Algorithmics: Basic definitions and concepts

The Algorithmics course

Already known (EDA level)

- Algorithms cost and Asymptotic notation

The Algorithmics course

Already known (EDA level)

- Algorithms cost and Asymptotic notation
- Sorting algorithms: Mergesort, Quicksort, ...

The Algorithmics course

Already known (EDA level)

- Algorithms cost and Asymptotic notation
- Sorting algorithms: Mergesort, Quicksort, ...
- Divide and conquer, recurrences, master theorem

The Algorithmics course

Already known (EDA level)

- Algorithms cost and Asymptotic notation
- Sorting algorithms: Mergesort, Quicksort, ...
- Divide and conquer, recurrences, master theorem
- Complexity, P and NP, reductions

The Algorithmics course

Already known (EDA level)

- Algorithms cost and Asymptotic notation
- Sorting algorithms: Mergesort, Quicksort, ...
- Divide and conquer, recurrences, master theorem
- Complexity, P and NP, reductions
- Foundations on probability

The Algorithmics course

Already known (EDA level)

- Algorithms cost and Asymptotic notation
- Sorting algorithms: Mergesort, Quicksort, ...
- Divide and conquer, recurrences, master theorem
- Complexity, P and NP, reductions
- Foundations on probability
- Basic data structures: Arrays, lists, stacks, queues, heaps, hashing ...

The Algorithmics course

Already known (EDA level)

- Algorithms cost and Asymptotic notation
- Sorting algorithms: Mergesort, Quicksort, ...
- Divide and conquer, recurrences, master theorem
- Complexity, P and NP, reductions
- Foundations on probability
- Basic data structures: Arrays, lists, stacks, queues, heaps, hashing ...
- Basics on graph theory, graph data structures

The Algorithmics course

Already known (EDA level)

- Algorithms cost and Asymptotic notation
- Sorting algorithms: Mergesort, Quicksort, ...
- Divide and conquer, recurrences, master theorem
- Complexity, P and NP, reductions
- Foundations on probability
- Basic data structures: Arrays, lists, stacks, queues, heaps, hashing ...
- Basics on graph theory, graph data structures
- Graph and digraph traversals (BFS, DFS) and applications.

The Algorithmics course

Already known (EDA level)

- Algorithms cost and Asymptotic notation
- Sorting algorithms: Mergesort, Quicksort, . . .
- Divide and conquer, recurrences, master theorem
- Complexity, P and NP, reductions
- Foundations on probability
- Basic data structures: Arrays, lists, stacks, queues, heaps, hashing . . .
- Basics on graph theory, graph data structures
- Graph and digraph traversals (BFS, DFS) and applications.
- Backtracking algorithms

The Algorithmics course

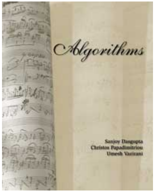
Topics to cover:

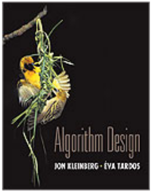
- Divide and conquer: Linear Selection
- Sorting in linear time (when? how?)
- Greedy algorithms
- Dynamic programming
- Distances in graphs
- Flow networks: problems, algorithms and applications
- Linear Programming
- Approximation algorithms
- Streaming algorithms

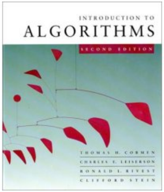
Provide models to solve real problems

References

Main references:







"The algorithmic lenses: C. Papadimitriou"

- In 1936 Alan Turing demonstrated the universality of computational principles with his mathematical model: the Turing machine.

"The algorithmic lenses: C. Papadimitriou"

- In 1936 Alan Turing demonstrated the universality of computational principles with his mathematical model: the Turing machine.
- Algorithms themselves have evolved into a complex set of techniques, self-learning, Web services, concurrent, distributed or parallel, etc... Each of them with ad-hoc relevant computational limitations and social implications.

"The algorithmic lenses: C. Papadimitriou"

- In 1936 Alan Turing demonstrated the universality of computational principles with his mathematical model: the Turing machine.
- Algorithms themselves have evolved into a complex set of techniques, self-learning, Web services, concurrent, distributed or parallel, etc... Each of them with ad-hoc relevant computational limitations and social implications.
- However, this course will be a course on classical algorithms, which are the core needed to understand more advanced computational material.

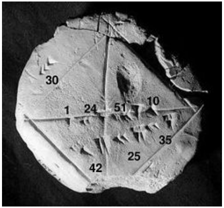
Algorithms

Algorithm: a step-by-step set of precise instructions or rules designed to perform a specific task or solve a defined problem. Each step of the process must be clear and unambiguous, and it should always yield a correct answer.

Algorithms

Algorithm: a step-by-step set of precise instructions or rules designed to perform a specific task or solve a defined problem. Each step of the process must be clear and unambiguous, and it should always yield a correct answer.

Sqrt (n)
$x_0 = 1$
for $i = 1$ to 6 **do**
  $x_i = (x_{i-1} + n/x_{i-1})/2$
end for

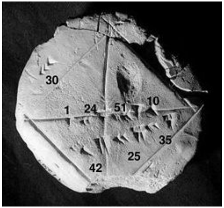


Babilònia (XVI BC)

Algorithms

Algorithm: a step-by-step set of precise instructions or rules designed to perform a specific task or solve a defined problem. Each step of the process must be clear and unambiguous, and it should always yield a correct answer.

Sqrt (n)
$x_0 = 1$
for $i = 1$ to 6 **do**
  $x_i = (x_{i-1} + n/x_{i-1})/2$
end for



Babilònia (XVI BC)

For $n = 20$, x's are 1 10.5 6.2023 4.7134 4.4783

We have designed an algorithm to solve a computational problem

We have designed an algorithm to solve a computational problem

What do we want to know?

We have designed an algorithm to solve a computational problem

What do we want to know?

- Correctness, it always does what it should?

We have designed an algorithm to solve a computational problem

What do we want to know?

- Correctness, it always does what it should?
- Performance,
  - computing time,
  - memory use
  - communication cost, ...

We have designed an algorithm to solve a computational problem

What do we want to know?

- Correctness, it always does what it should?
- Performance,
  - computing time,
  - memory use
  - communication cost, ...

For an algorithm $\mathcal{A}$, $t_{\mathcal{A}}(x)$ is the computing time on input x.

In this course, we use a worst case analysis: Given a problem, for which you designed an algorithm, you assume that your meanest adversary gives you the worst possible input.

We have designed an algorithm to solve a computational problem

What do we want to know?

- Correctness, it always does what it should?
- Performance,
  - computing time,
  - memory use
  - communication cost, ...

For an algorithm $\mathcal{A}$, $t_{\mathcal{A}}(x)$ is the computing time on input x.

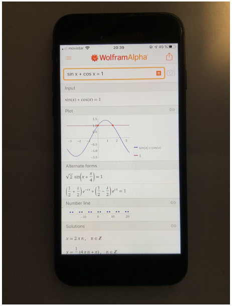
In this course, we use a worst case analysis: Given a problem, for which you designed an algorithm, you assume that your meanest adversary gives you the worst possible input.
We use as measure of time complexity or cost the function

$$T(n) = \max_{|x|=n} t_{\mathcal{A}}(x)$$

.

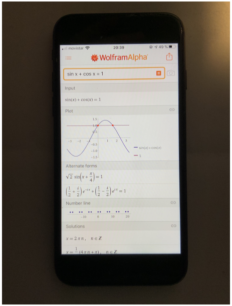
Time complexity

The time complexity must be independent of the "used" machine?



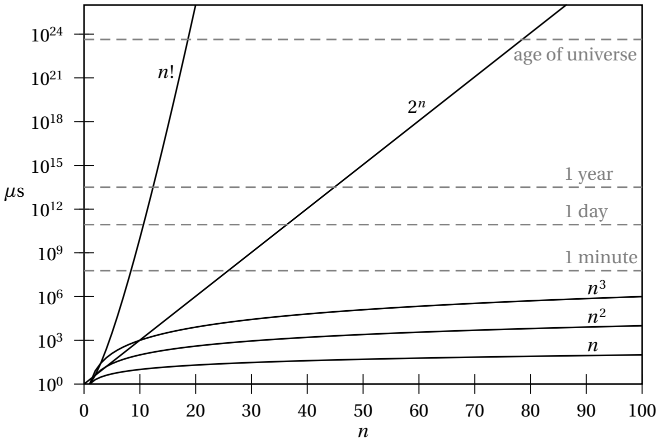
Time complexity

The time complexity must be independent of the "used" machine?



Must we consider carefully how operations scale with respect to size?

Computation time assuming that an input with size $n = 1$ can be solved in 1 μsecond:



From: Moore-Mertens, The Nature of Computation

Be careful

- Assume that the input to an algorithm is an integer x that uses 64 bits.

Be careful

- Assume that the input to an algorithm is an integer x that uses 64 bits.
- The cost of the algorithm is $O(x)$ and the time units are nanoseconds.

Be careful

- Assume that the input to an algorithm is an integer x that uses 64 bits.
- The cost of the algorithm is $O(x)$ and the time units are nanoseconds.
- Thus, processing this input takes more than

Be careful

- Assume that the input to an algorithm is an integer x that uses 64 bits.
- The cost of the algorithm is $O(x)$ and the time units are nanoseconds.
- Thus, processing this input takes more than 500 years

Be careful

- Assume that the input to an algorithm is an integer x that uses 64 bits.
- The cost of the algorithm is $O(x)$ and the time units are nanoseconds.
- Thus, processing this input takes more than 500 years

Note that the cost of this algorithm is a polynomial function on the input value not on the input size.

Be careful

- Assume that the input to an algorithm is an integer x that uses 64 bits.
- The cost of the algorithm is $O(x)$ and the time units are nanoseconds.
- Thus, processing this input takes more than 500 years

Note that the cost of this algorithm is a polynomial function on the input value not on the input size.
Such algorithms are classified as pseudopolynomial.

Efficient algorithms and practical algorithms

- We say that an algorithm is feasible if its cost is polynomial.

Efficient algorithms and practical algorithms

- We say that an algorithm is feasible if its cost is polynomial.
- However $n^{10^{10}}$ is a polynomial function but such computing time could be prohibitive!

Efficient algorithms and practical algorithms

- We say that an algorithm is feasible if its cost is polynomial.
- However $n^{10^{10}}$ is a polynomial function but such computing time could be prohibitive!
- In the same way, if we have cn^2 for constant $c = 10^{64}$, then c dominates inputs up to a size of $n > 10^{64}$.

Efficient algorithms and practical algorithms

- We say that an algorithm is feasible if its cost is polynomial.
- However $n^{10^{10}}$ is a polynomial function but such computing time could be prohibitive!
- In the same way, if we have cn^2 for constant $c = 10^{64}$, then c dominates inputs up to a size of $n > 10^{64}$.
- In this course, we will not enter in the analysis up to constants, but keep in mind that constants matter!!!!

Efficient algorithms and practical algorithms

- We say that an algorithm is feasible if its cost is polynomial.
- However $n^{10^{10}}$ is a polynomial function but such computing time could be prohibitive!
- In the same way, if we have cn^2 for constant $c = 10^{64}$, then c dominates inputs up to a size of $n > 10^{64}$.
- In this course, we will not enter in the analysis up to constants, but keep in mind that constants matter!!!!
- In practice, even a feasible algorithms with time complexity of for example n^4 could be too slow for $n \geq 1000$.

Asymptotic notation

Symbol	$L = \lim_{n\to\infty} \frac{f(n)}{g(n)}$	intuition
$f(n) = O(g(n))$	$L < \infty$	$f \leq g$
$f(n) = \Omega(g(n))$	$L > 0$	$f \geq g$
$f(n) = \Theta(g(n))$	$0 < L < \infty$	$f = g$
$f(n) = o(g(n))$	$L = 0$	$f < g$
$f(n) = \omega(g(n))$	$L = \infty$	$f > g$

Names used for specific function classes

Name	Definition
polylogarithmic	$f = O(\log^c n)$ (c constant)
polynomial	$f = O(n^c)$ (c constant) or $n^{O(1)}$
subexponential	$f = o(2^{n^\epsilon})$ $(0 < \epsilon < 1)$
exponential	$f = 2^{\mathrm{poly}(n)}$
double exponential	$f = 2^{\exp(n)}$

Notation:

$\lg \equiv \log_2$; $\ln \equiv \log_e$; $\log \equiv \log_{10}$.

Some math you should remember

Given an integer $n > 0$ and a real $a > 1$ and $a \neq 0$:

- Arithmetic summation: $\sum_{i=0}^{n} i = \frac{n(n+1)}{2}$.
- Geometric summation: $\sum_{i=0}^{n} a^i = \frac{1-a^{n+1}}{1-a}$.

Logarithms and Exponents: For $a, b, c \in \mathbb{R}^+$,

- $\log_b a = c \Leftrightarrow a = b^c \Rightarrow \log_b 1 = 0$
- $\log_b ac = \log_b a + \log_b c$, $\log_b a/c = \log_b a - \log_b c$.
- $\log_b a^c = c \log_b a \Rightarrow c^{\log_b a} = a^{\log_b c} \Rightarrow 2^{\log_2 n} = n$.
- $\log_b a = \log_c a / \log_c b \Rightarrow \log_b a = \Theta(\log_c a)$

Stirling: $n! = \sqrt{2\pi n}(n/e)^n + 0(1/n) + \gamma \Rightarrow n! + \omega((n/2)^n)$.

n-Harmonic: $H_n = \sum_{i=1}^{n} 1/i \sim \ln n$.

Graphs

See for ex. Chapter 3 of Dasgupta, Papadimitriou, Vazirani (DPV).

Graph: $G = (V, E)$, where V is the set of vertices, $n = |V|$, and $E \subset V \times V$ is the set of edges, $m = |E|$,

Graphs

See for ex. Chapter 3 of Dasgupta, Papadimitriou, Vazirani (DPV).

Graph: $G = (V, E)$, where V is the set of vertices, $n = |V|$, and $E \subset V \times V$ is the set of edges, $m = |E|$,

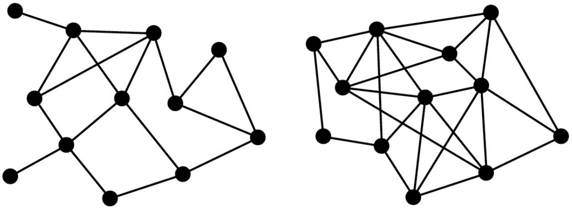
- Graphs: *undirected graphs (graphs)* and *directed graphs (digraphs)*
- The degree of v ($d(v)$) is the number of edges which are incident to v.
- A clique on n vertices (K_n) is a complete graph (with $m = n(n-1)/2$).
- A undirected G is said to be connected if there is a path between any two distinct vertices.
- If G is connected, then $m \geq n - 1$.

Directed graphs

- Edges are directed.
- The connectivity concept in digraphs is the so called strong connectivity: There is is a directed path between any two vertices.
- In a digraph $m \leq n(n-1)$.

Density of a graph

A G with n vertices is said to be dense when $m = \Theta(n^2)$.
When $m = o(n^2)$, G is said to be sparse.



Common graph's data structures

Let $G = (V, E)$ be a graph.

Adjacency list uses $O(|V| + |E|)$ space.

Adjacency matrix uses $O(|V|^2)$ space.

Adjacency matrix

- If G is undirected, its adjacency matrix $A(G)$ is symmetric.
- If A is the adjacency matrix of G, then A^2 gives, for $i, j \in V$, whether there is a path between i and j in G, with length 2.
  For $k > 0$, A^k indicates if there is a path with length k from i to j.
- If G has weights on edges, i.e. $w_{i,j}$ for each $(i, j) \in E$, $A(G)$ keeps w_{ij} in position (i, j).
- Adjacency matrices allow the use of tools from linear algebra.

Complexity issues between matrix and list DS

Complexity issues between matrix and list DS

- Adding a new edge to G:

Complexity issues between matrix and list DS

- Adding a new edge to G: In both data structures, we need $\Theta(1)$.

Complexity issues between matrix and list DS

- Adding a new edge to G: In both data structures, we need $\Theta(1)$.
- Edge query, for u and v in $V(G)$, $(u, v) \in E(V)$?:
  For matrix representation:

Complexity issues between matrix and list DS

- Adding a new edge to G: In both data structures, we need $\Theta(1)$.
- Edge query, for u and v in $V(G)$, $(u, v) \in E(V)$?:
  For matrix representation: $\Theta(1)$.

Complexity issues between matrix and list DS

- Adding a new edge to G: In both data structures, we need $\Theta(1)$.
- Edge query, for u and v in $V(G)$, $(u, v) \in E(V)$?:
  For matrix representation: $\Theta(1)$.
  For list representation:

Complexity issues between matrix and list DS

- Adding a new edge to G: In both data structures, we need $\Theta(1)$.
- Edge query, for u and v in $V(G)$, $(u, v) \in E(V)$?:
  For matrix representation: $\Theta(1)$.
  For list representation: $O(n)$.

Complexity issues between matrix and list DS

- Adding a new edge to G: In both data structures, we need $\Theta(1)$.
- Edge query, for u and v in $V(G)$, $(u, v) \in E(V)$?:
  For matrix representation: $\Theta(1)$.
  For list representation: $O(n)$.
- Explore all neighbours of vertex v:
  For matrix representation:

Complexity issues between matrix and list DS

- Adding a new edge to G: In both data structures, we need $\Theta(1)$.
- Edge query, for u and v in $V(G)$, $(u, v) \in E(V)$?:
  For matrix representation: $\Theta(1)$.
  For list representation: $O(n)$.
- Explore all neighbours of vertex v:
  For matrix representation: $\Theta(n)$
  For list representation:

Complexity issues between matrix and list DS

- Adding a new edge to G: In both data structures, we need $\Theta(1)$.
- Edge query, for u and v in $V(G)$, $(u, v) \in E(V)$?:
  For matrix representation: $\Theta(1)$.
  For list representation: $O(n)$.
- Explore all neighbours of vertex v:
  For matrix representation: $\Theta(n)$
  For list representation: $\Theta(|d(v)|)$

Complexity issues between matrix and list DS

- Adding a new edge to G: In both data structures, we need $\Theta(1)$.
- Edge query, for u and v in $V(G)$, $(u, v) \in E(V)$?:
  For matrix representation: $\Theta(1)$.
  For list representation: $O(n)$.
- Explore all neighbours of vertex v:
  For matrix representation: $\Theta(n)$
  For list representation: $\Theta(|d(v)|)$
- Erase an edge in G:

Complexity issues between matrix and list DS

- Adding a new edge to G: In both data structures, we need $\Theta(1)$.
- Edge query, for u and v in $V(G)$, $(u, v) \in E(V)$?:
  For matrix representation: $\Theta(1)$.
  For list representation: $O(n)$.
- Explore all neighbours of vertex v:
  For matrix representation: $\Theta(n)$
  For list representation: $\Theta(|d(v)|)$
- Erase an edge in G: The same as Edge query.

Complexity issues between matrix and list DS

- Adding a new edge to G: In both data structures, we need $\Theta(1)$.
- Edge query, for u and v in $V(G)$, $(u, v) \in E(V)$?:
  For matrix representation: $\Theta(1)$.
  For list representation: $O(n)$.
- Explore all neighbours of vertex v:
  For matrix representation: $\Theta(n)$
  For list representation: $\Theta(|d(v)|)$
- Erase an edge in G: The same as Edge query.
- Erase a vertex in G:
  For matrix representation:

Complexity issues between matrix and list DS

- Adding a new edge to G: In both data structures, we need $\Theta(1)$.
- Edge query, for u and v in $V(G)$, $(u, v) \in E(V)$?:
  For matrix representation: $\Theta(1)$.
  For list representation: $O(n)$.
- Explore all neighbours of vertex v:
  For matrix representation: $\Theta(n)$
  For list representation: $\Theta(|d(v)|)$
- Erase an edge in G: The same as Edge query.
- Erase a vertex in G:
  For matrix representation: $O(n^2)$.
  For list representation:

Complexity issues between matrix and list DS

- Adding a new edge to G: In both data structures, we need $\Theta(1)$.
- Edge query, for u and v in $V(G)$, $(u,v) \in E(V)$?:
  For matrix representation: $\Theta(1)$.
  For list representation: $O(n)$.
- Explore all neighbours of vertex v:
  For matrix representation: $\Theta(n)$
  For list representation: $\Theta(|d(v)|)$
- Erase an edge in G: The same as Edge query.
- Erase a vertex in G:
  For matrix representation: $O(n^2)$.
  For list representation: $O(m)$.

Basic graph algorithms

1. Breadth First Search
2. Depth First Search
3. Topological sort in directed acyclic graphs
4. Connected components in undirected graphs
5. Strongly connected components in directed graphs

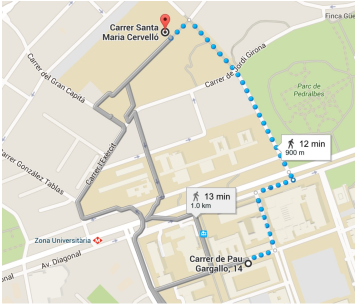
Basic graph algorithms

1. Breadth First Search
2. Depth First Search
3. Topological sort in directed acyclic graphs
4. Connected components in undirected graphs
5. Strongly connected components in directed graphs

All of them have cost $O(|V| + |E|)$ on graphs given by adjacency lists.

Searching a graph: Breadth First Search

1. Start with vertex v, visit v and all its neighbors.
2. Then, the non-visited neighbors of visited ones.
3. Repeat until all vertices are visited.



BFS use a QUEUE, (FIFO) to keep the neighbors of visited vertices.

Recall that vertices are labeled to avoid visiting them more than once.

Searching a graph: Depth First Search

explore

1. From current vertex, move to a neighbor.
2. Until you get stuck.
3. Then backtrack till new place to explore.

DFS use a STACK, (LIFO)

Time Complexity of DFS and BFS

For graphs given by adjacency lists: $O(|V| + |E|)$

For graphs given by adjacency matrix: $O(|V|^2)$

Therefore, both procedures can be implemented in linear time with respect to the size of the input graph.

Connected components in undirected graphs

A connected component is a maximal connected subgraph of G.

A connected graph has a unique connected component.

Connected components in undirected graphs

A connected component is a maximal connected subgraph of G.

A connected graph has a unique connected component.

Connected Components Problem
INPUT: undirected graph G
GOAL: Find all the connected components of G.

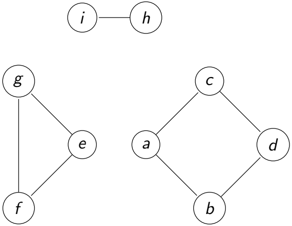
Connected components in undirected graphs

A connected component is a maximal connected subgraph of G.

A connected graph has a unique connected component.

Connected Components Problem
INPUT: undirected graph G
GOAL: Find all the connected components of G.

To find connected components in G use DFS/BFS and keep track of the set of vertices visited in each **explore** call.



The problem can be solved in $O(|V| + |E|)$.

Strongly connected components in a digraph

A digraph $G = (V, E)$, is *strongly connected*, if for all $u, v \in V$, there are paths $u \to v$ and $v \to u$.

A strongly connected component is a maximal strongly connected graph.

Strongly Connected Components Problem
INPUT: digraph G
GOAL: Find the strongly connected components of G.

Strongly connected components in a digraph

A digraph $G = (V, E)$, is *strongly connected*, if for all $u, v \in V$, there are paths $u \to v$ and $v \to u$.

A strongly connected component is a maximal strongly connected graph.

Strongly Connected Components Problem
INPUT: digraph G
GOAL: Find the strongly connected components of G.

Kosharaju-Sharir's algorithm: Uses DFS (twice). Complexity $T(n) = O(|V| + |E|)$

Strongly connected components in a digraph

A digraph $G = (V, E)$, is *strongly connected*, if for all $u, v \in V$, there are paths $u \to v$ and $v \to u$.

A strongly connected component is a maximal strongly connected graph.

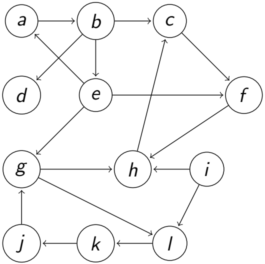
Strongly Connected Components Problem
INPUT: digraph G
GOAL: Find the strongly connected components of G.

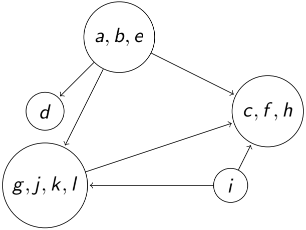
Kosharaju-Sharir's algorithm: Uses DFS (twice). Complexity $T(n) = O(|V| + |E|)$

Tarjan's algorithm: Using DFS (once). Complexity $T(n) = O(|V| + |E|)$

Strongly connected components in a digraph

A nice property: For every digraph, the graph on its strongly connected components is *acyclic*.





The classes P and NP

- Recall that a problem belong to the class P if there exists an algorithm that is polynomial in the worst-case analysis, (for the worst input given by a malicious adversary)

The classes P and NP

- Recall that a problem belong to the class P if there exists an algorithm that is polynomial in the worst-case analysis, (for the worst input given by a malicious adversary)
- A problem given in decisional form belong to the class NP non-deterministic polynomial time if, given a certificate of a solution, we can verify in polynomial time that indeed the certificate is a valid solution to the problem and those certificates have polynomial size

The classes P and NP

- Recall that a problem belong to the class P if there exists an algorithm that is polynomial in the worst-case analysis, (for the worst input given by a malicious adversary)
- A problem given in decisional form belong to the class NP non-deterministic polynomial time if, given a certificate of a solution, we can verify in polynomial time that indeed the certificate is a valid solution to the problem and those certificates have polynomial size
- It is easy to se that P$\subseteq$ NP, but it is an open problem to prove that P$=$NP or that P$\neq$NP.

The classes P and NP

- Recall that a problem belong to the class P if there exists an algorithm that is polynomial in the worst-case analysis, (for the worst input given by a malicious adversary)
- A problem given in decisional form belong to the class NP non-deterministic polynomial time if, given a certificate of a solution, we can verify in polynomial time that indeed the certificate is a valid solution to the problem and those certificates have polynomial size
- It is easy to se that P$\subseteq$ NP, but it is an open problem to prove that P$=$NP or that P$\neq$NP.
- The class NP-complete is the class of most difficult problems in decisional form that are in NP. Most difficult in the sense that if one of them is proved to be in P then P$=$NP.

Beyond worst-case analysis

- Under the hypothesis that P$\neq$NP, if the decision version of a problem is NP-complete, then the optimization problem will require at least exponential time, for some inputs.
- The classification of a problem as NP-complete is a case of worst-case analysis, and for many problems the "expensive inputs" are few, and far from practical typical inputs. We will see some examples through the course.
- Therefore, there are alternative ways to get in practice, solutions for NP-complete problems, with the use of alternative algorithmic techniques, as approximation (we will see some examples), heuristics and self-learning algorithms, that are deferred to other courses.

A powerful tool to solve problems: Reductions

You have been introduced in previous courses to the concept of reduction between decision problems, to define the class NP-complete.

A powerful tool to solve problems: Reductions

You have been introduced in previous courses to the concept of reduction between decision problems, to define the class NP-complete.

A polynomial time reduction $A \leq_m B$ is a function f, computable in polynomial tima such that f maps any input x to A, to an input $f(x)$ to B in such a way that x is a yes input for A iff $f(x)$ is a yes input for B.

We have to extend the concept to function problems.

Reductions

Given problems A and B, assume we have an algorithm $\mathcal{A}_B$ to solve the problem B on any input y.

Reductions

Given problems A and B, assume we have an algorithm $\mathcal{A}_B$ to solve the problem B on any input y.

A polynomial time reduction $A \leq B$ is an algorithm $\mathcal{A}$ solving A that uses calls to $\mathcal{A}_B$ such that $\mathcal{A}$ runs in polynomial time assuming that any call to $\mathcal{A}_B$ takes polynomial time.

Reductions

Given problems A and B, assume we have an algorithm $\mathcal{A}_B$ to solve the problem B on any input y.

A polynomial time reduction $A \leq B$ is an algorithm $\mathcal{A}$ solving A that uses calls to $\mathcal{A}_B$ such that $\mathcal{A}$ runs in polynomial time assuming that any call to $\mathcal{A}_B$ takes polynomial time.

Therefore if we have that $A \leq B$, if there is an algorithm to solve problem B in polynomial time, then we have an algorithm to solve A in polynomial time.

Evaluating a Boolean circuit

Evaluating a Boolean circuit

- A Boolean circuit can be represented by a DAG C in which in-degree 0 vertices are the input gates, out-degree 0 vertices are the output gates. Each vertex that is not an input has a label indicating the type Boolean function implemented by the gate (and,or,not).

Evaluating a Boolean circuit

- A Boolean circuit can be represented by a DAG C in which in-degree 0 vertices are the input gates, out-degree 0 vertices are the output gates. Each vertex that is not an input has a label indicating the type Boolean function implemented by the gate (and,or,not).
- A gate can be evaluated provided all its input values have been evaluated or it is an input gate.

Evaluating a Boolean circuit

- A Boolean circuit can be represented by a DAG C in which in-degree 0 vertices are the input gates, out-degree 0 vertices are the output gates. Each vertex that is not an input has a label indicating the type Boolean function implemented by the gate (and,or,not).
- A gate can be evaluated provided all its input values have been evaluated or it is an input gate.
- We can solve the problem using the following algorithm with input (C, x):

Evaluating a Boolean circuit

- A Boolean circuit can be represented by a DAG C in which in-degree 0 vertices are the input gates, out-degree 0 vertices are the output gates. Each vertex that is not an input has a label indicating the type Boolean function implemented by the gate (and,or,not).
- A gate can be evaluated provided all its input values have been evaluated or it is an input gate.
- We can solve the problem using the following algorithm with input (C, x):
  - Sort the vertices in C in such a way that any vertex is after their predecessors.
  - Assign to the input gates, the determined values
  - Evaluate gates in the computed order
  - Output the values of the output gates.

Evaluating a Boolean circuit

- We can solve the problem using the following algorithm, with input (C, x):

Evaluating a Boolean circuit

- We can solve the problem using the following algorithm, with input (C, x):
  - Sort the vertices in C in such a way that any vertex is after their predecessors.
  - Assign to the input gates, the values determined by x
  - Evaluate gates in the computed order
  - Output the values of the output gates.
- But that is:

Evaluating a Boolean circuit

- We can solve the problem using the following algorithm, with input (C, x):
  - Sort the vertices in C in such a way that any vertex is after their predecessors.
  - Assign to the input gates, the values determined by x
  - Evaluate gates in the computed order
  - Output the values of the output gates.
- But that is:
  - Do a topological sort of the vertices in C.
  - Assign to the input gates, the determined values
  - Evaluate gates in the computed order
  - Output the values of the output gates.

Evaluating a Boolean circuit

- - Do a topological sort of the vertices in C.
  - Assign to the input gates, the determined values
  - Evaluate gates in the computed order
  - Output the values of the output gates.
- Is this a polytime reduction from the problem of evaluating a circuit to the problem of computing a topological sort in a DAG?

Evaluating a Boolean circuit

  - Do a topological sort of the vertices in C.
  - Assign to the input gates, the determined values
  - Evaluate gates in the computed order
  - Output the values of the output gates.
- Is this a polytime reduction from the problem of evaluating a circuit to the problem of computing a topological sort in a DAG? Yes!
- Do we have a polynomial time algorithm that evaluates a Boolean circuit (given C and x)?

Evaluating a Boolean circuit

  - Do a topological sort of the vertices in C.
  - Assign to the input gates, the determined values
  - Evaluate gates in the computed order
  - Output the values of the output gates.
- Is this a polytime reduction from the problem of evaluating a circuit to the problem of computing a topological sort in a DAG? Yes!
- Do we have a polynomial time algorithm that evaluates a Boolean circuit (given C and x)? Yes!