COMPSCI 320S2C, 2008: Algorithmics

Mark C. Wilson

October 8, 2008

Organizational matters

- Lecturer: Dr Mark Wilson.

Organizational matters

- Lecturer: Dr Mark Wilson.
- Email mcw@cs.auckland.ac.nz. Phone extn 86643.

Organizational matters

- Lecturer: Dr Mark Wilson.
- Email `mcw@cs.auckland.ac.nz`. Phone extn 86643.
- Office: City 303.588.

Organizational matters

- Lecturer: Dr Mark Wilson.
- Email mcw@cs.auckland.ac.nz. Phone extn 86643.
- Office: City 303.588.
- Lectures: will stick mostly to textbook, but there may be some extra material. Please ask questions.

Organizational matters

- Lecturer: Dr Mark Wilson.
- Email `mcw@cs.auckland.ac.nz`. Phone extn 86643.
- Office: City 303.588.
- Lectures: will stick mostly to textbook, but there may be some extra material. Please ask questions.
- Other resources: course webpages, lecturers, tutorials, class forum, library (books on reserve).

Overview

- So far we have focused on problems that can be solved by “fast” algorithms, which we have designed using various paradigms.

Overview

- So far we have focused on problems that can be solved by "fast" algorithms, which we have designed using various paradigms.
- It turns out that many problems of practical interest seem very difficult to solve quickly using any of our algorithm design methods. Examples: travelling salesrep, independent set.

Overview

- So far we have focused on problems that can be solved by "fast" algorithms, which we have designed using various paradigms.
- It turns out that many problems of practical interest seem very difficult to solve quickly using any of our algorithm design methods. Examples: travelling salesrep, independent set.
- It also turns out that many of these problems are about equally hard, so solving any one quickly would yield a quick solution for all of them.

Overview

- So far we have focused on problems that can be solved by "fast" algorithms, which we have designed using various paradigms.
- It turns out that many problems of practical interest seem very difficult to solve quickly using any of our algorithm design methods. Examples: travelling salesrep, independent set.
- It also turns out that many of these problems are about equally hard, so solving any one quickly would yield a quick solution for all of them.
- It also turns out that for many of these problems it is easy to verify a solution once guessed, but apparently very hard to find a solution.

Overview

- So far we have focused on problems that can be solved by "fast" algorithms, which we have designed using various paradigms.
- It turns out that many problems of practical interest seem very difficult to solve quickly using any of our algorithm design methods. Examples: travelling salesrep, independent set.
- It also turns out that many of these problems are about equally hard, so solving any one quickly would yield a quick solution for all of them.
- It also turns out that for many of these problems it is easy to verify a solution once guessed, but apparently very hard to find a solution.
- It is widely believed that no algorithms exist to solve these problems quickly, but no one knows for sure. This is the most famous question in computer science: $P = NP$?

Polynomial-time reductions

- A problem is solvable in polynomial time if there is some polynomial p such that every instance of size n can be solved in time at most $p(n)$. Examples: almost everything in your courses so far.

Polynomial-time reductions

- A problem is solvable in polynomial time if there is some polynomial p such that every instance of size n can be solved in time at most $p(n)$. Examples: almost everything in your courses so far.
- If we can't even do this, then surely the problem is intractable (even n^4 grows very fast with n, let alone $n^{1000000}$).

Polynomial-time reductions

- A problem is solvable in polynomial time if there is some polynomial p such that every instance of size n can be solved in time at most $p(n)$. Examples: almost everything in your courses so far.
- If we can't even do this, then surely the problem is intractable (even n^4 grows very fast with n, let alone $n^{1000000}$).
- Suppose that we have problems X and Y, and that we can always solve Y using a polynomial number of calls to a subroutine that solves X, plus a polynomial number of other basic computational steps. Then we say that Y polynomially reduces to X and write $Y \leq_P X$.

Polynomial-time reductions

- A problem is solvable in polynomial time if there is some polynomial p such that every instance of size n can be solved in time at most $p(n)$. Examples: almost everything in your courses so far.
- If we can't even do this, then surely the problem is intractable (even n^4 grows very fast with n, let alone $n^{1000000}$).
- Suppose that we have problems X and Y, and that we can always solve Y using a polynomial number of calls to a subroutine that solves X, plus a polynomial number of other basic computational steps. Then we say that Y polynomially reduces to X and write $Y \leq_P X$.
- Note that the instances of X must be of polynomial size in the input, since we need to write them down before calling the black box.

Polynomial-time reductions

- A problem is solvable in polynomial time if there is some polynomial p such that every instance of size n can be solved in time at most $p(n)$. Examples: almost everything in your courses so far.
- If we can't even do this, then surely the problem is intractable (even n^4 grows very fast with n, let alone $n^{1000000}$).
- Suppose that we have problems X and Y, and that we can always solve Y using a polynomial number of calls to a subroutine that solves X, plus a polynomial number of other basic computational steps. Then we say that Y polynomially reduces to X and write $Y_{\leq P}X$.
- Note that the instances of X must be of polynomial size in the input, since we need to write them down before calling the black box.
- If $Y \leq_P X$ and X can be solved in polynomial time, so can Y. But if Y cannot be solved in polynomial time, neither can X.

Example: independent set and vertex cover

- $IS(k)$: given a graph, and a number k, does there exist a set of nodes of size at least k, no two of which are connected by an edge?

Example: independent set and vertex cover

- $IS(k)$: given a graph, and a number k, does there exist a set of nodes of size at least k, no two of which are connected by an edge?
- $VC(k)$: given a graph, and a number k, does there exist a set of nodes of size at most k, such that every edge contains at least one of these nodes?

Example: independent set and vertex cover

- $IS(k)$: given a graph, and a number k, does there exist a set of nodes of size at least k, no two of which are connected by an edge?
- $VC(k)$: given a graph, and a number k, does there exist a set of nodes of size at most k, such that every edge contains at least one of these nodes?
- Note that a subset S of edges is an independent set if and only if its complement is a vertex cover.

Example: independent set and vertex cover

- $IS(k)$: given a graph, and a number k, does there exist a set of nodes of size at least k, no two of which are connected by an edge?
- $VC(k)$: given a graph, and a number k, does there exist a set of nodes of size at most k, such that every edge contains at least one of these nodes?
- Note that a subset S of edges is an independent set if and only if its complement is a vertex cover.
- Thus $IS(k) \leq_P VC(k)$ and $VC(k) \leq_P IS(k)$.

Example: independent set and vertex cover

- $IS(k)$: given a graph, and a number k, does there exist a set of nodes of size at least k, no two of which are connected by an edge?
- $VC(k)$: given a graph, and a number k, does there exist a set of nodes of size at most k, such that every edge contains at least one of these nodes?
- Note that a subset S of edges is an independent set if and only if its complement is a vertex cover.
- Thus $IS(k) \leq_P VC(k)$ and $VC(k) \leq_P IS(k)$.
- This is a case of reduction by a simple equivalence of problems.

Example: vertex cover and set cover

- $SC(k)$: given a set U of elements, a collection of subsets $S_1, \ldots, S_m$ of U, and a number k, does there exist a collection of at most k of these sets whose union is all of U?

Example: vertex cover and set cover

- $SC(k)$: given a set U of elements, a collection of subsets $S_1, \ldots, S_m$ of U, and a number k, does there exist a collection of at most k of these sets whose union is all of U?
- Suppose we can solve $SC(k)$ with a black box. Given an instance of $VC(k)$, we encode it as an instance of $SC(k)$ as follows.

Example: vertex cover and set cover

- $SC(k)$: given a set U of elements, a collection of subsets $S_1, \ldots, S_m$ of U, and a number k, does there exist a collection of at most k of these sets whose union is all of U?
- Suppose we can solve $SC(k)$ with a black box. Given an instance of $VC(k)$, we encode it as an instance of $SC(k)$ as follows.
- Let U be the set of all edges of the graph. For each vertex i, let S_i be the set of all edges incident to i.

Example: vertex cover and set cover

- $SC(k)$: given a set U of elements, a collection of subsets $S_1, \ldots, S_m$ of U, and a number k, does there exist a collection of at most k of these sets whose union is all of U?
- Suppose we can solve $SC(k)$ with a black box. Given an instance of $VC(k)$, we encode it as an instance of $SC(k)$ as follows.
- Let U be the set of all edges of the graph. For each vertex i, let S_i be the set of all edges incident to i.
- Now U is the union of sets $S_i, i \in I$ if and only if I is a vertex cover.

Example: vertex cover and set cover

- $SC(k)$: given a set U of elements, a collection of subsets $S_1, \dots, S_m$ of U, and a number k, does there exist a collection of at most k of these sets whose union is all of U?
- Suppose we can solve $SC(k)$ with a black box. Given an instance of $VC(k)$, we encode it as an instance of $SC(k)$ as follows.
- Let U be the set of all edges of the graph. For each vertex i, let S_i be the set of all edges incident to i.
- Now U is the union of sets $S_i, i \in I$ if and only if I is a vertex cover.
- Thus $VC(k) \leq_P SC(k)$.

SAT and related problems

- Constraint satisfaction problems occur often in applications.

SAT and related problems

- Constraint satisfaction problems occur often in applications.
- Given set X of n Boolean variables $x_1, \ldots, x_n$, consider terms (a variable or its negation) and clauses $(t_1 \vee t_2 \vee \cdots \vee t_k$ where t_i are terms).

SAT and related problems

- Constraint satisfaction problems occur often in applications.
- Given set X of n Boolean variables $x_1, \ldots, x_n$, consider terms (a variable or its negation) and clauses $(t_1 \vee t_2 \vee \cdots \vee t_k$ where t_i are terms).
- A truth assignment is a function from X to $\{0, 1\}$. It satisfies a clause if the clause evaluates to 1 (iff at least one term in the clause evaluates to 1).

SAT and related problems

- Constraint satisfaction problems occur often in applications.
- Given set X of n Boolean variables $x_1, \ldots, x_n$, consider terms (a variable or its negation) and clauses $(t_1 \vee t_2 \vee \cdots \vee t_k$ where t_i are terms).
- A truth assignment is a function from X to $\{0, 1\}$. It satisfies a clause if the clause evaluates to 1 (iff at least one term in the clause evaluates to 1).
- SAT problem: given a set of k clauses as above, can they be simultaneously satisfied?

SAT and related problems

- Constraint satisfaction problems occur often in applications.
- Given set X of n Boolean variables $x_1, \ldots, x_n$, consider terms (a variable or its negation) and clauses $(t_1 \vee t_2 \vee \cdots \vee t_k$ where t_i are terms).
- A truth assignment is a function from X to $\{0, 1\}$. It satisfies a clause if the clause evaluates to 1 (iff at least one term in the clause evaluates to 1).
- SAT problem: given a set of k clauses as above, can they be simultaneously satisfied?
- $3\text{-}SAT$ problem: as for SAT, but restrict to $k = 3$.

Reduction of 3-SAT to IS

- Given an instance of $3\text{-}SAT$, we need to build an instance of IS whose solution will help solve the original problem (a gadget). One way is given below.

Reduction of 3-SAT to IS

- Given an instance of $3\text{-}SAT$, we need to build an instance of IS whose solution will help solve the original problem (a gadget). One way is given below.
- For each clause i, have 3 nodes v_{i1}, v_{i2}, v_{i3} where the second index j represents the jth term in the clause.

Reduction of 3-SAT to IS

- Given an instance of $3\text{-}SAT$, we need to build an instance of IS whose solution will help solve the original problem (a gadget). One way is given below.
- For each clause i, have 3 nodes v_{i1}, v_{i2}, v_{i3} where the second index j represents the jth term in the clause.
- Now add an edge for each conflict (when a variable occurs in one term but negated in another). This gives a graph G.

Reduction of 3-SAT to IS

- Given an instance of 3-SAT, we need to build an instance of IS whose solution will help solve the original problem (a gadget). One way is given below.
- For each clause i, have 3 nodes v_{i1}, v_{i2}, v_{i3} where the second index j represents the jth term in the clause.
- Now add an edge for each conflict (when a variable occurs in one term but negated in another). This gives a graph G.
- Now the original 3-SAT instance is satisfiable iff G has an independent set of size $\geq k$.

Reduction of 3-SAT to IS

- Given an instance of $3\text{-}SAT$, we need to build an instance of IS whose solution will help solve the original problem (a gadget). One way is given below.
- For each clause i, have 3 nodes v_{i1}, v_{i2}, v_{i3} where the second index j represents the jth term in the clause.
- Now add an edge for each conflict (when a variable occurs in one term but negated in another). This gives a graph G.
- Now the original $3\text{-}SAT$ instance is satisfiable iff G has an independent set of size $\geq k$.
- Thus $3\text{-}SAT \leq_P IS$.

P and NP: basic definitions

- We define input size as the size of a binary string encoding the input.

P and NP: basic definitions

- We define input size as the size of a binary string encoding the input.
- A decision problem D is an algorithmic problem with a yes/no answer. The set X of all input strings for D for which the answer is "yes" is a formal language.

P and NP: basic definitions

- We define input size as the size of a binary string encoding the input.
- A decision problem D is an algorithmic problem with a yes/no answer. The set X of all input strings for D for which the answer is "yes" is a formal language.
- An algorithm A which returns "yes" on an input string s precisely when $s \in X$ is correct for D. In other words, A solves the problem D.

P and NP: basic definitions

- We define input size as the size of a binary string encoding the input.
- A decision problem D is an algorithmic problem with a yes/no answer. The set X of all input strings for D for which the answer is "yes" is a formal language.
- An algorithm A which returns "yes" on an input string s precisely when $s \in X$ is correct for D. In other words, A solves the problem D.
- An algorithm B that takes input strings s and t and outputs "yes" for some value t^* of t precisely when $s \in X$ is a certifier for X (or for D).

P and NP: basic definitions

- We define input size as the size of a binary string encoding the input.
- A decision problem D is an algorithmic problem with a yes/no answer. The set X of all input strings for D for which the answer is "yes" is a formal language.
- An algorithm A which returns "yes" on an input string s precisely when $s \in X$ is correct for D. In other words, A solves the problem D.
- An algorithm B that takes input strings s and t and outputs "yes" for some value t^* of t precisely when $s \in X$ is a certifier for X (or for D).
- Think of t as being a proof (or certificate) that $s \in X$. B does not solve the problem, but can check certificates.

Example

- For $IS(k)$, a certificate might be the alleged independent set S. The verifier must check that $|S| \geq k$, and that no edge of the input graph contains two elements of S.

Example

- For $IS(k)$, a certificate might be the alleged independent set S. The verifier must check that $|S| \geq k$, and that no edge of the input graph contains two elements of S.
- For 3-SAT, a certificate might be a truth assignment of the variables. The verifier computes its value on each clause, and checks that these values all equal 1.

Example

- For $IS(k)$, a certificate might be the alleged independent set S. The verifier must check that $|S| \geq k$, and that no edge of the input graph contains two elements of S.
- For 3-SAT, a certificate might be a truth assignment of the variables. The verifier computes its value on each clause, and checks that these values all equal 1.
- How quickly can this be done?

P and NP: definitions

- P is the class of problems D for which there exists an algorithm A that solves D, and a polynomial p, such that for each input s, the running time of A on input s is at most $p(|s|)$. That is, there is a polynomial time solver.

P and NP: definitions

- P is the class of problems D for which there exists an algorithm A that solves D, and a polynomial p, such that for each input s, the running time of A on input s is at most $p(|s|)$. That is, there is a polynomial time solver.
- NP is the class of problems D for which there exists an algorithm B that verifies D, and polynomials p, q, such that the running time of B on input s, t is at most $p(|s|)$ for each input s, and $|t^*| \leq q(|s|)$. That is, there is a polynomial time certifier.

P and NP: definitions

- P is the class of problems D for which there exists an algorithm A that solves D, and a polynomial p, such that for each input s, the running time of A on input s is at most $p(|s|)$. That is, there is a polynomial time solver.
- NP is the class of problems D for which there exists an algorithm B that verifies D, and polynomials p, q, such that the running time of B on input s, t is at most $p(|s|)$ for each input s, and $|t^*| \leq q(|s|)$. That is, there is a polynomial time certifier.
- NP stands for "nondeterministic polynomial time" — it corresponds to a "nondeterministic Turing machine".

P and NP: definitions

- P is the class of problems D for which there exists an algorithm A that solves D, and a polynomial p, such that for each input s, the running time of A on input s is at most $p(|s|)$. That is, there is a polynomial time solver.
- NP is the class of problems D for which there exists an algorithm B that verifies D, and polynomials p, q, such that the running time of B on input s, t is at most $p(|s|)$ for each input s, and $|t^*| \leq q(|s|)$. That is, there is a polynomial time certifier.
- NP stands for "nondeterministic polynomial time" — it corresponds to a "nondeterministic Turing machine".
- Many important problems are in NP: Examples: VC, IS, 3-SAT, Hamiltonian cycle, travelling salesperson, graph colouring, graph isomorphism, subset sum.

P and NP: consequences

- Clearly $P \subseteq NP$. Biggest problem of theoretical CS: does $P = NP$?

P and NP: consequences

- Clearly $P \subseteq NP$. Biggest problem of theoretical CS: does $P = NP$?
- Most people believe $P \neq NP$.

P and NP: consequences

- Clearly $P \subseteq NP$. Biggest problem of theoretical CS: does $P = NP$?
- Most people believe $P \neq NP$.
- If $P \neq NP$, then there exist hundreds of important problems that will never have a polynomial time algorithm. This would save us wasting time looking for them. Also, hard problems are useful for applications like cryptography.

P and NP: consequences

- Clearly $P \subseteq NP$. Biggest problem of theoretical CS: does $P = NP$?
- Most people believe $P \neq NP$.
- If $P \neq NP$, then there exist hundreds of important problems that will never have a polynomial time algorithm. This would save us wasting time looking for them. Also, hard problems are useful for applications like cryptography.
- If $P = NP$, then all these hard-looking problems are actually easy, but we have not yet found the algorithms. Cryptography would be much harder to do. Mathematicians would be essentially out of business (computers could find all proofs of theorems of reasonable length).

P and NP: consequences

- Clearly $P \subseteq NP$. Biggest problem of theoretical CS: does $P = NP$?
- Most people believe $P \neq NP$.
- If $P \neq NP$, then there exist hundreds of important problems that will never have a polynomial time algorithm. This would save us wasting time looking for them. Also, hard problems are useful for applications like cryptography.
- If $P = NP$, then all these hard-looking problems are actually easy, but we have not yet found the algorithms. Cryptography would be much harder to do. Mathematicians would be essentially out of business (computers could find all proofs of theorems of reasonable length).
- The Clay Mathematics Institute offers US$1 000 000 for a solution to the $P = NP$? problem.

NP-completeness

- What are the hardest problems in NP?

NP-completeness

- What are the hardest problems in NP?
- Suppose that X lies in NP and for all Y in NP, we have $Y \leq_P X$. Then we call X NP-complete.

NP-completeness

- What are the hardest problems in NP?
- Suppose that X lies in NP and for all Y in NP, we have $Y \leq_P X$. Then we call X NP-complete.
- Why do any such problems exist? There is no obvious reason.

NP-completeness

- What are the hardest problems in NP?
- Suppose that X lies in NP and for all Y in NP, we have $Y \leq_P X$. Then we call X NP-complete.
- Why do any such problems exist? There is no obvious reason.
- Cook and Levin proved around 1971 that they do exist (for example, circuit satisfiability problem). It is now known that hundreds of natural problems are NP-complete, for example 3-SAT, VC, IS, SC.

NP-completeness

- What are the hardest problems in NP?
- Suppose that X lies in NP and for all Y in NP, we have $Y \leq_P X$. Then we call X NP-complete.
- Why do any such problems exist? There is no obvious reason.
- Cook and Levin proved around 1971 that they do exist (for example, circuit satisfiability problem). It is now known that hundreds of natural problems are NP-complete, for example 3-SAT, VC, IS, SC.
- Note that if any NP-complete problem is solvable in polynomial time, then $P = NP$. Thus it is widely believed that all of these problems are extremely hard to solve in the worst case.

Circuit-SAT

- Circuit-SAT takes as input an acyclic digraph representing a Boolean circuit.

Circuit-SAT

- Circuit-SAT takes as input an acyclic digraph representing a Boolean circuit.
    - Each source is labelled with a constant 0 or 1, or a variable (all of the variables are different, and these nodes are called inputs).

Circuit-SAT

- Circuit-SAT takes as input an acyclic digraph representing a Boolean circuit.
  - Each source is labelled with a constant 0 or 1, or a variable (all of the variables are different, and these nodes are called inputs).
  - Other nodes labelled by the Boolean operators $\wedge, \neg, \vee$. There is a single sink (the output).

Circuit-SAT

- Circuit-SAT takes as input an acyclic digraph representing a Boolean circuit.
  - Each source is labelled with a constant 0 or 1, or a variable (all of the variables are different, and these nodes are called inputs).
  - Other nodes labelled by the Boolean operators $\wedge, \neg, \vee$. There is a single sink (the output).
- The value of a node is computed by following the Boolean logic in the obvious way.

Circuit-SAT

- Circuit-SAT takes as input an acyclic digraph representing a Boolean circuit.
  - Each source is labelled with a constant 0 or 1, or a variable (all of the variables are different, and these nodes are called inputs).
  - Other nodes labelled by the Boolean operators $\wedge, \neg, \vee$. There is a single sink (the output).
- The value of a node is computed by following the Boolean logic in the obvious way.
- We ask whether there is a truth assignment to the inputs that causes the value of the output to be 1.

Reduction to Circuit-SAT

- Given an arbitrary problem X in NP, we can reduce it to Circuit-SAT. The proof is complicated (not given here) but the main idea is not.

Reduction to Circuit-SAT

- Given an arbitrary problem X in NP, we can reduce it to Circuit-SAT. The proof is complicated (not given here) but the main idea is not.
- Every algorithm for a decision problem that takes a fixed size input can be represented by a circuit whose value on a given input is the correct output for the algorithm on that input. The execution of the algorithm is modelled by the circuit.

Reduction to Circuit-SAT

- Given an arbitrary problem X in NP, we can reduce it to Circuit-SAT. The proof is complicated (not given here) but the main idea is not.
- Every algorithm for a decision problem that takes a fixed size input can be represented by a circuit whose value on a given input is the correct output for the algorithm on that input. The execution of the algorithm is modelled by the circuit.
- If the algorithm has polynomially bounded running time, then the circuit size is also polynomially bounded.

Reduction to Circuit-SAT

- Given an arbitrary problem X in NP, we can reduce it to Circuit-SAT. The proof is complicated (not given here) but the main idea is not.
- Every algorithm for a decision problem that takes a fixed size input can be represented by a circuit whose value on a given input is the correct output for the algorithm on that input. The execution of the algorithm is modelled by the circuit.
- If the algorithm has polynomially bounded running time, then the circuit size is also polynomially bounded.
- Given a polynomial-time certifier B for X, we convert it to a circuit with polynomial input size and use the black box for Circuit-SAT to solve X.

Circuit-SAT reduction example

- Given a graph G, does it contain an independent set of size 2? We encode the input via the adjacency matrix M.

Circuit-SAT reduction example

- Given a graph G, does it contain an independent set of size 2? We encode the input via the adjacency matrix M.
- We have one source for each edge (encoded as a constant based on M) and one input for each node.

Circuit-SAT reduction example

- Given a graph G, does it contain an independent set of size 2? We encode the input via the adjacency matrix M.
- We have one source for each edge (encoded as a constant based on M) and one input for each node.
- Build a circuit that checks that at least two nodes have been chosen. Build another than checks that we don't choose both ends of each edge. Then concatenate these with $\wedge$.

Circuit-SAT reduction example

- Given a graph G, does it contain an independent set of size 2? We encode the input via the adjacency matrix M.
- We have one source for each edge (encoded as a constant based on M) and one input for each node.
- Build a circuit that checks that at least two nodes have been chosen. Build another than checks that we don't choose both ends of each edge. Then concatenate these with $\wedge$.
- The size of the circuit is polynomial in n, the number of nodes.

k-colouring a graph

- Given a graph G, can we assign a colour to each vertex so that no two adjacent vertices have the same colour, using only k colours?

k-colouring a graph

- Given a graph G, can we assign a colour to each vertex so that no two adjacent vertices have the same colour, using only k colours?
- If $k = 2$, this is possible if and only if G is bipartite, and the question can be decided (either way) in linear time by breadth-first search.

k-colouring a graph

- Given a graph G, can we assign a colour to each vertex so that no two adjacent vertices have the same colour, using only k colours?
- If $k = 2$, this is possible if and only if G is bipartite, and the question can be decided (either way) in linear time by breadth-first search.
- If $k \geq 3$, it is an NP-complete problem, as we shall show.

k-colouring a graph

- Given a graph G, can we assign a colour to each vertex so that no two adjacent vertices have the same colour, using only k colours?
- If $k = 2$, this is possible if and only if G is bipartite, and the question can be decided (either way) in linear time by breadth-first search.
- If $k \geq 3$, it is an NP-complete problem, as we shall show.
  - First note that if H is a complete graph on l vertices, then H requires l colours.

k-colouring a graph

- Given a graph G, can we assign a colour to each vertex so that no two adjacent vertices have the same colour, using only k colours?
- If $k = 2$, this is possible if and only if G is bipartite, and the question can be decided (either way) in linear time by breadth-first search.
- If $k \geq 3$, it is an NP-complete problem, as we shall show.
  - First note that if H is a complete graph on l vertices, then H requires l colours.
  - Given an instance of 3-colouring, form such an H (with $k - 3$ vertices) and add an edge from every vertex of G to every vertex of H.

k-colouring a graph

- Given a graph G, can we assign a colour to each vertex so that no two adjacent vertices have the same colour, using only k colours?
- If $k = 2$, this is possible if and only if G is bipartite, and the question can be decided (either way) in linear time by breadth-first search.
- If $k \geq 3$, it is an NP-complete problem, as we shall show.
  - First note that if H is a complete graph on l vertices, then H requires l colours.
  - Given an instance of 3-colouring, form such an H (with $k-3$ vertices) and add an edge from every vertex of G to every vertex of H.
- G is 3-colourable if and only if the bigger graph is k-colourable. So k-colouring is NP-complete if 3-colouring is.

k-colouring a graph

- Given a graph G, can we assign a colour to each vertex so that no two adjacent vertices have the same colour, using only k colours?
- If $k = 2$, this is possible if and only if G is bipartite, and the question can be decided (either way) in linear time by breadth-first search.
- If $k \geq 3$, it is an NP-complete problem, as we shall show.
  - First note that if H is a complete graph on l vertices, then H requires l colours.
  - Given an instance of 3-colouring, form such an H (with $k - 3$ vertices) and add an edge from every vertex of G to every vertex of H.
- G is 3-colourable if and only if the bigger graph is k-colourable. So k-colouring is NP-complete if 3-colouring is.
- It remains to show that 3-colouring is NP-complete.

Reduction from 3-SAT to 3-colouring, I

- Given an instance of 3-SAT, create a graph G with nodes:

  and edges:

Reduction from 3-SAT to 3-colouring, I

- Given an instance of 3-SAT, create a graph G with nodes:
  - v_i and $\bar{v}_i$ for each variable x_i occurring

  and edges:

Reduction from 3-SAT to 3-colouring, I

- Given an instance of 3-SAT, create a graph G with nodes:
  - v_i and $\bar{v}_i$ for each variable x_i occurring
  - special nodes T, F, B

  and edges:

Reduction from 3-SAT to 3-colouring, I

- Given an instance of 3-SAT, create a graph G with nodes:
  - v_i and $\bar{v}_i$ for each variable x_i occurring
  - special nodes T, F, B

  and edges:
  - joining each v_i and $\bar{v}_i$

Reduction from 3-SAT to 3-colouring, I

- Given an instance of 3-SAT, create a graph G with nodes:
  - v_i and $\bar{v}_i$ for each variable x_i occurring
  - special nodes T, F, B

  and edges:
  - joining each v_i and $\bar{v}_i$
  - joining T and F

Reduction from 3-SAT to 3-colouring, I

- Given an instance of 3-SAT, create a graph G with nodes:
  - v_i and $\bar{v}_i$ for each variable x_i occurring
  - special nodes T, F, B

  and edges:
  - joining each v_i and $\bar{v}_i$
  - joining T and F
  - joining each other node to B.

Reduction from 3-SAT to 3-colouring, I

- Given an instance of 3-SAT, create a graph G with nodes:
  - v_i and $\bar{v}_i$ for each variable x_i occurring
  - special nodes T, F, B

  and edges:
  - joining each v_i and $\bar{v}_i$
  - joining T and F
  - joining each other node to B.
- Every 3-colouring of this corresponds to a truth assignment. We need to extend this graph so that only satisfying assignments yield valid 3-colourings.

Reduction from 3-SAT to 3-colouring, II

- For each clause, create a small graph H that attaches to G at the three terms in the clause, and at the nodes T, F, B so that if all terms in the clause are false, no 3-colouring extending in to H is possible. In other words, a valid 3-colouring yields at least one true term in the clause.

Reduction from 3-SAT to 3-colouring, II

- For each clause, create a small graph H that attaches to G at the three terms in the clause, and at the nodes T, F, B so that if all terms in the clause are false, no 3-colouring extending in to H is possible. In other words, a valid 3-colouring yields at least one true term in the clause.
- H can be found fairly easily, and has 6 vertices. Attach one for each clause, independently.

Reduction from 3-SAT to 3-colouring, II

- For each clause, create a small graph H that attaches to G at the three terms in the clause, and at the nodes T, F, B so that if all terms in the clause are false, no 3-colouring extending in to H is possible. In other words, a valid 3-colouring yields at least one true term in the clause.
- H can be found fairly easily, and has 6 vertices. Attach one for each clause, independently.
- 3-colourings of this graph correspond exactly to satisfying assignments of the given set of clauses.

Subset sum problem

- Given a set of integers $X = \{x_1, \ldots, x_n\}$, and target integer W, does there exist a subset of X whose sum equals W?

Subset sum problem

- Given a set of integers $X = \{x_1, \ldots, x_n\}$, and target integer W, does there exist a subset of X whose sum equals W?
- This is a special case of (the decision version of) the knapsack problem.

Subset sum problem

- Given a set of integers $X = \{x_1, \ldots, x_n\}$, and target integer W, does there exist a subset of X whose sum equals W?
- This is a special case of (the decision version of) the knapsack problem.
- A dynamic programming solution runs in time $O(nW)$. This is exponential in the input size, since the input is considered to be given in binary expansion.

Subset sum problem

- Given a set of integers $X = \{x_1, \ldots, x_n\}$, and target integer W, does there exist a subset of X whose sum equals W?
- This is a special case of (the decision version of) the knapsack problem.
- A dynamic programming solution runs in time $O(nW)$. This is exponential in the input size, since the input is considered to be given in binary expansion.
- The problem is clearly in NP (just perform the addition). We reduce 3-SAT to it, to show that it is NP-complete.

Subset sum problem reduction

- Given an instance of 3-SAT with n variables and k clauses, we encode it into the subset sum framework as follows (easily understood by writing a table).

Subset sum problem reduction

- Given an instance of 3-SAT with n variables and k clauses, we encode it into the subset sum framework as follows (easily understood by writing a table).
- For each variable $x_i, i = 0 \ldots n-1$ occurring in a term somewhere, create variables y_i and z_i, initially zero.

Subset sum problem reduction

- Given an instance of 3-SAT with n variables and k clauses, we encode it into the subset sum framework as follows (easily understood by writing a table).
- For each variable $x_i, i = 0 \ldots n-1$ occurring in a term somewhere, create variables y_i and z_i, initially zero.
  - Add 10^i to y_i and z_i.

Subset sum problem reduction

- Given an instance of 3-SAT with n variables and k clauses, we encode it into the subset sum framework as follows (easily understood by writing a table).
- For each variable $x_i, i = 0 \ldots n-1$ occurring in a term somewhere, create variables y_i and z_i, initially zero.
  - Add 10^i to y_i and z_i.
  - If x_i occurs in clause C_j, add 10^{n+j} to y_i; if $\bar{x}_i$ occurs in clause j, add 10^{n+j} to z_i.

Subset sum problem reduction

- Given an instance of 3-SAT with n variables and k clauses, we encode it into the subset sum framework as follows (easily understood by writing a table).
- For each variable $x_i, i = 0 \ldots n-1$ occurring in a term somewhere, create variables y_i and z_i, initially zero.
  - Add 10^i to y_i and z_i.
  - If x_i occurs in clause C_j, add 10^{n+j} to y_i; if $\bar{x}_i$ occurs in clause j, add 10^{n+j} to z_i.
- For each clause C_j, create variables g_j, h_j with value 1.

Subset sum problem reduction

- Given an instance of 3-SAT with n variables and k clauses, we encode it into the subset sum framework as follows (easily understood by writing a table).
- For each variable $x_i, i = 0 \ldots n-1$ occurring in a term somewhere, create variables y_i and z_i, initially zero.
  - Add 10^i to y_i and z_i.
  - If x_i occurs in clause C_j, add 10^{n+j} to y_i; if $\bar{x}_i$ occurs in clause j, add 10^{n+j} to z_i.
- For each clause C_j, create variables g_j, h_j with value 1.
- The target is a decimal whose lowest n digits are all 1 and whose next k digits are all 3. A solution to subset sum gives a satisfying assignment.

NP and co-NP

- Consider a decision problem such that if the answer is no, there is a polynomial sized certificate. The complement (swap yes and no) of this problem is in NP.

NP and co-NP

- Consider a decision problem such that if the answer is no, there is a polynomial sized certificate. The complement (swap yes and no) of this problem is in NP.
- Call the set of such problems co-NP. Example: testing an integer n for primality. If not, can verify that there are x, y with $xy = n$ in polynomial time.

NP and co-NP

- Consider a decision problem such that if the answer is no, there is a polynomial sized certificate. The complement (swap yes and no) of this problem is in NP.
- Call the set of such problems co-NP. Example: testing an integer n for primality. If not, can verify that there are x, y with $xy = n$ in polynomial time.
- Consider the intersection $NP \cap co - NP$. It consists of problems that have a fast verifier, no matter what the answer. Primality is known to be one.

NP and co-NP

- Consider a decision problem such that if the answer is no, there is a polynomial sized certificate. The complement (swap yes and no) of this problem is in NP.
- Call the set of such problems co-NP. Example: testing an integer n for primality. If not, can verify that there are x, y with $xy = n$ in polynomial time.
- Consider the intersection $NP \cap co - NP$. It consists of problems that have a fast verifier, no matter what the answer. Primality is known to be one.
- Primality has recently been shown to lie in P. It is unknown whether there are problems in $NP \cap co - NP$ that are not in P.

NP and co-NP

- Consider a decision problem such that if the answer is no, there is a polynomial sized certificate. The complement (swap yes and no) of this problem is in NP.
- Call the set of such problems co-NP. Example: testing an integer n for primality. If not, can verify that there are x, y with $xy = n$ in polynomial time.
- Consider the intersection $NP \cap co - NP$. It consists of problems that have a fast verifier, no matter what the answer. Primality is known to be one.
- Primality has recently been shown to lie in P. It is unknown whether there are problems in $NP \cap co - NP$ that are not in P.
- It is known that integer factorization is in $NP \cap co - NP$, but it is not known to be in P. Important cryptographic algorithms are based on the assumption that it is not.

Why use randomization?

- Protect against bad worst case input from a malicious adversary.

Why use randomization?

- Protect against bad worst case input from a malicious adversary.
- Algorithms are often conceptually simpler and easier to implement than deterministic ones.

Why use randomization?

- Protect against bad worst case input from a malicious adversary.
- Algorithms are often conceptually simpler and easier to implement than deterministic ones.
- Find multiple solutions to a given problem.

Why use randomization?

- Protect against bad worst case input from a malicious adversary.
- Algorithms are often conceptually simpler and easier to implement than deterministic ones.
- Find multiple solutions to a given problem.
- Break symmetry between competing agents, reducing conflict.

Why use randomization?

- Protect against bad worst case input from a malicious adversary.
- Algorithms are often conceptually simpler and easier to implement than deterministic ones.
- Find multiple solutions to a given problem.
- Break symmetry between competing agents, reducing conflict.
- Find a solution when there are many, but no clear structure to them.

Monte Carlo algorithms

- If the runtime on a given instance is deterministic and the answer is correct with bounded probability, we have a Monte Carlo algorithm.

Monte Carlo algorithms

- If the runtime on a given instance is deterministic and the answer is correct with bounded probability, we have a Monte Carlo algorithm.
- A MC algorithm for a decision problem is biased if one of the answers (yes/no) is always correct when given. In other words, for example, $P(Y|N) \equiv$ P(says Y given answer is N) $= 0, P(N|N) = 1, P(Y|Y) = p, P(N|Y) = 1 - p$.

Monte Carlo algorithms

- If the runtime on a given instance is deterministic and the answer is correct with bounded probability, we have a Monte Carlo algorithm.
- A MC algorithm for a decision problem is biased if one of the answers (yes/no) is always correct when given. In other words, for example, $P(Y|N) \equiv$ P(says Y given answer is N) $= 0, P(N|N) = 1, P(Y|Y) = p, P(N|Y) = 1 - p$.
- Examples: fingerprinting (verification of identities); primality testing.

Fingerprinting

- Suppose we have a set U and want to determine whether elements u, v are equal. It is often easier to choose a fingerprinting function f and compute whether $f(u) = f(v)$, in which case we return yes. Such methods are always biased: $P(N|Y) = 0$.

Fingerprinting

- Suppose we have a set U and want to determine whether elements u, v are equal. It is often easier to choose a fingerprinting function f and compute whether $f(u) = f(v)$, in which case we return yes. Such methods are always biased: $P(N|Y) = 0$.
- Example: X, Y, Z are $n \times n$ matrices, and we want to know whether $XY = Z$. Choose a random n bit vector r and compute XYr and Zr. Can show $P(Y|N) \leq 1/2$.

Fingerprinting

- Suppose we have a set U and want to determine whether elements u, v are equal. It is often easier to choose a fingerprinting function f and compute whether $f(u) = f(v)$, in which case we return yes. Such methods are always biased: $P(N|Y) = 0$.
- Example: X, Y, Z are $n \times n$ matrices, and we want to know whether $XY = Z$. Choose a random n bit vector r and compute XYr and Zr. Can show $P(Y|N) \leq 1/2$.
- Example: M is a symbolic matrix in variables $x_1, \ldots x_n$; we want to know whether $\det(M) = 0$. Choose a finite subset S of $\mathbb{C}$ and choose $r_1, \ldots r_n$ independently and uniformly from S. Substitute $x_i = r_i$ for all i and compute the determinant. Can show $P(Y|N) \leq m/|S|$ where m is the total degree of the polynomial $\det(M)$.

Fingerprinting

- Suppose we have a set U and want to determine whether elements u, v are equal. It is often easier to choose a fingerprinting function f and compute whether $f(u) = f(v)$, in which case we return yes. Such methods are always biased: $P(N|Y) = 0$.
- Example: X, Y, Z are $n \times n$ matrices, and we want to know whether $XY = Z$. Choose a random n bit vector r and compute XYr and Zr. Can show $P(Y|N) \leq 1/2$.
- Example: M is a symbolic matrix in variables $x_1, \ldots x_n$; we want to know whether $\det(M) = 0$. Choose a finite subset S of $\mathbb{C}$ and choose $r_1, \ldots r_n$ independently and uniformly from S. Substitute $x_i = r_i$ for all i and compute the determinant. Can show $P(Y|N) \leq m/|S|$ where m is the total degree of the polynomial $\det(M)$.
- There are many specific applications of the above examples.

Improving biased Monte Carlo algorithms

- Let $0 < p < 1$. Say a MC algorithm is p-correct if its error probability is at most $1 - p$ on every instance (sometimes p depends on the size, but never on the instance itself).

Improving biased Monte Carlo algorithms

- Let $0 < p < 1$. Say a MC algorithm is p-correct if its error probability is at most $1 - p$ on every instance (sometimes p depends on the size, but never on the instance itself).
- This means $P(Y|N) \leq 1 - p, P(N|Y) \leq 1 - p$.

Improving biased Monte Carlo algorithms

- Let $0 < p < 1$. Say a MC algorithm is p-correct if its error probability is at most $1 - p$ on every instance (sometimes p depends on the size, but never on the instance itself).
- This means $P(Y|N) \leq 1 - p, P(N|Y) \leq 1 - p$.
- If, say, NO is always right then we can improve our confidence in the answer by repeating the algorithm n times on the same instance. This is amplification of the stochastic advantage. If we ever get NO, report NO. Else report YES. Probability of error is at most $(1 - p)^n$. To reduce this to ε requires number of trials proportional to $\lg(1/\varepsilon)$ and to $-\lg(1 - p)$.

Improving unbiased Monte Carlo algorithms

- We need $p \geq 1/2$. Repeat n (odd) times and return the more frequent answer. Analysis is more complicated.

Improving unbiased Monte Carlo algorithms

- We need $p \geq 1/2$. Repeat n (odd) times and return the more frequent answer. Analysis is more complicated.
- Let $X_i = 1$ if ith run gives correct answer, 0 otherwise. Then X_i is a Bernoulli random variable and $X = \sum_{i=1}^{n} X_i$ is binomial with parameter p. Probability of error of repeated algorithm is $P(X < n/2)$. This is just $\sum_{j<n/2} \binom{n}{j} p^j (1-p)^{n-j}$.

Improving unbiased Monte Carlo algorithms

- We need $p \geq 1/2$. Repeat n (odd) times and return the more frequent answer. Analysis is more complicated.
- Let $X_i = 1$ if ith run gives correct answer, 0 otherwise. Then X_i is a Bernoulli random variable and $X = \sum_{i=1}^{n} X_i$ is binomial with parameter p. Probability of error of repeated algorithm is $P(X < n/2)$. This is just $\sum_{j<n/2} \binom{n}{j} p^j (1-p)^{n-j}$.
- Simplifying this could be done, but it is easier to use the normal approximation: X is approximately normal with mean np and variance $np(1-p)$ for n large enough. Use table of normal distribution to work out size of n for given ε. Answer is proportional to $\lg(1/\varepsilon)$ and $(p - 1/2)^{-2}$.

Primality testing

- We wish to know whether a given positive integer n is prime (has no proper factor other than 1).

Primality testing

- We wish to know whether a given positive integer n is prime (has no proper factor other than 1).
- Fermat (1640) proved that if n is prime, and $1 \leq a \leq n-1$, then $a^{n-1} \equiv 1 \mod n$. However if n is composite (not prime) then this equality may or may not hold.

Primality testing

- We wish to know whether a given positive integer n is prime (has no proper factor other than 1).
- Fermat (1640) proved that if n is prime, and $1 \leq a \leq n-1$, then $a^{n-1} \equiv 1 \mod n$. However if n is composite (not prime) then this equality may or may not hold.
- An obvious idea is: given n, choose a randomly and compute $a^{n-1} \mod n$ (recall that we can do this quickly, using a divide-and conquer approach). If the answer is not 1, we report NO (such an a is called a witness).

Primality testing

- We wish to know whether a given positive integer n is prime (has no proper factor other than 1).
- Fermat (1640) proved that if n is prime, and $1 \leq a \leq n-1$, then $a^{n-1} \equiv 1 \mod n$. However if n is composite (not prime) then this equality may or may not hold.
- An obvious idea is: given n, choose a randomly and compute $a^{n-1} \mod n$ (recall that we can do this quickly, using a divide-and conquer approach). If the answer is not 1, we report NO (such an a is called a witness).
- This gives a biased Monte Carlo algorithm with $P(N|N) = 1, P(Y|N) = 0$. What about $P(N|Y), P(Y|Y)$?

Primality testing

- We wish to know whether a given positive integer n is prime (has no proper factor other than 1).
- Fermat (1640) proved that if n is prime, and $1 \leq a \leq n-1$, then $a^{n-1} \equiv 1 \mod n$. However if n is composite (not prime) then this equality may or may not hold.
- An obvious idea is: given n, choose a randomly and compute $a^{n-1} \mod n$ (recall that we can do this quickly, using a divide-and conquer approach). If the answer is not 1, we report NO (such an a is called a witness).
- This gives a biased Monte Carlo algorithm with $P(N|N) = 1, P(Y|N) = 0$. What about $P(N|Y), P(Y|Y)$?
- Unfortunately $P(N|Y)$ can be made arbitrarily close to 1 (some n have a lot of false witnesses). So this algorithm is not p-correct for any $p > 0$.

Improving the primality testing algorithm

- There is a more complicated test (Miller-Rabin, 1976 - see textbook) based on Fermat's result that gives $P(N|Y) \leq 1/4$, and hence a $3/4$-correct algorithm.

Improving the primality testing algorithm

- There is a more complicated test (Miller-Rabin, 1976 - see textbook) based on Fermat's result that gives $P(N|Y) \leq 1/4$, and hence a $3/4$-correct algorithm.
- There is an even more complicated test (Agrawal-Kayal-Saxena 2002) that is in fact always correct, and this gives the first worst-case polynomial-time algorithm for primality. But it is not as fast in practice as the randomized algorithm above.

Las Vegas algorithms

- Nice properties: can find more than one solution even on same input; breaks the link between input and worse-case runtime.

Las Vegas algorithms

- Nice properties: can find more than one solution even on same input; breaks the link between input and worse-case runtime.
- Every Las Vegas algorithm can be converted to a Monte Carlo algorithm: just report a random answer if the running time gets too long.

Las Vegas algorithms

- Nice properties: can find more than one solution even on same input; breaks the link between input and worse-case runtime.
- Every Las Vegas algorithm can be converted to a Monte Carlo algorithm: just report a random answer if the running time gets too long.
- Examples:

Las Vegas algorithms

- Nice properties: can find more than one solution even on same input; breaks the link between input and worse-case runtime.
- Every Las Vegas algorithm can be converted to a Monte Carlo algorithm: just report a random answer if the running time gets too long.
- Examples:
  - randomized quicksort and quickselect;

Las Vegas algorithms

- Nice properties: can find more than one solution even on same input; breaks the link between input and worse-case runtime.
- Every Las Vegas algorithm can be converted to a Monte Carlo algorithm: just report a random answer if the running time gets too long.
- Examples:
  - randomized quicksort and quickselect;
  - randomized greedy algorithm for n queens problem;

Las Vegas algorithms

- Nice properties: can find more than one solution even on same input; breaks the link between input and worse-case runtime.
- Every Las Vegas algorithm can be converted to a Monte Carlo algorithm: just report a random answer if the running time gets too long.
- Examples:
    - randomized quicksort and quickselect;
    - randomized greedy algorithm for n queens problem;
    - integer factorization;

Las Vegas algorithms

- Nice properties: can find more than one solution even on same input; breaks the link between input and worse-case runtime.
- Every Las Vegas algorithm can be converted to a Monte Carlo algorithm: just report a random answer if the running time gets too long.
- Examples:
  - randomized quicksort and quickselect;
  - randomized greedy algorithm for n queens problem;
  - integer factorization;
  - universal hashing;

Las Vegas algorithms

- Nice properties: can find more than one solution even on same input; breaks the link between input and worse-case runtime.
- Every Las Vegas algorithm can be converted to a Monte Carlo algorithm: just report a random answer if the running time gets too long.
- Examples:
    - randomized quicksort and quickselect;
    - randomized greedy algorithm for n queens problem;
    - integer factorization;
    - universal hashing;
    - linear time MST.

Improving Las Vegas algorithms

- If a particular run takes too long, we can just stop and run again on the same input. If the expected runtime is fast, it is very unlikely that many iterations will fail to run fast.

Improving Las Vegas algorithms

- If a particular run takes too long, we can just stop and run again on the same input. If the expected runtime is fast, it is very unlikely that many iterations will fail to run fast.
- To quantify this: let p be probability of success, s the expected time to find a solution, and f the expected time when failure occurs. Then expected time t until we find a solution is given by $t = ps + (1-p)(f+t)$, so $t = s + (1-p)f/p$.

Improving Las Vegas algorithms

- If a particular run takes too long, we can just stop and run again on the same input. If the expected runtime is fast, it is very unlikely that many iterations will fail to run fast.
- To quantify this: let p be probability of success, s the expected time to find a solution, and f the expected time when failure occurs. Then expected time t until we find a solution is given by $t = ps + (1-p)(f+t)$, so $t = s + (1-p)f/p$.
- We can use this to optimize the repeated algorithm.