edge weight or cost
- particular cases often omitted: self-loops, multiple parallel edges
- degree, δ, Δ, outdegree, indegree
- path $P =< v_0, v_1, \ldots, v_k >$, $(v_0, v_1) \in E, \ldots, (v_{k-1}, v_k) \in E$, $< v_0, v_1 >$ has length 2, $< v_0, v_1, v_2, v_0 >$ cycle, walk, path
- directed acyclic digraph
- digraph strongly connected ($\forall u, v \;\exists (uv)$-path), strongly connected components
- G is a tree ($\exists$ path between any two vertices) $\iff$ G is connected and has $n - 1$ edges $\iff$ G is connected and contains no cycles.
- parent, children, sibling, height, depth

Representing Graphs

Operations:

- Access associated information (NodeArray, EdgeArray, Hashes)
- Navigation: access outgoing edges
- Edge queries: given u and v is there an edge?
- Update: add remove edges, vertices

Representing Graphs

Operations:

- Access associated information (NodeArray, EdgeArray, Hashes)
- Navigation: access outgoing edges
- Edge queries: given u and v is there an edge?
- Update: add remove edges, vertices

Data Structures:

- Edge sequences
- Adjacency arrays
- Adjacency lists
- Adjacency matrix

Representing Graphs

Operations:

- Access associated information (NodeArray, EdgeArray, Hashes)
- Navigation: access outgoing edges
- Edge queries: given u and v is there an edge?
- Update: add remove edges, vertices

Data Structures:

- Edge sequences
- Adjacency arrays
- Adjacency lists
- Adjacency matrix

How to choose?

- it depends on the graphs and the application
- if time and space not crucial no need to customize the structures
- use interfaces that make easy to change the data structure
- libraries offer different choices (Boost, Java `jdsl.graph`)

Outline

Motivations

Questions:

1. How good is the algorithm designed?
2. How hard, computationally, is a given a problem to solve using the most efficient algorithm for that problem?

Motivations

Questions:

1. How good is the algorithm designed?
2. How hard, computationally, is a given a problem to solve using the most efficient algorithm for that problem?

1. Asymptotic notation, running time bounds
   Approximation theory

2. Complexity theory

Asymptotic notation

$n \in \mathbf{N}$ instance size

max time	worst case	$T(n) = \max\{T(\pi) \;:\; \pi \in \Pi_n\}$
average time	average case	$T(n) = \frac{1}{\lvert\Pi_n\rvert}\{\sum_\pi T(\pi) \;:\; \pi \in \Pi_n\}$
min time	best case	$T(n) = \min\{T(\pi) \;:\; \pi \in \Pi_n\}$

Asymptotic notation

$n \in \mathbf{N}$ instance size

max time	worst case	$T(n) = \max\{T(\pi) \,:\, \pi \in \Pi_n\}$
average time	average case	$T(n) = \frac{1}{\lvert\Pi_n\rvert}\{\sum_\pi T(\pi) \,:\, \pi \in \Pi_n\}$
min time	best case	$T(n) = \min\{T(\pi) \,:\, \pi \in \Pi_n\}$

Growth rate or asymptotic analysis

$f(n)$ and $g(n)$ same growth rate if $c \le \frac{f(n)}{g(n)} \le d$ for n large

$f(n)$ grows faster than $g(n)$ if $f(n) \ge c \cdot g(n)$ for all c and n large

Asymptotic notation

$n \in \mathbf{N}$ instance size

max time	worst case	$T(n) = \max\{T(\pi) \ : \ \pi \in \Pi_n\}$
average time	average case	$T(n) = \frac{1}{\lvert\Pi_n\rvert}\{\sum_\pi T(\pi) \ : \ \pi \in \Pi_n\}$
min time	best case	$T(n) = \min\{T(\pi) \ : \ \pi \in \Pi_n\}$

Growth rate or asymptotic analysis

$f(n)$ and $g(n)$ same growth rate if $c \leq \frac{f(n)}{g(n)} \leq d$ for n large

$f(n)$ grows faster than $g(n)$ if $f(n) \geq c \cdot g(n)$ for all c and n large

big O	$O(f) = \{g(n) \ : \ \exists c > 0, \forall n > n_0 \ : \ g(n) \leq c \cdot f(n)\}$
big omega	$\Omega(f) = \{g(n) \ : \ \exists c > 0, \forall n > n_0 \ : \ g(n) \geq c \cdot f(n)\}$
theta	$\Theta(f) = O(f) \cap \Omega(f)$
(little o	$o(f) = \{g \ : \ g$ grows strictly more slowly$\}$)

Asymptotic notation

$n \in \mathbf{N}$ instance size

max time	worst case	$T(n) = \max\{T(\pi) \;:\; \pi \in \Pi_n\}$
average time	average case	$T(n) = \frac{1}{\lvert\Pi_n\rvert}\{\sum_\pi T(\pi) \;:\; \pi \in \Pi_n\}$
min time	best case	$T(n) = \min\{T(\pi) \;:\; \pi \in \Pi_n\}$

Growth rate or asymptotic analysis

$f(n)$ and $g(n)$ same growth rate if	$c \leq \frac{f(n)}{g(n)} \leq d$ for n large
$f(n)$ grows faster than $g(n)$ if	$f(n) \geq c \cdot g(n)$ for all c and n large

big O	$O(f) = \{g(n) \;:\; \exists c > 0, \forall n > n_0 \;:\; g(n) \leq c \cdot f(n)\}$
big omega	$\Omega(f) = \{g(n) \;:\; \exists c > 0, \forall n > n_0 \;:\; g(n) \geq c \cdot f(n)\}$
theta	$\Theta(f) = O(f) \cap \Omega(f)$
(little o	$o(f) = \{g \;:\; g$ grows strictly more slowly$\}$)

Outline

Machine model

For asymptotic analysis we use RAM machine

- sequential, single processor unit
- all memory access take same amount of time

It is an abstraction from machine architecture: it ignores caches, memories hierarchies, parallel processing (SIMD, multi-threading), etc.

Machine model

For asymptotic analysis we use RAM machine

- sequential, single processor unit
- all memory access take same amount of time

It is an abstraction from machine architecture: it ignores caches, memories hierarchies, parallel processing (SIMD, multi-threading), etc.

Total execution of a program = total number of instructions executed

We are not interested in constant and lower order terms

Outline

1. Basic Concepts from Previous Courses
   - Graphs
   - Notation and runtime
   - Machine model
   - Pseudo-code
   - Computational Complexity
   - Analysis of Algorithms

Pseudo-code

We express algorithms in natural language and mathematical notation, and in pseudo-code, which is an abstraction from programming languages C, C++, Java, etc.

(In implementation you can choose your favorite language)

Pseudo-code

We express algorithms in natural language and mathematical notation, and in pseudo-code, which is an abstraction from programming languages C, C++, Java, etc.

(In implementation you can choose your favorite language)

Programs must be correct.
Certifying algorithm: computes a certificate for a post condition (without increasing asymptotic running time)

Good Algorithms

We say that an algorithm A is

Efficient = good = polynomial time = polytime
iff
there exists $p(n)$ such that $T(A) = O(p(n))$

Good Algorithms

We say that an algorithm A is

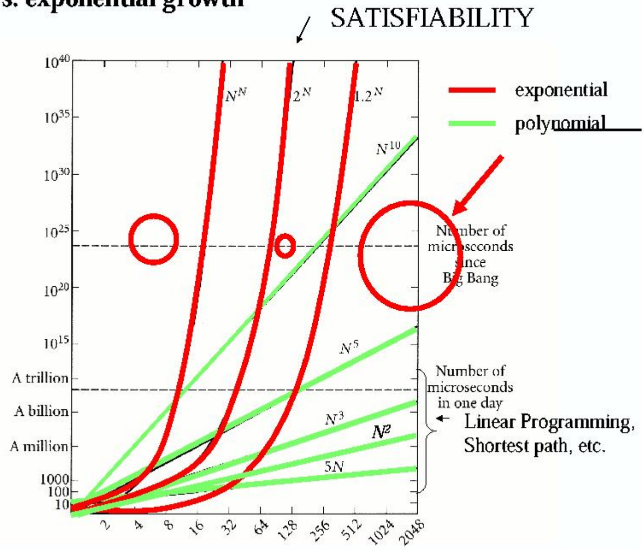
Efficient = good = polynomial time = polytime
iff
there exists $p(n)$ such that $T(A) = O(p(n))$

There are problems for which no polytime algorithm is known.
This course is about those problems.

Complexity theory classifies problems

Polynomial vs. exponential growth

(Harel 2000)



Outline

Complexity Classes
[Garey and Johnson, 1979]

Consider a Decision Search Problem Π:

Complexity Classes

[Garey and Johnson, 1979]

Consider a Decision Search Problem Π:

- Π is in P if $\exists$ algorithm $\mathcal{A}$ that finds a solution in polynomial time.

Complexity Classes
[Garey and Johnson, 1979]

Consider a Decision Search Problem Π:

- Π is in P if $\exists$ algorithm $\mathcal{A}$ that finds a solution in polynomial time.
- Π is in NP if $\exists$ verification algorithm $\mathcal{A}$ that verifies whether a binary certificate is a solution to the problem in polynomial time.

Complexity Classes

[Garey and Johnson, 1979]

Consider a Decision Search Problem Π:

- Π is in P if $\exists$ algorithm $\mathcal{A}$ that finds a solution in polynomial time.
- Π is in NP if $\exists$ verification algorithm $\mathcal{A}$ that verifies whether a binary certificate is a solution to the problem in polynomial time.
- a search problem Π' is (polynomially) reducible to Π ($\Pi' \longrightarrow \Pi$) if there exists an algorithm $\mathcal{A}$ that solves Π' by using a hypothetical subroutine $\mathcal{S}$ for Π and except for $\mathcal{S}$ everything runs in polynomial time.

Complexity Classes

[Garey and Johnson, 1979]

Consider a Decision Search Problem Π:

- Π is in P if $\exists$ algorithm $\mathcal{A}$ that finds a solution in polynomial time.
- Π is in NP if $\exists$ verification algorithm $\mathcal{A}$ that verifies whether a binary certificate is a solution to the problem in polynomial time.
- a search problem Π' is (polynomially) reducible to Π ($\Pi' \longrightarrow \Pi$) if there exists an algorithm $\mathcal{A}$ that solves Π' by using a hypothetical subroutine $\mathcal{S}$ for Π and except for $\mathcal{S}$ everything runs in polynomial time.
- Π is NP-complete if
  1. it is in NP
  2. there exists some NP-complete problem Π' that reduces to Π ($\Pi' \longrightarrow \Pi$)

Complexity Classes
[Garey and Johnson, 1979]

Consider a Decision Search Problem Π:

- Π is in P if $\exists$ algorithm $\mathcal{A}$ that finds a solution in polynomial time.
- Π is in NP if $\exists$ verification algorithm $\mathcal{A}$ that verifies whether a binary certificate is a solution to the problem in polynomial time.
- a search problem Π' is (polynomially) reducible to Π ($\Pi' \longrightarrow \Pi$) if there exists an algorithm $\mathcal{A}$ that solves Π' by using a hypothetical subroutine $\mathcal{S}$ for Π and except for $\mathcal{S}$ everything runs in polynomial time.
- Π is NP-complete if
  1. it is in NP
  2. there exists some NP-complete problem Π' that reduces to Π ($\Pi' \longrightarrow \Pi$)
- If Π satisfies property 2, but not necessarily property 1, we say that it is NP-hard:

- *NP*: Class of problems that can be solved in polynomial time by a non-deterministic machine.

- *NP*: Class of problems that can be solved in polynomial time by a non-deterministic machine.

  Note: non-deterministic $\neq$ randomized;
  non-deterministic machines are idealized models of computation that have the ability to make perfect guesses.

- *NP*: Class of problems that can be solved in polynomial time by a non-deterministic machine.

  Note: non-deterministic $\neq$ randomized;
  non-deterministic machines are idealized models of computation that have the ability to make perfect guesses.

- *NP*-complete: Among the most difficult problems in *NP*; believed to have at least exponential time-complexity for any realistic machine or programming model.

- *NP*: Class of problems that can be solved in polynomial time by a non-deterministic machine.

  Note: non-deterministic $\neq$ randomized;
  non-deterministic machines are idealized models of computation that have the ability to make perfect guesses.

- *NP*-complete: Among the most difficult problems in *NP*; believed to have at least exponential time-complexity for any realistic machine or programming model.

- *NP*-hard: At least as difficult as the most difficult problems in *NP*, but possibly not in *NP* (*i.e.*, may have even worse complexity than *NP*-complete problems).

SAT Problem
Satisfiability problem in propositional logic

Definitions:

- Formula in propositional logic: well-formed string that may contain
  - propositional variables $x_1, x_2, \ldots, x_n$;
  - truth values $\top$ ('true'), $\bot$ ('false');
  - operators $\neg$ ('not'), $\wedge$ ('and'), $\vee$ ('or');
  - parentheses (for operator nesting).

SAT Problem

Satisfiability problem in propositional logic

Definitions:

- Formula in propositional logic: well-formed string that may contain
  - propositional variables $x_1, x_2, \ldots, x_n$;
  - truth values $\top$ ('true'), $\bot$ ('false');
  - operators $\neg$ ('not'), $\wedge$ ('and'), $\vee$ ('or');
  - parentheses (for operator nesting).
- Model (or satisfying assignment) of a formula F: Assignment of truth values to the variables in F under which F becomes true (under the usual interpretation of the logical operators)

SAT Problem
Satisfiability problem in propositional logic

Definitions:

- Formula in propositional logic: well-formed string that may contain
  - propositional variables $x_1, x_2, \dots, x_n$;
  - truth values $\top$ ('true'), $\bot$ ('false');
  - operators $\neg$ ('not'), $\wedge$ ('and'), $\vee$ ('or');
  - parentheses (for operator nesting).

- Model (or satisfying assignment) of a formula F: Assignment of truth values to the variables in F under which F becomes true (under the usual interpretation of the logical operators)

- Formula F is satisfiable iff there exists at least one model of F, unsatisfiable otherwise.

SAT Problem (decision problem, search variant):

- **Given:** Formula F in propositional logic
- **Task:** Find an assignment of truth values to variables in F that renders F true, or decide that no such assignment exists.

SAT Problem (decision problem, search variant):

- **Given:** Formula F in propositional logic
- **Task:** Find an assignment of truth values to variables in F that renders F true, or decide that no such assignment exists.

SAT: A simple example

- **Given:** Formula $F := (x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2)$
- **Task:** Find an assignment of truth values to variables x_1, x_2 that renders F true, or decide that no such assignment exists.

Definitions:

- A formula is in conjunctive normal form (CNF) iff it is of the form

  $$\bigwedge_{i=1}^{m} \bigvee_{j=1}^{k_i} l_{ij} = (l_{11} \vee \ldots \vee l_{1k_1}) \wedge \ldots \wedge (l_{m1} \vee \ldots \vee l_{mk_m})$$

  where each literal l_{ij} is a propositional variable or its negation. The disjunctions $c_i = (l_{i1} \vee \ldots \vee l_{ik_i})$ are called clauses.

- A formula is in k-CNF iff it is in CNF and all clauses contain exactly k literals (*i.e.*, for all i, $k_i = k$).

Definitions:

- A formula is in conjunctive normal form (CNF) iff it is of the form

$$\bigwedge_{i=1}^{m} \bigvee_{j=1}^{k_i} l_{ij} = (l_{11} \vee \ldots \vee l_{1k_1}) \wedge \ldots \wedge (l_{m1} \vee \ldots \vee l_{mk_m})$$

  where each literal l_{ij} is a propositional variable or its negation. The disjunctions $c_i = (l_{i1} \vee \ldots \vee l_{ik_i})$ are called clauses.

- A formula is in k-CNF iff it is in CNF and all clauses contain exactly k literals (*i.e.*, for all i, $k_i = k$).

- In many cases, the restriction of SAT to CNF formulae is considered.
- For every propositional formula, there is an equivalent formula in 3-CNF.

Example:

$$
\begin{aligned}
F := \quad & \wedge (\neg x_2 \vee x_1) \\
& \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_3) \\
& \wedge (x_1 \vee x_2) \\
& \wedge (\neg x_4 \vee x_3) \\
& \wedge (\neg x_5 \vee x_3)
\end{aligned}
$$

- F is in CNF.
- Is F satisfiable?

Example:

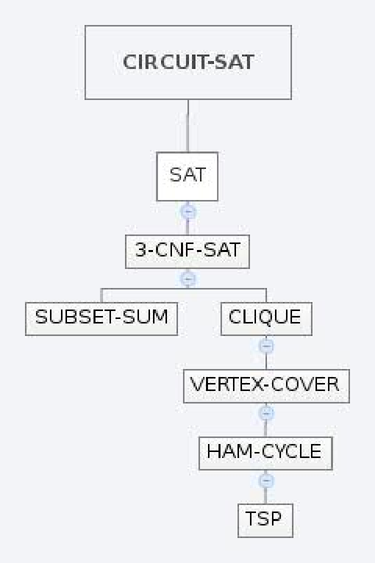
$$F := \begin{array}{l} \wedge (\neg x_2 \vee x_1) \\ \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_3) \\ \wedge (x_1 \vee x_2) \\ \wedge (\neg x_4 \vee x_3) \\ \wedge (\neg x_5 \vee x_3) \end{array}$$

- F is in CNF.
- Is F satisfiable?
  Yes, *e.g.*, $x_1 := x_2 := \top$, $x_3 := x_4 := x_5 := \bot$ is a model of F.

MAX-SAT (optimization problem)

Which is the maximal number of clauses satisfiable in a propositional logic formula F?

NP-Completeness Proofs



Many combinatorial problems are hard
but some problems can be solved efficiently

- Longest path problem is *NP*-hard
  but not shortest path problem

Many combinatorial problems are hard
but some problems can be solved efficiently

- Longest path problem is *NP*-hard
  but not shortest path problem

- SAT for 3-CNF is *NP*-complete
  but not 2-CNF (linear time algorithm)

Many combinatorial problems are hard
but some problems can be solved efficiently

- Longest path problem is *NP*-hard
  but not shortest path problem

- SAT for 3-CNF is *NP*-complete
  but not 2-CNF (linear time algorithm)

- Hamiltonian path is *NP*-complete
  but not the Eulerian path problem

Many combinatorial problems are hard
but some problems can be solved efficiently

- Longest path problem is *NP*-hard
  but not shortest path problem

- SAT for 3-CNF is *NP*-complete
  but not 2-CNF (linear time algorithm)

- Hamiltonian path is *NP*-complete
  but not the Eulerian path problem

- TSP on Euclidean instances is *NP*-hard
  but not where all vertices lie on a circle.

An online compendium on the computational complexity of optimization problems:
http://www.nada.kth.se/~viggo/problemlist/compendium.html

Outline

Theoretical Analysis

- Worst-case analysis (runtime and quality):
  worst performance of algorithms over all possible instances
- Probabilistic analysis (runtime):
  average-case performance over a given probability distribution of instances
- Average-case (runtime):
  overall possible instances for randomized algorithms
- Asymptotic convergence results (quality)
- Approximation of optimal solutions:
  sometimes possible in polynomial time (*e.g.*, Euclidean TSP),
  but in many cases also intractable (*e.g.*, general TSP);
- Domination
- Algorithm invariance

Approximation Algorithms

Definition: Approximation Algorithms

An algorithm $\mathcal{A}$ is said to be a δ-approximation algorithm if it runs in polynomial time and for every problem instance π with optimal solution value $OPT(\pi)$

$$\text{minimization:} \quad \frac{\mathcal{A}(\pi)}{OPT(\pi)} \leq \delta \quad \delta \geq 1$$

$$\text{maximization:} \quad \frac{\mathcal{A}(\pi)}{OPT(\pi)} \geq \delta \quad \delta \leq 1$$

(δ is called *worst case bound*, *worst case performance*, *approximation factor*, *approximation ratio*, *performance bound*, *performance ratio*, *error ratio*)

Approximation Algorithms

Definition: Polynomial approximation scheme

A family of approximation algorithms for a problem Π, $\{\mathcal{A}_\epsilon\}_\epsilon$, is called a polynomial approximation scheme (PAS), if algorithm $\mathcal{A}_\epsilon$ is a $(1+\epsilon)$-approximation algorithm and its running time is polynomial in the size of the input for each fixed ϵ

Definition: Fully polynomial approximation scheme

A family of approximation algorithms for a problem Π, $\{\mathcal{A}_\epsilon\}_\epsilon$, is called a fully polynomial approximation scheme (FPAS), if algorithm $\mathcal{A}_\epsilon$ is a $(1+\epsilon)$-approximation algorithm and its running time is polynomial in the size of the input and $1/\epsilon$

Useful Graph Algorithms

- Breadth first, depth first search, traversal
- Transitive closure
- Topological sorting
- (Strongly) connected components
- Shortest Path
- Minimum Spanning Tree
- Matching

Randomized Algorithms

Most often algorithms are randomized. Why?

Randomized Algorithms

Most often algorithms are randomized. Why?

- possibility of gains from re-runs
- adversary argument
- structural simplicity for comparable average performance,
- speed up,
- avoiding loops in the search
- ...

Randomized Algorithms

Definition: Randomized Algorithms

Their running time depends on the random choices made.
Hence, the running time is a random variable.

Randomized Algorithms

Definition: Randomized Algorithms

Their running time depends on the random choices made.
Hence, the running time is a random variable.

Las Vegas algorithm: it always gives the correct result but in random runtime (with finite expected value).

Monte Carlo algorithm: the result is not guaranteed correct. Typically halted due to bouned resources.

Randomized Heuristics

In the case of randomized optimization heuristics
both solution quality and runtime are random variables.

Randomized Heuristics

In the case of randomized optimization heuristics
both solution quality and runtime are random variables.

We distinguish:

- single-pass heuristics (denoted $\mathcal{A}^{\dashv}$): have an embedded termination, for example, upon reaching a certain state
(generalized optimization Las Vegas algorithms [B2])

- asymptotic heuristics (denoted $\mathcal{A}^{\infty}$): do not have an embedded termination and they might improve their solution asymptotically
(both probabilistically approximately complete
and essentially incomplete [B2])