Lecture 2: Combinatorial search and optimisation problems

- Different types of computational problems
- Examples of computational problems
- Relationships between problems
- Computational properties of different problems.

Computational problems

- A (computational) problem: an infinite set of possible instances with a question.
- A decision problem: a question with a yes/no answer

Example

REACHABILITY
INSTANCE: A graph (V,E) and nodes $v,u \in V$.
QUESTION: Is there a path in the graph from v to u?

Computational problems

Often more complicated questions are of interest:

- Search (function) problem:
  given an instance find a solution (object satisfying certain properties).
- Optimization problem:
  given an instance find a best solution according to some cost criterion.
  Typically this is formalized by specifying
  - what are feasible solutions for an instance and
  - a cost function which assigns a cost (typically a integer/real number) to each feasible solution.

  Now a solution to an optimization problem instance is a feasible solution that has the minimal (or maximal) cost.
- Counting problem:
  given an instance count the number of solutions.

Examples

- PATH
  INSTANCE: A graph (V,E) and nodes $v,u \in V$.
  QUESTION: Find a path from v to u.
- SHORTEST PATH
  INSTANCE: A graph (V,E) and nodes $v,u \in V$.
  QUESTION: Find a shortest path from v to u.
- #PATH
  INSTANCE: A graph (V,E) and nodes $v,u \in V$.
  QUESTION: Count the number of simple paths from v to u.

Easy and hard problems

- Many problems are computationally easy: there is a polynomial time algorithm for the problem, i.e. there is an algorithm solving the problem whose run time increases polynomially w.r.t. the size of the input instance. Consider, e.g., REACHABILITY.
- Some problems are not computationally easy: there is no known guaranteed polynomial time algorithm for the problem, i.e. for any known algorithm there is an infinite collection of instances for which the run time increases super-polynomially w.r.t. the size of the instance.
- This course focuses on methods for solving such problems in practice.

T–79.4201 Search Problems and Algorithms

Examples of hard problems

- SAT
  INSTANCE: a propositional formula in conjunctive normal form
  QUESTION:
  (D) Is the formula satisfiable?
  (S) Find a satisfiable truth assignment for the formula.
  (O) Find a truth assignment that satisfies the most clauses in the formula.
- GRAPH COLORING
  INSTANCE: A graph (V, E) and a positive integer k
  QUESTION:
  (D) Is there a k-coloring of the graph, i.e. an assignment of one of the k colors to each vertex such that vertices connected with an edge do not have the same color?
  (S) Find a k-coloring.
  (O) Find an l-coloring with the smallest number l of colors.

I.N. & P.O. Autumn 2006

T–79.4201 Search Problems and Algorithms

Examples of hard problems (II)

- CLIQUE
  INSTANCE: A graph (V, E) and a positive integer k
  QUESTION:
  (D) Is there a k-clique in the graph, i.e. a set of k nodes such that there is an edge between every pair of vertices from the set.
  (S) Find a k-clique.
  (O) Find an l-clique with the largest number l of vertices.
- SET COVER
  INSTANCE: A family of sets $F = \{S_1, \ldots, S_n\}$ of subsets of a finite set U and a positive integer k.
  QUESTION:
  (D) Is there k-cover of U, i.e., a set of k sets from F whose union is U.
  (S) Find a k-cover of U.
  (O) Find a set l-cover of U with the smallest number l of sets.

I.N. & P.O. Autumn 2006

T–79.4201 Search Problems and Algorithms

Examples of hard problems (III)

TSP (TRAVELING SALESPERSON)
INSTANCE: n cities $1, \ldots, n$ and a nonnegative integer distance d_{ij} between any two cities i and j (such that $d_{ij} = d_{ji}$) and a positive integer B.
QUESTION:
(D) Is there a tour of length at most B, i.e. a permutation π of the cities such that the length

$$\sum_{i=1}^{n} d_{\pi(i)\pi(i+1)}$$

is at most B (where $\pi(n+1) = \pi(1)$)?
(S) Find a tour of length at most B.
(O) Find the shortest tour of the cities.

I.N. & P.O. Autumn 2006

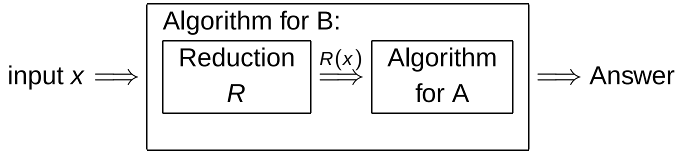
Relationship between problems

- An interesting relationship between two computational problems A and B is that of a reduction.
- B reduces to A ($B \sqsubseteq A$) if there is a transformation R which for every input instance x of B produces an equivalent input instance $R(x)$ of A (where equivalent means that the answer (yes/no) for $R(x)$ considered as the input of A is the correct answer to x as an input of B).
- For a reduction to be useful it needs to be relatively easy to compute (compared to the problems A and B).
- Typically it is assumed that the reduction can be computed in polynomial time.

Reduction

Reduction from B to A ($B \sqsubseteq A$) can be exploited in two interesting ways:

- an algorithm for B can be built on top of an algorithm for A.
- reduction implies that A is computationally at least as hard as B.



- The former is used extensively in the course.
- The latter is used in computational complexity theory (T-79.5103) to classify computational problems; $B \sqsubseteq A$ orders problems by difficulty.

Example: 3-COL $\sqsubseteq$ SAT

- 3-COL
  INSTANCE: a graph (V,E).
  QUESTION: is there a 3-coloring of the graph.
- Reduction from 3-COL to SAT

For each vertex $v \in V$:	For each edge $(v,u) \in E$:
$v(1) \vee v(2) \vee v(3)$	$\neg v(1) \vee \neg u(1)$
$\neg v(1) \vee \neg v(2)$	$\neg v(2) \vee \neg u(2)$
$\neg v(1) \vee \neg v(3)$	$\neg v(3) \vee \neg u(3)$
$\neg v(2) \vee \neg v(3)$	

- This is a reduction because
  (i) it can be computed efficiently and
  (ii) it produces from an instance of 3-COL an equivalent instance of SAT: the graph has a 3-coloring iff the set of clauses is satisfiable.

Example: 3-SAT $\sqsubseteq$ INDEPENDENT SET

- INDEPENDENT SET
  INSTANCE: A graph $G = (V,E)$ and an integer K.
  QUESTION: Is there an independent set $I \subseteq V$ with $|I| = K$.
  (A set $I \subseteq V$ is independent if $i,j \in I$ implies that there is no edge between i and j).
- Reduction from 3-SAT to INDEPENDENT SET
  Given a set ϕ of m clauses each with three literals, construct a graph whose vertices are the occurrences of the literals in ϕ and add edges so that for each clause there is a separate triangle and then add an edge between two vertices in different triangles if they correspond to complementary literals.
  Finally, set $K = m$.

Example: 3-SAT $\sqsubseteq$ INDEPENDENT SET—cont'd

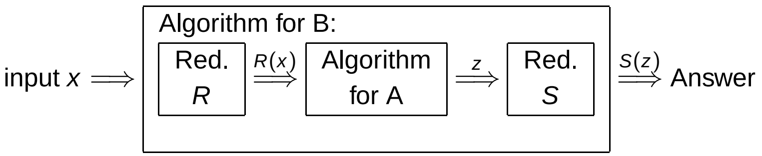
- This is a reduction because ϕ is satisfiable iff there is an independent set of size m for the graph.
  ($\Rightarrow$) If ϕ has a satisfying truth assignment, then take one vertex from each triangle for which the corresponding literal is true in the assignment and this gives an independent set of size m.
  ($\Leftarrow$) If there is an independent set of size m, then it contains exactly one vertex from each triangle and no two vertices corresponding to complementary literals. Hence, the set induces a truth assignment for which each clause has a true literal implying that ϕ is satisfiable.

Example: INDEPENDENT SET $\sqsubseteq$ CLIQUE

- Reduction from INDEPENDENT SET to CLIQUE
  Given a $G = (V, E)$ and an integer K, take the complement graph $G' = (V, \{(v,u) \mid v, u \in V, (v,u) \notin E\}$.
- This is a reduction because an independent set of a graph is a clique of the complement graph.
- Reductions compose (are transitive):
  3-SAT $\sqsubseteq$ INDEPENDENT SET and
  INDEPENDENT SET $\sqsubseteq$ CLIQUE imply
  3-SAT $\sqsubseteq$ CLIQUE
- Hence, using an algorithm for CLIQUE, we can solve INDEPENDENT SET, 3-SAT, 3-COL using reductions.

Reductions—cont'd

- Reductions for search problems need a translation of the result back to the original problem:
  A reduction from a search problem B to A is a pair of mappings (R, S) (both computable in polynomial time) such that for all x, z: if x is an instance of B, then $R(x)$ is an instance of A and if z is a correct output of $R(x)$, then $S(z)$ is a correct output of x.
- For optimization problems optimality needs to be preserved, too.



Size of the reductions

In practice not all polynomial time reductions are useful in building algorithms on top of others but the size of the translation matters.

Example

- Consider a problem A for which we have a $2^{n/1000}$ algorithm.
  Hence, an input of length n=20000 needs $2^{20000/1000} \approx 10^6$ steps.
- We want to use this algorithm to solve a difficult problem B for which we have a quadratic translation to A.
- Now the run time of the combined algorithm for B is $p(n) + 2^{n^2/1000}$ where $p(n)$ is a polynomial giving the run time of the translation from B to A.
- For an input of length n=20000 the run time is $p(20000) + 2^{20000^2/1000} \geq 2^{400000} \geq 10^{10000}$ steps!

Relationship between different kinds of problems

Decision problems vs search problems

- A decision problem reduces to the corresponding search problem trivially, i.e., if a search problem can be solved efficient so can the corresponding decision problem.
- But also a search problem reduces to the corresponding decision problem.

SET COVER(D) vs SET COVER(S)

- If SET COVER(S) can solved in polynomial time, then so can SET COVER(D).
- If SET COVER(D) can solved in polynomial time, then so can SET COVER(S) using the following algorithm given a family $F = \{S_1, \ldots, S_n\}$ of subsets of U and a positive integer k.

  if setcover(F, k) returns "no" then return "no";
  l := k-1;
  for all $S \in \{S_1, \ldots, S_n\}$ do
    if setcover($F[S := \textbf{true}]$, l) returns "yes" then
      $T(S) := \textbf{true}$; $F := F[S := \textbf{true}]$; l := l - 1
    else $T(S) := \textbf{false}$; $F := F[S := \textbf{false}]$;
  return T;

  where setcover(F,k) is a procedure deciding whether F has a k-cover; $F[S := \textbf{true}]$ denotes F with the set S and its elements removed from F and U; $F[S := \textbf{false}]$ is just the set S removed from F; and $\{S \in F \mid T(S) = \textbf{true}\}$ is the computed k-cover;

Decision vs optimization problems

Consider TSP(D) vs TSP(O)

- If TSP(O) can solved in polynomial time, then so can TSP(D).
- If TSP(D) can solved in polynomial time, then so can TSP(O).
- An optimal tour can be found using an algorithm which
  1. finds the cost C of an optimal tour by binary search (with a polynomial number of calls to the polynomial time algorithm for TSP(D));
  2. finds an optimal tour using C (with a polynomial number of calls to the polynomial time algorithm for TSP(D)).

TSP(D) vs TSP(O)

A TSP(O) algorithm using a TSP(D) algorithm as a subroutine:

/*Find the cost C of an optimal tour by binary search*/
$C := 0$; $C_u := 2^n$;
while $(C_u > C)$ do
  if there is a tour of cost $\lfloor (C_u + C)/2 \rfloor$ or less then
    $C_u := \lfloor (C_u + C)/2 \rfloor$
  else $C := \lfloor (C_u + C)/2 \rfloor + 1$;
/* Find an optimal tour */
For every intercity distance $d(i,j)$ do
  set the distance to $C + 1$;
  if there is a tour of cost C or less, freeze the distance to $C + 1$
  else restore the original distance and add (i,j) to the tour;
endfor

Different kinds of optimization problems

- Consider the traveling salesperson problem and two new variants:
  EXACT TSP: Given a distance matrix and an integer B, is the length of the shortest tour equal to B?
  TSP COST: Given a distance matrix, compute the length of the shortest tour.
- It can be shown that the four variants can be ordered in "increasing complexity" by reductions:
  TSP(D) ; EXACT TSP; TSP COST; TSP(O)
- All the four variants of TSP are polynomially equivalent: there is a polynomial-time algorithm for one iff there is one for all four (because TSP(D) and TSP(O) are).

Computational properties of problems

- The previous arguments indicate that for a problem the decision, search, and optimization variants are polynomially equivalent.
- However, this does not imply that they are equally easy to solve in practice.
- There are differences if no polynomial algorithm is known.
- For a decision problem the "yes" answer is often easy to verify.
  - Typically, the question is about existence of a certain objects (witness/certificate) such as a satisfying truth assignment, a coloring, . . .
  - If the witness is given, then the correctness of the "yes" answer can be checked in polynomial time.
  - However, the "no" answer is more challenging to verify because there is no obvious witness/certificate for the answer, e.g., for the lack of coloring.

Computational properties of problems (II)

- The same holds for search problems where the correctness of the found object can typically be checked in polynomial time but where the "no" answer is more challenging to verify.
- Notice that even if the verification of a solution is easy, this does not imply that finding a solution is easy.
- Many engineering problems fall into this class of problems
  - A typical problem is to construct a mathematical object satisfying certain specifications (path, solution of equations, routing, VLSI layout,. . .).
  - The decision version of the problem is determine whether at least one such an object exists for the input.
  - The object is usually not very large compared to the input.
  - The specifications of the object are usually simple enough to be checkable in polynomial time.

Computational properties of problems (III)

- The decision versions of this class of problems form the problem class NP, i.e., decision problems with polynomial size certificates that are checkable in polynomial time.
- The hardest problems in this class (w.r.t. $\sqsubseteq$) are called NP-complete problems and they include, for example, SAT, GRAPH COLORING, CLIQUE, SET COVER, TSP, . . .
- To learn more, see computational complexity theory, for example, course T-79.5103 in the autumn term.
- For optimization problems it is hard even to verify a solution.
  - Consider an instance of the traveling salesperson problem and its potential solution π.
  - There seems to be no obvious polynomial time test that could establish that π is actually a tour of the cities that has the shortest possible length.
- Counting problems are often even harder.

Computational properties of optimization problems

- The computational hardness of verifying a solution depends on the type of an optimization problem.
- EXACT TSP: checking whether the length of the shortest tour equals to B requires two calls to the decision problem:
  - check whether there is a tour of length at most B?
  - check whether there is not a tour of length at most $B - 1$?
- However, checking the length of the shortest tour seems to require polynomial number of adaptive calls to the decision procedure (see binary search above).
- The same holds for checking the shortest tour.

Algorithm design techniques for hard problems

- There are several approaches to developing efficient algorithms for computationally challenging problems such as:
  - identify special cases (using tools from complexity theory) and develop special algorithms for these
  - approximation algorithms
  - randomized algorithms
- However, it typically requires a substantial amount of expertise and resources to develop an efficient algorithm for a problem.
- For example, in practical applications it often happens that the problem specification is not “mathematically clean” but includes a number of “side conditions” and criteria which are fairly complicated to integrate into an algorithm. Moreover, these “side conditions” tend to change quite frequently.
- In this course we study search algorithms as a practical set of tools to solve such problems.