Aalto University
School of Science

T-79.4101 Discrete Models and Search

Emilia Oikarinen

Department of Information and Computer Science, Aalto University

Spring 2015

T–79.4101 Discrete Models and Search (5 ECTS)

At this course you will learn to represent combinatorial search problems in terms of propositional satisfiability, constraint programming, and integer programming formulations. You will obtain a basic understanding of linear programming methodology and become familiar with several types of local search techniques. Having completed the course, you will be able to translate your problem into an appropriate general formulation and use a generic problem solver to solve the problem, or design a local search method tailored specifically to your problem of interest.

The course material is based on material by Prof. Ilkka Niemelä, Prof. Pekka Orponen, Dr. André Schumacher and Dr. Emilia Oikarinen.

Why this course?

- With the increase in computing power, continually new computation-intensive application areas emerge (e.g. various types of planning & scheduling, data mining, bioinformatics,. . .)
- Many immediate problems in these areas are both computationally demanding & mathematically weakly structured (“Here is my messy objective function. Find a near-optimal solution to it – quickly!”)
- In such “quick-and-dirty” settings a search problem formulation is often the most effective (if not the only) approach.
- Moreover, the design and analysis of search algorithms is a fascinating research topic in itself!

Practical arrangements

Lectures: Tue 10–12 T6, Emilia Oikarinen

Tutorials: Thu 16–18 T3, Laura Koponen *(Starting Thu Jan. 15)*

Registration: by WebOodi: `https://oodi.aalto.fi/`

Prerequisites: Basic knowledge of problem representations and logic, facility in programming, data structures and algorithms; T-79.4202 Principles of Algorithmic Techniques recommended

Requirements: Examination (Apr. 8) and two programming assignments (due Feb. 22 and Apr. 12)
Recall Aalto SCI policy: You MUST register to the examinition beforehand!

Course home page: in Noppa:
`https://noppa.tkk.fi/noppa/kurssi/t-79.4101/`

Please: follow us in Noppa! (i.e., subscribe to the news)

Grading scheme

Exam: max 40 points

Programming: max 10 points

Extra assignments$^{(\star)}$: max 5 points

Tutorials: max 10 points

Note: The maximum number of points that can be obtained is 65.
(Note: we consider about half of the achievable points of programming assignments and tutorials as well as the points from extra assignment as bonus points)

The final grade is be computed as follows:
(see also the next slide for the requirements for passing the course!)

Total points	25	31	37	43	49
Grade	1	2	3	4	5

($\star$) Extra assignments are voluntary assignments related to some extra material and involving independent work.

Grading scheme—cont'd

To *pass the course* you need to satisfy **all** the following requirements:

- Exam: $\geq$ 20 points
- Total: $\geq$ 25 points
- *Both programming assignments completed successfully*

For the programming assignments, a correctly functioning program, returned *on time* with *appropriate work description* yields approx. half the points available. The remaining points are allocated based on an efficiency competition among the submitted programs and on the quality of the report. Details of the grading scheme for programming assignments will be introduced later.

Tutorial sessions

- Two sets of problems each week: demonstration problems and homework problems
- Assistant presents the solutions for the demonstration problems.
- Assistant is ready to answer questions
- Homework problems ($\approx$ 24 in total) to be solved until the following week
- Each problem one is ready to present at the whiteboard counts for the tutorial points as follows:

Solved problems	2	4	6	8	10	12	14	16	17	18
Points	1	2	3	4	5	6	7	8	9	10

- Even if you are not interested in tutorial points, come to the session and participate actively (saves you time later when studying for the exam)
- Be active, ask questions!

Programming assignments

- We will use automated system called Stratum[1] for distributing, submitting and checking the assignments
- Supported languages: Python and Java
- Start working early! (e.g., not on the day of the deadline)
- Read instructions carefully!
- If something is unclear, please ask (preferably early) $\rightarrow$ helps you and others
- Often main challenge: *how to model a problem*
- Assignments are **compulsory**!

[1] `https://puzzle.ics.hut.fi/`

Material

- No existing textbook: lectures cover a wide range of material from several textbooks & current scientific literature.
- Course problems based on lecture slides.
- Examples of reference material:
  - Aarts & Lenstra (Eds.), *Local Search in Combinatorial Optimization.* Wiley 1997.
  - Apt, *Principles of Constraint Programming.* Cambrigde University Press, 2003.
  - T. Bäck, *Evolutionary Algorithms in Theory and Practice.* Oxford University Press, 1996.
  - Hoos & Stützle, *Stochastic Local Search: Foundations and Applications.* Morgan Kaufmann 2005.
  - Russell & Norvig, *Artificial Intelligence: A Modern Approach*, Pearson Education, 2010.

A Motivating Example

- Twelve slightly different types of items, numbered 1...12, arrive for processing at a factory workshop.
- The workshop has four machines, numbered I...IV, and four workers, named A...D, who have different qualifications for working on the items.
- To make things more complicated, there are also four specialized tools, numbered $\alpha \ldots \delta$, that are needed for processing the various items.

The requirements of machines, tools, and workers for the items are indicated in the following table:

Machine				Tool				Worker			
I:	1	5	9	α:	1	2	3	A:	1	7	8
II:	2	6	10	β:	4	9	10	B:	2	3	4
III:	3	7	11	γ:	5	11	12	C:	5	6	12
IV:	4	8	12	δ:	6	7	8	D:	9	10	11

Let's say processing each item by a combination of the appropriate machine, tool & worker (e.g., the combination (III, α, B) to process item 3), requires 1 hour. Any given machine, tool, or worker can only work on one item at a time. Since there are 12 items and 4 machines (as well as tools & workers), processing all the items requires at least 3 hours. Can it be done in this minimal time?

How would you approach the preceding problem:

(a) By hand? (Design an appropriate schedule!)

(b) By computer, assuming that an arbitrary list of requirements such as above would be given as input? (The numbers of machines, tools, and workers do not need to be the same: this is just a peculiarity of the present example.)

Outline of this course (tentative schedule!)

Part I: Intro, search algorithms: complete & local search

Lecture 1	Introduction and general information **(today)**
Homework	Study material on propositional logic!
Lecture 2	Computational problems and their properties; reductions between problems from an algorithmic point-of-view **(13.1.)**
Lecture 3	*Complete search methods*: search spaces, backtrack and branch-and-bound search; *local search methods* (hill climbing, simulated annealing, and tabu search) **(20.1.)**
Assignment 1	Local search **(DL: 22.2.)**

Part II: CSP models and algorithms, Boolean circuits

Lecture 4 *Constraints satisfaction problems* (CSP): introduction & modeling **(27.1.)**

Lecture 5 CSP algorithms: complete & local methods **(3.2.)**

Lecture 6 Boolean circuits: introduction & modeling **(10.2.)**

Lecture 7 Algorithms for circuit SAT: complete & local **(24.2.)**

Lecture 8 Modern SAT solvers **(3.3.)**

Part III: LP and MIP models and algorithms

Lecture 9 *Linear and integer linear programming*: introduction & modeling **(10.3.)**

Lecture 10 Algorithms for LP's and MIP's: branch-and-bound and the simplex method **(17.3.)**

Assignment 2 MIP or SAT modeling problem **(DL: 12.4.)**

Lecture 11 Beyond MIP's and LP's **(24.3.)**

Part IV: Advanced topics

Lecture 12 Advanced topics, feedback, question session **(31.3.)**

Background: Propositional logic

Outline

- Syntax: propositional formulas
- Semantics
- Logical equivalence
- Normal form transformations
- Example: 3-coloring of a graph

Propositional formulas

- Syntax based on:
  Boolean variables (atoms) $X = \{x_1, x_2, \ldots\}$
  Boolean connectives $\vee, \wedge, \neg$
- The set of (propositional) formulas is the smallest set such that all Boolean variables are formulas and if ϕ_1 and ϕ_2 are formulas, so are $\neg\phi_1$, $(\phi_1 \wedge \phi_2)$, and $(\phi_1 \vee \phi_2)$.
  For example, $((x_1 \vee x_2) \wedge \neg x_3)$ is a formula but $((x_1 \vee x_2)\neg x_3)$ is not.
- A formula of the form x_i or $\neg x_i$ is called a *literal* where x_i is a Boolean variable.
- We employ usual shorthands:
  $\phi_1 \rightarrow \phi_2$: $\neg\phi_1 \vee \phi_2$
  $\phi_1 \leftrightarrow \phi_2$: $(\neg\phi_1 \vee \phi_2) \wedge (\neg\phi_2 \vee \phi_1)$
  $\phi_1 \oplus \phi_2$: $(\neg\phi_1 \wedge \phi_2) \vee (\phi_1 \wedge \neg\phi_2)$

Semantics

- Atomic propositions (Boolean variables) are either true or false and this induces a truth value for any formula as follows.
- A truth assignment T is mapping from a finite subset $X' \subset X$ to the set of truth values $\{\textbf{true}, \textbf{false}\}$.
- Consider a truth assignment $T : X' \longrightarrow \{\textbf{true}, \textbf{false}\}$ which is appropriate to ϕ, i.e., $X(\phi) \subseteq X'$ where $X(\phi)$ be the set of Boolean *variables appearing in* ϕ.
- $T \models \phi$ (T *satisfies* ϕ) is defined inductively as follows:
  If ϕ is a variable, then $T \models \phi$ iff $T(\phi) = \textbf{true}$.
  If $\phi = \neg\phi_1$, then $T \models \phi$ iff $T \not\models \phi_1$
  If $\phi = \phi_1 \wedge \phi_2$, then $T \models \phi$ iff $T \models \phi_1$ and $T \models \phi_2$
  If $\phi = \phi_1 \vee \phi_2$, then $T \models \phi$ iff $T \models \phi_1$ or $T \models \phi_2$

Example Let $T(x_1) = \textbf{true}$, $T(x_2) = \textbf{false}$.
Then $T \models x_1 \vee x_2$ but $T \not\models (x_1 \vee \neg x_2) \wedge (\neg x_1 \wedge x_2)$

Representing Boolean Functions

- A propositional formula ϕ with variables $x_1, \ldots, x_n$ *expresses* a n-ary Boolean function f if for any n-tuple of truth values $\mathbf{t} = (t_1, \ldots, t_n)$, $f(\mathbf{t}) = \textbf{true}$ if $T \models \phi$ and $f(\mathbf{t}) = \textbf{false}$ if $T \not\models \phi$ where $T(x_i) = t_i$, $i = 1, \ldots, n$.

Proposition. Any n-ary Boolean function f can be expressed as a propositional formula ϕ_f involving variables $x_1, \ldots, x_n$.

- The idea: model each case of the function f having value **true** as a disjunction of conjunctions.
- Let F be the set of all n-tuples $\mathbf{t} = (t_1, \ldots, t_n)$ with $f(\mathbf{t}) = \textbf{true}$.
  For each $\mathbf{t}$, let $D_{\mathbf{t}}$ be a conjunction of literals x_i if $t_i = \textbf{true}$ and $\neg x_i$ if $t_i = \textbf{false}$.
- Let $\phi_f = \bigvee_{\mathbf{t} \in F} D_{\mathbf{t}}$

Note that ϕ_f is big in the worst case: $O(n2^n)$.

Example.

x_1	x_2	f
0	0	0
0	1	1
1	0	1
1	1	0

$\phi_f =$
$(\neg x_1 \wedge x_2) \vee$
$(x_1 \wedge \neg x_2)$

Logical Equivalence

Definition

Formulas ϕ_1 and ϕ_2 are *equivalent* ($\phi_1 \equiv \phi_2$) iff for all truth assignments T appropriate to both of them, $T \models \phi_1$ iff $T \models \phi_2$.

Properties:

- Commutativity
  $(\phi_1 \vee \phi_2) \equiv (\phi_2 \vee \phi_1)$
  $(\phi_1 \wedge \phi_2) \equiv (\phi_2 \wedge \phi_1)$
- Associativity
  $((\phi_1 \vee \phi_2) \vee \phi_3) \equiv (\phi_1 \vee (\phi_2 \vee \phi_3))$
  $((\phi_1 \wedge \phi_2) \wedge \phi_3) \equiv (\phi_1 \wedge (\phi_2 \wedge \phi_3))$
- Distributivity
  $((\phi_1 \vee \phi_2) \wedge \phi_3) \equiv ((\phi_1 \wedge \phi_3) \vee (\phi_2 \wedge \phi_3))$

Properties of logical Equivalence *continued*

- De Morgan's law
  $\neg(\phi_1 \vee \phi_2) \equiv (\neg\phi_1 \wedge \neg\phi_2)$
  $\neg(\phi_1 \wedge \phi_2) \equiv (\neg\phi_1 \vee \neg\phi_2)$
- Laws of absorption
  $(\phi_1 \vee \phi_1) \equiv \phi_1$
  $(\phi_1 \wedge \phi_1) \equiv \phi_1$
- Double negation
  $\neg\neg\phi \equiv \phi$
- Identity
  $(\phi_1 \vee \mathbf{false}) \equiv \phi_1$
  $(\phi_1 \wedge \mathbf{true}) \equiv \phi_1$
- Nullity
  $(\phi_1 \vee \mathbf{true}) \equiv \mathbf{true}$
  $(\phi_1 \wedge \mathbf{false}) \equiv \mathbf{false}$
- Complement
  $(\phi_1 \vee \neg\phi_1) \equiv \mathbf{true}$
  $(\phi_1 \wedge \neg\phi_1) \equiv \mathbf{false}$

Notational shorthand

$\bigvee_{i=1}^{n} \varphi_i$ stands for $\varphi_1 \vee \cdots \vee \varphi_n$ and $\bigwedge_{i=1}^{n} \varphi_i$ stands for $\varphi_1 \wedge \cdots \wedge \varphi_n$

Normal Forms

- Many solvers for Boolean constraints require that the constraints are represented in a normal form (typically in conjunctive normal form).

Proposition. Every propositional formula is equivalent to one in conjunctive (respectively, disjunctive) normal form.

CNF: $(l_{11} \vee \cdots \vee l_{1n_1}) \wedge \cdots \wedge (l_{m1} \vee \cdots \vee l_{mn_m})$
DNF: $(l_{11} \wedge \cdots \wedge l_{1n_1}) \vee \cdots \vee (l_{m1} \wedge \cdots \wedge l_{mn_m})$
where each l_{ij} is a literal (Boolean variable or its negation).

- A disjunction $l_1 \vee \cdots \vee l_n$ is called a *clause*.
- A conjunction $l_1 \wedge \cdots \wedge l_n$ is called an *implicant*.

Normal Form Transformations

CNF/DNF transformation:

1. remove $\leftrightarrow$ and $\rightarrow$:

$$\alpha \rightarrow \beta \quad \leadsto \quad \neg\alpha \vee \beta \qquad (1)$$
$$\alpha \leftrightarrow \beta \quad \leadsto \quad (\neg\alpha \vee \beta) \wedge (\neg\beta \vee \alpha) \qquad (2)$$

2. Push negations in front of Boolean variables:

$$\neg\neg\alpha \quad \leadsto \quad \alpha \qquad (3)$$
$$\neg(\alpha \vee \beta) \quad \leadsto \quad \neg\alpha \wedge \neg\beta \qquad (4)$$
$$\neg(\alpha \wedge \beta) \quad \leadsto \quad \neg\alpha \vee \neg\beta \qquad (5)$$

3. CNF: move $\wedge$ connectives outside $\vee$ connectives:

$$\alpha \vee (\beta \wedge \gamma) \quad \leadsto \quad (\alpha \vee \beta) \wedge (\alpha \vee \gamma) \qquad (6)$$
$$(\alpha \wedge \beta) \vee \gamma \quad \leadsto \quad (\alpha \vee \gamma) \wedge (\beta \vee \gamma) \qquad (7)$$

DNF: move $\vee$ connectives outside $\wedge$ connectives:

$$\alpha \wedge (\beta \vee \gamma) \quad \leadsto \quad (\alpha \wedge \beta) \vee (\alpha \wedge \gamma) \qquad (8)$$
$$(\alpha \vee \beta) \wedge \gamma \quad \leadsto \quad (\alpha \wedge \gamma) \vee (\beta \wedge \gamma) \qquad (9)$$

Example

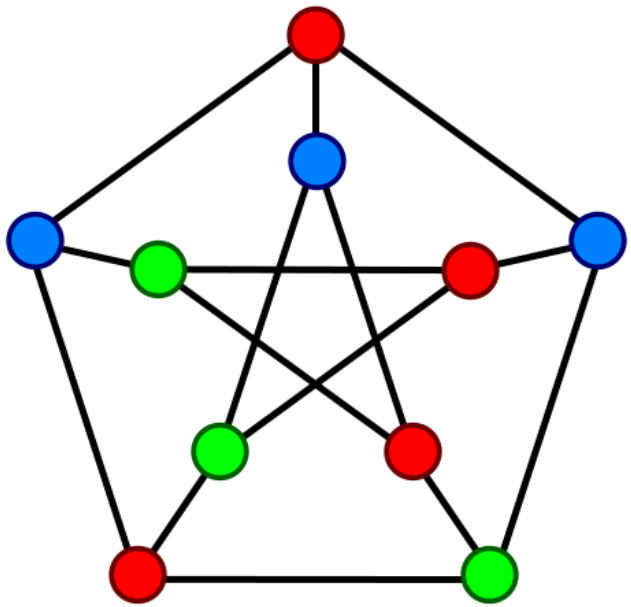
Transform $(A \vee B) \rightarrow (B \leftrightarrow C)$ to CNF.

$(A \vee B) \rightarrow (B \leftrightarrow C)$ (1,2)
$\neg(A \vee B) \vee ((\neg B \vee C) \wedge (\neg C \vee B))$ (4)
$(\neg A \wedge \neg B) \vee ((\neg B \vee C) \wedge (\neg C \vee B))$ (7)
$(\neg A \vee ((\neg B \vee C) \wedge (\neg C \vee B))) \wedge (\neg B \vee ((\neg B \vee C) \wedge (\neg C \vee B)))$ (6)
$((\neg A \vee (\neg B \vee C)) \wedge (\neg A \vee (\neg C \vee B))) \wedge (\neg B \vee ((\neg B \vee C) \wedge (\neg C \vee B)))$ (6)
$((\neg A \vee (\neg B \vee C)) \wedge (\neg A \vee (\neg C \vee B))) \wedge ((\neg B \vee (\neg B \vee C)) \wedge (\neg B \vee (\neg C \vee B)))$
$(\neg A \vee \neg B \vee C) \wedge (\neg A \vee \neg C \vee B) \wedge (\neg B \vee \neg B \vee C) \wedge (\neg B \vee \neg C \vee B)$

- We can assume that normal forms do not have repeated clauses/implicants or repeated literals in clauses/implicants (for example $(\neg B \vee \neg B \vee C) \equiv (\neg B \vee C)$).
- In the *worst case* CNF/DNF form can be *exponentially larger* than the original formula.

Example: Graph coloring

- Consider the problem of finding a 3-coloring for a graph.
- Note: the graph coloring problem asks for an assignment of colors to vertices such that no pair of adjacent vertices (vertices that share an edge) have the same color.
- In the special case of 3-coloring one restricts the number of colors to 3 (see example below).



Example: Graph coloring cont.

- 3-coloring can be encoded as a set of Boolean constraints as follows:
  - For each vertex $v \in V$, introduce three Boolean variables v_1, v_2, v_3 (intuition: v_i is true iff vertex v is colored with color i).
  - For each vertex $v \in V$ introduce the constraints

$$v_1 \vee v_2 \vee v_3$$
$$(v_1 \rightarrow \neg v_2) \wedge (v_1 \rightarrow \neg v_3) \wedge (v_2 \rightarrow \neg v_3)$$

  - For each edge $(v, u) \in E$ introduce the constraint

$$(v_1 \rightarrow \neg u_1) \wedge (v_2 \rightarrow \neg u_2) \wedge (v_3 \rightarrow \neg u_3)$$

- Now 3-colorings of a graph (V, E) and solutions to the Boolean constraints (satisfying truth assignments) correspond: vertex v colored with color i iff v_i assigned true in the solution.

Lecture 2: Combinatorial search problems

Outline

- Computational problems and their properties
  - decision problem
  - search problem
  - optimization problem
  - counting problem
- Examples of computational problems
- Reductions between problems from an algorithmic point-of-view

Goal for today: Learn to recognize and formulate different types of computational problems; learn how to use reductions between problems as an algorithmic technique for their solution.

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 (Search Problem)
  INSTANCE: A graph (V, E) and nodes $v, u \in V$.
  QUESTION: Find a path from v to u.

- SHORTEST PATH (Optimization Problem)
  INSTANCE: A graph (V, E) and nodes $v, u \in V$.
  QUESTION: Find a shortest path from v to u.

- #PATH (Counting Problem)
  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.*

Examples of hard problems

SAT (Boolean Satisfiability Problem)
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 satisfying the most clauses in the formula.

Propositional formulas consist of literals (variables and their negations), conjunctions ("and" $\wedge$) and disjunctions ("or" $\vee$).

Conjunctive normal form: conjunction of disjunctions (=clauses)

$$x_1 \wedge (x_2 \vee \neg x_2)$$

Example Consider an instance of the SAT problem

$$F(x_1, x_2) := (x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_2) \wedge (x_1 \vee x_2)$$

x_1	x_2	$F(x_1, x_2)$
false	**false**	**false**
false	**true**	**false**
true	**false**	**false**
true	**true**	**true**

This is satisfiable as the formula is satisfied by a truth assignment T_1 where $T_1(x_1) = \textbf{true}$, $T_1(x_2) = \textbf{true}$.

If we add a new conjunct $(\neg x_1 \vee \neg x_2)$, the instance turns unsatisfiable.

For the SAT(O) problem consider the instance

$$F'(x_1, x_2) := (x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_2) \wedge (x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2) \wedge \neg x_2.$$

The assignment T_1 is not optimal but $T_2(x_1) = \textbf{false}$, $T_2(x_2) = \textbf{false}$ is (satisfying 4 clauses).

Examples of hard problems (II)

- 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.

- 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.

Examples of hard problems (III)

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.

Example

Consider the family of sets: $F = \{S_1, S_2, S_3\}$ where
$S_1 = \{1,2\}, S_2 = \{2,3\}, S_3 = \{3,4\}$ and $U = \{1, \ldots, 4\}$

Now $\{S_1, S_2, S_3\}$ is a 3-cover of U, $\{S_1, S_3\}$ is a 2-cover of U but there are no 1-covers.

Examples of hard problems (IV)

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.

Relationship between problems

- Let us consider decision problems A and B.
- B *reduces* to A ($B \sqsubseteq A$) if there is a transformation R for which every input instance x of B produces an *equivalent* input instance $R(x)$ of A.
  - Here 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* R 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.

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

  Clauses for vertex $v \in V$:

  $v_b \vee v_r \vee v_g$
  $\neg v_b \vee \neg v_r$
  $\neg v_b \vee \neg v_g$
  $\neg v_r \vee \neg v_g$

  Clauses for edge $(v, u) \in E$:

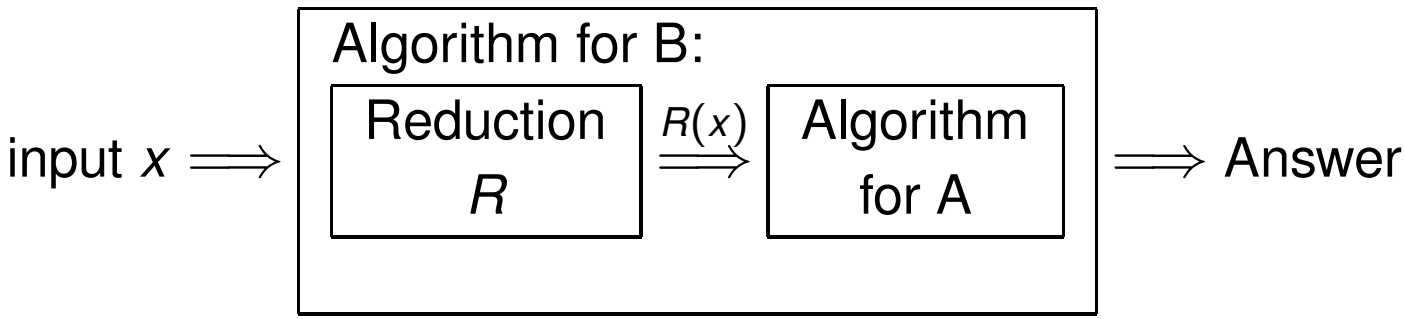
  $\neg v_b \vee \neg u_b$
  $\neg v_r \vee \neg u_r$
  $\neg v_g \vee \neg u_g$

- 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.

Reduction

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

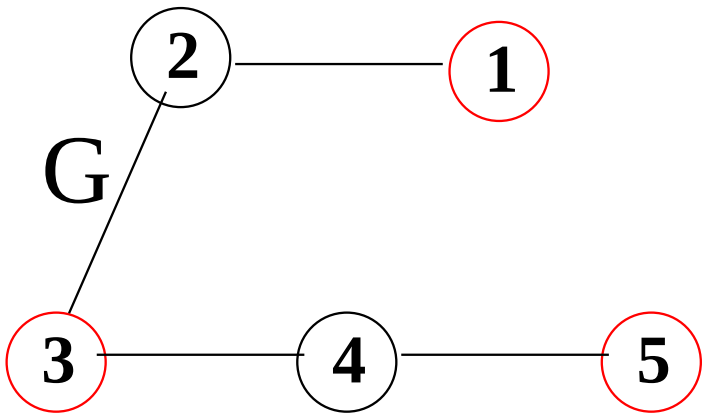
- An algorithm for B can be built on top of an algorithm for A.
  - Used extensively in this course.

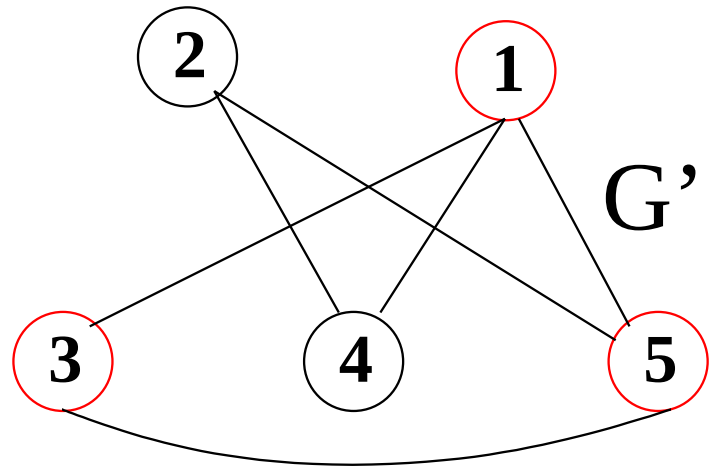


- Reduction implies that A is computationally *at least as hard as* B.
  - Used in *computational complexity theory* (T-79.5103) to classify computational problems; $B \sqsubseteq A$ orders problems by difficulty.

Example: INDEPENDENT SET $\sqsubseteq$ CLIQUE

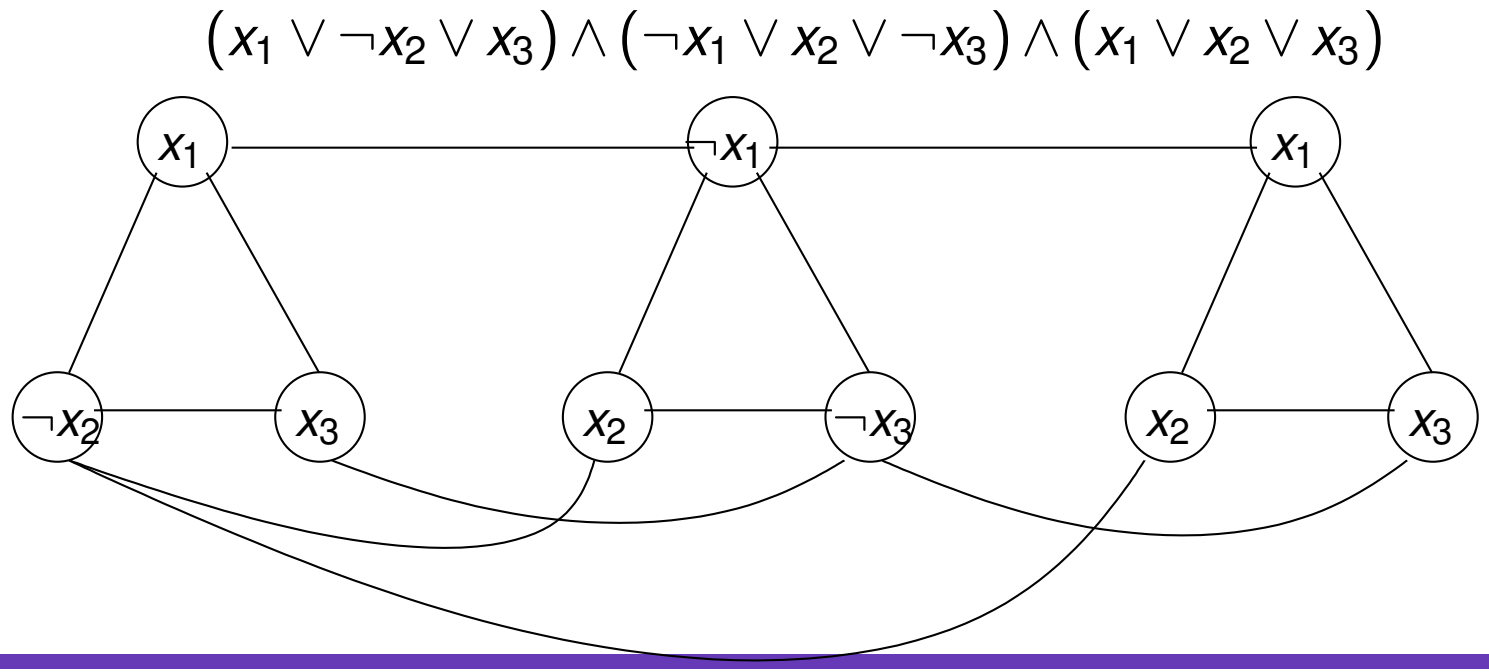
- INDEPENDENT SET
  INSTANCE: A graph $G = (V, E)$ and an integer K.
  QUESTION: Is there an independent set $I \subseteq V$ with $|I| = K$.
  (A set $I \subseteq V$ is independent if $i, j \in I$ implies that there is no edge between i and j).
- Reduction from INDEPENDENT SET to CLIQUE: Given a $G = (V, E)$ and an integer K, take the complement graph $G' = (V, \{(v, u) \mid v, u \in V, (v, u) \notin E\}$. (Note: an independent set of a graph is a clique of the complement graph.)





Example: 3-SAT $\sqsubseteq$ INDEPENDENT SET

- 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. Add the following edges: a) a separate triangle for each clause b) an edge between two vertices in different triangles corresponding 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.

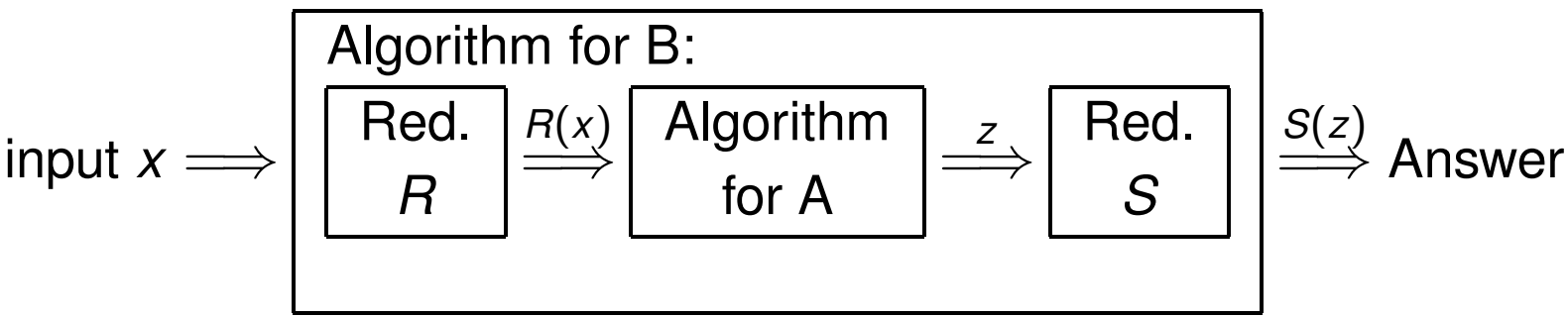
Reductions—cont'd

- 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 for search problems

- 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 often a search problem reduces to the corresponding decision problem.

SET COVER(D) vs SET COVER(S)

- Clearly, if SET COVER(S) can solved in polynomial time, then so can SET COVER(D).
- Next we show that if SET COVER(D) can be solved in polynomial time, then so can SET COVER(S).
- Assume that SET COVER(D) can be solved in polynomial time, i.e., there is a procedure setcover(F, U, k) such that given a family $F = \{S_1, \ldots, S_n\}$ of subsets of U and a positive integer k, it decides in polynomial time whether F has a k-cover or not.
- Now using the procedure setcover(F, U, k) a k-cover of F can be found by calling the procedure at most once for each member S_i of the family of sets $F = \{S_1, \ldots, S_n\}$.
- Hence, the run time remains polynomial.

A Procedure for Solving SET COVER(S)

if setcover(F, U, k) returns “no” then return “no”;
$l := k - 1$; $C := \emptyset$;
for all $S \in \{S_1, \ldots, S_n\}$ do
  if setcover(F/S, $U - S$, l) returns “yes” then
    $C := C \cup \{S\}$; $F := F/S$; $U := U - S$;
    $l := l - 1$;
  else
    $F := F - \{S\}$;
  endfi
return C;

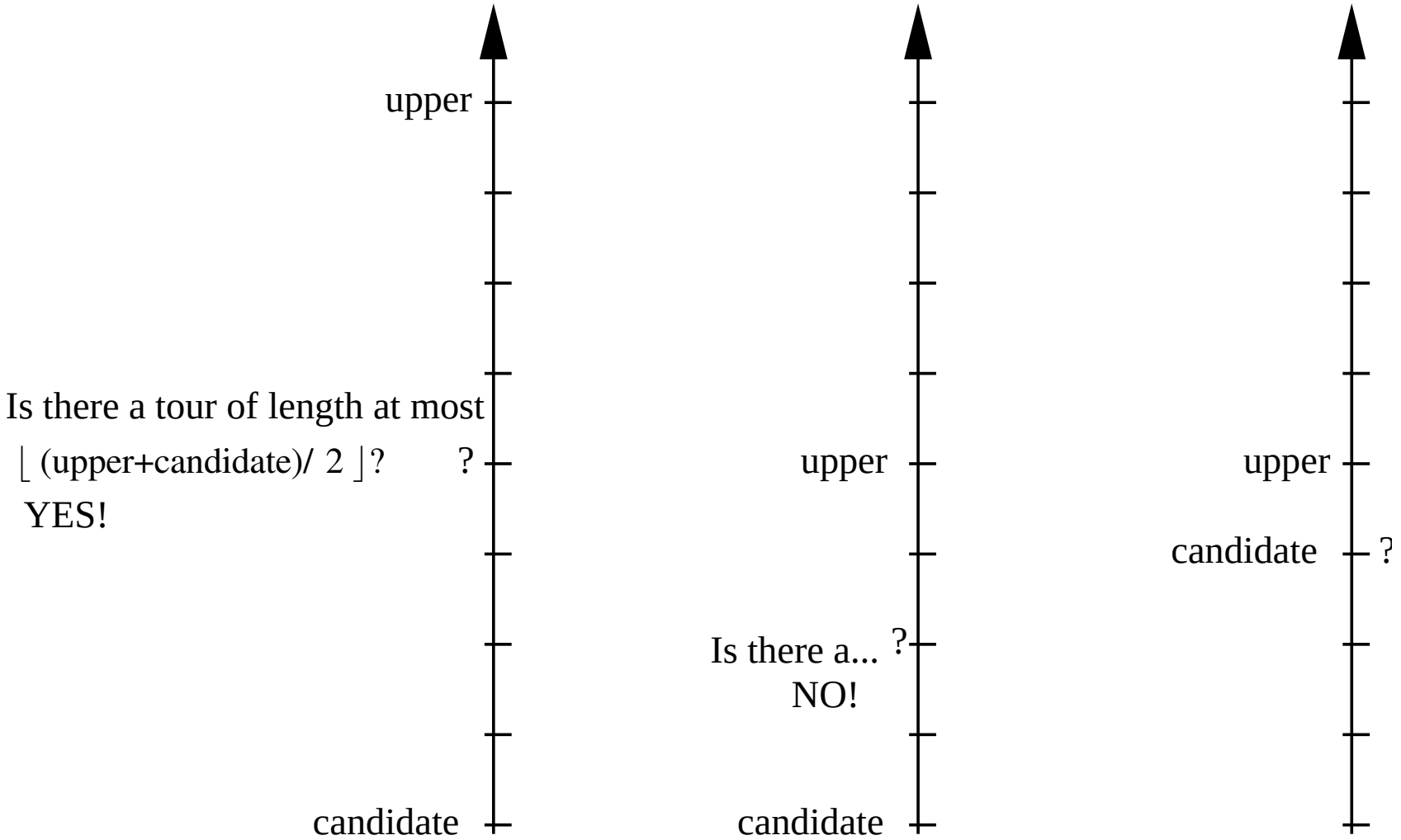
where C is the computed k-cover.
Note: F/S denotes F with the set S removed and all elements of S deleted from other sets in the family F; $U - S$ denotes the set with elements of U but elements also in S removed (“setminus”); similarly, $F - \{S\}$ is the remaining family of sets with S removed.

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 (=length) 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)).

How to find C by binary search



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**/
$C := 0$; $C_u := D$; /* D is the sum of maximal distances from each city */
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
    $C := \lfloor (C_u + C)/2 \rfloor + 1$;
  endfi
/* Find an optimal tour given the cost C of 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 the decision, search, and optimization variants of problems are polynomially equivalent.
- 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 object (witness/certificate), e.g., satisfying truth assignment, 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.
- For optimization problems it is hard even to verify a solution.
  - Consider the traveling salesperson problem and a potential solution π.
  - There seems to be no obvious polynomial time test that could establish that π is actually a tour that has minimum length.
- Counting problems are often even harder.

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.

Lecture 3: Intro to complete and local search methods

Outline

- Complete search
  - Search spaces and objective functions
  - Methods: backtrack, branch-and-bound
- Local search
  - Search spaces and neighborhood structures
  - Methods: hill climbing, simulated annealing, tabu search, etc.

Goal for today: For a given high-level description of a computational problem (search or optimization), learn to

1. formulate a suitable search space for the problem and
2. devise
   a) a complete search/optimization method based on the high-level algorithms introduced today
   b) a local search algorithm for solving the problem based on the high-level methods introduced today

Search spaces and objective functions

- A combinatorial search or optimization problem Π determines a *search space* X of candidate solutions for each of its instances I.
- The computational difficulty in such problems arises from the fact that X is typically exponential in the size of I (= HUGE).
- In general, complete search methods have to be able to find (generate) any solution in X.
- E.g. SAT(S):

Instance:	F = propositional formula on n variables $\{x_1, \ldots, x_n\}$.
Search space:	X = all truth assignments $t : \{x_1, \ldots, x_n\} \to \{0, 1\}$.
Goal:	find $t \in X$ that makes F true.

Size of $X = 2^n$ points (0/1-vectors).

Search spaces and objective functions—cont'd

Recall that since SAT formulas are required to be in conjunctive normal form, it can also be viewed as an optimization problem:

SAT(O):

Instance:	F = family of m clauses on n variables $\{x_1, \ldots, x_n\}$.
Search space:	X = all truth assignments $t : \{x_1, \ldots, x_n\} \to \{0, 1\}$.
Objective function:	$c(t)$ = # clauses not satisfied by t.
Goal:	minimize $c(t)$.

Size of $X = 2^n$ points (0/1-vectors).

Search spaces and objective functions—cont'd

TSP(O):

Instance:	An $n \times n$ matrix D of distances d_{ij} between n "cities".
Search space:	X = all permutations ("tours") π of $\{1, \ldots, n\}$.
Objective function:	$d(\pi) = \sum_{i=1}^{n-1} d_{\pi(i)\pi(i+1)} + d_{\pi(n)\pi(1)}$.
Goal:	minimize $d(\pi)$.

Note: Here $|X| = n!$. (More precisely: $|X| = (n-1)!/2$, if the starting points and orientations of tours are ignored.)

Search spaces and objective functions—cont'd

MAX CUT(O):

Instance:	A graph $G = (V, E)$ and a function c giving each edge $(u, v) \in E$ an integer capacity $c(u, v)$.
Search space:	X = all cuts in G, which are partitions of V into S and $V - S$, where $S \subset V$ and $S \neq \emptyset$.
Objective function:	$c(S) = \sum_{(u,v) \in E, u \in S, v \notin S} c(u, v)$.
Goal:	maximize $c(S)$.

Note: Here $|X| = 2^n - 2$. (More precisely: $|X| = 2^{n-1} - 1$, since there is no "direction" of a cut.)

Backtrack search

- *Backtrack search* is a systematic method to search for a satisfying, or an optimal solution x in a search space X (the pseudo-code below terminates when the first solution is found).
- Note: from here on onwards, x may be also a *partial solution*; but: $x \in X$ iff x is a complete solution!

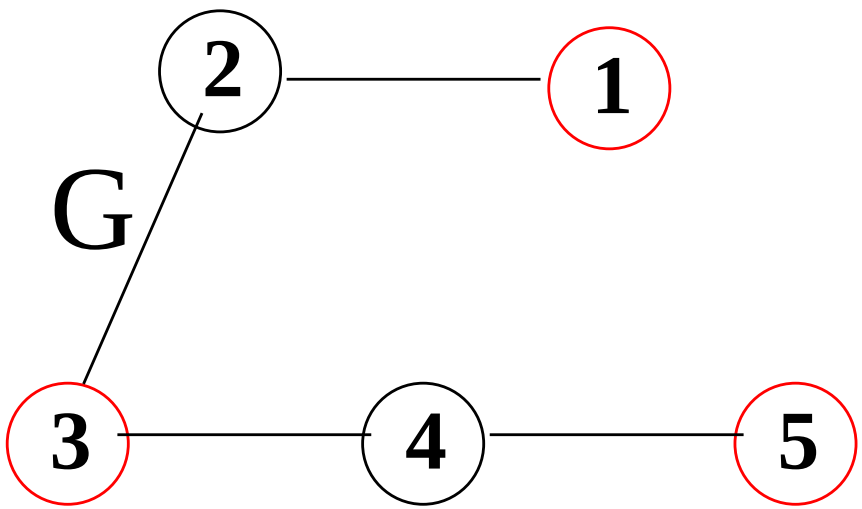
function backtrack(I:instance; x:partialsol):
    if $x \in X$ (x is a complete solution) and feasible **then**
        return x;
    else
        for all extensions $e_1, \ldots e_k$ to x **do**
            $x' \leftarrow$ backtrack($I, x \oplus e_i$);
            if $x' \in X$ and feasible **then return** x'
        end for;
        return fail
    end if.

Backtrack search: VERTEX COVER

- Recall the (optimization variant) of the vertex cover problem.

VERTEX COVER(O):

Instance: A graph $G = (V, E)$.
Goal: minimize $c(x) = |x|$ over all subsets $x \subseteq V$ s.t. for all $(u, v) \in E$, $u \in x$ or $v \in x$ (or both).



Backtrack search: VERTEX COVER—cont'd

- Let x^* be the currently best known solution.

initially: $x^* \leftarrow V$; $x \leftarrow \emptyset$;
function simpleBacktrackVC($G = (V, E)$:instance; $x \subseteq V$:partialsol):
    if x is a vertex cover **then**
        if $|x| < |x^*|$ (x is better than current best) **then**
            $x^* \leftarrow x$;
    else
        for all $v \in V \setminus x$ **do**
            simpleBacktrackVC((V, E), $x \cup \{v\}$);
        end for;
    end if.

- *Note:* This is inefficient (repeats solutions!).

Branch-and-bound search (1/2)

- Pruning techniques can greatly improve the efficiency of backtrack search in optimization problems.
- Example modification to the VC algorithm: only recurse by calling simpleBacktrackVC((V, E), $x \cup \{v\}$), if $|x| < |x^*| - 1$.
- General idea: Assume for partial solution x we have a lower bound $l(x)$ on the cost of any complete solution that can be constructed from x.
- If we know of a complete solution with cost c (which is an upper bound on minimum cost), we can prune the search tree at x if $l(x) \geq c$. (VC example: $c = |x^*|), l(x) = |x| + 1$)
- Note: larger lower bounds are better than smaller ones! (more effective pruning)

Branch-and-bound search (2/2)

initially: $c \leftarrow \infty$; $x^* \leftarrow ()$;
function branch_and_bound(I:instance; x:partialsol):
    if x is a complete (and feasible) solution **then**
        if $\mathrm{cost}(x) < c$ **then**
            $c \leftarrow \mathrm{cost}(x)$; $x^* \leftarrow x$;
        end if;
    else
        for all extensions $e_1, \ldots e_k$ to x **do**
            $x' \leftarrow x \oplus e_i$;
            if $I(x') < c$ **then**
                branch_and_bound(I, x');
            else // *prune, do nothing*
            end if;
        end for;
    end if.

Branch-and-bound search: TSP

Consider e.g. the TSP(O) problem and choose:

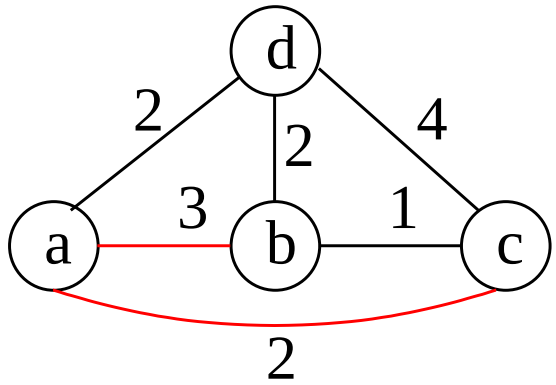
Partial solution: A set of edges (links) chosen either to be included or excluded from the complete solution tour (here: set of all candidate edges = $N \times N$, $N = \{1, \ldots, n\}$).

Bounding heuristic: Let the TSP instance under consideration be given by distance matrix $D = d_{ij}$ (where $d_{ij} = d_{ji}$). Then the following inequality holds for any complete tour π:

$$\begin{aligned} d(\pi) &= \frac{1}{2} \sum_j \{(d_{ij} + d_{jk}) \mid \text{at city } j \text{ tour } \pi \text{ uses links } ij \text{ and } jk\} \\ &\geq \frac{1}{2} \sum_j \min_{i,k:i \neq k} (d_{ij} + d_{jk}). \end{aligned}$$

Intuition: even the (globally) optimal tour needs to enter and leave each city and thus cannot achieve a shorter length than the locally best way to do so.

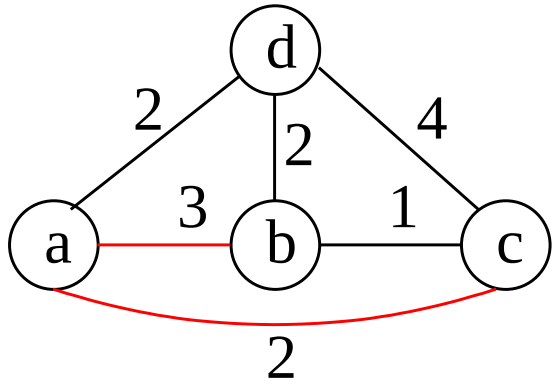
TSP bounding heuristic (1/2)



Say, the edges a-b and a-c have been chosen as part of the solution. The bounding heuristic then results in the following:

$$\begin{aligned}
&\tfrac{1}{2}\left(d_{ab} + d_{ac} + \min_{i,k:i\neq k}(d_{ib} + d_{bk}) + \min_{i,k:i\neq k}(d_{ic} + d_{ck})\right. \\
&\qquad \left. + \min_{i,k:i\neq k}(d_{id} + d_{dk})\right) \\
&= \tfrac{1}{2}(5 + (1+2) + (1+2) + (2+2)) \\
&= \tfrac{15}{2}
\end{aligned}$$

TSP bounding heuristic (2/2)



However, we can do better: since we need to be able to complete the partial solution to a complete tour, we can exclude certain edges from the bound and force the inclusion of others.

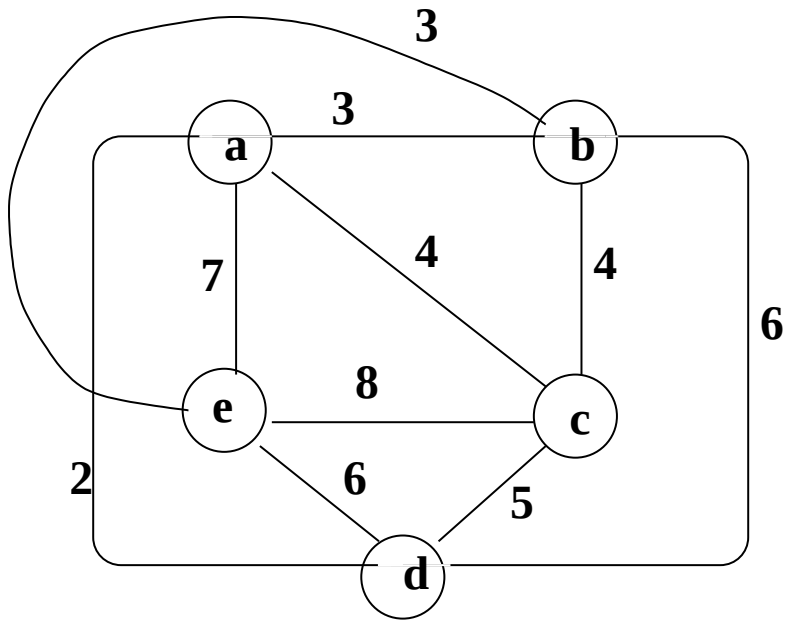
Example: Including a-b and a-c leads to excluding c-b (incomplete tour).

$$\frac{1}{2}\left(d_{ab} + d_{ac} + (d_{ab} + d_{bd}) + (d_{bd} + d_{dc}) + (d_{dc} + d_{ca})\right)$$
$$= 11$$

Special case of constraint propagation, which we discuss later in detail.

Example for TSP

Consider the following small TSP instance:



TSP example cont'd

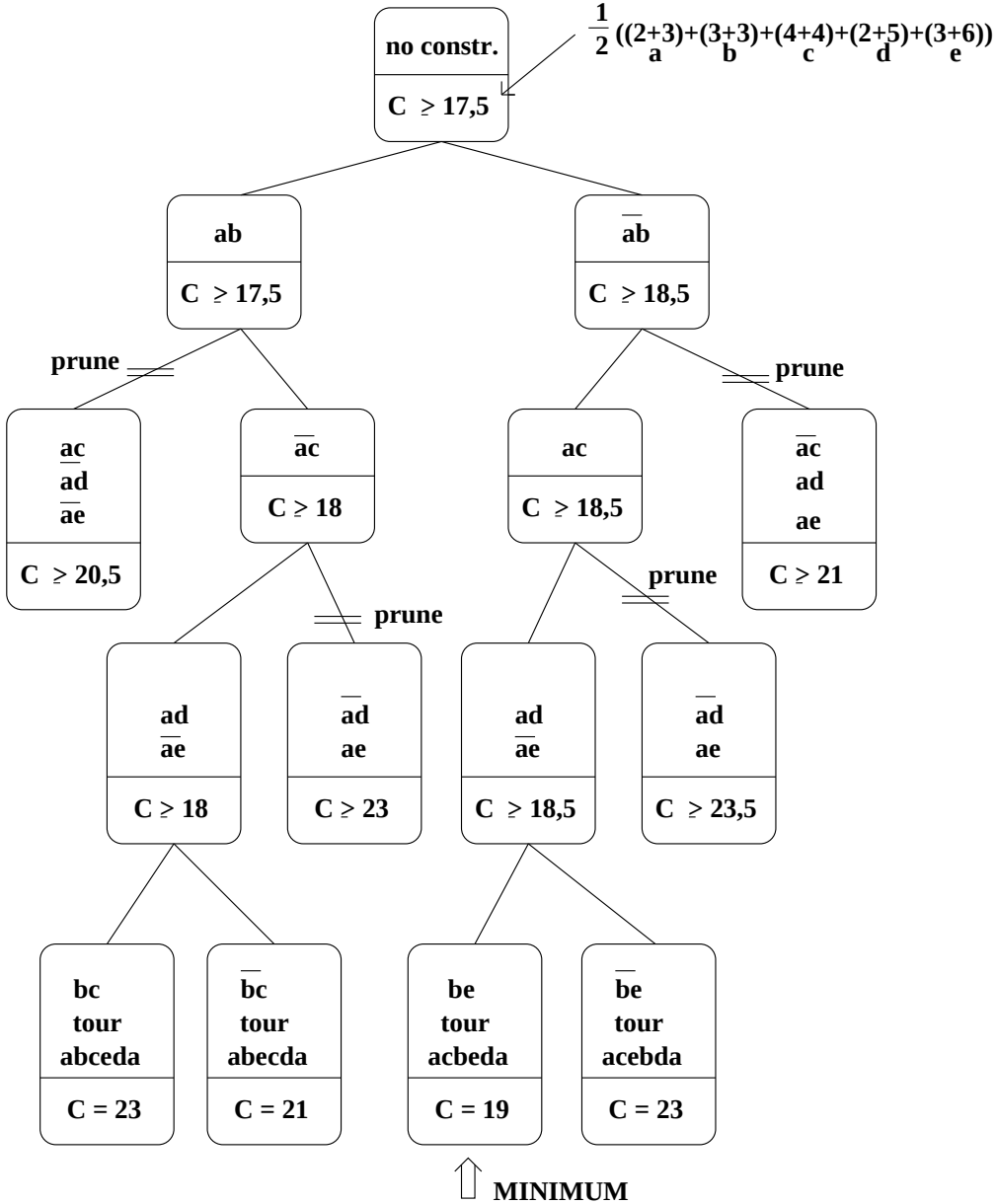
We introduce a modified version of the branch-and-bound algorithm, using the following procedures:

- find_initial_tour(D): Always choose the link with the smallest cost from the current node to an unvisited node. Once all the nodes have been visited, a tour is found.
  For the example, when starting from node a this simple heuristics returns a complete tour "adcbea" with cost
  $d_{ad} + d_{dc} + d_{cb} + d_{be} + d_{ea} = 2 + 5 + 4 + 3 + 7 = 21$.
- propagate(π): exclude/include edges to the (partial) tour π in case their exclusion/inclusion is necessary in order to complete tour (as described on slide 69)

Using the lower-bounding heuristic and the modified branch-and-bound algorithm, the search tree for the minimum tour on this instance can be pruned as presented on slide 73.

Modified Branch-and-bound search for TSP example

initially: $\pi^* \leftarrow$ find_initial_tour(D); $c \leftarrow cost(\pi^*)$;
function branch_and_bound_TSP(D: distance matrix; π: partial tour):
    if π is a complete tour **then**
        if $\mathrm{cost}(\pi) < c$ **then**
            $c \leftarrow \mathrm{cost}(\pi)$; $\pi^* \leftarrow \pi$;
        end if;
    else
        for all extensions $e_1, \ldots e_k$ to π **do**
            $\pi' \leftarrow \pi \oplus e_i$; $\pi' \leftarrow propagate(\pi')$;
            if $l(\pi') < c$ **then**
                branch_and_bound(D, π');
            else // *prune, do nothing*
            end if;
        end for;
    end if.


no constr.
C ≥ 17,5
1/2 ((2+3)+(3+3)+(4+4)+(2+5)+(3+6))
a b c d e
ab
C ≥ 17,5
ab
C ≥ 18,5
prune
prune
ac ad ae
C ≥ 20,5
ac
C ≥ 18
ac
C ≥ 18,5
ac ad ae
C ≥ 21
prune
prune
ad ae
C ≥ 18
ad ae
C ≥ 23
ad ae
C ≥ 18,5
ad ae
C ≥ 23,5
bc tour abceda
C = 23
bc tour abecda
C = 21
be tour acbeda
C = 19
be tour acebda
C = 23
MINIMUM

Final remarks on Branch-and-bound search

Branch-and-bound search and SAT:

- Most naturally, the partial solutions are chosen to correspond to partial truth assignments $t : \{x_1, \ldots, x_i\} \to \{0, 1\}$.
- Each partial assignment has two possible extension e_0 and e_1: e_0 assigns value 0 to variable x_{i+1} and e_1 assigns value 1.
- DPLL (Davis-Putnam-Logemann-Loveland) procedure, a backtrack search method for testing satisfiability of a set of clauses will be introduced in detail later during the course.

Maximization problems:

- For maximization problems "orientation" of bounds is reversed: one prunes if $u(x) \leq p$, where $u(x)$ is an upper bound on the objective value of any complete solution that can be constructed from x and p is the objective value of the best complete solution known so far.

Local search techniques

- For realistic problems, complete search trees can be extremely large and difficult to prune effectively.
- Often, it is more important to get a reasonably good solution fast, rather than the globally optimal one after a long wait.
- Therefore, *local search* methods provide an interesting alternative.

Assume that the search space X has some *neighborhood structure* N, whereby for each solution $x \in X$, a set of "structurally close" solutions $N(x) \subseteq X$ can be easily generated from x by local transformations.

Note: here x are complete solutions (cmp. partial solutions for complete search methods) although possibly infeasible.

Examples

For instance, in the case of SAT(O) one could have:

$N(x) =$ {truth assignments x' that differ from x at exactly one variable},

... in the case of MAX CUT(O):

$N(x) =$ {cuts x' that result from x by moving one vertex to the other side of the cut},

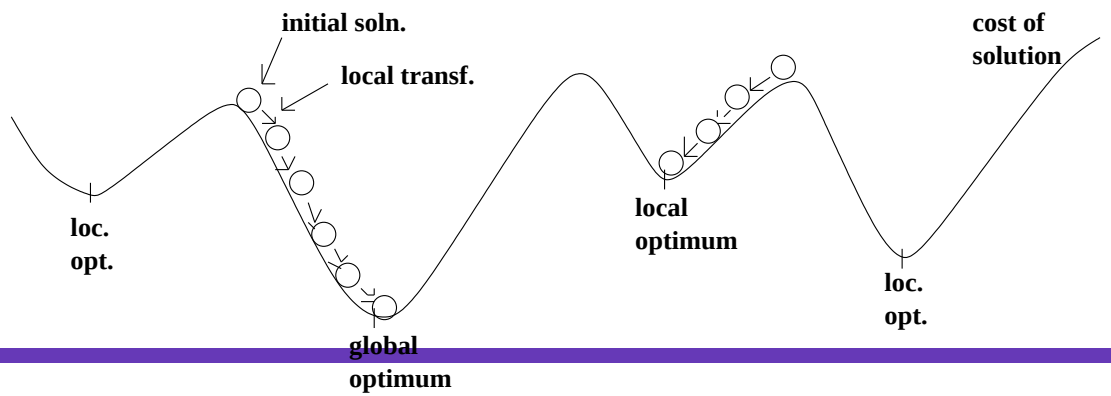
and for the case of GRAPH COLORING(O):

$N(x) =$ {colorings x that result from x by changing the color of a single node}.

Hill climbing

The *hill climbing* (simple local search, iterative improvement) method works by iteratively improving a given solution by neighborhood transformations, as long as possible:

function simple_LS (X, N, c):
    choose arbitrary initial solution $x \in X$;
    repeat
        find some $x' \in N(x)$ such that $c(x') < c(x)$;
        $x \leftarrow x'$;
    until no such x' can be found;
    return x.



Hill climbing—cont'd

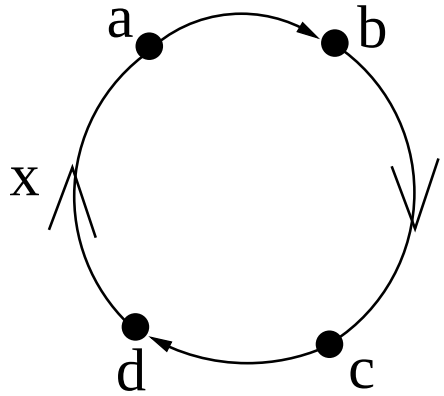
- Simple hill climbing just picks any better solution in the neighborhood.
- *Steepest descent* (or ascent for maximization problems) picks the best solution (the one that achieves the largest improvement) among all neighbors.
- Very simple technique, can be combined with *random restarts*.
- Obvious problem: search gets trapped in local optima, although random restarts may improve the best solution found.

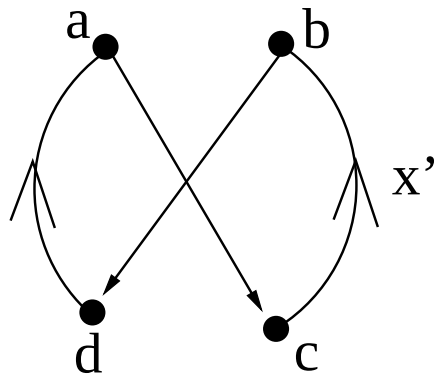
Example: Hill climbing for TSP

- Consider a hill climbing method for TSP: candidate solutions x are a sequence of edges forming a cycle in the TSP instance.
- Principle of *Lin-Kernighan k-opt neighborhoods*: solutions x and x' are neighbors, if x can be transformed into x' by replacing k edges that are in the tour x with k other edges not in x.
- More generally, the LK algorithm considers different values for k during a run ($k = 2$, $k = 3$, etc.) to improve the current solution.
- The resulting method has been experimentally shown to produce quite good results for the TSP, sometimes only a few % longer than optimum.
- For implementations and more information, e.g., see `http://www.akira.ruc.dk/~keld/research/LKH/`

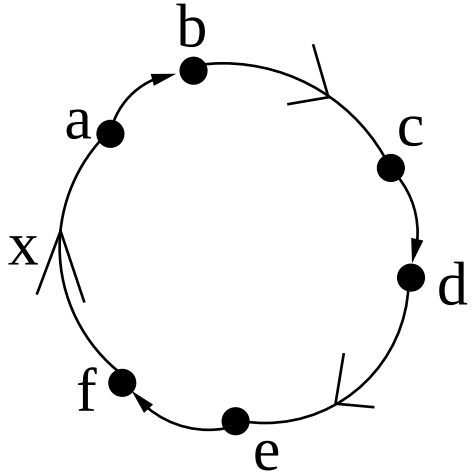
2-opt and 3-opt moves

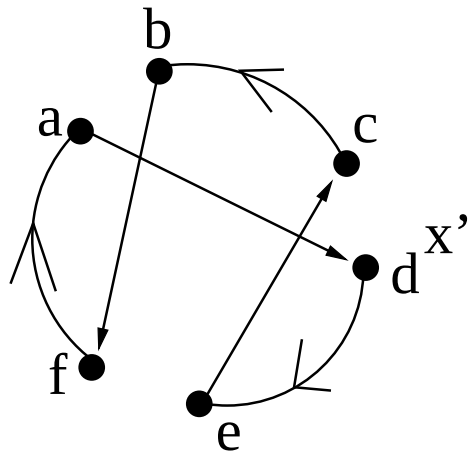
2-opt move: Replace $\{(a,b),(c,d)\}$ by $\{(a,c),(b,d)\}$



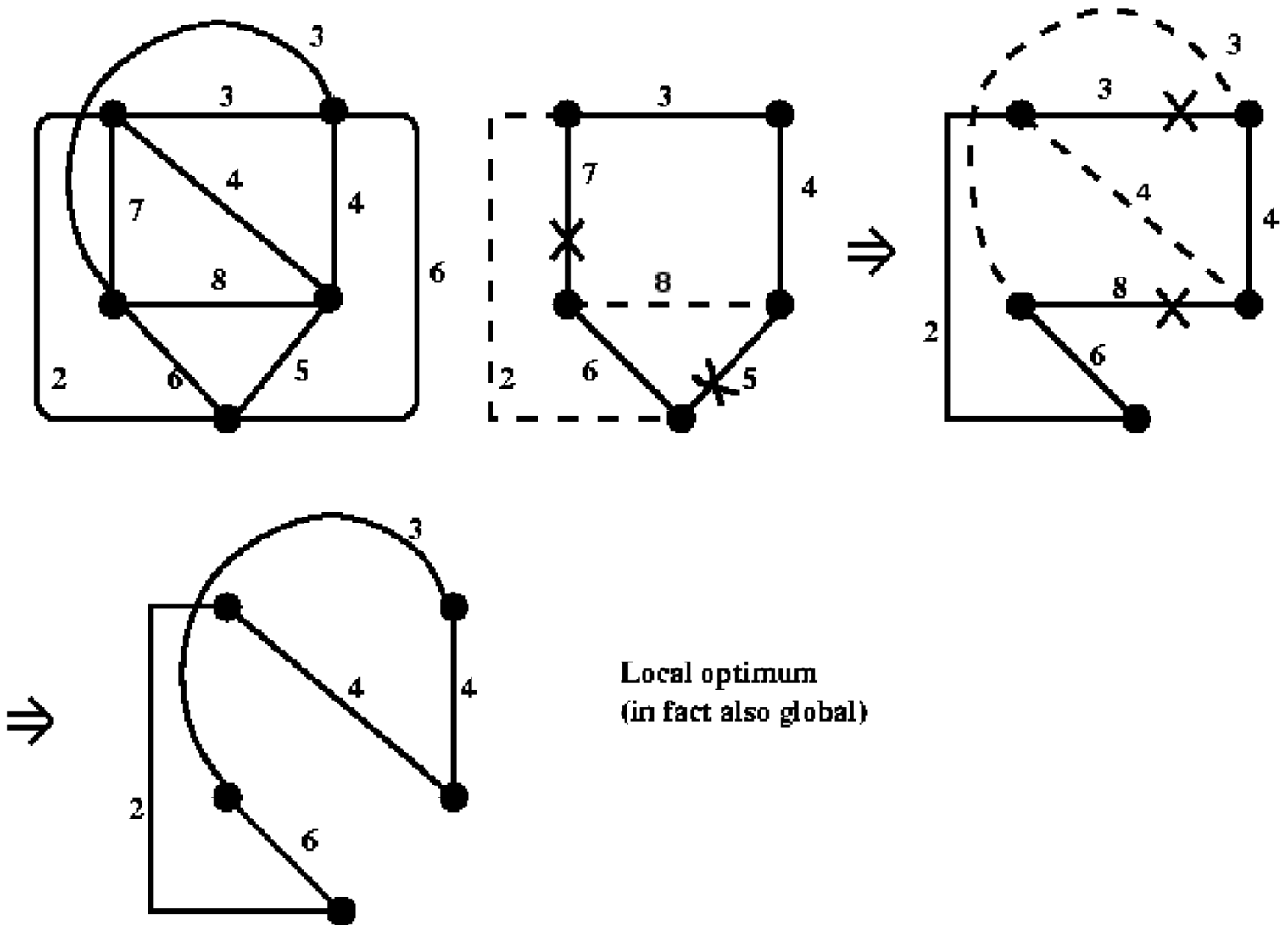


3-opt move: Replace $\{(a,b),(c,d),(e,f)\}$ by $\{(a,d),(e,c),(b,f)\}$





A 2-Opt descent to local optimum for TSP



Simulated annealing: basic idea

- Local (nonglobal) minima are a problem for deterministic local search, and many heuristics have been developed for escaping from them.
- One of the most widely used is *simulated annealing* (Kirkpatrick, Gelatt & Vecchi 1983, Černy 1985).
- Generate random neighbor of current solution (not necessarily with uniform probability).
- Always move to new solution if move improves objective value
- Sometimes move to worse solutions to escape local optima.
- Probability of accepting a worse solution varies over runtime (monotonically decreasing over iterations).
- Analogy in metallurgy: heating and controlled cooling of material to reduce defects in its crystal structure.

Simulated annealing: implementation

- Assume again a minimization problem.
- Amount of stochasticity is regulated by a *computational temperature* parameter T.
- Value of T during the search is decreased from some large initial value $T_{init} \gg 0$ to some final value $T_{final} \approx 0$.
- Search is allowed to proceed for several iterations with the same temperature (called *sweep*).
- Proposed move from a solution x to a worse solution x' is accepted with probability $e^{-\Delta c/T}$, where $\Delta c > 0$ is the cost difference of the solutions.

function SA(X, N, c):

$T \leftarrow T_{init}$;

$x \leftarrow x_{init}$; $x^* \leftarrow x$;

while $T > T_{final}$ **do**

$L \leftarrow$ sweep(T);

for L times **do**

choose $x' \in N(x)$ uniformly at random;

$\Delta c \leftarrow c(x') - c(x)$;

if $\Delta c \leq 0$ **then** $x \leftarrow x'$ **else**

choose $r \in [0, 1)$ uniformly at random;

if $r \leq \exp(-\Delta c / T)$ **then** $x \leftarrow x'$;

if $c(x) < c(x^*)$ **then** $x^* \leftarrow x$;

end for;

$T \leftarrow$ lower(T);

end while;

return x^*.

Cooling schedules

- An important question in applying SA is how to choose appropriate functions lower(T) and sweep(T), i.e. what is a good "cooling schedule" $\langle T_0, L_0 \rangle, \langle T_1, L_1 \rangle, \ldots$
- Theoretical results (Markov chain theory) guarantee that if the cooling is "sufficiently slow", then the algorithm almost surely converges to globally optimal solutions. Unfortunately these theoretical cooling schedules are astronomically slow.
- In practice, one normally just starts from some "high" temperature T_0, and after each "sufficiently long" sweep L decrease the temperature by some "cooling factor" $\alpha \approx 0.8 \ldots 0.99$, i.e. to set $T_{k+1} = \alpha T_k$.
- Theoretically this is much too fast, but often seems to work well enough. (No one really understands why.)

Convergence of simulated annealing

- View the search space X with neighborhood structure N as a graph (X, N). Assume that this graph is undirected, connected, and of degree r. (Each node=solution has exactly r neighbors.)

- Denote by $X^* \subseteq X$ the set of globally optimal solutions. The following result was proved by Geman & Geman (1984) and Mitra, Romeo & Sangiovanni-Vincentelli (1986):

Convergence of simulated annealing cont.

Theorem. Consider a simulated annealing computation on structure (X, N, c). Assume the neighborhood graph (X, N) is connected and regular of degree r. Denote:

$$\Delta = \max\{c(x') - c(x) \mid x \in X, x' \in N(x)\}.$$

Choose

$$L \geq \min_{x^* \in X^*} \max_{x \notin X^*} \text{dist}(x, x^*),$$

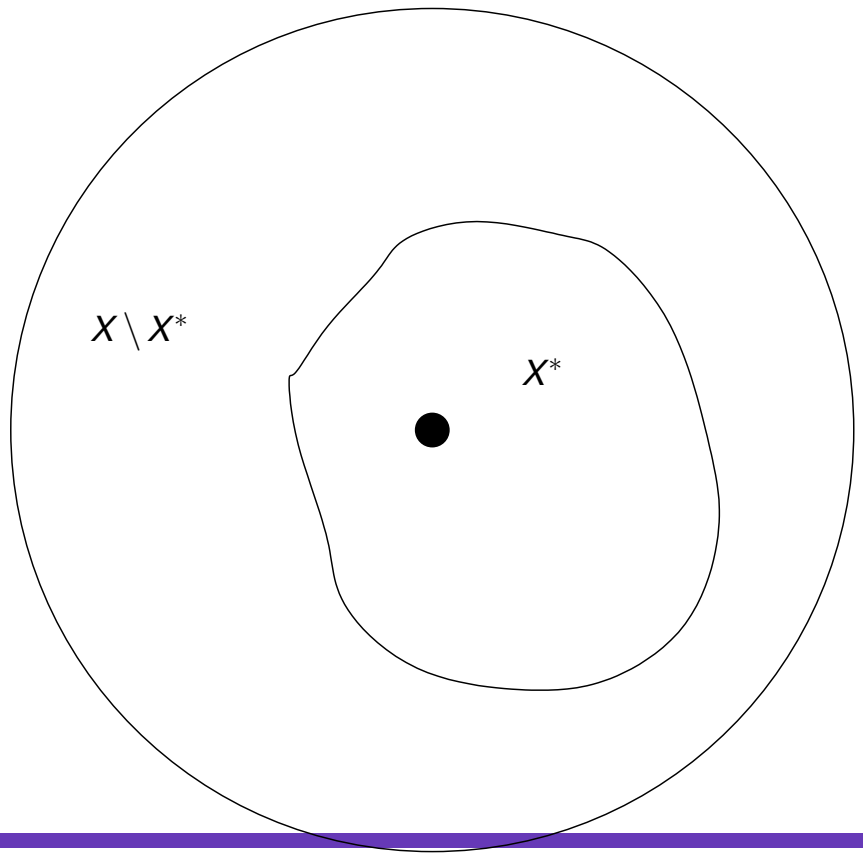
where $\text{dist}(x, x^*)$ is the shortest-path distance in graph (X, N) from node x to node x^*. Suppose the cooling schedule used is of the form $\langle T_0, L\rangle, \langle T_1, L\rangle, \langle T_2, L\rangle, \ldots$, where for each cooling stage $\ell \geq 2$:

$$T_\ell \geq \frac{L\Delta}{\ln \ell} \quad (\text{but } T_\ell \xrightarrow[\ell \to \infty]{} 0).$$

Convergence of simulated annealing cont.

Then the distribution of states visited by the computation converges in the limit to π^*, where

$$\pi^*_x = \begin{cases} 0, & \text{if } x \in X \setminus X^*, \\ 1/|X^*|, & \text{if } x \in X^*. \end{cases}$$



$$L \geq \min_{x^* \in X^*} \max_{x \notin X^*} \text{dist}(x, x^*)$$

Tabu search (Glover 1986)

Idea: Prevent a local search algorithm from getting stuck at a local minimum, or cycling through equally good solutions, by recording recently visited solutions (*tabu list*) and excluding moves to these. Sometimes finite convergence to an optimal solution can be shown (Hanafi 2001).

function TABU(c, tt):
    $x \leftarrow x_{init}$; $x^* \leftarrow x$; initialize TL to $\{x\}$;
    while moves $<$ max_moves **do**
        remove from TL solutions entered there more than tt moves ago;
        choose an $x' \in N(x) \setminus TL$ of minimum cost;
        $x \leftarrow x'$;
        add x to TL;
        if $c(x) < c(x^*)$ **then** $x^* \leftarrow x$;
    end while;
    return x^*;

Tabu search: practical considerations

- To save tabu list memory and access time, it may be worthwhile not to store complete solutions in the list, but just the recent *moves* (local transformations).
- Potential problem: a move may be tabu at time t (in the context of some earlier solution $x_{t'}$, $t' < t$), whereas it would lead to an interesting new solution in the context of solution x_t.
- To resolve this issue, heuristics for overriding the tabu rule (so-called *aspiration rules*) have been introduced, such as "always accept objective-improving moves" (i.e. such that $c(x') < c(x)$).

Tabu search applied to SAT(O)

Given propositional formula F on n variables $\{x_1, \ldots, x_n\}$ in conjunctive normal form, choose:

- Feasible solutions: truth assignments $t : \{x_1, \ldots, x_n\} \to \{0, 1\}$.
- Objective function: $c(t) =$ number of clauses unsatisfied by t.
- Neighborhood structure: $N(t) =$ truth assignments t' that differ from t in exactly one variable.
- Full tabu list: recently visited truth assignments.
- Abbreviated tabu list: recently flipped variables.

Other paradigms

A large number of other local search paradigms have been discussed in the literature, making use of dynamically changing neighborhood structures, adaptive evaluation functions etc.

Classification by Hoos & Stützle (2005):
Iterative improvement (II), Randomized iterative improvement (RII), Variable neighborhood descent (VND), Variable depth search (VDS), Simulated annealing (SA), Tabu search (TS), Dynamic local search (DLS), Iterated local search (ILS), Greedy randomized 'adaptive' search (GRASP), Adaptive iterated construction scheme (AICS), Ant colony optimization (ACO), and Memetic algorithm (MA).

See `http://www.sls-book.net/Sample-Pages/glossary.pdf` or also `http://cs.gmu.edu/~sean/book/metaheuristics/`.

Lecture 4: Constraint satisfaction problems

Outline

- Constraint satisfaction problems (CSP's) and constrained optimization problems (COP's), concepts and models
- Example encodings of several computational problems
- Alternative encodings and potential benefits they offer

Goals for today: Learn to recognize and formulate CSP's and COP's; when given a high-level description of a computational problem (search or optimization), learn to

a) encode the problem into a CSP/COP

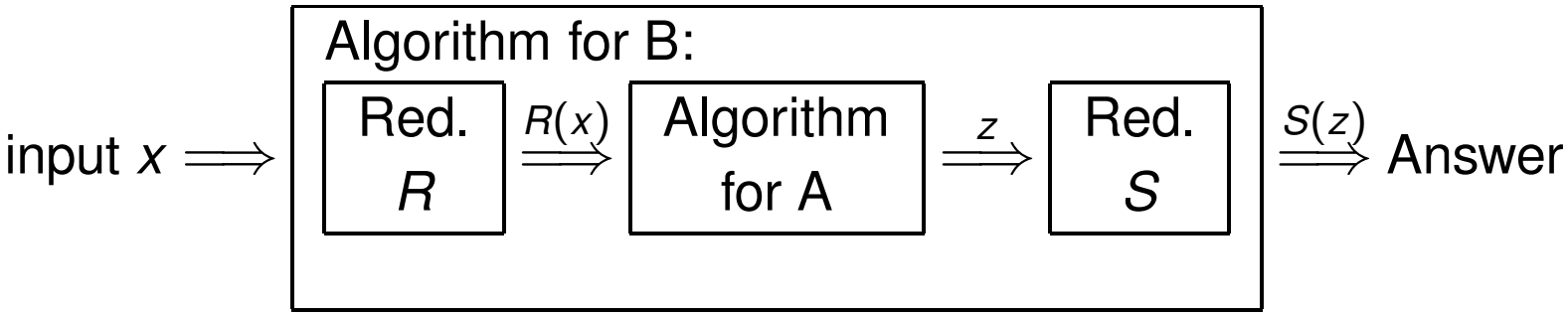
b) evaluate the benefit when choosing one of different encodings of the same problem

Constraint satisfaction: motivation

- When solving a search problem the most efficient solution methods are typically based on special purpose algorithms.
- In previous lectures important approaches to developing such algorithms have been discussed.
- However, developing a special purpose algorithm for a given problem requires typically a substantial amount of expertise and considerable resources.
- Another approach is to exploit an efficient algorithm already developed for some problem through *reductions* (introduced in Lecture 2).

Recall: exploiting reductions

- Given an (efficient) algorithm for a problem A we can solve a problem B by developing a (efficient) reduction from B to A and translating the solution of A back to a solution to B.



- *Constraint satisfaction problems* (CSP's) offer attractive target problems for reductions (CSP=Problem A).

Reduction to CSP's

1. Encode the computational problem as a CSP (i.e., compute the reduction R).
2. Solve the CSP via a complete or local search methods (see next lecture).
3. Extract from a solution to the CSP encoding a solution to the original problem (i.e., compute the reduction S).

- *Constraint programming* offers tools to build efficient algorithms for solving CSP's for a wide range of constraints.
- Constraint programming differs from, e.g., imperative programming, in the property that one does not specify instructions to be executed but properties of solutions to be found.
- There are efficient software packages that can be directly used for solving interesting classes of constraints.

Constraint Programming: toy example

- Consider the following problem of finding a course schedule for the ICS department.
- Assume there are three courses all taught in spring: DMS, Logic and Combinatorics.
- Say, at each time there are two lecture rooms available: lecture halls T_1 and T_2.
- Each day, there are time slots $10-12$ and $14-16$ available.
- Assume (unrealistically), every course has a lecture at every day of the week.
- Find time slots and rooms for each course on a given day!

Constraint Programming: toy example—cont'd

- For each course, its time and assigned room are variables.
- With each variable, we associate a set of potential values, which is its *domain*.
- Here, each room-choice variable has the domain $\{T_1, T_2\}$ and the domain for the time-choice variables is $\{(10-12), (14-16)\}$.
- Domains are typically finite and (usually) discrete.
- *Constraints* limit the possible assignments of values.
- Here there is only one type of constraint: *if two courses have been assigned the same time, they must be lectured in different rooms.*
- Question: how to represent constraints?

Constraint Programming: toy example—cont'd

- Answer: constraints are formalized as subsets of the Cartesian product (product set) of the domains of the affected variables.
- Consider the constraint

  $C :=$ *if DMS and Combinatorics have been assigned the same time, they must be lectured in different rooms.*

  which can be formalized as the set of tuples of values that satisfy the constraint:

$$
\begin{aligned}
C := \{&((10-12),(10-12),T_1,T_2),((10-12),(10-12),T_2,T_1),\\
&((14-16),(14-16),T_1,T_2),((14-16),(14-16),T_2,T_1),\\
&((10-12),(14-16),T_1,T_1),((14-16),(10-12),T_1,T_1),\\
&((10-12),(14-16),T_2,T_2),((14-16),(10-12),T_2,T_2)\}\\
&((10-12),(14-16),T_1,T_2),((14-16),(10-12),T_1,T_2),\\
&((10-12),(14-16),T_2,T_1),((14-16),(10-12),T_2,T_1)\}\\
\subseteq\ &\{(10-12),(14-16)\}\times\{(10-12),(14-16)\}\times\{T_1,T_2\}\times\{T_1,T_2\}.
\end{aligned}
$$

Constraints: formal model

- Consider some variables $x_1, \ldots, x_k$ and their domains $D_1, \ldots D_k$.
- Formally, a *constraint* C on variables $x_1, \ldots, x_k$ is a subset of $D_1 \times \cdots \times D_k$.
- The number of affected variables k is the *arity* of the constraint.
- If $k = 1$, the constraint is called *unary* and if $k = 2$, *binary*.

Example. Consider variables x_1, x_2 both having the domain $D_i = \{0, 1, 2\}$. Then the set

$$\{(0,1),(0,2),(1,0),(1,2),(2,0),(2,1)\} \subseteq D_1 \times D_2$$

can be taken as a binary constraint on x_1, x_2 and then we denote it by $NotEq(x_1, x_2)$.

Constraints: formal model—cont'd

- From now on we use a shorthand notation for constraints by giving directly the condition on the variables when it is clear how to interpret the condition on the domain elements.
- Hence, $cond(x_1, \ldots, x_k)$ on variables $x_1, \ldots, x_k$ with domains $D_1, \ldots D_k$ denotes the constraint

  $$\{(d_1, \ldots, d_k) \mid d_i \in D_i \text{ for } i = 1, \ldots, k \text{ and } cond(d_1, \ldots, d_k) \text{ holds } \}.$$

- *Note:* If there are in total n variables, then each constraint $cond(x_1, \ldots, x_k)$ is defined with respect to an *ordered subset* of the set of all variables $\{x_1, x_2, \ldots, x_n\}$.

Constraints—cont'd

Example

The condition $x_1 \neq x_2$ on variables x_1, x_2 with domains D_1, D_2 denotes the constraint

$$\{(d_1, d_2) \mid d_1 \in D_1, d_2 \in D_2, d_1 \neq d_2\}.$$

So if x_1, x_2 both have the domain $\{0,1,2\}$, then $x_1 \neq x_2$ denotes the constraint $\mathit{NotEq}(x_1, x_2)$ above.

Example

The condition $x_1 \leq \frac{x_2}{2} + \frac{1}{4}$ on x_1, x_2 both having the domain $\{0,1,2\}$ denotes the constraint

$$\{(d_1, d_2) \mid d_1, d_2 \in \{0,1,2\}, d_1 \leq \frac{d_2}{2} + \frac{1}{4}\} = \{(0,0),(0,1),(0,2),(1,2)\}.$$

Constraint Satisfaction Problems (CSP's)

- Given variables $x_1, \ldots, x_n$ and domains $D_1, \ldots D_n$, a *constraint satisfaction problem* (CSP):

$$\langle \mathbf{C}; x_1 \in D_1, \ldots, x_n \in D_n \rangle$$

where **C** is a set of constraints each defined on an ordered subset of $\{x_1, \ldots, x_n\}$.

Example

$$\langle \{NotEq(x_1, x_2), NotEq(x_1, x_3), NotEq(x_2, x_3)\}, \\ x_1 \in \{0,1,2\}, x_2 \in \{0,1,2\}, x_3 \in \{0,1,2\} \rangle$$

is a CSP. We often use shorthands for the constrains and write

$$\langle \{x_1 \neq x_2, x_1 \neq x_3, x_2 \neq x_3\}, x_1 \in \{0,1,2\}, x_2 \in \{0,1,2\}, x_3 \in \{0,1,2\} \rangle$$

CSP's: Value assignment

- For a CSP $\langle \mathbf{C}; x_1 \in D_1, \ldots, x_n \in D_n \rangle$ a potential solution is given by a *value assignment* which is a mapping T from $\{x_1, \ldots, x_n\}$ to $D_1 \cup \cdots \cup D_n$ such that for each variable x_i, $T(x_i) \in D_i$.
- A value assignment T *satisfies* a constraint C on variables $x_{i_1}, \ldots, x_{i_m}$ if $(T(x_{i_1}), \ldots, T(x_{i_m})) \in C$.

Example

A value assignment $T = \{x_1 \mapsto 1, x_2 \mapsto 2, \ldots, x_n \mapsto n\}$ satisfies the constraint *NotEq* on x_1, x_2 because

$(T(x_1), T(x_2)) = (1,2) \in \mathit{NotEq}(x_1, x_2)$

but $T' = \{x_1 \mapsto 1, x_2 \mapsto 1, \ldots, x_n \mapsto 1\}$ does not as

$(T'(x_1), T'(x_2)) = (1,1) \notin \mathit{NotEq}(x_1, x_2)$.

CSP's: Solution

- A *solution to a CSP* $\langle \mathbf{C}, x_1 \in D_1, \ldots, x_n \in D_n \rangle$ is a value assignment that satisfies every constraint $C \in \mathbf{C}$.

Example

Consider a CSP
$\langle \{x_1 \neq x_2, x_1 \neq x_3, x_2 \neq x_3\}, x_1 \in \{0,1,2\}, x_2 \in \{0,1,2\}, x_3 \in \{0,1,2\} \rangle$

The assignment $\{x_1 \mapsto 0, x_2 \mapsto 1, x_3 \mapsto 2\}$ is a solution to the CSP as it satisfies all the constraints but $\{x_1 \mapsto 0, x_2 \mapsto 1, x_3 \mapsto 1\}$ is not as it does not satisfy the constraint $x_2 \neq x_3$ ($\mathit{NotEq}(x_2, x_3)$).

Example: Graph k-Coloring Problem

Given a graph G, the coloring problem can be encoded as a CSP:

- For each node v_i in the graph introduce a variable V_i with the domain $\{1, \ldots, k\}$ where k is the number of available colors.
- For each edge (v_i, v_j) in the graph introduce a constraint $V_i \neq V_j$.
- This is a *reduction* of the k-coloring problem to a CSP because the solutions to the CSP correspond exactly to the solutions of the coloring problem:
  a value assignment $\{V_1 \mapsto t_1, \ldots, V_n \mapsto t_n\}$ satisfying all the constraints gives a valid coloring of the graph where node v_i is colored with color t_i.

Example: SEND + MORE = MONEY

- Replace each letter by a different digit so that

$$\begin{array}{r} \text{SEND} \\ +\ \text{MORE} \\ \hline \text{MONEY} \end{array}$$

is a correct sum.

$$\begin{array}{r} 9567 \\ +\ 1085 \\ \hline 10652 \end{array}$$

The unique solution.

- Variables: S, E, N, D, M, O, R and Y.
- Domains: {1,. . . ,9} for S, M and {0,. . . ,9} for E, N, D, O, R, Y.
- Constraints:

$$\begin{aligned} & 1000 \cdot S + 100 \cdot E + 10 \cdot N + D \\ + & 1000 \cdot M + 100 \cdot O + 10 \cdot R + E \\ = & 10000 \cdot M + 1000 \cdot O + 100 \cdot N + 10 \cdot E + Y \end{aligned}$$

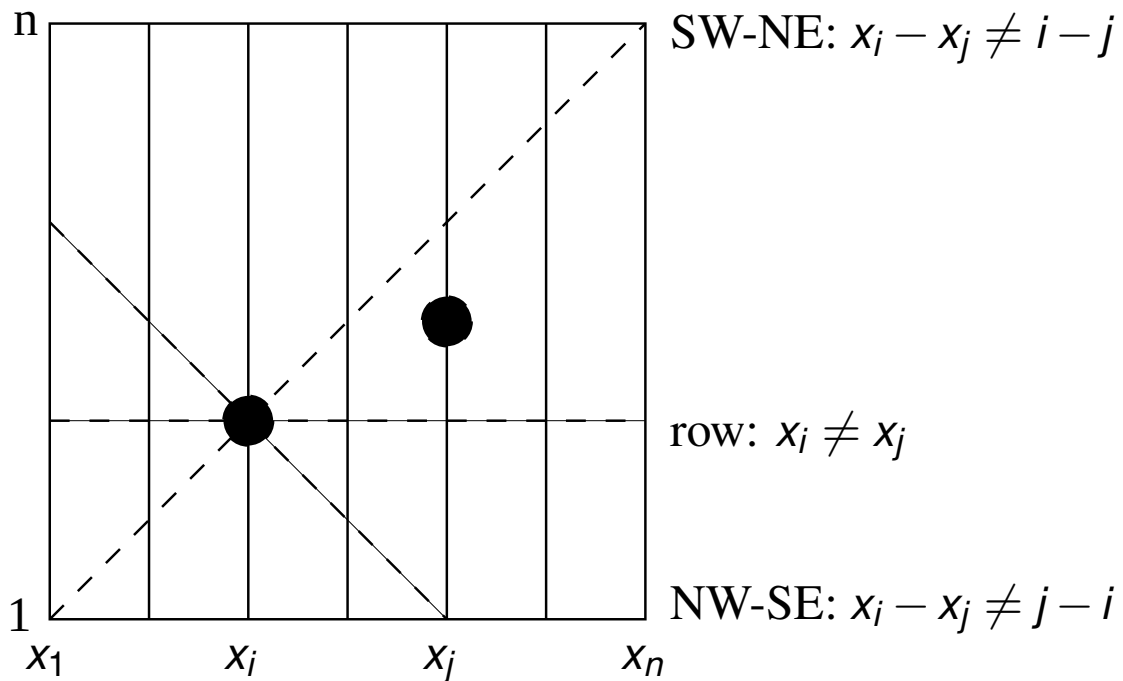
and $x \neq y$ for every variable pair x, y in {S, E, N, D, M, O, R, Y}.

- It is easy to check that the value assignment
$\{S \mapsto 9, E \mapsto 5, N \mapsto 6, D \mapsto 7, M \mapsto 1, O \mapsto 0, R \mapsto 8, Y \mapsto 2\}$
satisfies the constraints, i.e., is a solution to the problem.

N Queens

Problem: Place n queens on a $n \times n$ chess board so that they do not attack each other.

- Variables: $x_1, \ldots, x_n$ (x_i gives the row-position of the queen on the i-th column)
- Domains: $\{1, \ldots, n\}$ for each $x_i, i = 1, \ldots, n$
- Constraints: for $i \in \{1, \ldots, n-1\}$ and $j \in \{i+1, \ldots, n\}$:



Constrained Optimization Problems

- Given: a CSP $P := \langle \mathbf{C}; x_1 \in D_1, \ldots, x_n \in D_n \rangle$ and a function *obj* which maps solutions of the CSP to real numbers.
- (P, obj) is a *constrained optimization problem* (COP) where the task is to find a solution T to P for which the value $obj(T)$ is optimal.
- Both versions, minimization and maximization, of the objective function are possible, of course.
- Note1: in practice, instead of considering *obj* to be a function of T, one usually formulates it as a function of the product set of the domains, i.e., $obj : D_1 \times D_2 \times \ldots D_n \mapsto \mathbb{R}$.
- Note2: in some sense the CSP is the search-problem version of the COP, which asks for the best feasible solution.

Constrained Optimization Problems—cont'd

- **Example.** KNAPSACK: a knapsack of a fixed volume and n objects, each with a volume and a value. Find a collection of these objects with maximum total value that fits in the knapsack.
- Representation as a COP:
  Given: knapsack volume v and n objects with volumes $a_1, \ldots, a_n$ and values $b_1, \ldots, b_n$.

  Variables: $x_1, \ldots, x_n$ (Idea: x_i has value 1 iff item i is included in the collection of items.)
  Domains: $\{0, 1\}$
  Constraint: $\sum_{i=1}^{n} a_i \cdot x_i \leq v$,
  Objective function: $\sum_{i=1}^{n} b_i \cdot x_i$.

More examples

Examples of problems encountered in practice:

- Scheduling / timetabling problems: production panning, plant refueling, course timetabling, vehicle routing, car sequencing, etc.
- Rostering: airline crews, hospital staff, etc.
- Network optimization: routing, capacity provisioning, transmission scheduling, etc.

Applications typically fall into the domain of *operations research*.

Solving CSP's

- Different encodings of a problem as a CSP utilizing different sets of constraints can have substantial different computational properties.
- However, it is not obvious which encodings lead to the best computational performance.
- In the course we consider more carefully two classes of constraints: *Boolean constraints* (Lecture 6) and *linear constraints* (Lecture 9).
- *Linear constraints* are an example of a class of constraints which has efficient special purpose algorithms.
- For others *general methods* consisting of *constraint propagation algorithms* and search methods are available (Lecture 5).

General remarks on CSP encoding

Not all models are equal! Sometimes, the following guidelines are worth considering.

- *Try to avoid high-arity constraints, unless explicitly supported by an available constraint solver.*
- Constraints involving a large number of variables may cause trouble for solvers that rely on constraint propagation techniques, which frequently evaluate the effect of changes of variable values on constraint satisfiability depending on the remaining variables.
- One solution: replace a single constraint involving many variables by a few with only a small number of variables.
- Recall SEND + MORE = MONEY example:

SEND	9567
+ MORE	+ 1085
MONEY	10652

Alternative formulation

- Old constraint:

$$\begin{aligned} & 1000 \cdot S + 100 \cdot E + 10 \cdot N + D \\ + & 1000 \cdot M + 100 \cdot O + 10 \cdot R + E \\ = \; & 10000 \cdot M + 1000 \cdot O + 100 \cdot N + 10 \cdot E + Y \end{aligned}$$

- New variables C_1, C_2, C_3, C_4, C_5 with domains $\{0, 1\}$ and replacement constraints

$$\begin{aligned} D + E &= Y + C_1 \cdot 10 \\ C_1 + N + R &= E + C_2 \cdot 10 \\ C_2 + E + O &= N + C_3 \cdot 10 \\ C_3 + S + M &= O + C_4 \cdot 10 \\ C_4 &= M \end{aligned}$$

- Advantage? Maximum arity 5 (incl. 2 binary variables) compared to 8 in the original (translates to $4 \cdot 10^3$ vs 10^8).

General remarks on CSP encoding—cont'd

- *Prefer a few variables with large domains over many variables with small domains.*
- Recall N Queens: n variables $x_1, \ldots, x_n$, each with domain $\{1, \ldots, n\}$, corresponding to the row-position of the queen on the i-th column.
- Consider replacing these with Boolean variables

$$x_{ij} = \begin{cases} 1, & \text{if and only if there is a queen in column } i, \text{ row } j, \\ 0, & \text{otherwise.} \end{cases}$$

- The search space increased from n^n to 2^{n^2}!
- Additionally n new constraints *queen i is assigned to one row exactly*, which were satisfied implicitly earlier!

General remarks on CSP encoding—cont'd

- *Try to avoid symmetry in solutions.*
- Example: if two tasks in a scheduling problem can be interchanged with no effects on constraints and cost, introduce artificial pairwise order constraints.
- Intuitive explanation: symmetries may fool complete methods into performing unnecessary computation by exploring branches of the search-tree that may not lead to better solutions than currently known ones.
- For more and an introduction to CSP's in general see the review by Brailsford, Potts and Smith '99 (see Noppa, additional-reading page, *voluntary but encouraged*).

CSP's in practice

- Interesting invited talk by Laurent Perron (Google) from 2011:
  `http://www.dmi.unipg.it/cp2011/invited.html`;
  some quotes about solving COP's/CSP's in practice:

1. *Getting the right problem with the right people is hard.*
2. *Getting clean data is hard.*
3. *Solving the problem is easy.*
4. *Reporting the result/explaining the implications is hard.*

- *Time spent is 50 / 25 / 5 / 20 %.*

Lecture 5: Complete and local search methods for CSP's

Outline

- Algorithms for solving constraint satisfaction and constrained optimization problems
- Complete algorithms and local search methods
- Constraint propagation: concept and methods

Goal for today: When faced with a given class of constraint satisfaction/optimization problems, learn to devise a complete/local search method for finding solutions by employing some of the techniques discussed today.

Constraint satisfaction: Algorithms

- For some classes of constraints there are efficient special purpose algorithms (domain specific methods/constraint solvers).
- But now we consider general methods consisting of *constraint propagation techniques* and *search methods*.
- Note: to simplify presentation we use CSP's for the discussion, although the algorithms naturally extend to COP's.
- Recall: Given variables $x_1, \ldots, x_n$ and domains $D_1, \ldots D_n$, a *constraint satisfaction problem* (CSP) is formulated as:

$$\langle \mathbf{C}; x_1 \in D_1, \ldots, x_n \in D_n \rangle$$

  where $\mathbf{C}$ is a set of constraints each defined on an ordered subset of $\{x_1, \ldots, x_n\}$.
- Example:

$$\langle \{x_1 \neq x_2, x_1 \neq x_3, x_2 \neq x_3\}, x_1 \in \{0,1,2\}, x_2 \in \{0,1,2\}, x_3 \in \{0,1,2\} \rangle$$

Solve

- The first method (here called *Solve*) is very similar to complete search methods seen in Lecture 3, such as backtrack search.
- The procedure Solve takes as input a constraint satisfaction problem (CSP) and transforms it until it is *solved*.
- It employs a number of subprocedures: Happy, Preprocess, Constraint Propagation, Atomic, Split, Proceed by Cases; (*Principles of Constraint Programming*, Krzysztop R. Apt, 1999).

Happy, Atomic	check of termination condition
Preprocess, Constraint Propagation	transformation of CSP to another one that is equivalent to it
Split	division of CSP into two or more CSP's whose union is equivalent to the CSP
Proceed by Cases	specifies what search techniques are used to process the CSP's generated by Split

Constraint Programming: Basic Framework

```
procedure Solve(CSP/COP P):
P ← Preprocess(P);
P ← Constraint_Propagation(P);
if not Happy(P) then
   if Atomic(P) then
      return;     /* impossible to split */
   else
      (P_1, P_2, ...) ← Split(P);
      Proceed_by_Cases(P_1, P_2, ...);     /* may lead to recursive calls */
   end
end
```

Equivalence of CSP's

- To understand Solve we need the notion of equivalence of CSP's.
- Informally, CSP's $\mathbf{P}_1$ and $\mathbf{P}_2$ are *equivalent* if they have the same set of solutions (satisfying assignments of values to variables).
- However, transformations can add new variables to a CSP and then equivalence is understood w.r.t. the original variables.
- Recall: For a CSP $\langle \mathbf{C}; x_1 \in D_1, \ldots, x_n \in D_n \rangle$ a *value assignment* (a.k.a. potential solution) is a mapping T from $\{x_1, \ldots, x_n\}$ to $D_1 \cup \cdots \cup D_n$ such that for each variable x_i, $T(x_i) \in D_i$.
- We say that two value assignments T and T' *agree on a set of variables* X iff $T(x) = T'(x)$ for all $x \in X$.
- Then define equivalence w.r.t. variables in set X based on solutions to the CSP's that agree on X as follows.

Equivalence of CSP's—cont'd

We say that two CSP's $\mathbf{P}_1$ and $\mathbf{P}_2$ are equivalent w.r.t. a set of variables X iff

- for every solution T_1 of P_1 there exists a solution T_2 of P_2 such that T_1 and T_2 agree on variables X and
- for every solution T_2 of P_2 there exists a solution T_1 of P_1 such that T_2 and T_1 agree on variables X.

Equivalence of CSP's – Example

Consider the following two CSP's that are equivalent on $X = \{x_1, x_2\}$:
$P_1 = \langle \{x_1 < x_2\}; x_1 \in \{1,3\}, x_2 \in \{1,3\} \rangle$
$P_2 = \langle \{x_1 < x_3, x_3 \leq x_2\}; x_1 \in \{1,3\}, x_2 \in \{1,3\}, x_3 \in \{1,2,3\} \rangle$

- for the unique solution $T_1 = \{x_1 \mapsto 1, x_2 \mapsto 3\}$ of P_1 there is a corresponding solution $T_{21} = \{x_1 \mapsto 1, x_2 \mapsto 3, x_3 \mapsto 3\}$ of P_2 such that T_1 and T_{21} agree on variables X and ...
- for the solutions T_{21} and $T_{22} = \{x_1 \mapsto 1, x_2 \mapsto 3, x_3 \mapsto 2\}$ of P_2, T_1 is a corresponding solution of P_1 agreeing on X.

Equivalence of CSP's cont'd

We extend the notion of equivalence to several CSP's as follows:
A union of CSP's $\mathbf{P}_1, \ldots, \mathbf{P}_m$ is equivalent to a CSP $\mathbf{P}_0$ w.r.t. X iff

- for every solution T_0 of P_0 there exists a solution T_i of P_i for some $1 \leq i \leq m$ such that T_0 and T_i agree on variables X and
- for each $1 \leq i \leq m$ and for every solution T_i of P_i there exists a solution T_0 of P_0 such that T_i and T_0 agree on variables X.

For instance, CSP

$$P_0 = \langle \{x_1 < x_2\}; x_1 \in \{1, \ldots, 10\}, x_2 \in \{1, \ldots, 10\} \rangle$$

is equivalent w.r.t. $\{x_1, x_2\}$ to the union of the two CSP's

$$P_{01} = \langle \{x_1 < x_2\}; x_1 \in \{1, \ldots, 5\}, x_2 \in \{1, \ldots, 10\} \rangle$$

$$P_{02} = \langle \{x_1 < x_2\}; x_1 \in \{6, \ldots, 10\}, x_2 \in \{1, \ldots, 10\} \rangle$$

Solved and Failed CSP's

- For termination one needs to define when a CSP has been solved and when it is failed.
- Let C be a constraint on variables $y_1, \ldots, y_k$ with domains $D_1, \ldots, D_k$ ($C \subseteq D_1 \times \cdots \times D_k$).
- C is *solved* if $C = D_1 \times \cdots \times D_k$ and $C \neq \emptyset$.
- A CSP is *solved* if
  a) all its constraints are solved and
  b) none of the domains is empty.
- A CSP is *failed* if
  a) it contains the empty (false) constraint $\bot$ or
  b) one of its domains is empty.

Happy

test applied to the current CSP to see whether the goal conditions set for the original CSP have been achieved. Typical conditions include:

- a solution has been found,
- all solutions have been found,
- a solved form has been reached from which one can generate all solutions,
- it is determined that no solution exists (the CSP is failed),
- an optimal solution w.r.t. some objective function has been found,
- all optimal solutions have been found.

Example For a CSP $\langle \{x_1 + x_2 = x_3, x_1 - x_2 = 0\}; x_i \in D_i \rangle$ the solved form could be, for example, $\langle \{x_1 = x_2, x_3 = 2x_2\}; x_i \in D_i \rangle$.

Transformations

- In the following we represent transformations of CSP's by means of proof rules.

- A rule

$$\frac{\mathbf{P}_0}{\mathbf{P}_1}$$

  transforms the CSP $\mathbf{P}_0$ to the CSP $\mathbf{P}_1$.

- A rule

$$\frac{\mathbf{P}_0}{\mathbf{P}_1 \mid \cdots \mid \mathbf{P}_n}$$

  transforms the CSP $\mathbf{P}_0$ to the set of CSP's $\mathbf{P}_1, \ldots, \mathbf{P}_n$.

Preprocess

- The aim is to bring constraints to a desired syntactic form.
- Example: Constraints on reals.
  Desired syntactic form: no inequalities in more than one variable

$$\frac{x + y \geq 5}{x + y - z = 5, z \geq 0}$$

(Notice that a new variable is introduced.)

Atomic

- This is a test applied to the current CSP to see whether the CSP is available for splitting.
- Typically a CSP is considered atomic if the domains of the variables are either singletons or empty.
- But a CSP can be viewed as atomic also if it is clear that search 'under' this CSP is not needed.
  For example, this could be the case when the CSP is "solved" or an optimal solution can be computed directly from the CSP.

Split

- After Constraint Propagation, Split is called when the test Happy fails but the CSP is not yet Atomic.
- A call to Split replaces the current CSP $\mathbf{P}_0$ by CSP's $\mathbf{P}_1, \ldots, \mathbf{P}_n$ such that the union of $\mathbf{P}_1, \ldots, \mathbf{P}_n$ is equivalent to $\mathbf{P}_0$, i.e., the rule

$$\frac{\mathbf{P}_0}{\mathbf{P}_1 \mid \cdots \mid \mathbf{P}_n}$$

  is applied.
- A split can be implemented by splitting domains or constraints.
- For efficiency an important issue is the splitting *heuristics*, i.e., which split to apply and in which order to consider the resulting CSP's.

Split — a domain

- D finite (Enumeration) : $$\frac{x \in D}{x \in \{a\} \mid x \in D \setminus \{a\}}$$
- D finite (Labeling) : $$\frac{x \in \{a_1, \ldots, a_k\}}{x \in \{a_1\} \mid \ldots \mid x \in \{a_k\}}$$
- D interval of reals (Bisection) : $$\frac{x \in [a, b]}{x \in [a, \frac{a+b}{2}] \mid x \in (\frac{a+b}{2}, b]}$$

Split — a constraint

- Disjunctive constraints like
  Start[task1] + Duration[task1] $\leq$ Start[task2] $\vee$
  Start[task2] + Duration[task2] $\leq$ Start[task1]
  can be split using the rule: $\dfrac{C1 \vee C2}{C1 \mid C2}$

- Constraints in "compound" form:
  $$\frac{|x+y| = a}{x+y = a \mid x+y = -a}$$

Heuristics

Which

- variable to choose,
- value to choose,
- constraint to split.

Examples:

(i) Select a variable that appears in the largest number of constraints (most constrained variable).

(ii) For a domain being an integer interval: select the middle value.

Proceed by Cases

- Various search techniques can be applied.
- A typical solution is to use
  - backtracking or
  - branch and bound
- and combine these with
  - efficient constraint propagation and
  - intelligent backtracking (e.g., conflict directed backjumping)
- As the search trees are often very big, you tend to avoid techniques where much more than the current branch of the search tree needs to be stored.

Constraint Propagation

- Intuition: Replace a CSP by an equivalent one that is "simpler".
- Basic idea: exploit dependence of variables and constraints to *reduce* ("shrink") domains of some variables and/or constraints
- By *constraint propagation* we mean applying repeatedly reduction steps.
- Efficient constraint propagation enabling substantial reductions is a key issue for overall performance.
- Note: the following examples use integer interval notation: $[5..10] = \{5, 6, 7, 8, 9, 10\}$

Constraint Propagation—cont'd

Domain Reduction

- Linear inequalities on integers:

$$\frac{\langle x < y; x \in [l_x..h_x], y \in [l_y..h_y]\rangle}{\langle x < y; x \in [l_x..h'_x], y \in [l'_y..h_y]\rangle}$$

where $h'_x = min(h_x, h_y - 1), l'_y = max(l_y, l_x + 1)$

Example:

$$\frac{\langle x < y; x \in [50..200], y \in [0..100]\rangle}{\langle x < y; x \in [50..99], y \in [51..100]\rangle}$$

Constraint Reduction

Usually by introducing new constraints.

- Transitivity of $<$: $\dfrac{\langle x < y, y < z; x \in D_x, y \in D_y, z \in D_z\rangle}{\langle x < y, y < z, x < z; x \in D_x, y \in D_y, z \in D_z\rangle}$

This rule introduces new constraint, $x < z$.

Repeated Domain Reduction: Example

- Consider $\langle x < y, y < z; x \in [50..200], y \in [0..100], z \in [0..100] \rangle$
- Apply the rule from previous slide to $x < y$:
  $\langle x < y, y < z; x \in [50..99], y \in [51..100], z \in [0..100] \rangle$.
- Apply it now to $y < z$:
  $\langle x < y, y < z; x \in [50..99], y \in [51..99], z \in [52..100] \rangle$
- Apply it again to $x < y$:
  $\langle x < y, y < z; x \in [50..98], y \in [51..99], z \in [52..100] \rangle$

Constraint Propagation Algorithms

- The efficient scheduling of atomic reduction steps quickly becomes nontrivial.
- Constraint propagation algorithms perform reduction steps with the goal of achieving *local consistency*; depending on the class of constraints there are different notions of local consistency.
- The projection rule is a widely applicable and efficient general reduction rule.
- Note: to simplify implementation the rule is formulated as an update rule for the domains and keeps constraints unchanged.

Projection rule:

Take a constraint C on variables $x_1, \ldots, x_k$. From these variables, choose a variable x_i with domain D_i. Remove from D_i each value d for which there is no $(d_1, \ldots, d_i, \ldots, d_k) \in D_1 \times \cdots \times D_k$ such that $(d_1, \ldots, d_i, \ldots, d_k) \in C$ and $d_i = d$.

CSP: $\langle C_1(x,y,z), C_2(x,z); x \in \{1,2,3\}, y \in \{1,2,3\}, z \in \{1,2,3\}\rangle$
where $C_1 = \{(1,1,2),(1,2,1),(2,3,3)\}$, $C_2 = \{(1,1),(2,2),(3,3)\}$.

- Applying Projection rule to $C_1(x,y,z)$ and variables x,y,z yields $\langle C_1(x,y,z), C_2(x,z); x \in \{1,2\}, y \in \{1,2,3\}, z \in \{1,2,3\}\rangle$
- Applying Projection rule to $C_2(x,z)$ yields $\langle C_1(x,y,z), C_2(x,z); x \in \{1,2\}, y \in \{1,2,3\}, z \in \{1,2\}\rangle$
- Applying Projection rule to $C_1(x,y,z)$ yields $\langle C_1(x,y,z), C_2(x,z); x \in \{1\}, y \in \{1,2\}, z \in \{1,2\}\rangle$
- Applying Projection rule to $C_2(x,z)$ yields $\langle C_1(x,y,z), C_2(x,z); x \in \{1\}, y \in \{1,2\}, z \in \{1\}\rangle$
- Applying Projection rule to $C_1(x,y,z)$ yields $\langle C_1(x,y,z), C_2(x,z); x \in \{1\}, y \in \{2\}, z \in \{1\}\rangle$
  (This CSP is *hyper-arc consistent* and happens to be solved).

Hyper-Arc Consistency

If the projection rule is the only atomic reduction step and it is applied as long as new reductions can be made, then the constraint propagation algorithm achieves a local consistency notion called *hyper-arc consistency*:

> *A CSP is hyper-arc consistent if for every constraint C on variables $x_1, \ldots x_k$ and every x_i with domain D_i, for each $d \in D_i$, there is some $(d_1, \ldots, d_i, \ldots, d_k) \in D_1 \times \cdots \times D_k$ such that $(d_1, \ldots, d_i, \ldots, d_k) \in C$ and $d_i = d$.*

More formally: A CSP is hyper-arc consistent if

$$\forall C(x_1, \ldots, x_k) \in \mathbf{C}\Big(\forall x_i \in \{x_1, \ldots, x_k\}\big(d \in D_i \rightarrow$$

$$\exists (d_1, \ldots, d_i, \ldots, d_k) \in C(x_1, \ldots, x_k) \bigcap (D_1 \times \cdots \times D_k) \text{ such that } d_i = d\big)\Big)$$

Note: after one application of the projection rule the CSP satisfies the condition for one (constraint, variable in that constraint) pair.

Example

Consider the Solve procedure and a CSP

$$\langle \mathbf{C}; x_1 \in \{1,2,3\}, x_2 \in \{1,2,3\} \rangle$$

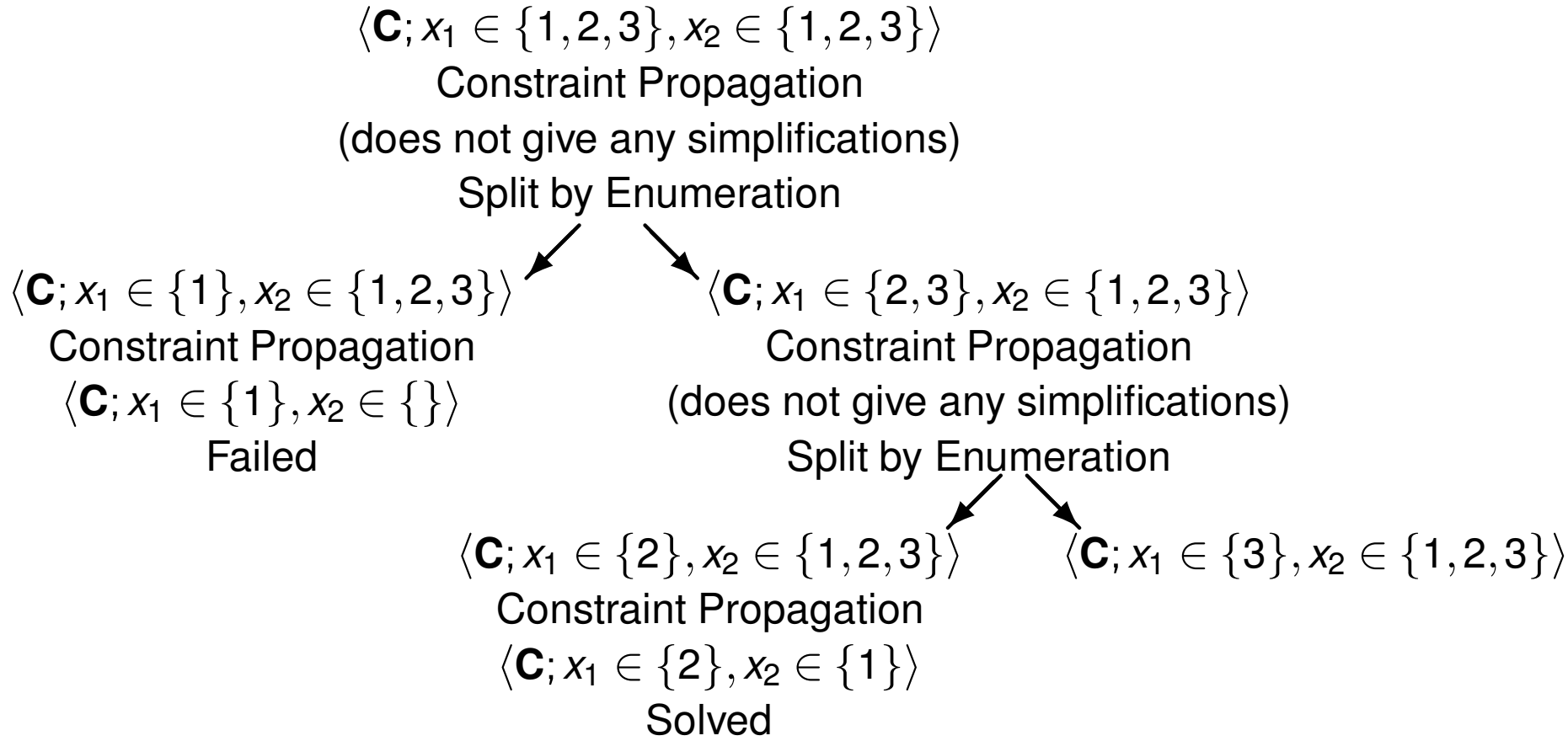
given as its input where

$$\mathbf{C} = \{x_1 \neq x_2, x_1 \geq x_2\}$$

Below the behaviour of Solve is given when (i) the goal is to find one solution (Happy), (ii) no Preprocessing is done, (iii) Constraint Propagation is based on the Projection rule, (iv) Splitting is based on enumeration and (v) search (Proceed by Cases) on depth first backtracking search.

Example—cont'd

(Here: $\mathbf{C} = \{x_1 \neq x_2, x_1 \geq x_2\}$)



Global Constraints

- Constraint programming systems often offer constraints with special purpose constraint propagation (filtering) algorithms. Such a constraint can typically be seen as an encapsulation of a set of simpler constraints and is called a *global constraint*.
- A global constraint is an *expressive and concise condition involving a non-fixed number of variables*. (see Global Constraint Catalog)
- A representative example is the alldiff constraint:

$$\text{alldiff}(x_1, \ldots, x_n) = \{(d_1, \ldots, d_n) \mid d_i \neq d_j, \text{for } i \neq j\}$$

Example. A value assignment $\{x_1 \mapsto a, x_2 \mapsto b, x_3 \mapsto c\}$ satisfies $\text{alldiff}(x_1, x_2, x_3)$ but $\{x_1 \mapsto a, x_2 \mapsto b, x_3 \mapsto a\}$ does not.

- $\text{alldiff}(x_1, \ldots, x_n)$ can be seen as an encapsulation of a set of binary constraints $x_i \neq x_j$, $1 \leq i < j \leq n$.

Global Constraints: alldiff

Global constraints enable *compact encodings* of problems.
Recall the N Queens problem:

Example. Place n queens on a $n \times n$ chess board so that they do not attack each other.

- Variables: $x_1, \ldots, x_n$ (x_i gives the row-position of the queen on the i-th column)
- Domains: {1,. . . ,n}
- Constraints: for $i \in \{1, \ldots, n-1\}$ and $j \in \{i+1 \ldots, n\}$:
  (i) $\text{alldiff}(x_1, \ldots, x_n)$ *(rows)*
  (ii) $x_i - x_j \neq i - j$ (SW-NE diagonals)
  (iii) $x_i - x_j \neq j - i$ (NW-SE diagonals)

Global Constraints: Propagation

- In addition to compactness global constraints often provide *more powerful propagation* than the same condition expressed as the set of corresponding simpler constraints.
- Consider variables x_1, x_2, x_3 with domains $D_1 = \{a, b, c\}, D_2 = \{a, b\}, D_3 = \{a, b\}$.
- Now $\mathrm{alldiff}(x_1, x_2, x_3)$ is not hyper-arc consistent and the projection rule removes values a, b from the domain of x_1.
- However, the corresponding set of constraints $x_1 \neq x_2, x_1 \neq x_3, x_2 \neq x_3$ is hyper-arc consistent and the projection rule is not able to remove any values.
- There is a wide range of such global constraints (e.g., see the Global Constraint Catalog `http://www.emn.fr/x-info/sdemasse/gccat/`).
- For some special-purpose algorithms exist (e.g., for $\mathrm{alldiff}$).

Local Search for CSP/COP

Many of the methods of Lecture 3 can be adapted to CSP's and COP's. As a different example we consider *Min Conflict Heuristic* (MCH) algorithm (Minton et al, 1990). Given a CSP instance C:

- Initialize each variable by selecting a value uniformly at random from its domain.
- In each local step select a variable x_i uniformly at random from the conflict set, which is the set of variables appearing in a constraint that is unsatisfied under the current assignment.
- A new value v for x_i is selected from the domain of x_i such that by assigning v to x_i the number of conflicting constraints is minimized.
- If there is more than one new value with that property, one of the minimizing values is chosen uniformly at random.

Example

Consider a run of MCH on a CSP
$\langle \{x_1 \leq x_2, x_2 \leq x_3, x_3 \leq x_1\}, x_1 \in \{1,2,3\}, x_2 \in \{1,2,3\}, x_3 \in \{1,2,3\} \rangle$

- First a value is selected for each variable uniformly at random from its domain, say $\{x_1 \mapsto 1, x_2 \mapsto 2, x_3 \mapsto 3\}$.
- For this assignment, the conflict set is $\{x_1, x_3\}$ from which, say, x_1 is randomly selected.
- Each possible assignment $x_1 \mapsto 1 / x_1 \mapsto 2 / x_1 \mapsto 3$ leaves one conflict and, hence, one of them is randomly selected, say $x_1 \mapsto 2$.
- For the resulting assignment $\{x_1 \mapsto 2, x_2 \mapsto 2, x_3 \mapsto 3\}$, the conflict set is $\{x_1, x_3\}$, from which x_3 is randomly selected.
- Now assignments $x_3 \mapsto 1 / x_3 \mapsto 3$ leave one conflict but $x_3 \mapsto 2$ leaves none.
- Hence, $x_3 \mapsto 2$ is selected leading to a solution $\{x_1 \mapsto 2, x_2 \mapsto 2, x_3 \mapsto 2\}$.

MCH—cont'd

Sometimes MCH appears to be too greedy and gets stuck quickly. Ways to mitigate this problem (see, e.g., Wallace and Freuder, 1995):

- A noise parameter p is introduced.
- Then in each local step: with probability p a new value for the variable from the conflict set is chosen randomly and with probability $1 - p$ the normal min conflict heuristics is followed.

MCH can also be extended with a tabu search mechanism (Steinmann et al. 1997):

- After each search step where the value of a variable x_i has changed from v to v', the assignment $x_i \mapsto v$ is declared tabu for the next tt steps.
- While $x_i \mapsto v$ is tabu, value v is excluded from the selection of values for x_i except if assigning v to x_i leads to an improvement in the evaluation function over the current assignment (*aspiration criterion*).

CSP: Tabu Search—cont'd

- A similar algorithm modifies MCH to choose over all non-tabu (variable, value) pairs the best move.
- *TS-GH* algorithm (Galinier and Hao, 1997):
  - In each local step: consider all non-tabu variable-value assignments $x \mapsto v$, where x appears in a currently unsatisfied constraint and v is in the domain of x.
  - Choose the assignment that leads to the maximal decrease in the number of violated constraints.
  - If there are multiple such assignments, one of them is chosen uniformly at random.
  - After changing the assignment of x from v to v', the assignment $x \mapsto v$ is declared tabu for tt steps (except when leading to an improvement).
- For competitive performance, the evaluation function for $x \mapsto v$ moves should use caching and incremental update techniques.

Example

Consider a local step of TS-GH on a CSP
$\langle \{x_1 \leq x_2, x_2 \leq x_3, x_3 \leq x_1\}, x_1 \in \{1,2,3\}, x_2 \in \{1,2,3\}, x_3 \in \{1,2,3\} \rangle$
where the current assignment is $\{x_1 \mapsto 2, x_2 \mapsto 2, x_3 \mapsto 3\}$

- Variables x_1, x_3 appear in an unsatisfiable constraint ($x_3 \leq x_1$).
- In MCH one of these would be randomly selected but in TS-GH we consider all assignments

$$x_1 \mapsto 1 / x_1 \mapsto 2 / x_1 \mapsto 3 / x_3 \mapsto 1 / x_3 \mapsto 2 / x_3 \mapsto 3$$

  and select an assignment leading to the maximal decrease in the number of violated constraints.
- Assignment $x_3 \mapsto 2$ leaves no violated constraints but other assignments leave a violated constraint.
- Hence, $x_3 \mapsto 2$ is selected leading to a solution $\{x_1 \mapsto 2, x_2 \mapsto 2, x_3 \mapsto 2\}$.

Tools for CSP

- Constraint programming systems offer a rich set of supported constraint types with efficient propagation algorithms and primitives for implementing search.

- See, for example, `http://4c.ucc.ie/web/archive/solver.jsp` and also `http://en.wikipedia.org/wiki/Constraint_programming` for solvers and libraries. Some examples:

  `CLAIRE, ECLiPse, GNU Prolog, Oz,`
  `Sicstus Prolog, ILOG Solver, ...`

Lecture 6: Boolean circuits

Outline

- Boolean circuits and circuit satisfiability
- Concepts and model, example circuits
- Tseitin's translation for BC's to CNF

Goals for today: Learn to represent propositional formulas as Boolean circuits and how to convert between Boolean circuits and propositional formulas in CNF.

Review

- In the last two lectures we discussed CSP's and COP's as targets for reductions of computational problems.
- Sometimes the expressivity of CSP's is not necessary.
  - Instead one can consider special cases (e.g., Boolean variables and constraints, i.e., SAT) that enable special purpose algorithmic solutions that typically perform well in practice.
  - Despite the simplicity of SAT, it provides a highly efficient approach for solving various hard computational problems
  - This is due to very efficient algorithms and solvers available
  - In addition more general constraints can be reduced to SAT
- In this lecture we discuss *Boolean circuits* as a viable method for propositional satisfiability.
- In the next lecture we consider complete and local search methods for finding solutions to SAT.

Recall: Representing Boolean Functions

- Consider an n-ary Boolean function $f : \{\textbf{true}, \textbf{false}\}^n \mapsto \{\textbf{true}, \textbf{false}\}$ with Boolean variables $x_1, \ldots, x_n$.
- Denote its Boolean inverse (negation) as $\overline{f} := \neg f$.

Example.

x_1	x_2	f	$\neg f$
0	0	0	1
0	1	1	0
1	0	1	0
1	1	0	1

- Disjunctive Normal Form (DNF):

$$f = (\neg x_1 \wedge x_2) \vee (x_1 \wedge \neg x_2)$$

- Conjunctive Normal Form (CNF):

$$\begin{aligned} f = \neg\neg f &= \neg((\neg x_1 \wedge \neg x_2) \vee (x_1 \wedge x_2)) \\ &= (x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2) \end{aligned}$$

- *Note*: The f in this example is the xor (exclusive or) function and $\overline{f}$ correspondingly the xnor (sometimes called equiv for *equivalent*) function.

Boolean Circuits

- CNF/DNF normal forms are often quite an unnatural way of encoding problems and it is more convenient to use full propositional logic.
- In many applications the encoding is of considerable size and different parts of the encoding have a substantial amount of common substructure.
- Boolean circuits offer an attractive formalism for representing the required Boolean functions where compactness is enhanced by sharing common substructure.
- *Note*: here we talk about the computational model of Boolean circuits, which is different from (although related to) actual real-life digital circuits that for example form the basis of today's computing technologies.

Boolean Circuits

- A *Boolean circuit* C is a triple (V, E, s).
- Here, (V, E) is a finite directed acyclic graph whose nodes are called *gates*. The nodes are divided into three categories.
- *Note*: for every node $v \in V$, one calls the number of incoming (outgoing) edges the *indegree* (*outdegree*) of v.
- Similarly, the set of nodes $\{u \in V \mid (u, v) \in E\}$ is the set of *incoming neighbors*; the set of *outgoing neighbors* is defined analogously.
- *Output gates* can have any indegree but have outdegree 0.
- *Intermediate gates* can have any indegree and outdegree larger than 0.
- *Input gates* can have any outdegree but have indegree 0.

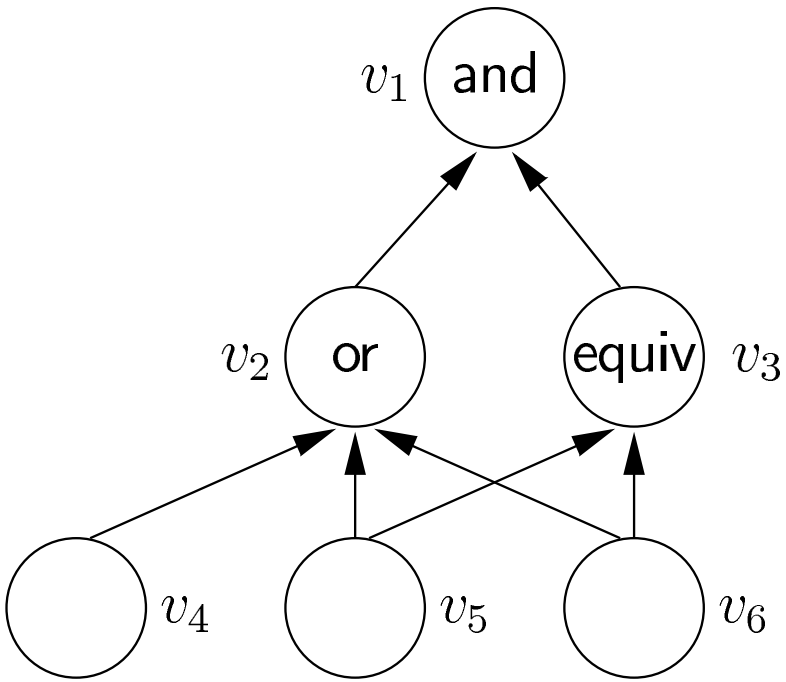
Boolean Circuits—cont'd

- The function $s : V \mapsto \{\textbf{true}, \textbf{false}\}^{\{\textbf{true},\textbf{false}\}^{|V|}}$ assigns a Boolean function $s(g)$ to each intermediate and output gate g of appropriate arity corresponding to the indegree of the gate.
- Typical Boolean functions used in the gates are: and/n (n-input AND function), $or/n, not/1, equiv/2, xor/2, \ldots$

For example:

x_1	x_2	$equiv/2$	$xor/2$
0	0	1	0
0	1	0	1
1	0	0	1
1	1	1	0

Example: Boolean Circuit



$s(v_1) = and/2$
$s(v_2) = or/3$
$s(v_3) = equiv/2$

v_1 is the output gate of the circuit
v_4, v_5, v_6 are the input gates

Boolean Circuits—Semantics

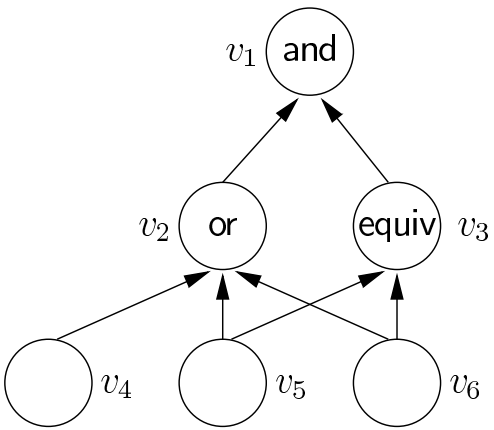
- Consider first the graph (V, E). We can order its nodes (C's gates) *topologically*, such that for every edge from u to v node u appears before v in the ordering; this is always possible because the graph (circuit) is acyclic.
- *Note*: in this sequence all gates appear after their input gates.
- For a given circuit $C = (V, E, s)$ a *truth assignment* $T : X(C) \longrightarrow \{\textbf{true}, \textbf{false}\}$ assigns a truth value to each gate in $X(C)$, where $X(C)$ is the set of input gates of C.
- The truth value $T'(g)$ for gate v is then determined inductively:

$$T'(g) = \begin{cases} T(g), & \text{if } g \in X(C), \text{ else} \\ f(T'(g_1), \ldots, T'(g_n)) & \text{where the } g_i \text{ are the incoming neighbors of } g \\ & \text{and } f = s(g) \text{ is the function associated with } g. \end{cases}$$

Boolean Circuits—Semantics cont'd

Example

For the previous example circuit C, $X(C) = \{v_4, v_5, v_6\}$, and a valid topological ordering would be the sequence $v_4, v_5, v_6, v_3, v_2, v_1$. Consider a truth assignment $T(v_4) = T(v_5) = T(v_6) =$ **false** and the values it induces:

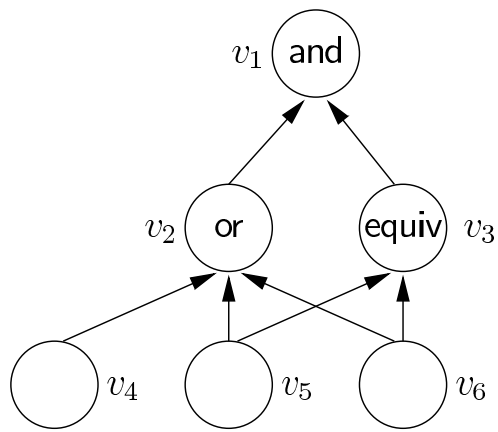


- $T(v_3) = equiv(T(v_5), T(v_6)) = equiv(\textbf{false}, \textbf{false}) = \textbf{true}$,
- $T(v_2) = or(T(v_4), T(v_5), T(v_6)) = \textbf{false}$, and
- $T(v_1) = and(T(v_2), T(v_3)) = \textbf{false}$.

Circuit Satisfiability Problem

- An interesting computational (search) problem related to circuits is the *circuit satisfiability* problem.
- A *constrained Boolean circuit* is a pair (C, α) with a circuit C and *constraints* α assigning truth values for some gates.
- Given a constrained Boolean circuit (C, α) a truth assignment T *satisfies* (C, α) if it satisfies the constraints α, i.e., for each gate g for which α gives a truth value, $\alpha(g) = T(g)$ holds.
- *CIRCUIT SAT* problem: Given a constrained Boolean circuit find a truth assignment T that satisfies it.

Example. Consider the circuit with constraints $\alpha(v_4) =$ **false**, $\alpha(v_1) =$ **true**.
This circuit has a satisfying truth assignment $T(v_4) =$ **false**, $T(v_5) = T(v_6) =$ **true**.
If the constraints are $\alpha(v_2) =$ **false**, $\alpha(v_1) =$ **true**, the circuit is unsatisfiable.



Example—cont'd

- There are solvers available for solving constrained circuit satisfiability problems.
- One example: bczchaff (Junttila and Niemelä, 2000) based on the SAT solver zchaff (Zhang et al.).

Input:

```
BC1.0
v_4;
v_5;
v_6;
v_2 := OR(v_4, v_5, v_6);
v_3 := EQUIV(v_5, v_6);
v_1 := AND(v_2, v_3);
ASSIGN v_1, ~v_4;
```

Output:

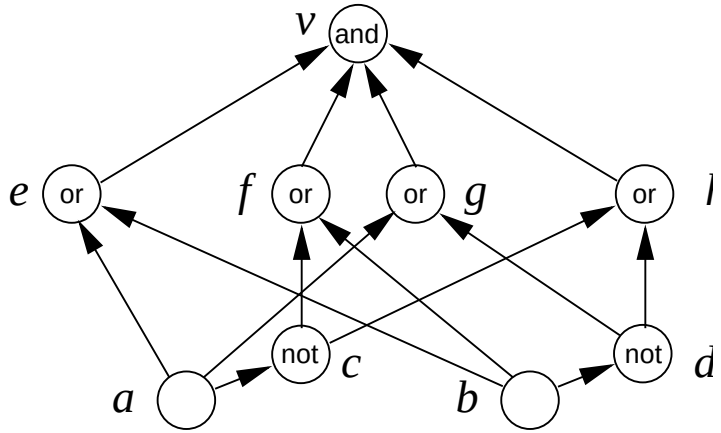
```
./bczchaff test.txt
v_5 v_6 v_2 v_3 v_1 ~v_4
Satisfiable
```

- `http://users.ics.aalto.fi/tjunttil/bcsat/` (bczchaff), `http://www.princeton.edu/~chaff/zchaff.html` (zchaff).

Boolean Circuits vs. Propositional Formulas

- For each propositional formula ϕ, there is a corresponding Boolean circuit C_ϕ such that for any T appropriate for both, $T(g_\phi) = \textbf{true}$ iff $T \models \phi$ for an output gate g_ϕ of C_ϕ .
  Idea: just introduce a new gate for each subexpression.

$(a \vee b) \wedge (\neg a \vee b) \wedge$
$(a \vee \neg b) \wedge (\neg a \vee \neg b)$



- For each Boolean circuit C, there is a corresponding formula ϕ_C.
- Notice that Boolean circuits allow shared subexpressions but formulas do not.
  For instance, in the circuit above gates a, b, c, d.

Circuits Compute Boolean Functions

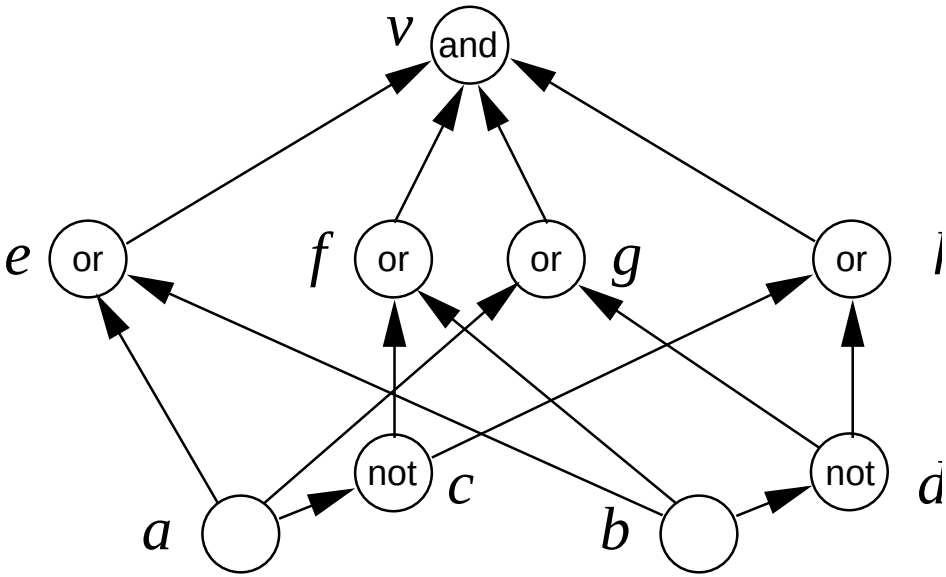
- A Boolean circuit with output gate g and input gates (corresponding to variables) $x_1, \ldots, x_n$ *computes* an n-ary Boolean function f if for any n-tuple of truth values $\mathbf{t} = (t_1, \ldots, t_n)$, $f(\mathbf{t}) = T(g)$ where $T(x_i) = t_i$, $i = 1, \ldots, n$.
- Any n-ary Boolean function f can be computed by a Boolean circuit involving variables $x_1, \ldots, x_n$.
- Not every Boolean function can be computed using a concise circuit.

Theorem

For any $n \geq 2$ there is an n-ary Boolean function f such that no Boolean circuit with $\frac{2^n}{2n}$ or fewer gates can compute it.

Boolean Circuits as Equation Systems

A Boolean circuit can be written as a system of equations.



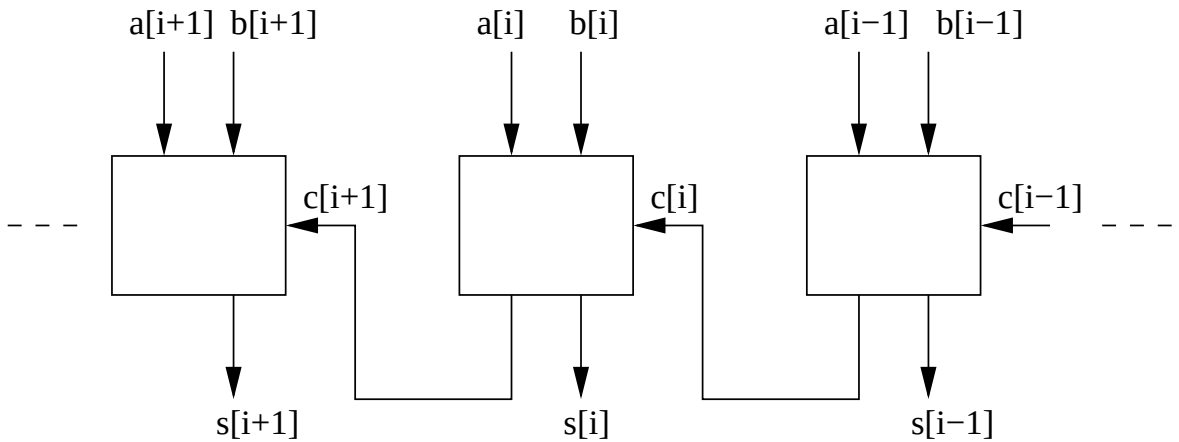
$$v = \text{and}(e, f, g, h)$$
$$e = \text{or}(a, b)$$
$$f = \text{or}(b, c)$$
$$g = \text{or}(a, d)$$
$$h = \text{or}(c, d)$$
$$c = \text{not}(a)$$
$$d = \text{not}(b)$$

Boolean Modelling

- Propositional formulas/Boolean circuits offer a natural way of modelling many interesting Boolean functions.
- Example. IF-THEN-ELSE $\text{ite}(a, b, c)$ (if a then b else c.).
  As a formula:
  $\text{ite}(a, b, c) \equiv (a \wedge b) \vee (\neg a \wedge c)$
  As a circuit:
  $ite = \text{or}(i_1, i_2)$
  $i_1 = \text{and}(a, b)$
  $i_2 = \text{and}(a_1, c)$
  $a_1 = \text{not}(a)$
- Given gates a, b, c, $\text{ite}(a, b, c)$ can be thought as a shorthand for a subcircuit given above.

Example

Binary adder. Given input bits $a[i]$, $b[i]$ and $c[i]$ compute output bits $s[i]$ and $c[i+1]$ (c: carry bit).



As a formula:

$$s[i] \equiv ((a[i] \oplus b[i]) \oplus c[i])$$
$$c[i+1] \equiv (a[i] \wedge b[i]) \vee (c[i] \wedge (a[i] \oplus b[i]))$$

As a circuit:

$$s[i] = \text{xor}(x, c[i])$$
$$c[i+1] = \text{or}(l, r)$$
$$l = \text{and}(a[i], b[i])$$
$$r = \text{and}(c[i], x)$$
$$x = \text{xor}(a[i], b[i])$$

Encoding Problems Using Circuits

- Circuits can be used to encode problems in a structured way.
- Example. Given three bits a, b, c find their values such that if at least two of them are ones then either a or b is one else a or c is one (note: first *or* is exclusive, second *or* is inclusive) .
- We use IF-THEN-ELSE and adder circuits to encode this as a CIRCUIT SAT problem (replacing $a[i] \leftarrow a$, $b[i] \leftarrow b$, $c[i] \leftarrow c$):

  $p = \text{ite}(c[i+1], x, p_1)$

  $p_1 = \text{or}(a, c)$

  `% full adder; sum output gate omitted`

  $c[i+1] = \text{or}(l, r)$

  $l = \text{and}(a, b)$

  $r = \text{and}(c, x)$

  $x = \text{xor}(a, b)$
- Now each satisfying truth assignment for the circuit with constraint $\alpha(p) = \textbf{true}$ gives a solution to the problem.

Example: Reachability

Given a graph $G = (V = \{1, \ldots, n\}, E)$, construct a circuit $R(G)$ such that $R(G)$ is satisfiable iff there is a (simple) path from 1 to n in G.

- Basic idea very similar to *dynamic programming* solution: introduce Boolean variables for triples of nodes (i, j, k) that indicate whether node k is larger or equal to any node on a path from i to j (disregarding endpoints of the paths).
- The gates of $R(G)$ are of the form
  g_{ijk} with $1 \leq i, j \leq n$ and $0 \leq k \leq n$
  h_{ijk} with $1 \leq i, j, k \leq n$
- g_{ijk} is **true**: there is a path in G from i to j where the largest intermediate node is k or smaller.
- h_{ijk} is **true**: there is a path in G from i to j for which k is the largest intermediate node.

Example—cont'd

$R(G)$ is the following circuit:

- For $k = 0$, g_{ijk} is an input gate.
- For $k = 1, 2, \ldots, n$:
  $h_{ijk} = \text{and}(g_{ik(k-1)}, g_{kj(k-1)})$
  $g_{ijk} = \text{or}(g_{ij(k-1)}, h_{ijk})$
- g_{1nn} is the output gate of $R(G)$.
- Constraints α:
  For the output gate: $\alpha(g_{1nn}) =$ **true**
  For the input gates: $\alpha(g_{ij0}) =$ **true** if $i = j$ or (i, j) is an edge in G
  else $\alpha(g_{ij0}) =$ **false**.

Example—cont'd

- Because of the constraints α on input gates there is at most one possible truth assignment T.
- It can be shown by induction on $k = 0, 1, \ldots, n$ that in this assignment the truth values of the gates correspond to their given intuitive readings.
- From this follows:
  $R(G)$ is satisfiable iff $T(g_{1nn}) =$ **true** in the truth assignment iff there is a (simple) path from 1 to n in G without any intermediate nodes bigger than n iff there is a path from 1 to n in G.

From Circuits to CNF

- Translating Boolean Circuits to an equivalent CNF formula can lead to exponential blow-up in the size of the formula.
- Often exact equivalence is not necessary but auxiliary variables can be used as long as at least satisfiability is preserved.
- Then a linear size CNF representation can be obtained, e.g., using the so-called *Tseitin's translation*.

From Circuits to CNF—cont'd

- Given a Boolean circuit C the corresponding CNF formula is obtained as follows.
- For each gate of the circuit a new variable is introduced.
- Clauses are formed by the gate equations (taken as an equivalences) written in clausal form for each intermediate and output gate.
- For each constraint $\alpha(g) = t$, the corresponding literal for g is added.
- This transformation preserves satisfiability and even truth assignments in the following sense:
  If C is a Boolean circuit and Σ its Tseitin translation, then for every truth assignment T of C satisfying α, there is a satisfying truth assignment T' of Σ which agrees with T and vice versa.

Example—cont'd

Example 1.

- Assume we are given a simple circuit with input gates v_3, v_4, v_5, intermediate gate v_2 and output gate v_1.
- Further assume the circuit is unconstrained and contains the following gates: $v_1 = \text{and}(v_2, v_5)$ and $v_2 = \text{or}(v_3, v_4)$
- We obtain for the first gate the clauses

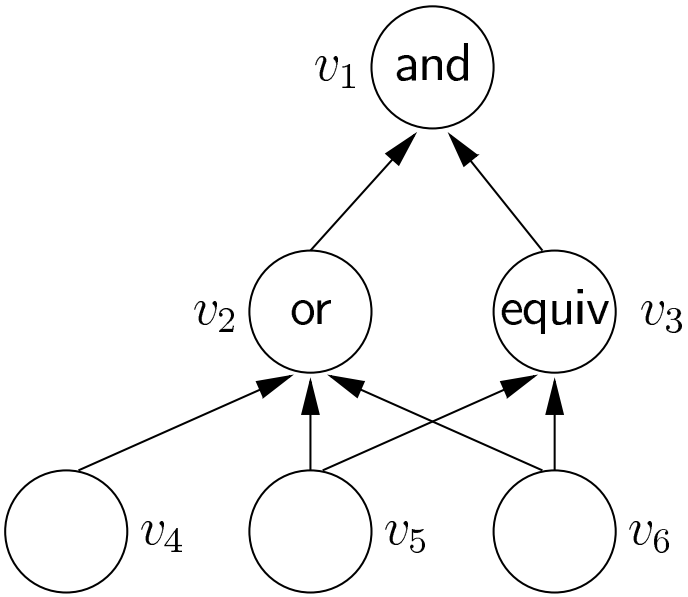
$$\begin{aligned} v_1 \leftrightarrow (v_2 \wedge v_5) &\equiv (v_1 \rightarrow (v_2 \wedge v_5)) \wedge ((v_2 \wedge v_5) \rightarrow v_1) \\ &\equiv (\neg v_1 \vee (v_2 \wedge v_5)) \wedge (\neg v_2 \vee \neg v_5 \vee v_1) \\ &\equiv (\neg v_1 \vee v_2) \wedge (\neg v_1 \vee v_5) \wedge (\neg v_2 \vee \neg v_5 \vee v_1). \end{aligned}$$

- For the second gate we then obtain the clauses

$$\begin{aligned} v_2 \leftrightarrow (v_3 \vee v_4) &\equiv (v_2 \rightarrow (v_3 \vee v_4)) \wedge ((v_3 \vee v_4) \rightarrow v_2) \\ &\equiv (\neg v_2 \vee v_3 \vee v_4) \wedge ((\neg v_3 \wedge \neg v_4) \vee v_2) \\ &\equiv (\neg v_2 \vee v_3 \vee v_4) \wedge (\neg v_3 \vee v_2) \wedge (\neg v_4 \vee v_2). \end{aligned}$$

Example—cont'd

Example 2.



Consider the circuit with constraints $\alpha(v_1) = \textbf{true}, \alpha(v_4) = \textbf{false}$.
Gate equations (taken as equivalences) for non-input gates:

$v_1 \leftrightarrow (v_2 \wedge v_3)$

$v_2 \leftrightarrow (v_4 \vee v_5 \vee v_6)$

$v_3 \leftrightarrow (v_5 \leftrightarrow v_6)$

The resulting CNF for the translation:

$$(\neg v_1 \vee v_2) \wedge (\neg v_1 \vee v_3) \wedge (v_1 \vee \neg v_2 \vee \neg v_3) \wedge$$
$$(v_2 \vee \neg v_4) \wedge (v_2 \vee \neg v_5) \wedge (v_2 \vee \neg v_6) \wedge (\neg v_2 \vee v_4 \vee v_5 \vee v_6) \wedge$$
$$(v_3 \vee v_5 \vee v_6) \wedge (v_3 \vee \neg v_5 \vee \neg v_6) \wedge (\neg v_3 \vee v_5 \vee \neg v_6) \wedge (\neg v_3 \vee \neg v_5 \vee v_6) \wedge$$
$$v_1 \wedge \neg v_4 \text{ [for constraints]}$$

Outlook

- There is plenty of literature on the topic of *circuit complexity*, which is a subfield of computational complexity theory.
- Circuit complexity theory involves the study and classification of Boolean functions according to their computability with Boolean circuits of fixed size or depth (length of longest path from input to output).
- Next week we discuss local and complete methods for solving satisfiability problems.
- Some of these can be considered special cases of algorithms for constraint satisfaction problems.

Lecture 7: Complete and local search methods for SAT

Outline

- Algorithms for solving Boolean satisfiability problems
- Complete algorithms and local search methods

Goal for today: Understand how constraint propagation techniques give rise to efficient complete methods for Boolean satisfiability. Learn how to apply local search methods to SAT.

SAT

- Recall: SAT (Boolean Satisfiability Problem)
  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 satisfying the most clauses in the formula.
- SAT(O) also called MAX-SAT.
- In this lecture we first discuss complete methods for SAT(S), then outline some local search algorithms.
- Note: an instance of SAT is essentially a CSP with Boolean variables and clauses as constraints.

Recall: equivalence of CSP's

Recall: CSPs $\mathbf{P}_1$ and $\mathbf{P}_2$ are equivalent w.r.t. a set of variables X iff

- for every solution T_1 of P_1 there is a solution T_2 of P_2 such that T_1 and T_2 agree on variables X and
- for every solution T_2 of P_2 there is a solution T_1 of P_1 such that T_2 and T_1 agree on variables X.

Complete search methods make use of Boolean constraint propagation techniques that transform a given SAT instance (set of clauses) into (a simpler) one that is equivalent to the original.

Recall: the basic framework for solving CSP's

```
procedure Solve(CSP/COP P):
P ← Preprocess(P);
P ← Constraint_Propagation(P);
if not Happy(P) then
   if Atomic(P) then
      return;     /* impossible to split */
   else
      (P_1, P_2, ...) ← Split(P);
      Proceed_by_Cases(P_1, P_2, ...);     /* may lead to recursive calls */
   end
end
```

Solving Boolean Constraints

Applying the general method to SAT:

- Preprocess: remove satisfied clauses, remove variables that only appear as the same literal in all clauses (together with their clauses).
  Example.
  $C_1 = \{\neg x_1 \vee \neg x_2\},\ C_2 = \{\neg x_1 \vee x_2 \vee \neg x_3\},\ C_3 = \{\neg x_2 \vee x_3\}$
  gets processed as follows: $\{C_1, C_2, C_3\} \to \{C_3\} \to \emptyset$.

- Happy: a satisfying truth assignment has been found (SAT) or it was determined that no truth assignment can satisfy more clauses than a previously found one (MAX-SAT).

- Atomic: No more undecided variables to choose from, or empty clause was found.

Propagation for Boolean Constraints

A basic reduction step is the so-called *unit clause* rule:

$$\frac{S \cup \{l\}}{S'},$$

where S is a set of clauses, l is a unit clause (a literal) and S' is obtained from S by removing

(i) every clause that contains l and

(ii) the complement of l from every remaining clause.

(The complement of a literal: $\overline{v} = \neg v$ and $\overline{\neg v} = v$.)

Intuition: a satisfying assignment must satisfy all clauses, hence also the unit clause, which can only be satisfied by l evaluating to true.

Example.

$\{\neg v_1, v_1 \vee \neg v_2, v_2 \vee v_3, \neg v_3 \vee v_1, \neg v_1 \vee v_4\} \rightsquigarrow \{\neg v_2, v_2 \vee v_3, \neg v_3\}$

Propagation for Boolean Constraints—cont'd

Unit propagation (UP) (aka Boolean Constraint Propagation (BCP)):

Apply the unit clause rule until

- a conflict (empty clause; denoted by $\bot$) is obtained
- or no new unit clauses are available.

Compare: *consistency condition for constraint propagation*

Example.

$\{\neg v_2, v_2 \vee v_3, \neg v_3\} \rightsquigarrow \{v_3, \neg v_3\} \rightsquigarrow \{\bot\}$ (conflict)

Using an efficient variable$\rightarrow$clauses lookup table, one can implement unit propagation in linear time (in the total number of literals in the set of clauses).

Boolean Constraints—cont'd

- Split:
  Apply the enumeration rule:

$$\frac{x \in \{0,1\}}{x \in \{0\} \mid x \in \{1\}}$$

  There exists a wide variety of heuristics for choosing the split variables, some popular ones:
  random choice, choice based on the number of unsatisfied clauses it appears in, based on their size, based on the occurrence of positive and negative literals, etc.
- Proceed by cases: backtrack with unit propagation
- This gives the *DPLL-algorithm*
  (Davis-Putnam-Loveland-Logemann, 1962) which is the basis of most of the state-of-the-art complete SAT solvers.

Input: S: a set of clauses; M: a set of literals
Output: If there is an assignment satisfying the clauses S, then a set of literals describing such an assignment is returned otherwise 'UNSAT' is returned.

$\mathrm{DPLL}(S, M)$
$\langle S', M' \rangle := \mathrm{simplify}(S, M)$;
if $S' = \emptyset$ **then** return M'
else if $\bot \in S'$ **then** return 'UNSAT'
else
  $L := \mathrm{choose}(S', M')$;
  $M'' := \mathrm{DPLL}(S' \cup \{L\}, M' \cup \{L\})$;
  if $M'' =$ 'UNSAT' **then** return $\mathrm{DPLL}(S' \cup \{\overline{L}\}, M' \cup \{\overline{L}\})$
  else return M''
  end if
end if

Initial call: $\mathrm{DPLL}(S, \{\})$

Basic DPLL

DPLL uses two subprocedures:

- The call to $\mathrm{simplify}(S, M)$ returns $\langle S', M' \rangle$ where S' is the set of clauses obtained by applying unit propagation to S and M' is M extended with unit literals found in the process.
- $\mathrm{choose}(S', M')$ implements the search heuristics, i.e., decides for which variable the splitting rule is applied and which of the branches is considered first.
- The performance of the procedure depends crucially on the constraint propagation techniques and search heuristics.

Example

We use DPLL to decide whether a set of clauses $S = \{c1, \ldots, c8\}$, where

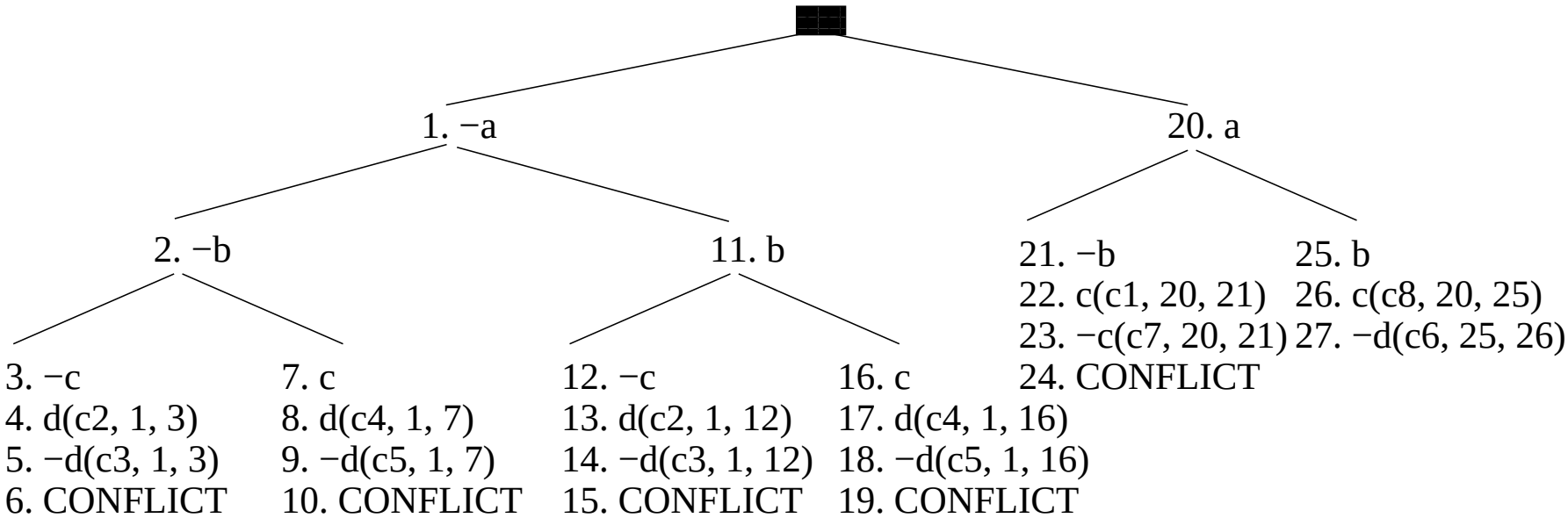
$$
\begin{array}{llll}
c1: & \neg a \vee b \vee c & c5: & a \vee \neg c \vee \neg d \\
c2: & a \vee c \vee d & c6: & \neg b \vee \neg c \vee \neg d \\
c3: & a \vee c \vee \neg d & c7: & \neg a \vee b \vee \neg c \\
c4: & a \vee \neg c \vee d & c8: & \neg a \vee \neg b \vee c
\end{array}
$$

is satisfiable. We illustrate the behavior of DPLL by giving a possible DPLL search tree for S.

A DPLL search tree can be taken as a tree where nodes are literals obtained by unit propagation or by the splitting rule. For literals derived by unit propagation we give the clause and the literals earlier in the branch of the search tree by which it is obtained and mark with "CONFLICT" the branches with a conflict. (*NB: the picture assumes that no Preprocess step as on slide 182 is performed, only unit propagation.*)

Example—cont'd. A DPPL search tree for the set of clauses S:

$$
\begin{array}{llll}
c1: & \neg a \vee b \vee c & c5: & a \vee \neg c \vee \neg d \\
c2: & a \vee c \vee d & c6: & \neg b \vee \neg c \vee \neg d \\
c3: & a \vee c \vee \neg d & c7: & \neg a \vee b \vee \neg c \\
c4: & a \vee \neg c \vee d & c8: & \neg a \vee \neg b \vee c
\end{array}
$$



For example: Node 1 is obtained by the splitting rule and node 4 by unit propagation.

When detecting a conflict (6.) DPLL backtracks to the next untried alternative (7. c).

The set S is satisfiable as the last branch (ending with 27.) does not have a conflict and S is satisfied in a truth assignment with literals $a, b, c, \neg d$ true.

Local search for SAT

- Very large SAT instances (or particularly “difficult” ones) are out of reach of complete methods, which justifies the investigation of local search methods.
- However, standard local search methods have the fundamental limitation of only being applicable for finding satisfying truth assignments for satisfiable instances.
- For this task they still outperform complete search methods, particularly for very large instances and also those that are generated randomly.
- Recently, also local search methods employing techniques from complete algorithms (e.g., clause learning) have been developed and some work on local search for unsatisfiability has been published (see, e.g., Audemard and Simon, 2007).

GSAT (Selman et al. 1992)

- The algorithm is essentially the *steepest descent* variant of the simple local search method (see slides 77, 78) applied to SAT.
- *Idea:* Candidate solutions are truth assignments t; the algorithm aims to minimize $c(t)$, which is defined as the number of unsatisfied clauses under the truth assignment t.
- The set of neighboring solutions of t are the truth assignments that differ from t in one variable ("one variable flipped").
- Further extensions include: restart rules, tabu list, etc.

GSAT (Selman et al. 1992)—cont'd

Input: propositional formula F in CNF

function GSAT(F):
    $t \leftarrow$ initial truth assignment;
    while flips $<$ max_flips **do**
        if t satisfies F **then return** t
        else
            find a variable x whose flipping in t causes
                largest decrease in $c(t)$ (if no decrease is
                possible, then smallest increase);
            $t \leftarrow$ (t with variable x flipped)
    end while;
    return t.

(Note that the algorithm requires some rule for breaking ties in the case that multiple variables qualify for being flipped. One option would be to pick any of these at random with equal probability.)

NoisyGSAT (Selman et al. $\sim$ 1996)

Idea: Augment GSAT by a fraction p of random walk moves.

Input: propositional formula F in CNF, parameter p

function NoisyGSAT(F,p):
    $t \leftarrow$ initial truth assignment;
    while flips $<$ max_flips **do**
        if t satisfies F **then return** t
        else
            with probability p, pick any variable x
                uniformly at random;
            with probability $(1-p)$, do basic GSAT move:
                find a variable x whose flipping causes
                largest decrease in $c(t)$ (if no decrease is
                possible, then smallest increase);
            $t \leftarrow (t$ with variable x flipped)
    end while;
    return t.

WalkSAT (Selman et al. 1996)

Idea: modified NoisyGSAT that *focuses* on unsatisfied clauses.

Input: propositional formula F in CNF, parameter p

function WalkSAT(F,p):
    $t \leftarrow$ initial truth assignment;
    while flips $<$ max_flips **do**
        if t satisfies F **then return** t **else**
            choose a random unsatisfied clause C in F;
            if some variables in C can be flipped without
                breaking any presently satisfied clauses,
                then pick one such variable x at random; else:
            with probability p, pick a variable x in C unif. at random;
            with probability $(1-p)$, do basic GSAT move:
                find a variable x in C whose flipping causes
                largest decrease in $c(t)$;
            $t \leftarrow (t$ with variable x flipped)
    end while;
    return t.

WalkSAT vs. NoisyGSAT

- The focusing seems to be important: in the (unsystematic) experiments in Selman et al. (1996), WalkSAT outperforms NoisyGSAT by several orders of magnitude. Later experimental evidence by other authors supports this.

- Good values for the "noise" parameter p seem to be about $p \approx 0.5$. For instance, for large randomly generated 3-SAT formulas with clauses-to-variables ratio α near the "satisfiability threshold" $\alpha = 4.267$, the optimal value of p seems to be about $p = 0.57$.

- In experiments by Seitz, Alava & Orponen (2005), a focused variant of a different local search method is competitive with WalkSAT on large randomly generated 3-SAT instances. What about other focused local search algorithms (e.g. focused tabu search)?

Adaptive local search for SAT

- Local search methods have difficulties with structured problem instances.
- For good performance parameter tuning is essential.
  (For example in WalkSAT: the noise parameter p and the max_flips parameter.)
- Finding good parameter values is a non-trivial problem which typically requires substantial experimentation and experience.
- One minor extension that is usually performed: introduce restarts that let the solver run multiple times (typically from different, randomly generated initial solutions).
- Algorithms (e.g., WalkSAT) can be also made greedier using a history-based variable selection mechanism, biasing the selection of variables to flip on those that have been flipped least-recently (least recently=furthest in the past).

Novelty (McAllester et al. 1997)

After choosing an unsatisfiable clause as done in WalkSAT, the variable to be flipped is selected from the variables in this clause as follows:

- Sort variables according to decrease in the number of unsatisfied clauses, breaking ties by placing least-recently flipped ones before others with the same number.
- If the first variable in that order is not the one most (!) recently flipped, it is always selected.
- Else it is only selected with probability $1 - p$, where p is a parameter called *noise setting*.
- Otherwise the variable on the second position is selected.

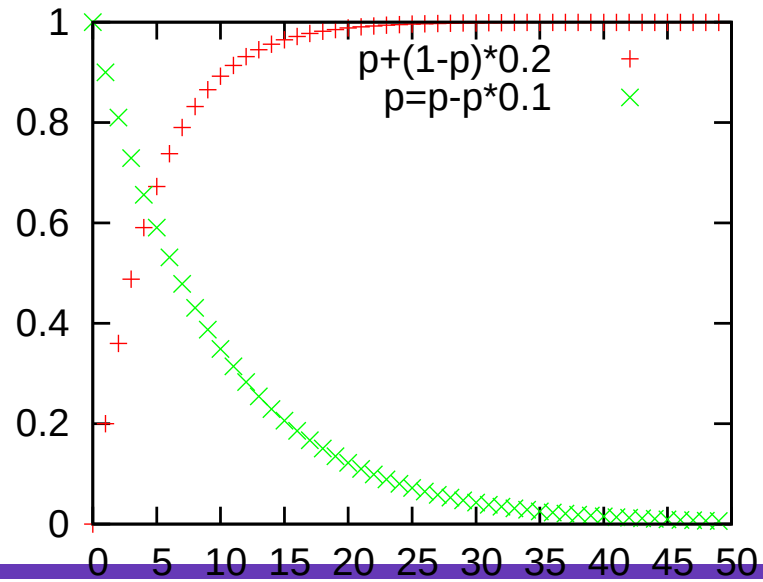
Adaptive WalkSat and Adaptive Novelty+

- In *Novelty+* (Hoos 1998) a random walk step (with probability *wp*) is added: with probability $1 - wp$ the variable to be flipped is selected according to the Novelty mechanism and in the other case it is randomly selected from the chosen unsatisfied clause.
- A suitable value for the noise parameter p (and wp for Novelty+) is crucial for competitive performance of WalkSAT and its variants.
- Too low noise settings lead to stagnation behavior and too high settings to long running times.
- Instead of a static setting, a dynamically changing noise parameter can be computed by the following method. (see Hoos, 2002)

Adaptive WalkSat and Adaptive Novelty+

Two parameters θ and $0 \leq \phi \leq 1$ are given

- At the beginning the search is maximally greedy ($p = 0$).
- There is a search stagnation if no improvement in the evaluation function value has been observed over the last $m\theta$ search steps where m is the number of clauses in the instance.
- In this situation the noise value is increased by $p := p + (1 - p)\phi$.
- If there is an improvement in the evaluation function value, then the noise value is decreased by $p := p - p\phi/2$.



Adaptive WalkSat and Adaptive Novelty+

- Notice the asymmetry between increases and decreases in the noise setting.
- Stagnation is more difficult to detect compared to improvement; also, there is empirical evidence that approaching the "optimal" noise setting from above yields better performance.
- When this mechanism of adapting the noise level is applied to WalkSat and Novelty+, we obtain *Adaptive WalkSat and Adaptive Novelty+* (Hoos, 2002).
- The performance of the adaptive versions is more robust w.r.t. the settings of θ and ϕ than the performance of the non-adaptive versions w.r.t. to the settings of p.
- For example, for Adaptive Novelty+ setting $\theta = 1/6$ and $\phi = 0.2$ seem to lead to robust overall performance (while there appears to be no such setting for p in the non-adaptive case).

Tools for SAT

- The development of SAT solvers is strongly driven by *SAT competitions* (`http://www.satcompetition.org/`)
- There is a large number of solvers available in the public domain.
- Solvers that ranked well in previous SAT competitions:
  SAT 2005:
  `SatELiteGTI, MiniSAT 1.13, zChaff_rand, HaifaSAT`
  SAT COMPETITION 2009:
  `PrecoSAT, SATzilla, glucose, clasp, TNM, March_hi, ...`
  SAT Competition 2011: `(p)lingeling, ppfolio, glucose, clasp, ...`
  SAT Competition 2013:
  `(p)lingeling aqw, glucose 2.3, Riss3g cert, BreakIDGlucose 1, probSAT SC13, CSHC, ...`

Lecture 8: Modern SAT solvers

Outline

- Conflict-driven clause learning solver
- Lazy data structures
- Restarts

Goal for today: Understand some of the main techniques and data structures used in a modern conflict-driven clause learning SAT solver.

Key contributions to CDCL solvers

- *Conflict clauses*; Grasp (Marques-Silva & Sakallah, 1996).
- *Restart strategies* (Gomes et.al 1997, Luby et al. 1993)
- *2-watch pointers and VSIDS*; zChaff (Moskewicz et al. 2001)
- Efficient (open source) implementation; Minisat (Een & Sörensson, 2003)
- Phase-saving; Rsat (Pipatsrisawat & Darwiche, 2007)
- Conclict-clause minimization (Sörensson & Biere, 2009)
- . . . combined with pre- and in-processing techniques

Overall conflict-driven clause learning algorithm

procedure CDCL(CNF F):
$F := \text{simplify}(F)$
while `true` **do**
  $l_d := \text{getDecisionLiteral}()$
  if no l_d exists **then return** `SAT` /* All variables assigned */
  $F := \text{simplify}(F(l_d \leftarrow 1))$
  while F contains $C_{falsified}$ **do**
    $C_{conflict} \leftarrow \text{analyzeConflict}(C_{falsified})$
    if $C_{conflict} = \emptyset$ **then return** `UNSAT`
    $\text{backtrack}(C_{conflict})$
    $F := \text{simplify}(F \cup \{C_{conflict}\})$
  endwhile
endwhile

Conflict-driven clause learning

- Basic idea: maintain *implication graph* that contains variable-value assignments and edges between them if one was implied by the other in the propagation.
- Whenever a conflict occurs, one adds a clause that corresponds to the variable-value assignments that caused the conflict.
- Same procedure allows *non-chronological backtracking*, since the implication graph also keeps track of the level in the search tree at which the assignment values were decided.
- This type of backtracking essentially prunes the search space, similarly to the bounding heuristic for branch&bound search.

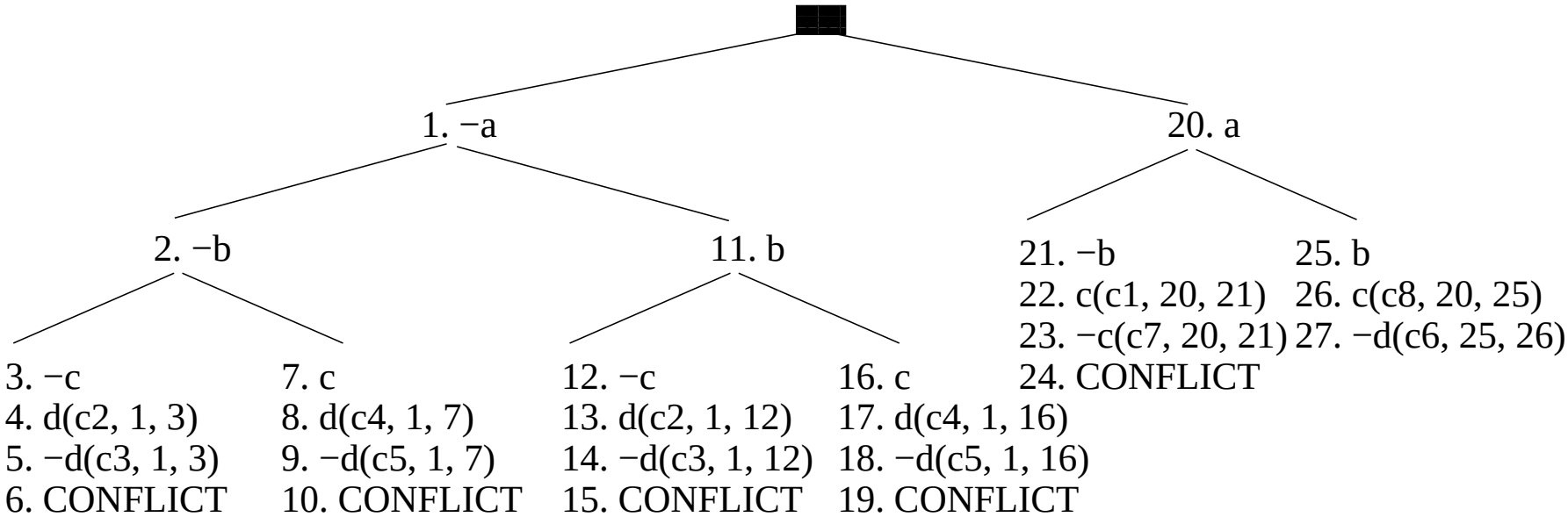
Recall the DPPL search tree for the set of clauses S from last lecture:

$c1: \neg a \vee b \vee c$ $\quad$ $c5: a \vee \neg c \vee \neg d$

$c2: a \vee c \vee d$ $\quad$ $c6: \neg b \vee \neg c \vee \neg d$

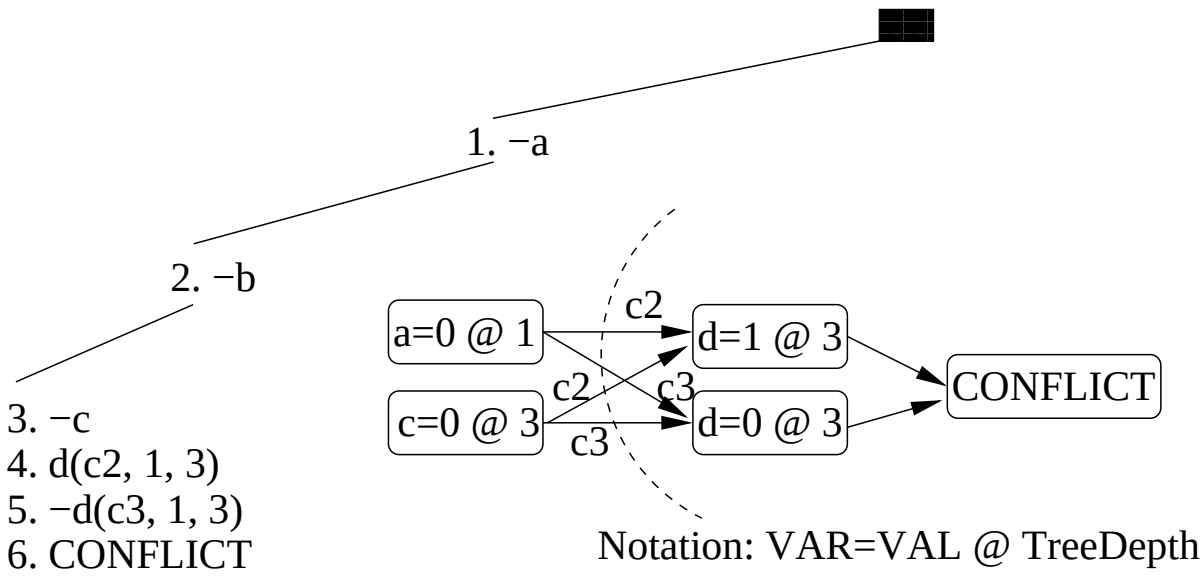
$c3: a \vee c \vee \neg d$ $\quad$ $c7: \neg a \vee b \vee \neg c$

$c4: a \vee \neg c \vee d$ $\quad$ $c8: \neg a \vee \neg b \vee c$



Conflict driven clause learning

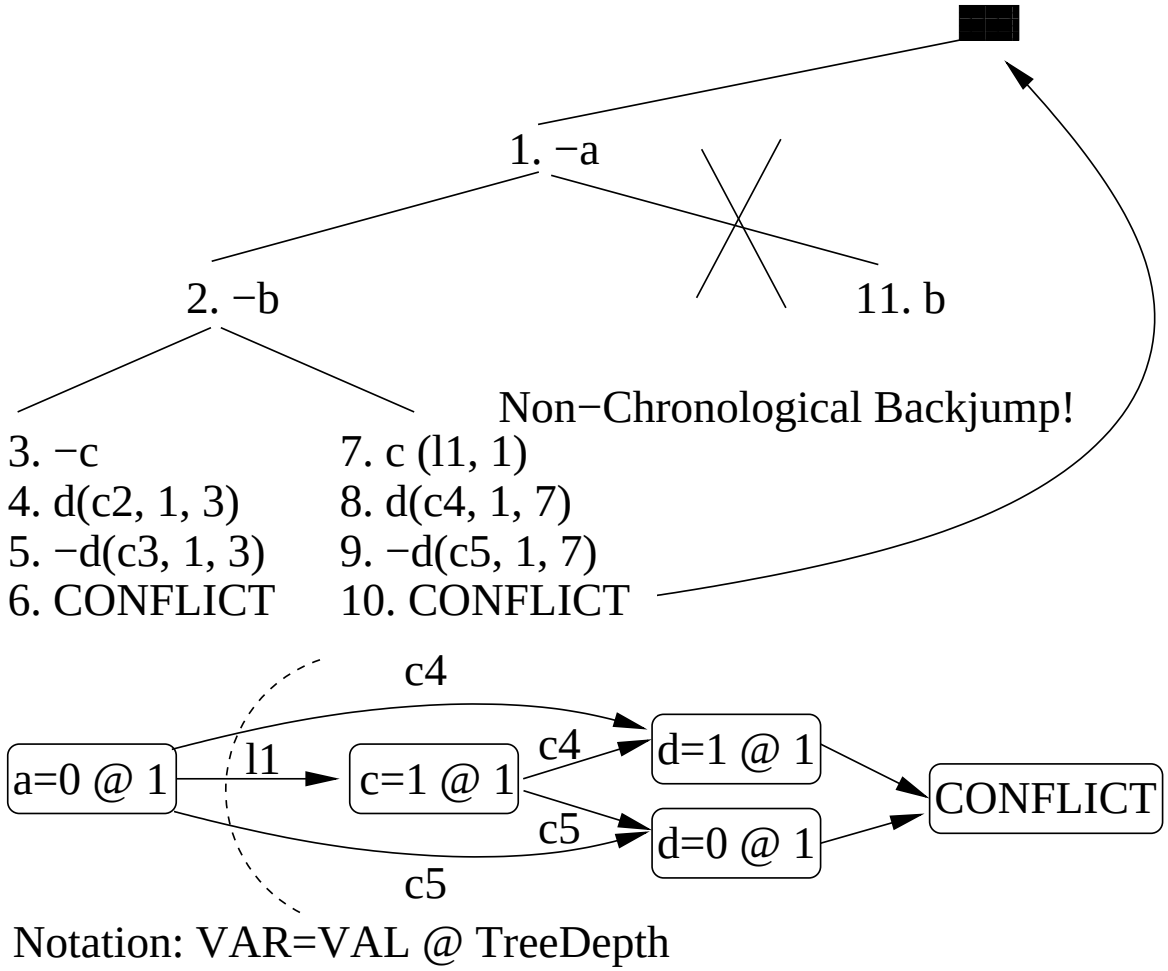
Consider the first conflict of the search tree:



Learned first conflict clause: $l1: \quad \neg(\neg a \wedge \neg c) \equiv a \vee c$

Conflict driven clause learning—cont'd

. . . and now the second conflict:



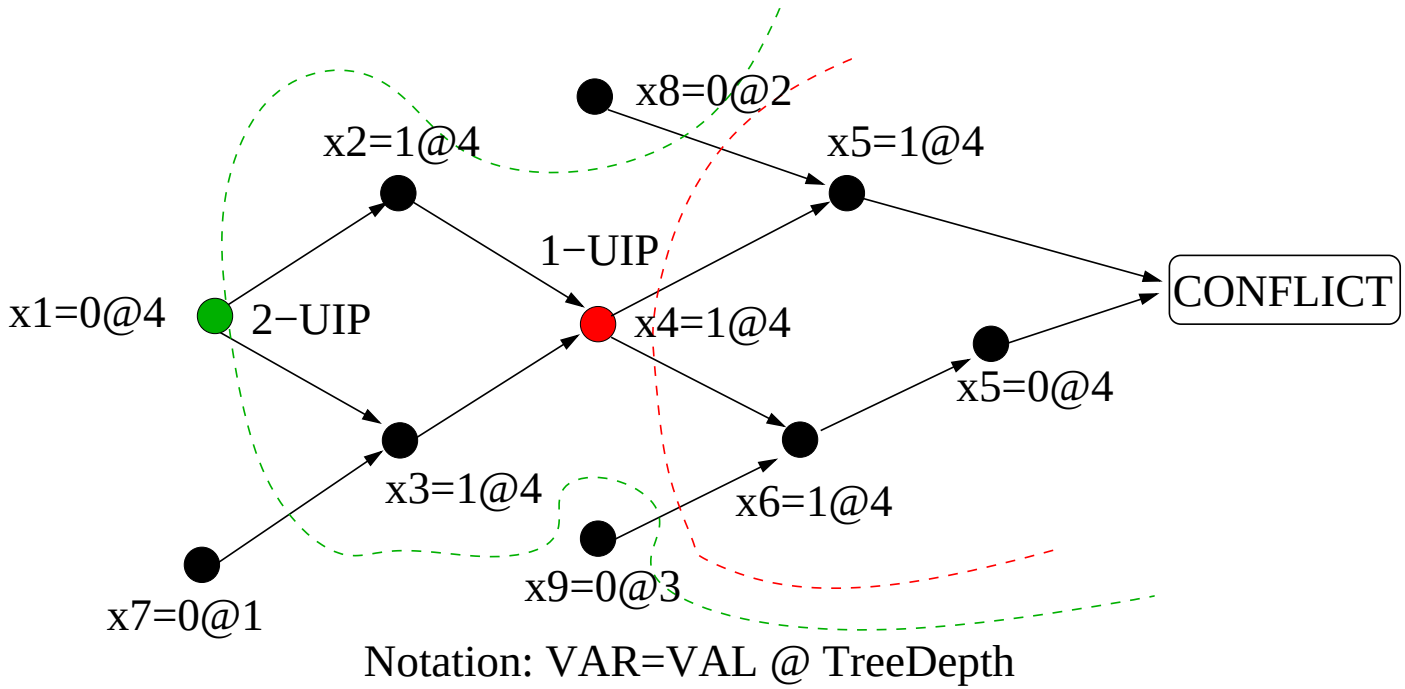
Learned second conflict clause: $l2$: $\neg\neg a \equiv a$

How to learn a clause

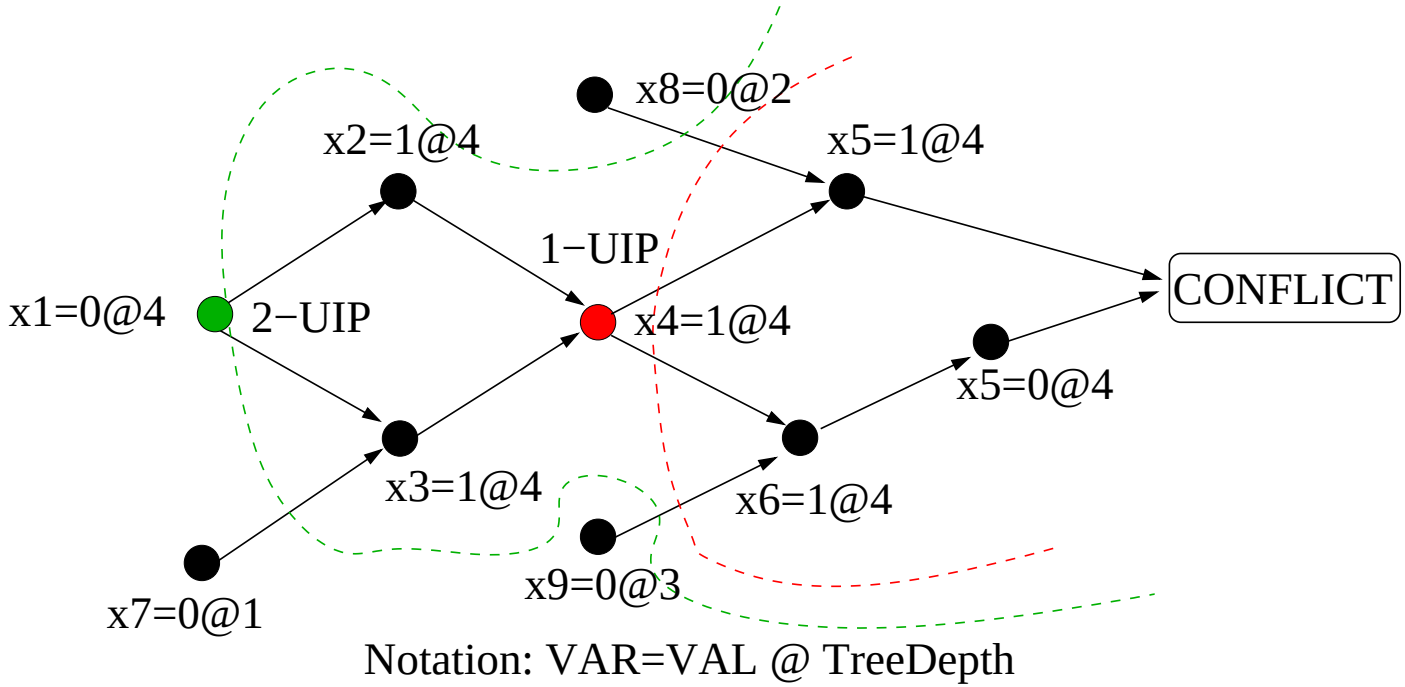
- Use a *cut in the implication graph*
- How to choose the cut?
  - Many different alternatives (conflict nodes on one side of the cut, reason nodes on the other side)
  - Short learned clauses are better than long ones
  - Conflict clause should be fast (=linear time) to compute
  - *1-UIP cut* is shown to be optimal in terms of backtrack level compared to the other possible UIPs [Audemard et al. 2008]

Unique implication point

- A *unique implication point (UIP)* is any node at the current decision level such that any path from the decision variable to the conflict node must pass through it.
- *1-UIP* is a UIP that is closest to the conflict node

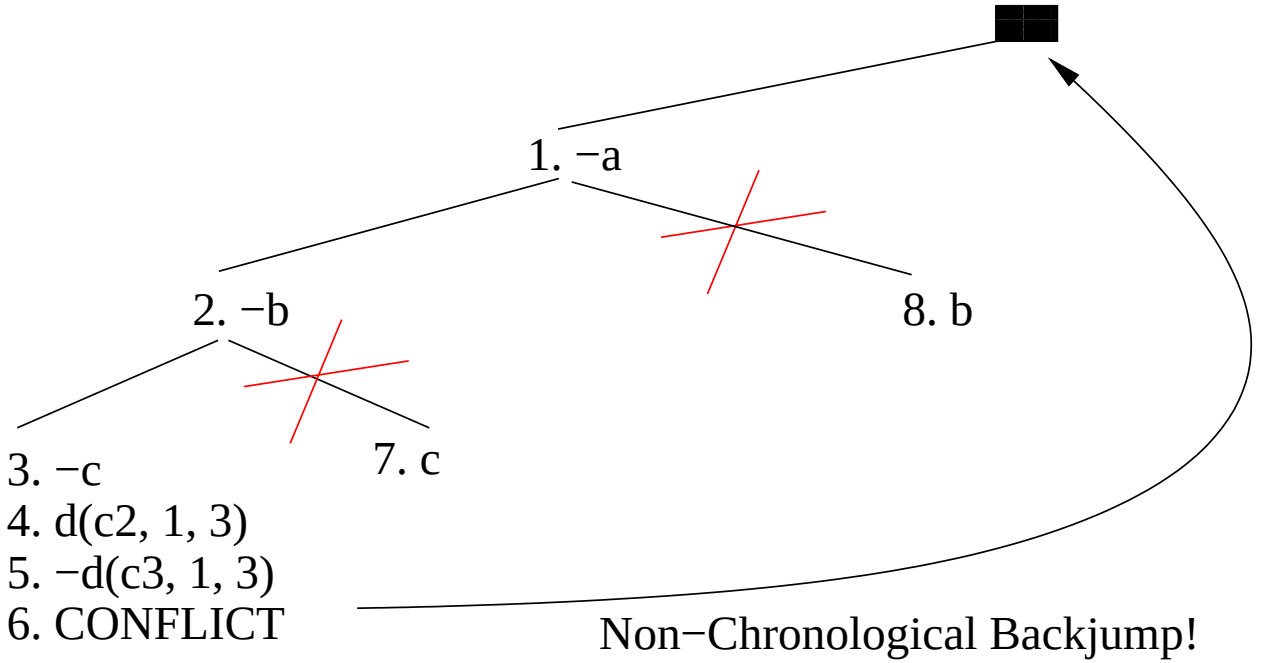


UIP and conflict clause



- 1-UIP: conflict clause $\neg x_4 \vee x_8 \vee x_9$
- 2-UIP: conflict clause $x_1 \vee x_7 \vee x_8 \vee x_9$
- Backtrack level is determined by analyzing the conflict clause C:
  $\max\{\mathit{TreeDepth}(x) \mid x \in C \setminus \{l\}, l \in C \text{ is assigned at conflict level}\}$

Example revisited



Notation: VAR=VAL @ TreeDepth

- Learned conflict clause: $l1: \quad \neg(\neg a \wedge \neg c) \equiv a \vee c$
- Backtrack to TreeDepth=1
- Note: Assignment of $c = 0$ is not discarded

Efficient data structures

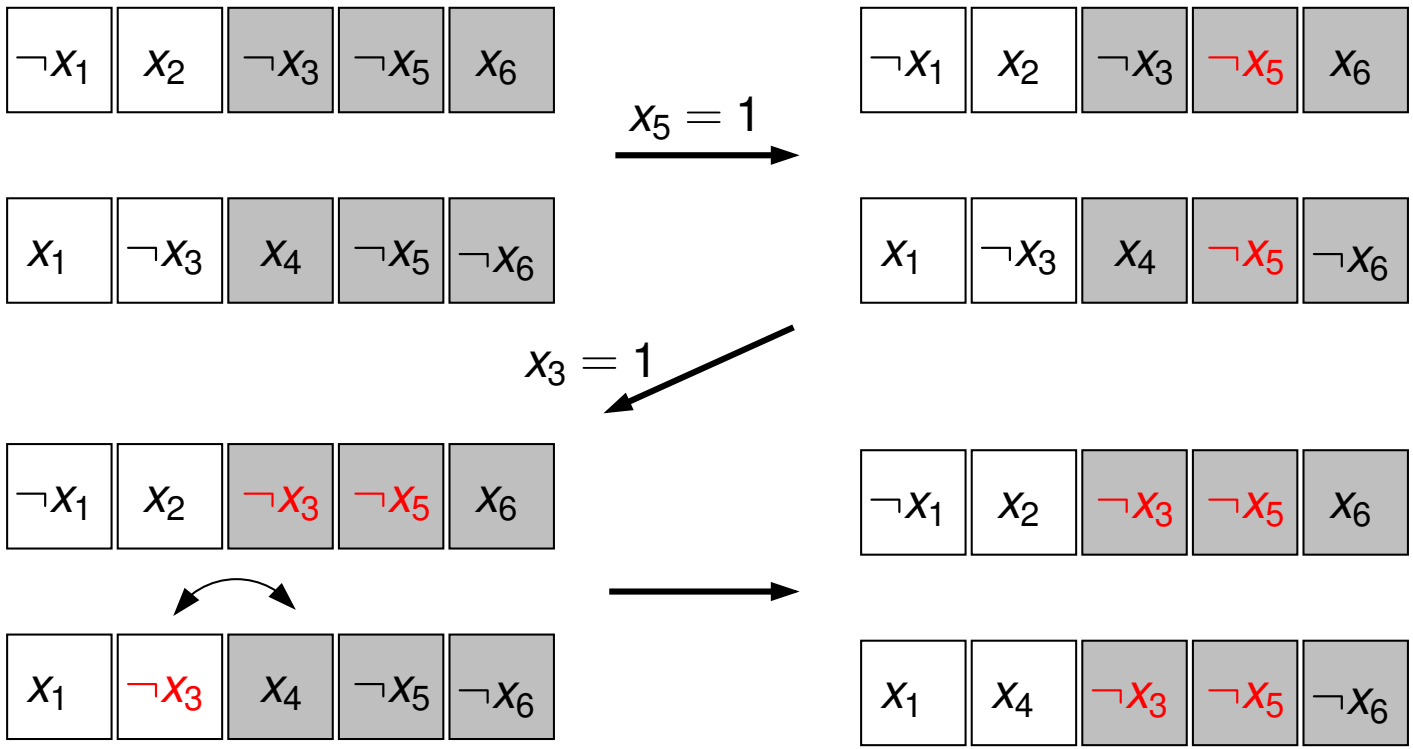
- During search SAT solver spends time on...
  - variable selection $\approx 10\%$
  - unit propagation $\approx 80\%$
  - conflict analysis $\approx 10\%$
- Thus, it is highly important to optimize unit propagation!
- In unit propagation one needs to detect *unit clauses and conflicting clauses*
- No need to detect that all clauses are true!
- No traversing the whole set of clauses, instead:
  - for each literal, store the clauses in which it appears
  - when literal l is added to assignment, only clauses in which $\overline{l}$ appears need to be visited

Efficient data structures: Watch pointers

- Unit clause / conflict detection can be based on two *watched literals* per clause
  - A clause with two non-false literals cannot be unit clause or conflicting clause
- Clause needs to be visited only when its watched literal becomes false – clauses are visited less frequently
- When backtracking, nothing needs to be done (just unassign variables)
- Very effective on long clauses
- Not used for binary clauses (special data structures)

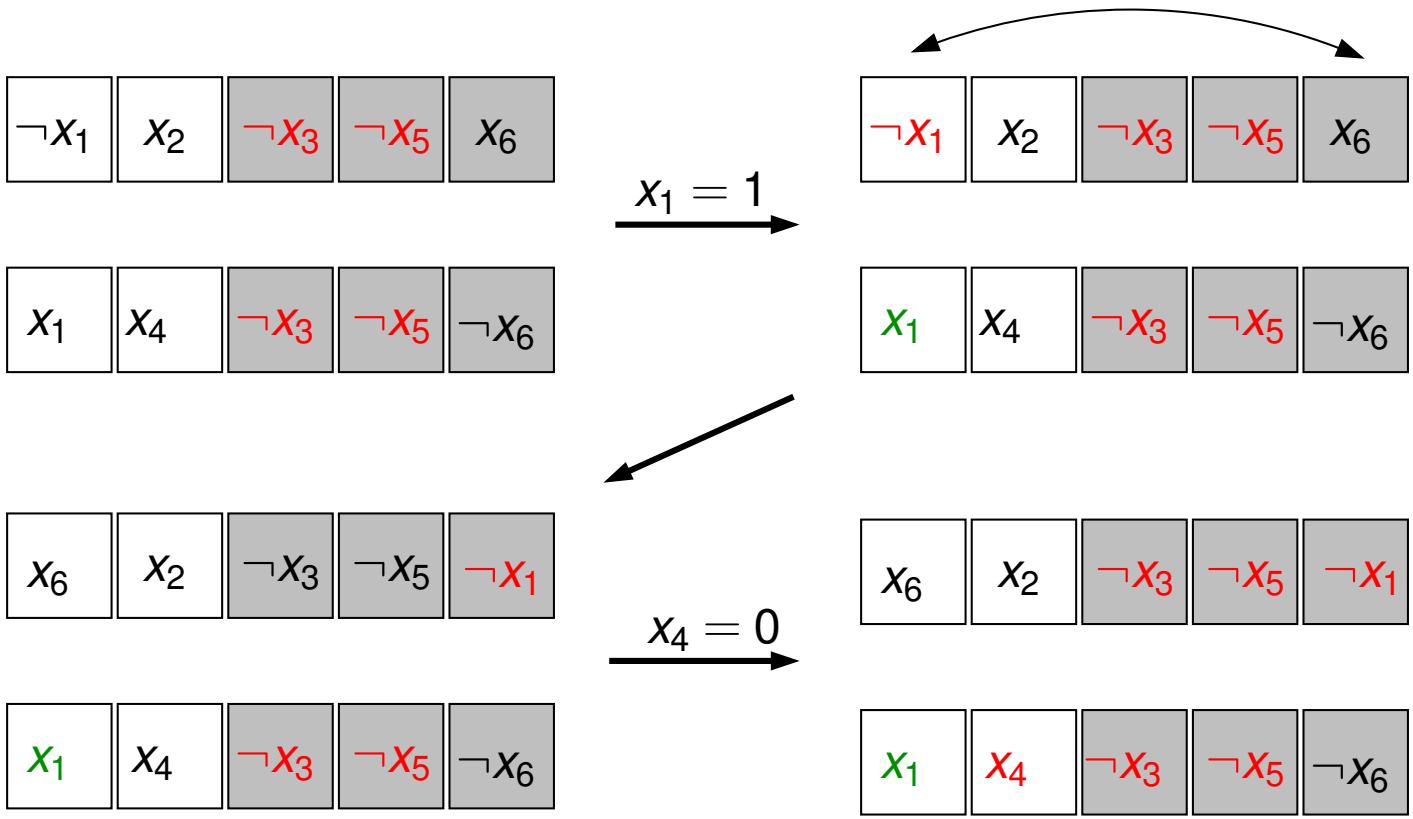
Watched literals — Example

$$\psi = \{x_1 = u, x_2 = u, x_3 = u, x_4 = u, x_5 = u, x_6 = u\}$$



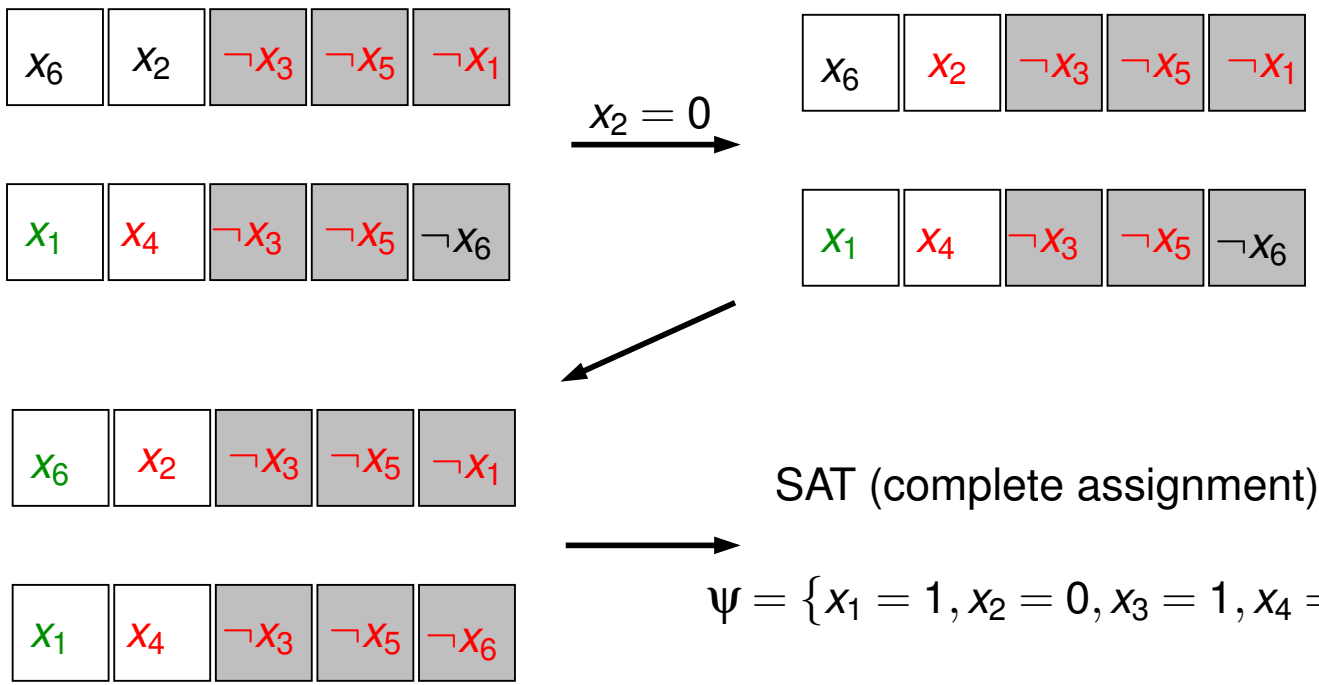
Watched literals — Example cont'd

$$\psi = \{x_1 = u, x_2 = u, x_3 = 1, x_4 = u, x_5 = 1, x_6 = u\}$$



Watched literals — Example cont'd

$$\psi = \{x_1 = 1, x_2 = u, x_3 = 1, x_4 = 0, x_5 = 1, x_6 = u\}$$



Choosing variable — VSIDS

Variable state independent decaying sum (VSIDS):

- Each literal has score
- Score based on the number of occurrences of the literals in the CNF
- Score updated when a new clause is learned
- Pick the unassigned literal with the highest score (break ties uniformly at random)
- Updating:
  - zChaff: every 256 conflicts divide scores by a constant factor 2
  - Minisat: for each conflict, increase the score of involved variables by δ and increase $\delta := 1.05\delta$
- Compatible with the lazy data structure

Restarting

- Runtimes of SAT solver can experince *heavy tail phenomenon*
  - In rare cases, the solver can get trapped on a very long run while most of the time the run times could be short
- Restarts are introduced to avoid this behaviour
  - At restart: unassign all variables but keep the (dynamic) heuristics and learned clauses
- In order to guarantee completeness, restart strategy with increasing cutoff needed, e.g.
  - Geometrical restart: 100, 150, 225, 333, 500, 750, ...
  - Luby sequence: 100, 100, 200, 100, 100, 200, 400 , ...

DPLL vs. CDCL solver

- The success of CDCL is not only due to advances in implementation and clever data structures
- Fundamental reason for better performance of CDCL is that it is a *stronger proof system* than DPLL
  - There exists an infinite family of CNFs F_n ($n = 1, 2, \ldots$) such that the *length of shortest proof* (of unsatisfiability) using DPLL is exponentially larger than the length of shortest proof using CDCL
  - For the other direction, "DPLL proof is always a CDCL proof"
- CDCL with restarts is as strong as *general resolution*

Other techniques

There are many more techniques that efficient CDCL SAT solvers implement:

- Preprocessing
- Inprocessing
- Clause forgetting
- Conflict-clause minimization
- Phase saving
- ...

Lecture 9: Intro to linear and integer linear programming

Outline

- Introduction to linear and integer linear programs
- Examples of constraints and problems modeled as mixed integer linear programs (MIP's)

Goal for today: Learn to recognize and formulate LP's and MIP's; for a given computational problem, learn to

a) encode different types of constraints as an LP/MIP

b) transform one form of a problem into another

Review

- Previously, you have learned how to represent computational problems via reductions to CSP and SAT instances.
- Both types of models allow for complete and local search methods for their solutions.
- SAT solvers exploit the binary domains of variables to obtain efficient constraint propagation techniques (unit propagation) and even learn new instance-specific constraints during execution.
- Now we consider *linear programs* (LP's) for the representation of computational search and optimization problems.
- Theoretically, there is a fundamental difference in the computational complexity of *mixed integer linear programs* (MIP's), which may contain integer variables, and LP's whose variables take real numbers as values.

Introduction

- Today we focus on the modeling aspects of linear programming, while next week we address algorithms.
- Although LP's in general are very versatile from a modeling perspective, some types of constraints are formulated more naturally than others.
- The examples we discuss today involve some of the more frequently encountered types of constraints.
- **Note:** although theoretically (and practically) there are important differences between MIP's and LP's, we simply consider LP's to be a special case of MIP's, where no variable is constrained to take integral values.

General Linear Programs

In a general linear program

$$\min \quad f(x_1, \ldots, x_n) := \sum_{j=1}^{n} c_j x_j \quad \text{such that (s.t.)}$$

$$g_i(x_1, \ldots, x_n) := \sum_{j=1}^{n} a_{ij} x_j = b_i, \quad i = 1, \ldots, m$$

$$l_j \leq x_j \leq u_j$$

inequalities with $\leq$ or $\geq$ can occur in addition to equalities ($=$), maximization can be used instead of minimization, and some of the variables can be unrestricted (do not have bounds).

Note that the x_j are variables, while the l_j, u_j, a_{ij}, c_j and b_i are all given constants. Further, we will always specify explicitly if some (or all) of the x_j are required to take integer values.

Standard and Canonical Forms

- A general LP can be transformed to an equivalent (w.r.t. the set of original variables) but simpler form, for instance, to a canonical or standard form (introduced below).
- Two forms are equivalent (w.r.t. a set of variables) if they have the same set of optimal solutions (w.r.t. the set of variables) or are both infeasible or both unbounded.

An LP is in *canonical form* when

- the objective function is minimized,
- all constraints are inequalities of the form $\sum_{j=1}^{n} a_{ij} x_j \geq b_i$, and
- all variables are non-negative, i.e., bounded by the constraint $x_j \geq 0$.

Standard and Canonical Forms—cont'd

Thus, an LP in canonical form is formulated as

$$\min \quad \sum_{j=1}^{n} c_j x_j \quad \text{s.t.}$$

$$\sum_{j=1}^{n} a_{ij} x_j \geq b_i, \quad i = 1, \ldots, m$$

$$x_j \geq 0, \quad j = 1, \ldots, n$$

The *standard form* is similar but all constraints (different from bounds) are of the form $\sum_{j=1}^{n} a_{ij} x_j = b_i$.

Standard and Canonical Forms—cont'd

An LP can be converted to standard or canonical form using the following transformations:

- Maximization of a function is equivalent to minimization of its negation, since $\max f(x_1, \ldots, x_n) = -\min -f(x_1, \ldots, x_n)$
- An equality can be transformed to a pair of inequalities

$$\sum_{j=1}^{n} a_{ij} x_j = b_i \Leftrightarrow \begin{cases} \sum_{j=1}^{n} a_{ij} x_j \geq b_i \\ \sum_{j=1}^{n} -a_{ij} x_j \geq -b_i \end{cases}$$

- An inequality can be transformed into an equality by *adding a slack (subtracting a surplus) variable*

$$\sum_{j=1}^{n} a_{ij} x_j \leq b_i \Leftrightarrow \begin{cases} \sum_{j=1}^{n} a_{ij} x_j + s = b_i \\ s \geq 0 \end{cases}$$

$$\sum_{j=1}^{n} a_{ij} x_j \geq b_i \Leftrightarrow \begin{cases} \sum_{j=1}^{n} a_{ij} x_j - s = b_i \\ s \geq 0 \end{cases}$$

Transformations—cont'd

- An unrestricted variable x_j (a variable that can take positive and negative values) can be eliminated by introducing two non-negative variables x_j^+, x_j^- as follows: replace every occurrence of x_j with $x_j^+ - x_j^-$ and add the constraints $x_j^+ \geq 0, x_j^- \geq 0$.
- Non-positivity constraints can be expressed as non-negativity constraints: to express $x_j \leq 0$, replace x_j everywhere with $-y_j$ and impose $y_j \geq 0$.
- These transformations are sometimes needed when modeling if the tool used does not support a feature exploited in the LP model, for example, non-positive or unrestricted variables.

Example

- Consider the problem of transforming the LP on the right to standard form. We illustrate the transformation in two steps.

$$\begin{aligned}&\max\ x_2 - x_1 \text{ s.t.}\\ &3x_1 - x_2 \geq 0\\ &x_1 + x_2 \leq 6\\ &-2 \leq x_1 \leq 0\end{aligned}$$

- First:
  turn maximization to minimization,
  turn the unrestricted variable x_2 to a pair of non-negative variables and
  treat bounds as constraints
  to obtain:

$$\begin{aligned}&\min\ -(x_2^+ - x_2^-) + x_1 \text{ s.t.}\\ &3x_1 - (x_2^+ - x_2^-) \geq 0\\ &x_1 + (x_2^+ - x_2^-) \leq 6\\ &x_1 \geq -2\\ &x_1 \leq 0\\ &x_2^+ \geq 0, x_2^- \geq 0\end{aligned}$$

Example—cont'd

- Second:
  eliminate non-positivity constraints and transform inequalities to equalities with slack and surplus variables to obtain:

$$\min\ -x_2^+ + x_2^- - y_1 \text{ s.t.}$$
$$-3y_1 - x_2^+ + x_2^- - s_1 = 0$$
$$-y_1 + x_2^+ - x_2^- + s_2 = 6$$
$$-y_1 - s_3 = -2$$
$$y_1 \geq 0$$
$$x_2^+ \geq 0, x_2^- \geq 0$$
$$s_1 \geq 0, s_2 \geq 0, s_3 \geq 0$$

Modeling

The diet problem: (a typical problem suitable for linear programming)

- We are given the following constants
  a_{ij}: amount of the i-th nutrient in a unit of the j-th food item
  r_i: yearly requirement of the i-th nutrient
  c_j: cost per unit of the j-th food item

- Build a yearly diet (decide yearly consumption of n food items) such that it satisfies the minimal nutritional requirements for m nutrients and is as inexpensive as possible.

- LP solution: take variables x_j to represent yearly consumption of the j-th food item

$$\min \quad \sum_j c_j \, x_j \text{ s.t.}$$
$$\sum_j a_{ij} \, x_j \geq r_i, \quad \forall i$$

Knapsack

(a typical problem suitable for (0-1) integer programming)

- Given: a knapsack of a fixed volume v and n objects, each with a volume a_j and a value b_j.
- Find a collection of these objects with maximal total value that fits in the knapsack.
- IP solution: for each item j take a binary variable x_j to model whether item j is included ($x_j = 1$) or not ($x_j = 0$)

$$\max \sum_j b_j \, x_j \text{ s.t.}$$
$$\sum_j a_j \, x_j \leq v$$
$$0 \leq x_j \leq 1, \quad \forall j$$
$$x_j \text{ is integer } \forall j$$

Facility Location Problem

(A slightly more complicated 0-1 IP problem)

- There is a set of n customers who need to be assigned to one of the m potential facility locations.
- Customers can only be assigned to an open facility, with there being a cost of c_j for opening facility j.
- An open facility can serve an arbitrary number of customers (assigning customer i to facility j incurs a cost of d_{ij}).
- Choose a set of facility locations that minimizes the overall costs of serving all the n customers.
- IP solution: introduce binary variables
  x_j representing the decision to open facility j
  y_{ij} representing the decision to assign customer i to facility j

Facility Location Problem—cont'd

- Objective function to minimize:

$$\sum_{j=1}^{m} c_j x_j + \sum_{i=1}^{n} \sum_{j=1}^{m} d_{ij} y_{ij}$$

- Customers are assigned to exactly one facility:

$$\sum_{j=1}^{m} y_{ij} = 1 \quad \text{for all } i = 1, \ldots, n$$

- Customers can be assigned only to an open facility. Two approaches:
  - If a facility is open, it can serve all n customers:

$$\sum_{i=1}^{n} y_{ij} \leq n \cdot x_j \quad \text{for all } j = 1, \ldots, m$$

  - If a customer i is assigned to facility j, it must be open:

$$y_{ij} \leq x_j \quad \text{for all } j = 1, \ldots, m \text{ and } i = 1, \ldots, n$$

Benefits of Optimal Solutions

- The previous example demonstrates that by making the (implicit) assumption that a resulting solution will minimize the objective value one can “weed out” undesired solutions.
- For example: the previous problem does not indicate whether a solution which opens facilities without assigning customers to it is considered feasible.
- Assume the problem formulation requires that an open facility must have at least some customers assigned to it.
- Since a solution that opens a facility with no customers assigned to it is clearly suboptimal (assuming $c_j > 0$), this assumption is satisfied implicitly.

Expressing Constraints in MIP's

- Some constraints cannot be represented straightforwardly using linear constraints.
- An implication is a typical example which can sometimes be encoded by introducing an additional variable and a new large constant.
- **Example.** Consider a binary variable y and the constraint "if $y = 1$ then $\sum_{j=1}^{n} x_j \geq b_i$" where each x_j is non-negative. Using a large constant M this can be expressed as follows:

$$\sum_{j=1}^{n} x_j \geq b_i - M(1 - y)$$

Notice that here if $y = 1$, then $\sum_{j=1}^{n} x_j \geq b_i$ must hold but if $y = 0$, then $\sum_{j=1}^{n} x_j \geq b_i - M$ imposes no constraint on variables $x_1, \ldots, x_n$ if we choose some $M \geq b_i$.

Expressing Constraints—cont'd

- A frequently occurring situation involves combining constraints "disjunctively".

- **Example.** Consider a disjunctive constraint "$x \geq 5$ or $y \leq 6$" where x and y are non-negative and $y \leq 1000$ due to other constraints.
  This constraint can be encoded by introducing a new binary variable b and constants M_1, M_2 as follows

$$x + M_1 b \geq 5$$
$$y - M_2(1 - b) \leq 6$$

  where we choose the constants $M_1 \geq 5$ and $M_2 \geq 994$.

Example—cont'd

Let us choose $M_1 = 5$ and $M_2 = 994$.

$$x + 5\,b \geq 5$$
$$y - 994\,(1 - b) \leq 6$$

Consider the two possible cases, depending on the value of b:

- If $b = 0$, we have constraints $x \geq 5$ and $y - 994 \leq 6 \leftrightarrow y \leq 1000$ where the latter is satisfied by every (relevant) value of y, since $y \leq 1000$ by definition of the problem.
- If $b = 1$, we have constraints $x + 5 \geq 5 \leftrightarrow x \geq 0$ and $y \leq 6$ where the former is satisfied by every (relevant) value of x.

These techniques for expressing disjunctions are are not general and choosing values for the constants is often non-trivial.

Example: Scheduling Constraints

- In a scheduling application typically following types of variables are used:
  s_j: starting time for job j
  x_{ij}: binary variable representing whether job i occurs before job j
- Consider now a typical constraint:
  "If job 1 occurs before job 2, then job 2 starts at least 10 time units after the end of job 1"
- This is an implication that can be represented by introducing a suitably large constant M (d_1 is the duration of job 1):

$$s_2 \geq s_1 + d_1 + 10 - M(1 - x_{12})$$

  - If $x_{12} = 1$: we get $s_2 \geq s_1 + d_1 + 10$ as required.
  - If $x_{12} = 0$: we get $s_2 \geq s_1 + d_1 + 10 - M$, which implies no restriction on s_2 if M is sufficiently large.

Example: Scheduling Constraints—cont'd

- Disjunctive constraints on binary variables can be expressed straightforwardly.
- For example, to enforce that the values of the variables x_{ij} in the previous example are assigned consistently according to their intuitive meaning following constraints need to be added.
  - "Either i occurs before j or the reverse but not both"
    This is an exclusive-or constraint which can be encoded directly:

$$x_{ij} + x_{ji} = 1 \quad (i \neq j)$$

  - "If i occurs before j and j before k, then i occurs before k."
    This can be seen as a disjunction $\neg x_{ij} \vee \neg x_{jk} \vee x_{ik}$ of binary variables x_{ij}, x_{jk}, x_{ik} (equivalent to $(x_{ij} \wedge x_{jk}) \rightarrow x_{ik}$):

$$(1 - x_{ij}) + (1 - x_{jk}) + x_{ik} \geq 1 \text{ (or equivalently } x_{ij} + x_{jk} - x_{ik} \leq 1)$$

    (A potential problem: $\mathrm{O}(n^3)$ constraints are needed where n is the number of jobs.)

Joint Replenishment Problem

- Consider the problem of scheduling the production of N types of products in a factory to satisfy the demands (orders) of customers that arrive over time periods $1, 2, \ldots T$.
- Producing any amount of product i, $1 \leq i \leq N$, incurs a fixed *production cost* of c_i, in addition to a joint *shipping cost* c_0 (production is assumed to be instantaneous).
- Note that by aggregating products one can save shipping cost!
- A demand $d = (t_d, i_d, q_d)$ arrives at time t_d and asks for q_d units of product i_d.
- In the *make-to-order* variant products are produced after demands for them have been communicated to the factory.
- Note that this means that if a demand d is *satisfied* by a production event at time t, then it must be that $t \geq t_d$.

Joint Replenishment Problem–cont'd

- In addition to the cost incurred due to the production of some types of products over time, unfulfilled demands collect a penalty that we call *delay cost*.
- More precisely, delaying a demand d from its arrival time t_d for some time units δ, incurs a cost equal to $\delta \cdot q_d$, which is proportional to the quantity of product requested by the demand.
- So the problem becomes how to balance delay and production/shipping costs.
- Note that here we assume that all cost factors c_i and all quantities q_d are positive.
- Without loss of generality, we consider only candidate production times that are integers in the set $\{1, \ldots, T\}$.

Joint Replenishment Problem–cont'd

We begin by introducing the following variables:

- x_t: binary variable that when having value 1 indicates that there is a shipment taking place at time period t.
- y_{it}: binary variable that when having value 1 indicates that there is a production of type i in time period t.
- z_{dt}: binary variable that when having value 1 indicates that demand d was delayed from time t until (at least) time $t+1$.

We then formulate the objective function to minimize as follows:

$$\sum_{t=1}^{T} \left(c_0 x_t + \sum_i c_i y_{it} \right) + \sum_d \sum_{t \geq t_d} q_d z_{dt}$$

Joint Replenishment Problem–cont'd

The first type of constraint couples the decision to delay an order to the production timepoints:

- Demands are delayed until they are eventually satisfied

$$\left(\sum_{t=t_d}^{s} y_{i_d t} \right) + z_{ds} \geq 1 \quad \text{for all } d,\ t_d \leq s \leq T-1$$

The second type of constraint couples the production and the shipping decision variables:

- Whenever some product is produced, there is a shipment taking place:

$$y_{it} \leq x_t \quad \text{for all } i = 1, \ldots, N \text{ and } t = 1, \ldots, T$$

Routing Constraints

(An example of a problem where finding a compact MIP encoding is challenging).

- Consider the Hamiltonian cycle problem:
  INSTANCE: An undirected graph (V, E).
  QUESTION: Is there a cycle visiting all nodes of the graph exactly once?
- Note that also the optimization variant is possible, which asks for the shortest (smallest total length) Hamiltonian cycle in the given graph.
- Variations of this problem are frequently encountered in practice (e.g., passenger transportation, parcel delivery, flight routing, ...).
- The optimization variant can be also considered a generalization of the TSP problem, which assumes a complete graph.
- Let us model the search variant of the Hamiltonian cycle problem.

Hamiltonian Cycle

- For simplicity of presentation, we treat the edges as being directed (introduce edges (i,j) and (j,i) for each $\{i,j\}$).
- Introduce a binary variable x_{ij} for each edge $(i,j) \in E$ indicating whether the edge is included in the cycle ($x_{ij} = 1$) or not ($x_{ij} = 0$).
- Constraints:
  - The cycle leaves each node i through exactly one edge:

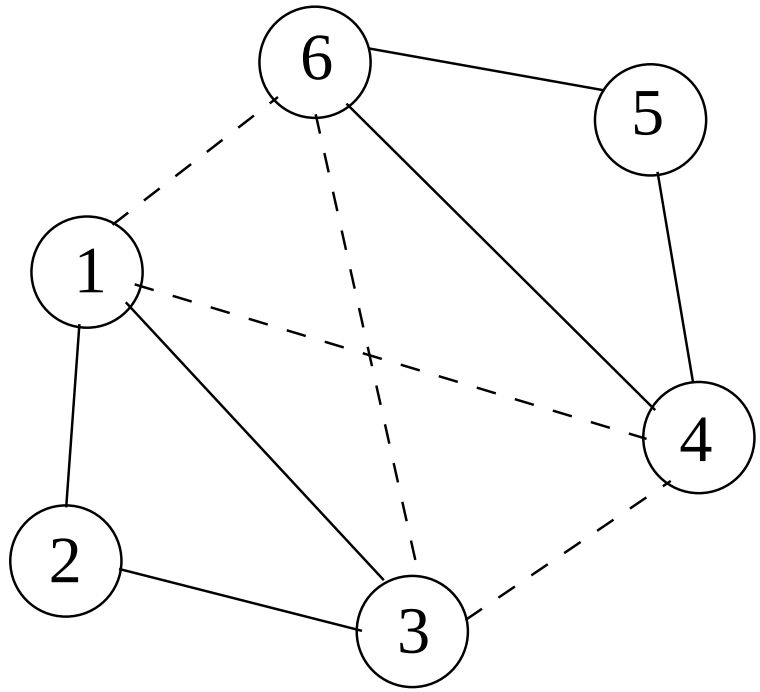
$$\text{for each node } i\text{:} \quad \sum_{(i,j)\in E} x_{ij} = 1$$

  - The cycle enters each node i through exactly one edge:

$$\text{for each node } i\text{:} \quad \sum_{(j,i)\in E} x_{ji} = 1$$

Hamiltonian Cycle—cont'd

- However, the constraints above are not sufficient.
- Consider, for example, a graph with 6 nodes such that variables $x_{1,2}, x_{2,3}, x_{3,1}, x_{4,5}, x_{5,6}, x_{6,4}$ are set to 1 and all others to 0. This solution satisfies the constraints but does not represent a Hamiltonian cycle (two separate cycles).



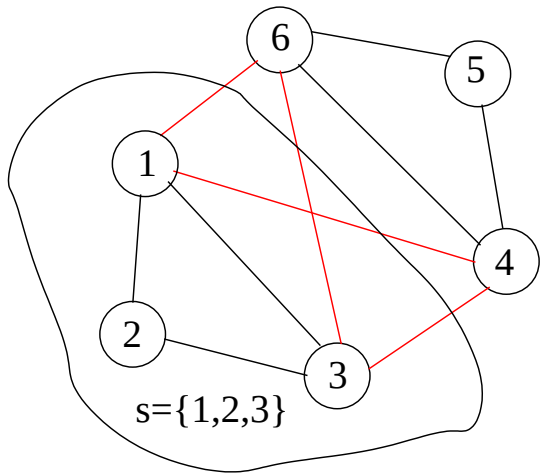
- Enforcing a single cycle is non-trivial.

Hamiltonian Cycle–cont'd

- A solution for small graphs is to require that the cycle leaves every proper subset of the nodes, that is, to have a constraint

$$\sum_{(i,j)\in E, i\in s, j\notin S} x_{ij} \geq 1$$

for every proper subset $S \subset V$ of the nodes V (note: $S \neq V$, since proper subset).



- A potential problem for bigger graphs: $\mathrm{O}(2^n)$ constraints needed where n is the number of nodes.

Hamiltonian Cycle–cont'd

- Another approach, where the number of constraints remains polynomial, is to introduce an integer variable p_i for each node $i = 1, \ldots, n$ in the graph to represent the position of the node i in the cycle, that is, $p_i = k$ means that node i is kth node visited in the cycle.
- In order to enforce a single cycle we need the following conditions.
- Each p_i has a value in $\{1, \ldots, n\}$: $1 \leq p_i \leq n$
- This value is unique, that is, for all pairs of nodes i and j with $i \neq j$, $p_j \neq p_i$ holds.

Hamiltonian Cycle–cont'd

- For all pairs of nodes i and j if node j is the next node after i there must be an edge $(i,j) \in E$, that is,
  - $(p_j = p_i + 1) \rightarrow (i,j) \in E \equiv$ for all $(i,j) \notin E, i \neq j : p_j \neq p_i + 1$
  - $(p_i = n \wedge p_j = 1) \rightarrow (i,j) \in E$
    $\equiv$ for all $(i,j) \notin E, i \neq j : p_i = n \rightarrow p_j \geq 2$
- For condition 'if $p_i = n$, then $p_j \geq 2$" we can use the technique for implications:

$$p_j \geq 2 - (n - p_i)$$

  Notice that
  - if $p_i = n$, then we get $p_j \geq 2$ and
  - if $p_i < n$, then the constraint is satisfied for all value of p_j $(1 \leq p_j \leq n)$.
- To complete the encoding in IP we need to express disequality $(\neq)$.

Expressing Disequality

- For expressing an arbitrary disequality $x \neq y$ of two bounded *integer* variables x and y we reformulate the disequality as "$x > y$ or $y > x$" or equivalently "$x - y \geq 1$ or $x - y \leq -1$".
- Now we can model the disjunction using a binary variable b and constants M_1, M_2 and the constraints

$$x - y \geq 1 - M_1 b$$
$$x - y \leq M_2(1 - b) - 1$$

Notice that

- if $b = 0$, then we get $x - y \geq 1, x - y \leq M_2 - 1$ and
- if $b = 1$, then we get $x - y \geq 1 - M_1, x - y \leq -1$

where the constraints involving M_1, M_2 are satisfied by all values of x, y given large enough M_1, M_2 w.r.t. to the bounds on the values of x, y.

MIP Tools

- There are several efficient commercial MIP solvers.
- Also a large variety of public domain systems exist.
- Different MIP encodings typically lead to different solver runtimes.
- See, for example, `http://www.neos-guide.org/lp-faq` for MIP systems and other information and frequently asked questions.

MIP Solvers

- A MIP solver can typically take its input via an input file and an API.
- There a number of widely used input formats (like mps) and tool specific formats (`lp_solve`, CPLEX, LINDO, GNU MathProg, LPFML XML, ...)
- Most MIP solvers do not require the input program to be in a standard form and typically quite general MIP's are allowed, that is
  - both minimization and maximization are supported and
  - operators "$=$", "$\leq$", and "$\geq$" can all be used.
  - Many solvers provide an API that allows the integration into user-generated programs from various programming languages.

Lecture 10: Linear relaxation and the simplex method

Outline

- Algorithms for solving mixed integer linear programs (MIP's): branch-and-bound and linear relaxation
- Simplex method for solving linear relaxations

Goal for today: Learn how to apply the branch-and-bound method for solving MIP's and the simplex method for solving LP's.

Review: Linear and Integer Programming

- Recall: in the previous lecture we modeled computational problems in the form of linear programs, such as

$$\min \quad c(x) := \sum_{j=1}^{n} c_j x_j \quad \text{s.t.}$$

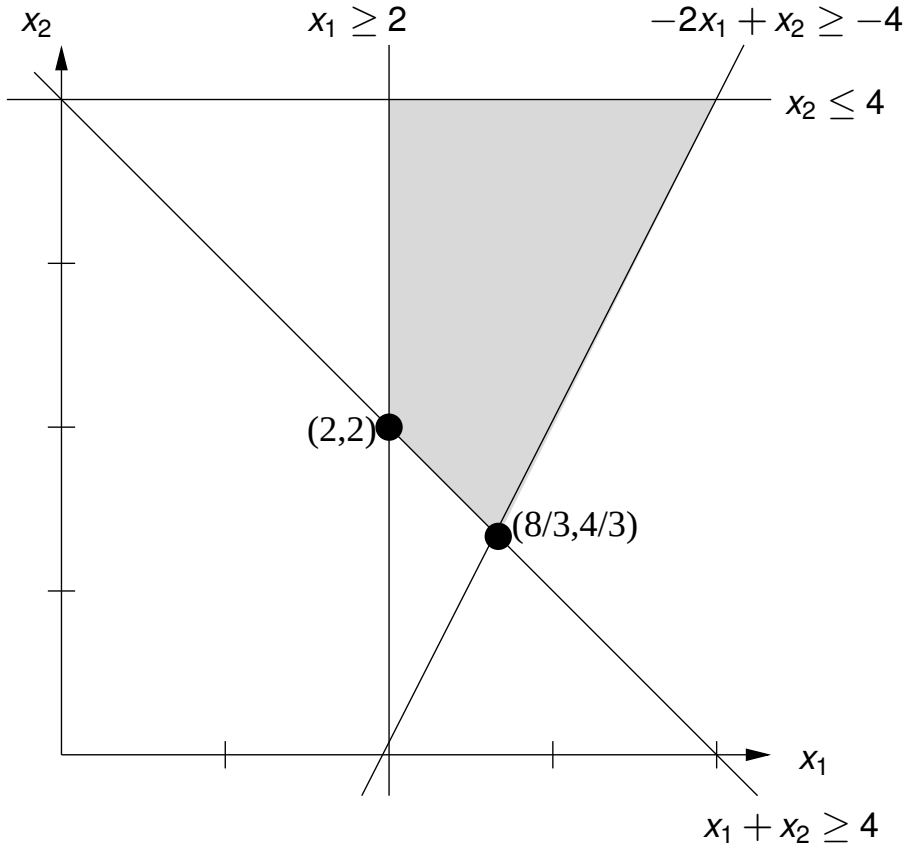
$$\sum_{j=1}^{n} a_{ij} x_j = b_i, \quad i = 1, \ldots, m$$

$$x_j \geq 0, \quad j = 1, \ldots, n$$

- Every LP can be brought into this so-called standard form.
- In mixed-integer linear programs some subset of variables $I \subseteq \{x_1, \ldots, x_n\}$ is required to take *integer values*.
- Finding optimal solutions to LP's can be done in polynomial time (e.g., ellipsoid method), whereas solving integer linear programs is NP-complete.

An Example MIP

$$\begin{array}{lll} \min & x_1 + 2x_2 & \text{s.t.} \\ & x_1 + x_2 & \geq \ 4 \\ & -2x_1 + x_2 & \geq -4 \\ & 2 \leq x_1 & \\ & x_2 \leq 4 & \\ & x_2 \text{ is integer} & \end{array}$$



Note that $\frac{8}{3} + 2\frac{4}{3} = \frac{16}{3} < 2 + 2 * 2 = 6$ but the first values do not satisfy the integrality constraint for x_2.

Solving MIP's

- A typical approach is use *branch and bound search* with a suitable *relaxation*.
- A relaxation of a problem removes constraints in order to obtain a problem that is "easier" to solve.
- Branch and bound search was introduced in Lecture 3 and is readily applicable to solving MIP's (no special-purpose bounding heuristic required).
- Instead of employing a custom bounding heuristic, at each stage of the search the relaxed version of a *subproblem* of the original problem is solved, which is an LP version of a corresponding MIP.

Problem relaxations: intuition

- A *relaxation* $R(P)$ of a problem P has strictly less restrictive constraints but the same objective function.
- Hence, an optimal solution to $R(P)$ can not be worse than an optimal solution for P.
- Example: problem P: Find cheapest flight from HEL to LYS s.t.
  1. Departure date 2.11. departure after 6pm
  2. Direct flight or connecting flights but not via CDG
  3. Airline either Finnair, Air France, or Lufthansa

  Consider now a relaxed version of P, denoted by $R(P)$:
  1. Departure date 2.11. departure at any time
  2. Airline either Finnair, Air France, or Lufthansa
- Clearly, when comparing the objective value of optimal solutions to P and $R(P)$: $OPT(R(P)) \leq OPT(P)$.
- If an optimal solution to $R(P)$ is feasible for P, then it is also optimal for P. However, if $R(P)$ is infeasible, so is P.

Linear relaxation

- For a given MIP P, in order to apply branch and bound search, its relaxation $R(P)$ should be a problem satisfying the very same three conditions (for a minimization problem P):

  R1: $OPT(P) \geq OPT(R(P))$.

  R2: If the optimal solution to $R(P)$ is feasible to P, it is optimal for P.

  R3: If $R(P)$ is infeasible, then so is P.

- A useful relaxation of a MIP P satisfying these condition is the *linear relaxation* of P which is obtained by removing the integrality constraints from P.

Linear relaxation—cont'd

Problem P

$$\begin{array}{lll} \min & x_1 + 2x_2 & \text{s.t.} \\ & x_1 + x_2 & \geq \ \ 4 \\ & -2x_1 + x_2 & \geq -4 \\ & 2 \leq x_1 & \\ & x_2 \leq 4 & \\ & x_2 \text{ is integer} & \end{array}$$

Problem $LR(P)$

$$\begin{array}{lll} \min & x_1 + 2x_2 & \text{s.t.} \\ & x_1 + x_2 & \geq \ \ 4 \\ & -2x_1 + x_2 & \geq -4 \\ & 2 \leq x_1 & \\ & x_2 \leq 4 & \end{array}$$

- The linear relaxation satisfies conditions R1–R3 because feasible solutions of $LR(P)$ include all feasible solutions of P.
- It is also computationally interesting because it is a strong relaxation which provides a global view on the constraints.

Branch and bound for MIP

- Note: from now on, $R(P) := LR(P)$.
- Applying branch and bound search for solving MIP's is very similar to the CSP case: given a problem P, the branching operation creates new subproblems $P_1, \ldots, P_k$, whose union is equivalent to P.
- The new subproblems, however, are based on an optimal solution x^* to $R(P)$ that is not feasible to P and neither to any of $R(P_1), \ldots, R(P_k)$.
- Given optimal solution x^* to $R(P)$, x^* is not feasible to P iff there is a integer variable x_j in P that has a fractional value x_j^* in x^*.
- For such a variable x_j with a fractional value x_j^*, we can create two subproblems (here: $k = 2$):
  - $P-$, which has the additional constraint $x_j \leq \lfloor x_j^* \rfloor$;
  - $P+$, which has the additional constraint $x_j \geq \lfloor x_j^* \rfloor + 1$.

Branch and bound for MIP—cont'd

- The bounding heuristic is replaced by a solver for the linear relaxation of each of the problems encountered in the search.
- Let c be the cost of a known feasible solution to the original MIP (possibly suboptimal).
- Then whenever we encounter a (sub)problem P whose relaxation $R(P)$ has the optimal solution value $OPT(R(P)) \geq c$, we prune the search.
- This is because by R1 $OPT(P) \geq OPT(R(P))$ and, hence, it is not possible to find a solution with a smaller objective value than c among the feasible solutions "below" P inside the search tree.

Branch and bound for MIP—cont'd

initially: $c \leftarrow \infty$;
procedure MIP_Branch&Bound(MIP P):
    if $R(P)$ is infeasible **then**
        return;
    else
        Solve $R(P)$ to get an optimal relaxation solution x^*;
        if x^* is feasible and thus optimal for P **then**
            if $OPT(P) := c(x^*) < c$ **then**
                $c \leftarrow c(x^*)$;
        else
            if $OPT(R(P)) := c(x^*) < c$ **then**
                split P into $P_1, \ldots, P_k$ by applying *branching rules*;
                for all $1 \leq p \leq k$ **do**
                    MIP_Branch&Bound(P_p);
            end
        end
    end
end

Branch and bound for MIP—cont'd

- One can show that if the set of feasible solutions of $R(P)$ is bounded, the algorithm terminates in finite time.
- Note that in this case P has only a finite number of feasible solutions (if any) if all variables are required to take integer values in P.
- If the set of feasible solutions $R(P)$ is not known to be bounded, one can replace P by a more constrained P' (for which this holds) that has an optimal solution not worse than the original P.

Example. Branch and Bound search using linear relaxation

Consider the integer program P:

$\min\ -8x_1 - 11x_2 - 6x_3 - 4x_4$ s.t.

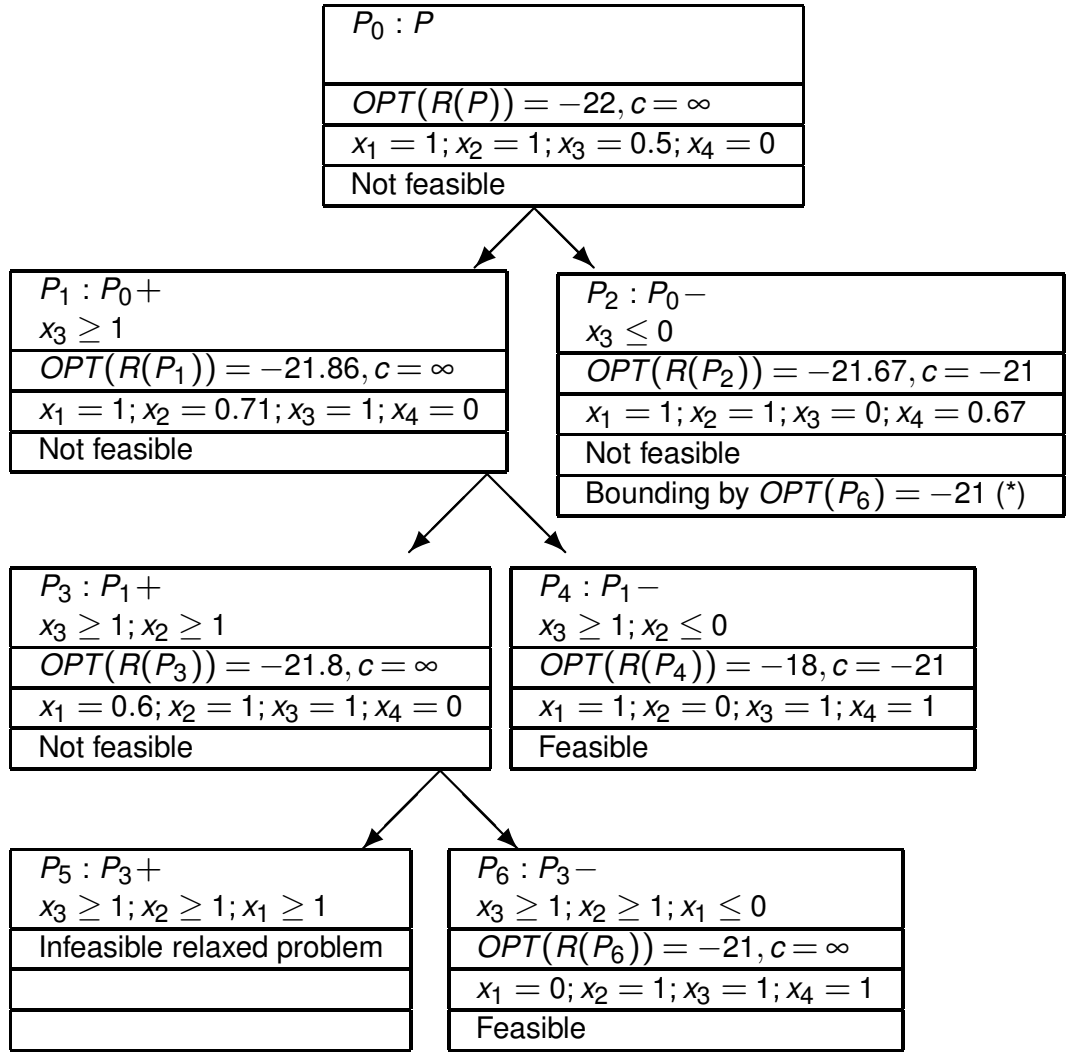
$5x_1 + 7x_2 + 4x_3 + 3x_4 \leq 14$

$0 \leq x_i \leq 1$

x_i is integer, $i = 1, \ldots, 4$

Each node $P_1, P_2, \ldots$ gives the new problem after branching and an optimal solution of the corresponding relaxed problem. The optimal solution to problem P is obtained from P_6.

Note (*): For P_2 the optimal solution satisfies $OPT(R(P_2)) \geq -21.67$ but because in the objective function all coefficients are integers, $OPT(P_2)$ has also an integer value and, thus, $OPT(P_2) \geq \lceil OPT(R(P_2)) \rceil = -21$.



Improving Effectiveness

- Careful formulation
  - Strong relaxations typically work well but are often bigger in size.
  - Break symmetries.
  - Multiple “big-M” values often lead to performance problems.
  - Deciding which formulation works better needs often experimentation.
- Special branching rules
  In many systems, for example, Special Ordered Sets are available.
- Cutting planes
  These are constraints that are added to a relaxation to “cut off” the optimal relaxation solution x^*. Often are problem specific but there are also general techniques (e.g. Gomory cuts).

lp_solve

- `lp_solve` is a a public domain MIP solver, see `http://lpsolve.sourceforge.net/` for latest version.
- `lp_solve` accepts a number of input formats
- **Example.** `lp_solve` native format

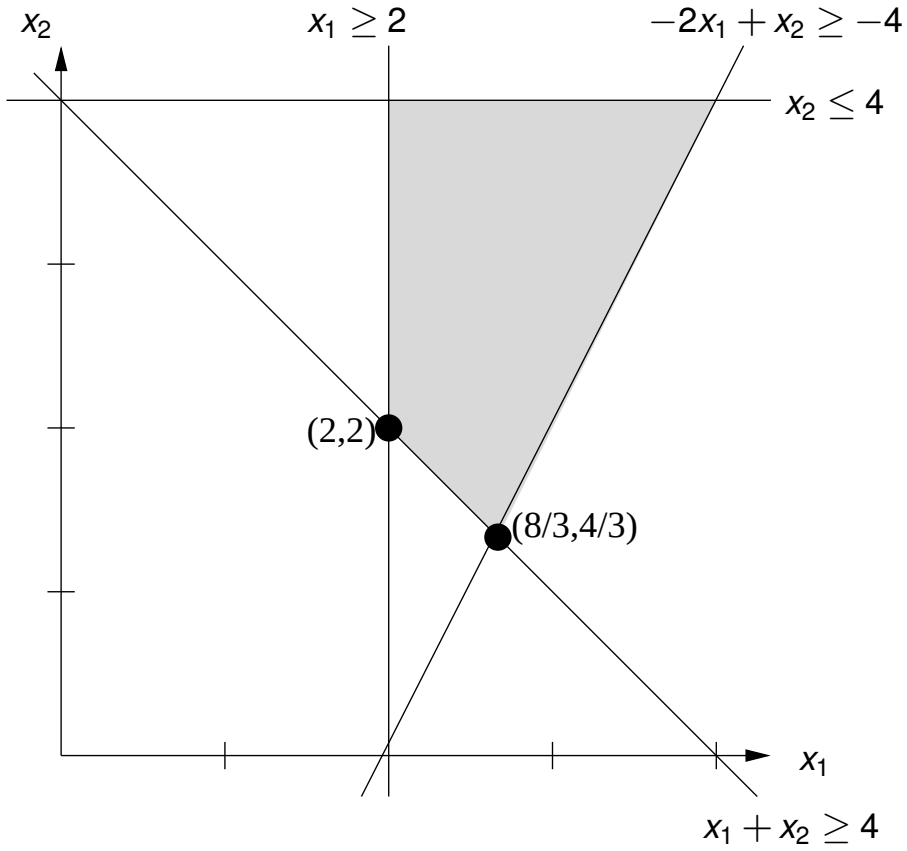
```
min: x1 + x2 + 3x3;
     x1 - x2 <= 1;
     2x2 - 2.5x3  >= 1;
     -7x3 + x2 = 3;
> lp_solve < example
Value of objective function: 3
Actual values of the variables:
x1                                0
x2                                3
x3                                0
```

Solving Linear Relaxations

- Linear Relaxation of a MIP gives a linear program (LP).
- There are a number of well-known techniques for solving LPs
  - Simplex method
    The oldest and most widely used method with very mature implementation techniques.
    Worst-case time complexity exponential but seems to work fairly well in practice.
  - Interior point methods
    A newer approach; polynomial time worst case time complexity; implementation techniques advancing
- Next, the Simplex method is reviewed as an example.

Solving LP's

$$\begin{aligned} \min \quad & x_1 + 2x_2 \quad \text{s.t.} \\ & x_1 + x_2 \geq 4 \\ & -2x_1 + x_2 \geq -4 \\ & 2 \leq x_1 \\ & x_2 \leq 4 \end{aligned}$$



One can show that an optimal solution occurs at an *extreme points* ("corner point") of the feasible region of the LP. Considering all of these in turn, one can thus find an optimal solution.

Simplex Method

- Assumes that the linear program is in standard form:

$$\min \quad \sum_{j=1}^{n} c_j x_j \text{ s.t.}$$

$$\sum_{j=1}^{n} a_{ij} x_j = b_i, \quad i = 1, \ldots, m$$

$$x_j \geq 0, \quad j = 1, \ldots, n$$

- Extreme points of its feasible region correspond to so-called *basic feasible solutions*.
- The basic idea: start from a basic solution and look at the adjacent ones. If an improvement in cost is possible by moving to an adjacent solution, we do so. An optimal solution has been found if no improvement is possible.

Basic Feasible Solutions

- Assume an LP in standard form with m linear equations and n variables $x_1, \ldots, x_n$, $m < n$, and that the columns of the constraint matrix $A = (a_{ij})$ are linearly independent.
- A *basic solution* satisfies the following conditions:
  - $n - m$ variables are set to 0 and
  - the assignment for the other m variables (the basis) gives a unique solution to the resulting set of m linear equations.
- This means that a basic solution is obtained by choosing m variable as the basis, setting the other $n - m$ variables to zero and solving the resulting set of equations for the basic variables. If there is a unique solution, this is gives a basic solution.
- A basic feasible solution (bfs) is a basic solution such that every variable is assigned a value ≥ 0.

Example

- Consider the LP

$$
\begin{aligned}
\min\ & 2x_2 + x_4 + 5x_7 \\
& x_1 + x_2 + x_3 + x_4 = 4 \\
& x_1 + x_5 = 2 \\
& x_3 + x_6 = 3 \\
& 3x_2 + x_3 + x_7 = 6 \\
& x_1, \ldots, x_7 \geq 0
\end{aligned}
$$

- For example, the basis (x_4, x_5, x_6, x_7) gives a basic feasible solution $x0 = (0, 0, 0, 4, 2, 3, 6)$ because $x_4 = 4, x_5 = 2, x_6 = 3, x_7 = 6$ is the unique solution to the resulting set of equations:

$$
\begin{aligned}
& 0 + 0 + 0 + x_4 = 4 \\
& 0 + x_5 = 2 \\
& 0 + x_6 = 3 \\
& 3 \cdot 0 + 0 + x_7 = 6
\end{aligned}
$$

Moving from bfs to bfs

- When moving from one bfs to another the idea is to remove one variable from the basis and replace it with another. This is called *pivoting*.
- In the Simplex algorithm, this is organized as a manipulation of a *tableau* where, for instance, a set of equations

$$\begin{aligned} 3x_1 + 2x_2 + x_3 &= 1 \\ 5x_1 + x_2 + x_3 + x_4 &= 3 \\ 2x_2 + 5x_2 + x_3 + x_5 &= 4 \end{aligned}$$

is represented as

	x_1	x_2	x_3	x_4	x_5
1	3	2	1	0	0
3	5	1	1	1	0
4	2	5	1	0	1

Tableaux

- Pivoting is handled by keeping the set of equations *diagonalized* with respect to the basic variables.
- This can be achieved using elementary row operations (Gaussian elimination): multiplying a row with a non-zero constant; adding a row to another.

Example.

Consider the set of equations

	x_1	x_2	x_3	x_4	x_5
1	3	2	1	0	0
3	5	1	1	1	0
4	2	5	1	0	1

Given a basis $B = (x_3, x_4, x_5)$, we can transform the tableau to a diagonalized form w.r.t. it by multiplying Row 1 with -1 and adding it to Rows 2 and 3:

	x_1	x_2	x_3	x_4	x_5
1	3	2	1	0	0
2	2	-1	0	1	0
3	-1	3	0	0	1

Tableaux—cont'd

- We denote by $x_{i,j}$ the entry on the ith row and jth column in a tableau.
- Notice that in the diagonalized form column 0 gives the values of the basic variables in the bfs $x0$ in question:

$$x0_{B(i)} = x_{i,0}, i = 1, \ldots, m$$

where $B(i)$ denotes the column of the ith basic variable.

- **Example**. Consider the set of equations:

	x_1	x_2	x_3	x_4	x_5
1	3	2	1	0	0
2	2	-1	0	1	0
3	-1	3	0	0	1

Given the basis $B = (x_3, x_4, x_5)$, $B(1) = 3, B(2) = 4, B(3) = 5$ and for its basic solution $x0$ holds: $x0_3 = 1, x0_4 = 2, x0_5 = 3$

Pivoting

- In pivoting a chosen variable x_j enters the basis and another variable x_i leaves it.
- In the tableau this defines a *pivot element* $x_{l,j}$ where column j corresponds to the entering variable x_j and row l to the leaving variable x_i such that $B(l) = i$. We say that we *pivot on* $x_{l,j}$.

Example

Consider the tableau

	x_1	x_2	x_3	x_4	x_5
1	3	2	1	0	0
2	2	-1	0	1	0
3	-1	3	0	0	1

and the case where x_1 enters and x_3 leaves the basis.

Now the pivot element is $x_{1,1}$ as $B(1) = 3$.

Pivoting

- In pivoting the tableau is brought to the diagonalized form w.r.t. the new basis using elementary row operations (Gaussian elimination):
  - for the pivot row l, all elements are divided by the pivot element and, hence, the pivot element in the new tableau is 1;
  - for other rows i, the resulting pivot row multiplied by $x_{i,j}$ is subtracted from the row, and, hence all elements in column j (except the pivot element) are 0 in the new tableau.
- This means that

$$\begin{aligned} x'_{l,q} &= \frac{x_{l,q}}{x_{l,j}} && q = 0, \ldots, n \\ x'_{i,q} &= x_{i,q} - x_{i,j} x'_{l,q} && q = 0, \ldots, n, i = 1, \ldots, m; i \neq l, \end{aligned}$$

where $x_{i,j}$ and $x'_{i,j}$ are the old and new tableaux, respectively.

Example

- Consider the tableau below and the pivot element $x_{1,1}$.

	x_1	x_2	x_3	x_4	x_5
1	3	2	1	0	0
2	2	-1	0	1	0
3	-1	3	0	0	1

- After pivoting we obtain a new tableau:

	x_1	x_2	x_3	x_4	x_5
$\frac{1}{3}$	1	$\frac{2}{3}$	$\frac{1}{3}$	0	0
$\frac{4}{3}$	0	$-\frac{7}{3}$	$-\frac{2}{3}$	1	0
$\frac{10}{3}$	0	$\frac{11}{3}$	$\frac{1}{3}$	0	1

For example: $x_{2,1} = 2 - 2 \cdot 1 = 0$, $x_{2,2} = -1 - 2 \cdot \frac{2}{3} = -\frac{7}{3}$ $x_{2,3} = 0 - 2 \cdot \frac{1}{3} = -\frac{2}{3}$ and $x_{3,2} = 3 - (-1) \cdot \frac{2}{3} = \frac{11}{3}$.

- The new basis is (x_1, x_4, x_5) and, hence, $B(1) = 1, B(2) = 4, B(3) = 5$.

Cost Function in the Tableau

- A cost function $z = \sum_{i=1}^{n} c_i x_i$ can be added as an extra equation $-z + \sum_{i=1}^{n} c_i x_i = 0$ to the tableau (no need to add a column for z).
- To start, we need a bfs and to make zero the c_js for the basic variables.
- This can be done using elementary row operations.
- Consider the example with x_3, x_4, and x_5 as the basis.
- After transformation to the diagonalized form, subtract the resulting Rows 1, 2, 3 from Row 0, to get the desired form.

- Our running example and a cost function $z = x_1 + x_2 + x_3 + x_4 + x_5$ lead to a tableau:

	x_1	x_2	x_3	x_4	x_5
0	1	1	1	1	1
1	3	2	1	0	0
3	5	1	1	1	0
4	2	5	1	0	1

	x_1	x_2	x_3	x_4	x_5
-6	-3	-3	0	0	0
1	3	2	1	0	0
2	2	-1	0	1	0
3	-1	3	0	0	1

Choosing a Profitable Column

- It turns out that the cost function can be improved if we move to a bfs containing a non-basic variable x_j where the corresponding value c_j in the tableau is negative.
- If no such c_j exists, then an optimal solution has been found.
- Consider the previous example with the basis (x_3, x_4, x_5). Now the equation for the cost function is $-z - 3x_1 - 3x_2 = -6$, i.e., $z = -3x_1 - 3x_2 + 6$. Hence, we can improve (decrease) the value of the cost function by increasing the value of x_1 or x_2 (because $c_1 = c_2 = -3 < 0$) and, hence, the current bfs $x0 = (0, 0, 1, 2, 3)$ is not an optimal one.
- Hence, we could move to a new bfs with entering variable x_1 or x_2 to improve the cost function.
- But how to choose the leaving variable?

Choosing the Leaving Variable

- The idea is to move to an adjacent bfs containing the entering variable x_j.
- In order not to miss an adjacent bfs we need to choose a pivot element $x_{k,j}$ with the smallest positive ratio $\frac{x_{k,0}}{x_{k,j}}$, that is, a $x_{k,j}$ such that

$$\frac{x_{k,0}}{x_{k,j}} = \min_{\substack{i \\ x_{i,j}>0}} \left(\frac{x_{i,0}}{x_{i,j}}\right)$$

- Then the leaving variable is $B(k)$.
- Note that the rule for choosing the leaving variable is sufficient for maintaining a feasible solution (all variables, including the basic ones, stay non-negative).

Example

- Consider the tableau

	x_1	x_2	x_3	x_4	x_5
-6	-3	-3	0	0	0
1	3	2	1	0	0
2	2	-1	0	1	0
3	-1	3	0	0	1

- If x_2 is the entering variable, the ratios are:

i	$\frac{x_{i,0}}{x_{i,2}}$
1	$\frac{1}{2}$
2	$-\frac{2}{1}$
3	$\frac{3}{3}$

- Then the pivot element is $x_{1,2}$ because the smallest positive ratio $\frac{x_{i,0}}{x_{i,2}}$ is $\frac{1}{2}$ for $i = 1$ and the leaving variable is x_3 as $B(1) = 3$.

Simplex algorithm

procedure Simplex
opt := “no”; unbounded := “no”;
while opt = “no” and unbounded = “no” **do**
    if $c_j \geq 0$ for all j **then** opt := “yes”
    else
        choose any j such that $c_j < 0$;
        if $x_{i,j} \leq 0$ for all i **then** unbounded := “yes”
        else
            find $\min_{\substack{i \\ x_{i,j}>0}} \left(\frac{x_{i,0}}{x_{i,j}}\right) = \frac{x_{k,0}}{x_{k,j}}$
            and pivot on $x_{k,j}$
        end if
    end if
end while.

Example

- Consider the tableau on the right (above).
- Running Simplex on this tableau, we notice that for variables x_1 and x_2, $c_j < 0$.
- If we choose c_2, then we need to pivot on $x_{1,2}$ as argued in the previous example.
- Then the new tableau is on the right (below).
- Here all c_js are non-negative and, hence, an optimal solution $(0, \frac{1}{2}, 0, \frac{5}{2}, \frac{3}{2})$ has been found with cost $\frac{9}{2}$ $(-z = -\frac{9}{2})$.

	x_1	x_2	x_3	x_4	x_5
-6	-3	-3	0	0	0
1	3	2	1	0	0
2	2	-1	0	1	0
3	-1	3	0	0	1

	x_1	x_2	x_3	x_4	x_5
$-\frac{9}{2}$	$\frac{3}{2}$	0	$\frac{3}{2}$	0	0
$\frac{1}{2}$	$\frac{3}{2}$	1	$\frac{1}{2}$	0	0
$\frac{5}{2}$	$\frac{7}{2}$	0	$\frac{1}{2}$	1	0
$\frac{3}{2}$	$-\frac{11}{2}$	0	$-\frac{3}{2}$	0	1

Further Issues

Implementations of the Simplex methods also need to address:

- Finding the first bfs to start Simplex:
  Implementations of the simplex method perform an initial phase with artificial variables (one for each constraint) whose sum is to be minimized to obtain a bfs for the original problem (or determine infeasibility).
- Treating *degenerate solutions* (some basic variables have zero value) which may lead to cycling:
  Bland's rule avoids cycling by using the variable index for determining the entering variable and to break ties between leaving variable candidates.
- How to choose the entering variable:
  nonbasic gradient method (choosing the most negative c_j), greatest increment method, Bland's rule ...

Summary: Solving MIP's

- Experiment with different formulations as well as different solvers and parameters/methods to see which performs best.
- Avoid multiple "big-M" values.
- Try to break symmetries.
- Do not introduce unnecessary integer variables.
- Scale the coefficients in the constraints to values as small as possible.
- Try to use sparse matrix representations if the problem is large and memory consumption becomes an issue.

Lecture 11: Introduction to Convex Optimization

Outline

- Introduction to non-linear convex optimization
- Projected gradient method for problems with “box constraints”
- Newton’s method for unconstrained optimization

Goal for today: For a given suitable (convex) non-linear optimization problem, learn to apply the projected gradient and Newton’s method for obtaining an iterative solution method.

Review: MIP and LP

- In the last two lectures we discussed linear programming models for combinatorial optimization problems and a general-purpose method for solving these (branch-and-bound and linear relaxation).
- Linear relaxations then lead to linear programs with no integer variables, which can be solved, e.g., by the simplex method.
- Often one also encounters problems that have a non-linear objective function and/or constraints.
- Examples for non-linear optimization problems with integer variables: cross-layer network optimization, supply chain optimization, some types of problems that involve random variables (with known probability distribution).

General optimization problems

- Consider a general optimization problem with no integer constraints

$$\begin{aligned} &\min \quad f(x) \quad \text{s.t.} \\ &x \in X \end{aligned}$$

  where $x = (x_1, \ldots, x_n)^T$, $f : \mathbb{R}^n \to \mathbb{R}$ and $X \subseteq \mathbb{R}^n$.
- For example: $X = \{x \in \mathbb{R}^n \mid g_i(x) \leq 0, \forall i = 1 \ldots m\}$, where $g_i : \mathbb{R}^n \to \mathbb{R}$.
- For $f(x) = \sum_{j=1}^{n} c_j x_j$ and $g_i(x) = b_i - \sum_{j=1}^{n} a_{ij} x_j$ we obtain the special case of *linear programming*.
- Note that the formulation above may result from the relaxation of a problem with integer variables and non-linear constraints and objective function.

General optimization problems —cont'd

- In general it is hard to come up with (provably) good methods that terminate in reasonable time.
- In fact, the worst-case complexity of current global optimization methods is exponential in problem size.
- However, if the problem is "sufficiently small" and constraint and objective functions are "nice behaving" one can sometimes obtain methods that converge reasonably fast to good solutions.
- For example if the f and g_i are convex and differentiable, one can typically find good solutions in reasonable time (all local minima of convex functions are global minima).
- The branch-and-bound method can be extended to these cases when there are integer variables.

Notation and important concepts

- Recall the *dot product* (a.k.a. *scalar product*)

$$a \cdot b = ||a||\,||b||\,\cos\theta = \sum_i a_i b_i = a^T b,$$

where $a, b \in \mathbb{R}^n$ are column vectors (given w.r.t. the standard basis) and θ is the angle between them.

- For a differentiable function $f : \mathbb{R}^n \to \mathbb{R}$, its *gradient* $\nabla f(x)$ has as entries the partial derivatives of f, i.e., $\nabla f_j = \partial f / \partial x_j$.
- For a fixed point $\hat{x}$, its *first-order Taylor expansion* at $\hat{x}$ leads to an approximation $f(x) \approx f(\hat{x}) + \nabla f(\hat{x}) \cdot (x - \hat{x})$.
- ... and its *second-order Taylor expansion* at $\hat{x}$

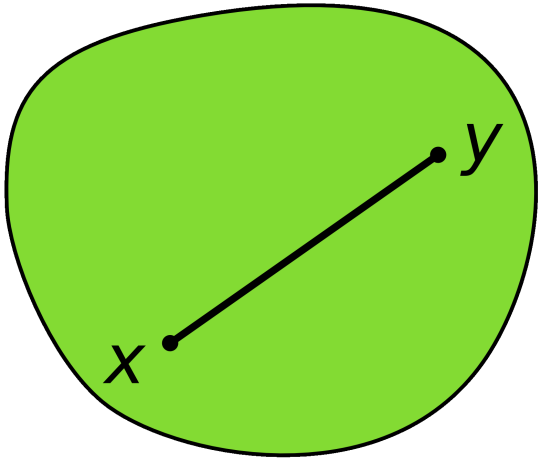
$$f(x) \approx f(\hat{x}) + \nabla f(\hat{x}) \cdot (x - \hat{x}) + \frac{1}{2}(x - \hat{x}) \cdot \nabla f^2(\hat{x}) \cdot (x - \hat{x}),$$

where $\nabla f^2(\hat{x})$ is the matrix containing the second-order partial derivatives of f at $\hat{x}$, a.k.a. *Hessian matrix*.

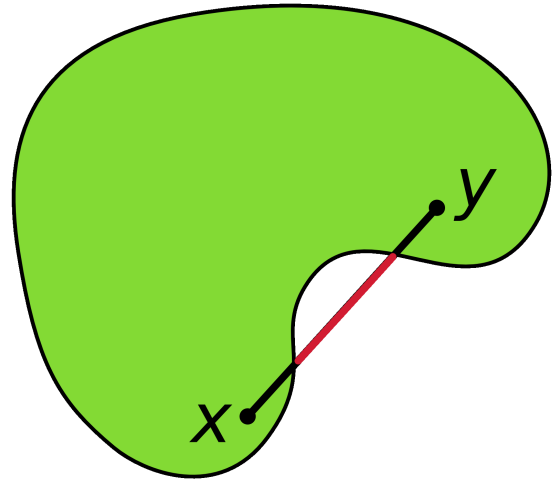
Convex sets

Pictures: Wikipedia

Convex set



Non-convex set

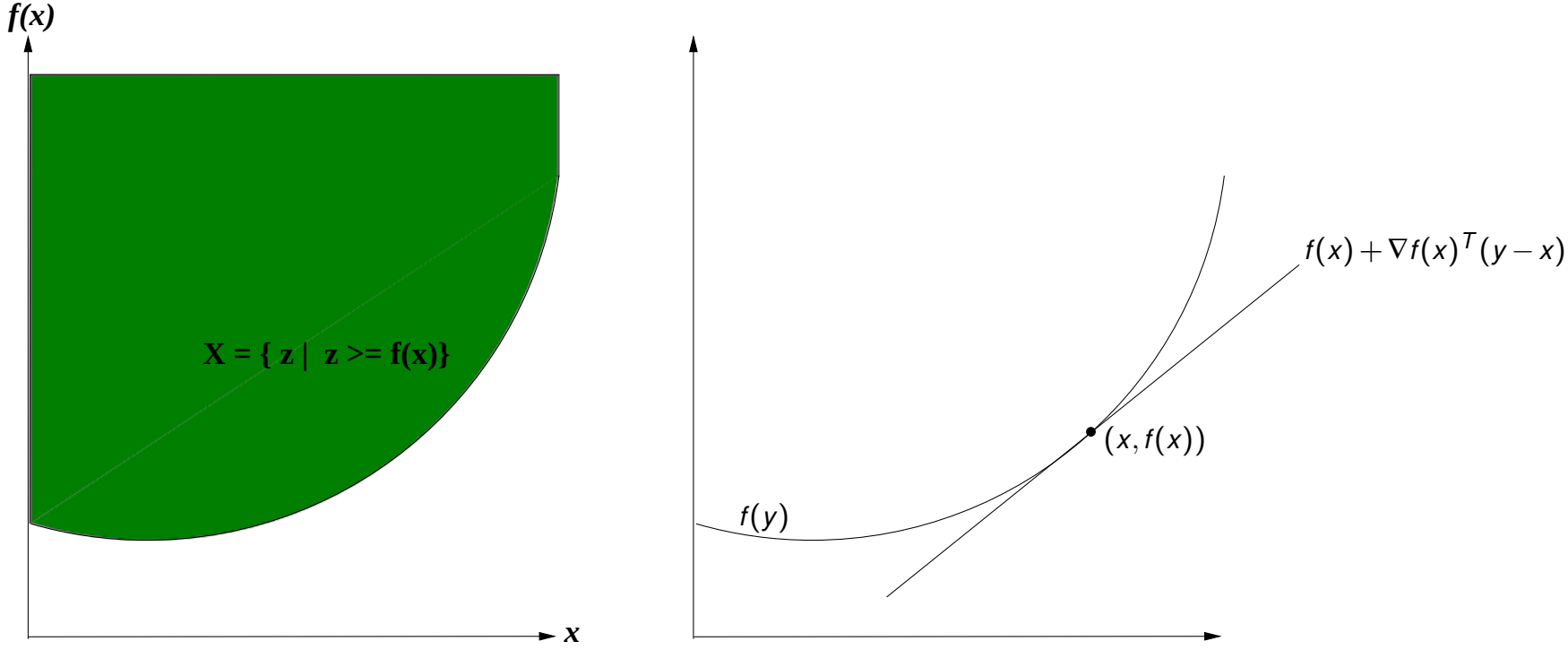


- A set $X \subseteq \mathbb{R}^n$ is said to be *convex* if, for all $x, y \in X$ and all $t \in [0,1]$ we have $z \in X$ where z is the point

$$z := (1-t)\,x + t\,y.$$

- Note that this means that we can travel along a line segment from one point in the set to any other point in the set without ever leaving X.

Convex function



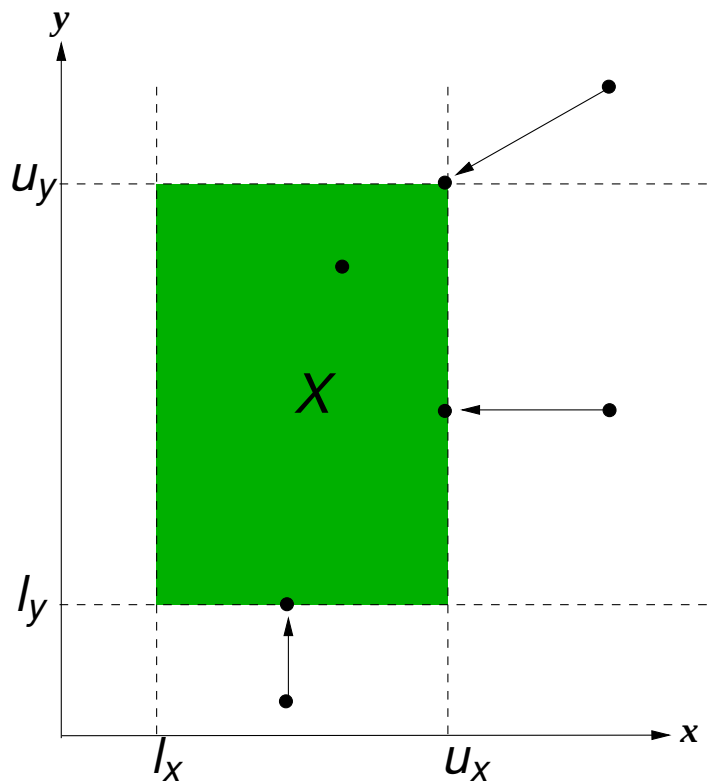
Necessary and sufficient condition for (differentiable) convex functions:

$$f(y) \geq f(x) + \nabla f(x)^T (y - x),$$

where $\nabla f(x)$ is the gradient of f at x. Note that this implies that a local optimum x^* (with $\nabla f(x^*) = 0$) is always a global optimum.

Euclidean projection

- We assume that X itself is a convex non-empty set.
- In this context it also often useful to consider the *projection* $[x']^+$ of a point $x' \in \mathbb{R}^n$ onto X, which is the closest point to x' that is in X.
- Assume for simplicity $X = \{x \mid l_j \leq x_j \leq u_j \text{ for all } j\}$ ("box constraints").



We can explicitly give the projection as

$$[x']_j^+ = \begin{cases} l_j & \text{if } x'_j \leq l_j, \\ u_j & \text{if } x'_j \geq u_j, \\ x'_j & \text{otherwise.} \end{cases}$$

Projected gradient method

- A popular and simple method that often serves as a first candidate for problems with differentiable objective functions is the *projected gradient method* (a.k.a. *steepest descent*).
- Basic idea: similar to local search methods for discrete optimization, the method maintains a feasible current solution and performs “small updates” in iterations.
- Search guided by the gradient of the objective function, which is a good approximation within the close neighborhood of the current solution.
- Note: the objective function f decreases along the direction of $-\nabla f$.
- Particularly useful for “black box” or “ distributed optimization”.

Projected gradient method —cont'd

- Fix an initial feasible solution $x(0) \in X$ and at each iteration $k = 1, 2, \ldots$ do

$$x(k+1) = [x(k) - \delta(k)\nabla f(x(k))]^+,$$

  where $[\cdot]^+$ denotes the projection onto the set X, $\delta(k)$ is a small *step size* and $\nabla f(x(k))$ is the gradient of the objective at $x = x(k)$.

- It can be shown that if the gradient ∇f is Lipschitz continuous and the step sizes are sufficiently small, then fixpoints of this update rule are local optima (see Chapter 2.3 in Bertsekas 1999).

- The resulting method in some sense very similar to steepest-descent local search.

Projected gradient method: example

- Simple example: let

$$f(x_1, x_2) = \log(e^{2x_1+1} + e^{x_2})$$
$$\text{and} \quad X = \{(x_1, x_2) \mid x_1, x_2 \geq 0\}$$

- For the gradient we have

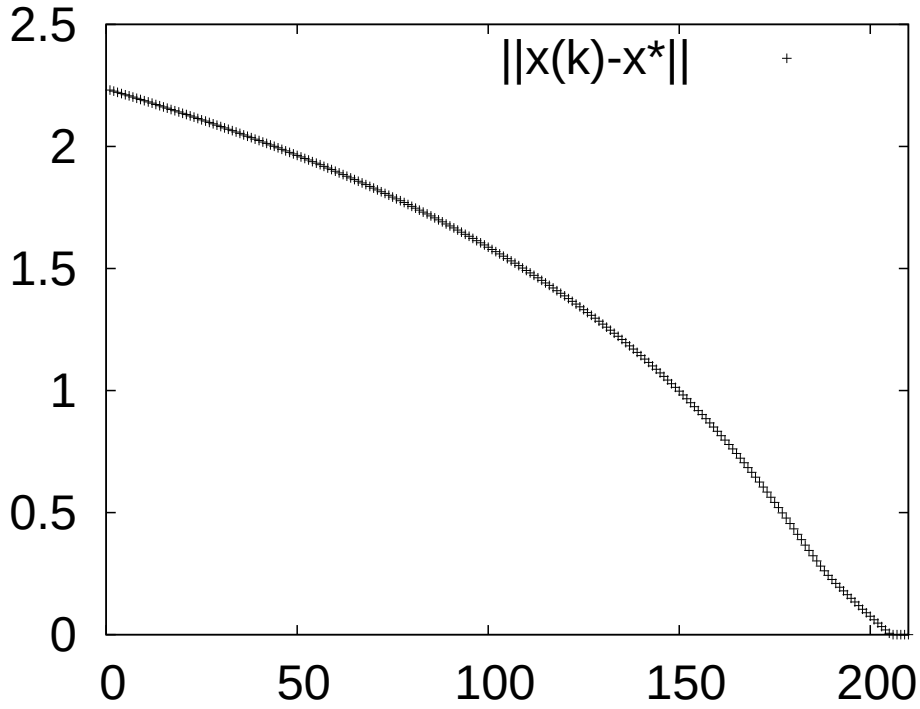
$$\nabla f(x_1, x_2) = \frac{1}{e^{2x_1+1} + e^{x_2}} \begin{pmatrix} 2\, e^{2x_1+1} \\ e^{x_2} \end{pmatrix}$$

- Fix step size $\delta(k) = \delta$, so the method for this case becomes:

$$x_1(k+1) = [x_1(k) - \frac{\delta}{e^{2x_1+1} + e^{x_2}} * 2\, e^{2x_1+1}]^+$$
$$x_2(k+1) = [x_2(k) - \frac{\delta}{e^{2x_1+1} + e^{x_2}} * e^{x_2}]^+$$

- Note: this function is convex and has a (global) minimum at $x^* = (0, 0)$ with $f(x^*) \approx 1.3133$.

Projected gradient method: example —cont'd



- Plot shows distance to optimal over the number of iterations.
- Here we have chosen: $x(0) = (2, 1)$ and $\delta(k) = 0.05$ for all k (based on experimentation).

Step-size rules

There are quite a few rules for choosing step size, for example:

- *constant step size*: $\delta(k) = \delta$; simple but if set too large does not allow convergence, when set to small leads to very slow convergence.
- *diminishing step size*: choose step sizes so that

$$\delta(k) \to 0, \quad \sum_{k=0}^{\infty} \delta(k) = \infty$$

  may lead to slow convergence but offers good theoretical convergence properties. Example: $\delta(k) = 1/k$.
- *minimization rule*: minimize objective function along the chosen direction,i.e., choose $\delta(k)$ such that

$$f(x(k) - \delta(k)\nabla f(x(k))) = \min_{\delta \geq 0} f(x(k) - \delta \nabla f(x(k)).$$

Linesearch

- Some step-size rules may lead to slow convergence or even divergence.
- The minimization rule (or restricted versions limiting the search to some small interval) are good candidates. However, it may be computationally intensive to solve this problem at each iteration.
- One popular alternative is the so-called *backtracking linesearch*.
- Its approach is based on the idea of starting at a large value (e.g., 1) and decreasing the step-size until improvement is observed.
- Formulated more generally for any *descent direction* $\Delta x(k)$ (note: earlier we had $\Delta x(k) := -\nabla f(x(k))$) that satisfies

$$\nabla f(x(k))^T \Delta x(k) < 0.$$

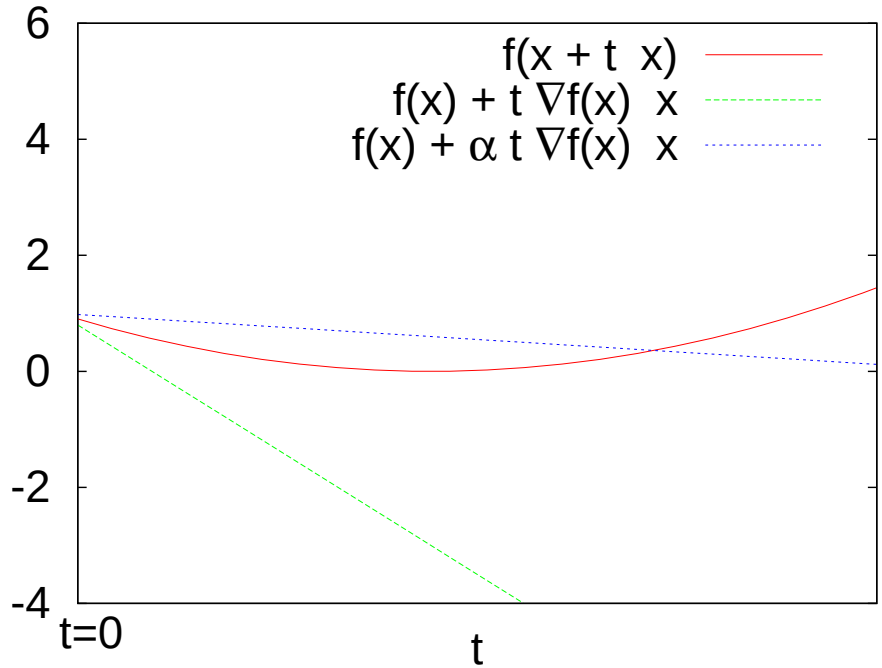
Backtracking line search

procedure BacktrackingLS
choose descent direction $\Delta x(k)$ for f at $x(k) \in X$;
choose $\alpha \in (0, 0.5), \beta \in (0, 1)$;
$t := 1$;
while $f(x(k) + t\Delta x(k)) > f(x(k)) + \alpha\, t\, \nabla f(x(k))^T \Delta x(k)$ **do**
    $t \leftarrow \beta\, t$;
end while.

Note: after termination of BacktrackingLS we set

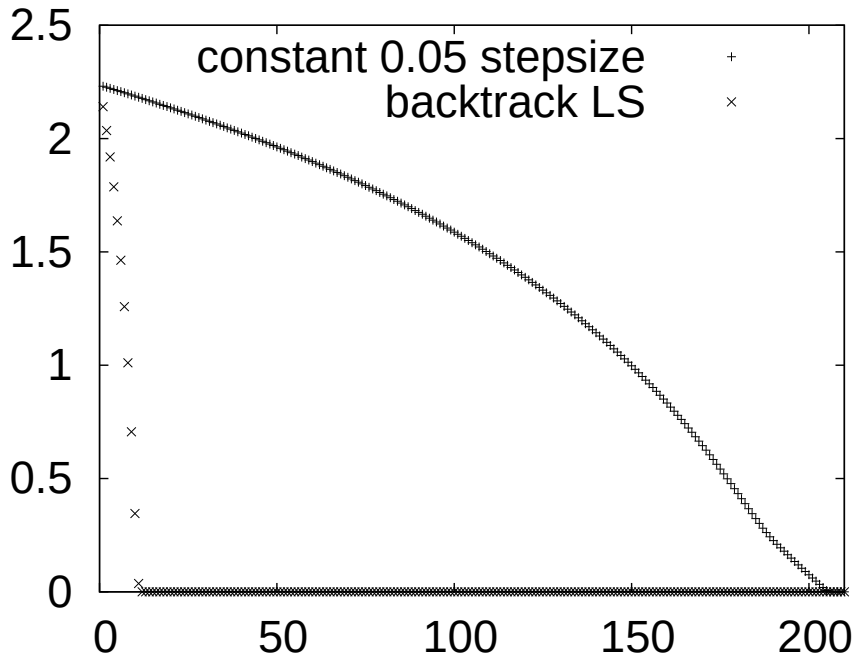
$$\delta(k) \leftarrow t,$$

where t is the final value for the step size that satisfies the termination condition.



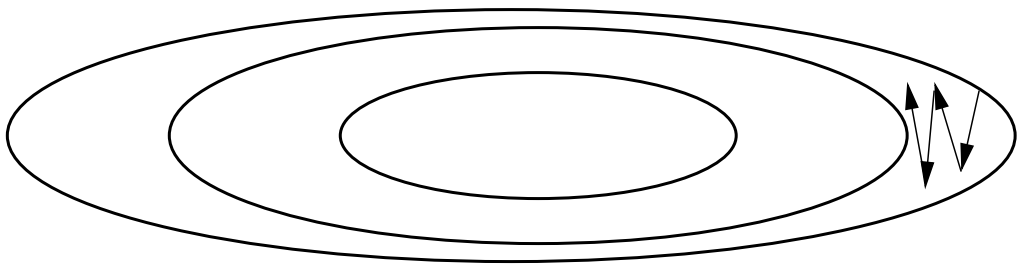
Example —cont'd

- For $\alpha = 0.4$ and $\beta = 0.5$, the projected gradient method applied to the previous example converges within 12 iterations
- Note: here $\Delta x(k) := -\nabla f(x(k))$
- In practice, some fine tuning of these parameters are required



Slow convergence of projected gradient method

- In some cases the convergence can still be slow
- Intuition: the method is prone to exhibit “zig-zaging” behavior in cases when the gradient is close to orthogonal to the direction towards a (local) optimum

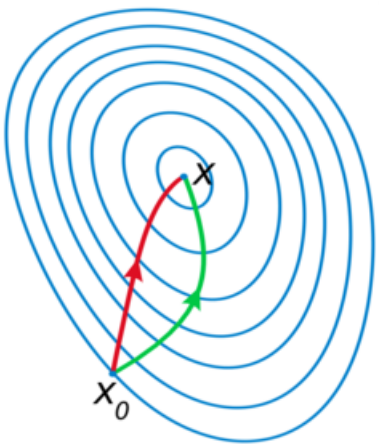


Newton's method

- Another popular method that is typically even faster (in number of iterations) than the previous one is *Newton's method* (a.k.a. *Newton Raphson method*)
- Basic idea: instead of using the original function, try to minimize a local approximation of it (i.e., its second-order Taylor expansion)

$$\hat{f}(x(k)+v) := f(x(k)) + \nabla f(x(k))^T v + \frac{1}{2} v^T \nabla^2 f(x(k)) v,$$

where v is a vector and $\nabla^2 f(x(k))$ is the Hessian at $x(k)$.



Comparison of gradient method (shown in green) and Newton's method (shown in red).

Picture: Wikipedia

Newton's method —cont'd

- Assume for now $X = \mathbb{R}^n$, so no projection to X is required.
- Consider for a fixed $x(k)$ the function $\hat{f}$, which is a quadratic function in v:

$$\hat{f}(x(k)+v) := f(x(k)) + \nabla f(x(k))^T v + \frac{1}{2} v^T \nabla^2 f(x(k)) v.$$

- Determine the gradient w.r.t. v and solve for v, so one obtains

$$v = -\nabla^2 f(x(k))^{-1} \nabla f(x(k)).$$

- Basic idea: move along direction v as determined above until

$$f(x(k)) - \hat{f}(x(k)+v) < \varepsilon,$$

where ε is a "small" positive number.

Newton's method cont.

- Note: for every iteration k we compute the *Newton step*

$$v(k) = -\nabla^2 f(x(k))^{-1} \nabla f(x(k)),$$

  which includes computing the inverse of the Hessian for $x = x(k)$.
- Additionally, to determine the step size $\delta(k)$, one typically performs a line search as earlier shown for the projected gradient method. Then one has the update rule

$$x(k+1) = x(k) + \delta(k)\, v(k).$$

- Although matrix inversion is expensive, the additional effort of the Newton method cmp. to gradient methods usually pays off in faster convergence.
- Under some assumptions one can show strong convergence results (super-linear) for the Newton method (see for example Boyd and Vandenberghe 2004).

Example cont.

- Consider the objective function of the previous example

$$f(x_1, x_2) = \log(e^{2x_1+1} + e^{x_2})$$

- We have for the gradient

$$\nabla f(x_1, x_2) = \frac{1}{e^{2x_1+1} + e^{x_2}} \begin{pmatrix} 2\, e^{2x_1+1} \\ e^{x_2} \end{pmatrix}$$

and for the Hessian

$$\nabla^2 f(x_1, x_2) = \frac{2\, e^{2x_1+x_2+1}}{(e^{2x_1+1} + e^{x_2})^2} \begin{pmatrix} 2 & -1 \\ -1 & 1 \end{pmatrix}$$

Example cont.

- We then obtain

$$\nabla^2 f(x(k))^{-1} = \frac{(e^{2x_1+1} + e^{x_2})^2}{2\, e^{2x_1+x_2+1}} \begin{pmatrix} 1 & 1 \\ 1 & 2 \end{pmatrix}$$

- So the Newton step becomes

$$\begin{aligned} v(k) &= -\nabla^2 f(x(k))^{-1} \nabla f(x(k)) \\ &= -\frac{(e^{2x_1+1} + e^{x_2})^2}{2\, e^{2x_1+x_2+1}} \begin{pmatrix} 1 & 1 \\ 1 & 2 \end{pmatrix} \frac{1}{e^{2x_1+1} + e^{x_2}} \begin{pmatrix} 2\, e^{2x_1+1} \\ e^{x_2} \end{pmatrix} \\ &= -\frac{e^{2x_1+1} + e^{x_2}}{2\, e^{2x_1+x_2+1}} \begin{pmatrix} 2\, e^{2x_1+1} + e^{x_2} \\ 2\, e^{2x_1+1} + 2\, e^{x_2} \end{pmatrix} \end{aligned}$$

Comments regarding Newton method

- Note that in unfortunate cases the Hessian is singular or close to singular (i.e., cannot be inverted) and thus the Newton method cannot be applied directly.
- One way to mitigate this is to consider the slightly modified matrix

$$\nabla^2 f(x) + \varepsilon ID,$$

  where ID is the identity matrix and ε is a small positive constant.
- Further, since matrix inversion is a relatively costly operation, implementations use techniques to offset accuracy by performance (e.g., approximate Hessian, only recompute now and then).

Constrained optimization

- Typically, however, having no constraints (except simple variable domains) is more of an exception.
- Newton's method can be extended to handle equality constraints.
- One method for dealing with constraints is to eliminate them via *Langrangian relaxation*.
- *Interior point methods* are also based on this approach.

Lecture 12: Advanced topics

Outline

- Introduction to genetic algorithms

Goal for today: For a given high-level description of a computational optimization problem, learn to devise a genetic algorithm.

Genetic algorithms

- Belong to class of *evolutionary algorithms*, inspired by evolutionary biology (inheritance, mutation, selection,..)
- Evolutionary algorithms more generally are examples of so-called *metaheuristics*, which include local search as another special case.
- GA's are general-purpose "black-box" optimization methods proposed by J. Holland (1975) and K. DeJong (1975).
- Method has attracted lots of interest, but theory is still incomplete and the empirical results inconclusive.
- Main idea: *encode* solutions to an optimization problem and let solutions evolve from one generation to the next.

The basic algorithm

- We consider the so called “simple genetic algorithm”; also many other variations exist.
- Assume we wish to *maximize a utility function c* defined on n-bit binary strings:

$$c : \{0,1\}^n \to \mathbb{R}.$$

  Other types of domains must be encoded into binary strings, which is a nontrivial problem. (Examples later.)
- View each of the candidate solutions $s \in \{0,1\}^n$ as an *individual* or *chromosome*.
- At each stage (*generation*) t the algorithm maintains a *population* of individuals $p_t = (s_1, \ldots, s_m)$.
- The population may contain multiple copies of the same individual.

The Basic Algorithm—cont'd

Three operations defined on populations:

- *selection* $\sigma(p)$ ("survival of the fittest")
- *recombination* $\rho(p)$ ("mating", "crossover")
- *mutation* $\mu(p)$

The *Simple Genetic Algorithm*:

function SGA(σ, ρ, μ):
    $p \leftarrow$ random initial population;
    while p "not converged" **do**
        $p' \leftarrow \sigma(p)$;
        $p'' \leftarrow \rho(p')$;
        $p \leftarrow \mu(p'')$
    end while;
    return p (or "fittest individual" in p).
end.

Selection

- From the current generation, some individuals are selected to form the basis for the next generation (sometimes called *mating pool*); the same individual may be selected multiple times
- Let m be the size of the population.
- Denote by $\Omega = \{0,1\}^n$ the set of all binary strings of length n.
- The selection operator $\sigma : \Omega^m \to \Omega^m$ maps populations probabilistically:

  > Given an individual $s \in p$, the expected number of copies of s in $\sigma(p)$ is proportional to the *fitness* of s in p. The fitness is a function of the utility of s compared to the utilities of other $s' \in p$.

- This class of selection methods is also referred to as *proportional selection*.

Selection—cont'd

Some possible fitness functions $f(s,p)$:

- Relative *utility* ($\Rightarrow$ "canonical GA"):

$$f(s,p) = \frac{c(s)}{\frac{1}{m}\sum_{s' \in p} c(s')} \triangleq \frac{c(s)}{\bar{c}}.$$

- Relative *rank*:

$$f(s,p) = \frac{r(s)}{\frac{1}{m}\sum_{s' \in p} r(s')} = \frac{2}{m+1} \cdot r(s),$$

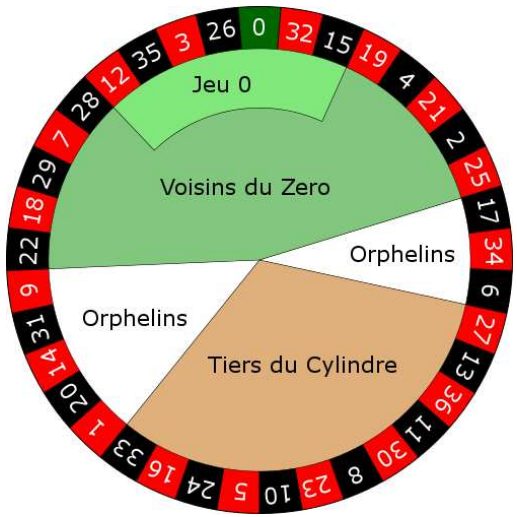
  where $r(s)$ is the rank of s in a worst-to-best ordering of p according to c (worst s_w has $r(s_w) = 1$, best s_b has $r(s_b) = m$).

- Note: for both cases $\sum_{s \in p} f(s,p) = m$.

Selection—cont'd

- There are many ways to perform selection based on fitness.
- A popular one: *Roulette-wheel selection* ("stochastic sampling with replacement")

- Assign to each individual $s \in p$ a probability to be selected in proportion to its fitness value $f(s,p)$. Select m individuals according to this distribution.
- Pictorially: Divide a roulette wheel into m sectors of width proportional to $f(s_1,p),\ldots,f(s_m,p)$. Spin the wheel m times.



Picture: Betzaar / Wikipedia

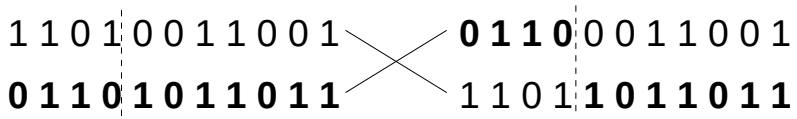
Recombination

- Recall order of operations:
  1. selection 2. recombination 3. mutation.
- Given a population p, choose two individuals $s, s' \in p$ uniformly at random. With probability p_ρ, apply a *crossover operator* $\rho(s, s')$ to produce two new offspring individuals t, t' that replace s, s' in the (new) population (with probability $1 - p_\rho$ parents s, s' remain).
- Repeat the operation $m/2$ times, so that on average each individual participates once. Denote the total effect on the population as $p' = \rho(p)$.
- Practical implementation: choose $\frac{p_\rho}{2} \cdot m$ random pairs from p and apply crossover deterministically (and let the remaining $(1 - p_\rho) \cdot m$ individuals stay unmodified).
- Typically $p_\rho \approx 0.7 \ldots 0.9$.

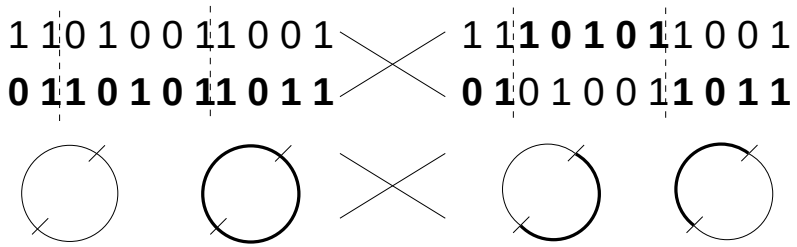
Recombination—cont'd

Possible crossover operators:

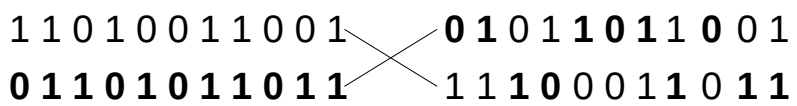
- *1-point crossover:*



- *2-point crossover:*



- *uniform crossover:*



Mutation

- Recall order of operations:
  1. selection 2. recombination 3. mutation.
- Given population p, consider each bit of each individual and flip it with some small probability p_μ. Denote the total effect on the population as $p' = \mu(p)$.
- Typically, $p_\mu \approx 0.001 \ldots 0.01$. Apparently good choice: $p_\mu = 1/n$ for n-bit strings.
- Theoretically mutation is disruptive. Recombination and selection should take care of optimization; mutation is needed only to (re)introduce "lost alleles", alternative values for bits that have the same value in all current individuals.
- In practice mutation + selection = local search. Mutation, even with quite high values of p_μ, can be efficient and is often more important than recombination.

Data Representations

General comments on coding:

- If the function to be optimized is not naturally defined on binary strings, then the domain must be *encoded*. This is a nontrivial task for GA's, because the representation influences the computation.
- Real numbers can be block-coded into sequences of integers.
- For integers, the *Gray code* should be considered as an alternative to the standard binary representation.
- Advantage of Gray code: a transition from integer k to $k+1$ requires only one mutation in Gray code, but may require more in the standard representation.
- Other encodings are possible, e.g., cycles/permutations, trees, graphs ...

Gray code conversion

integer (k)	*standard* ($a_1 a_2 a_3$)	*Gray* ($b_1 b_2 b_3$)
0	000	000
1	001	001
2	010	011
3	011	010
4	100	110
5	101	111
6	110	101
7	111	100

- standard $\rightarrow$ Gray conversion: $b_i = \begin{cases} a_i, & i = 1, \\ a_{i-1} \oplus a_i, & i > 1 \end{cases}$
- Gray $\rightarrow$ standard conversion: $a_i = \bigoplus_{j=1}^{i} b_j$

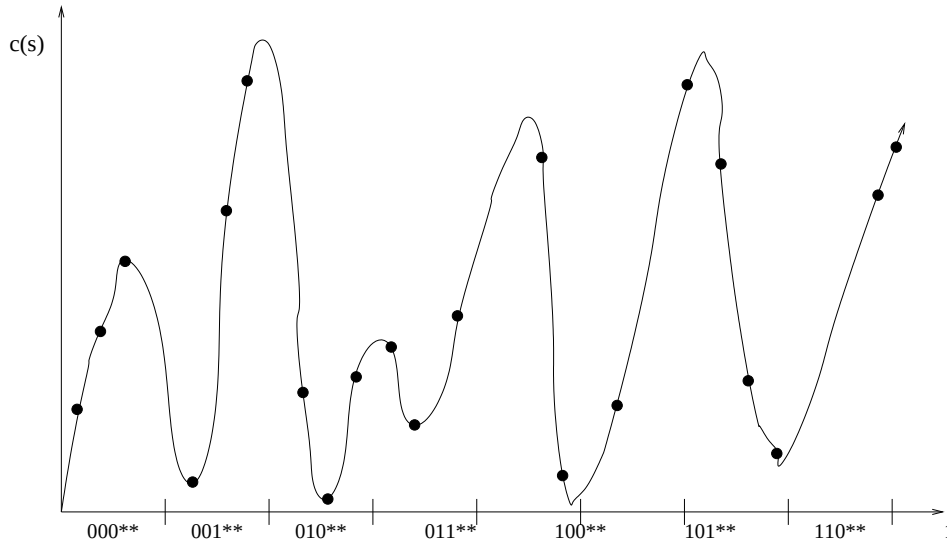
Analysis of GA's

Hyperplane sampling:

- A heuristic view of how a genetic algorithm works.
- A *hyperplane* (actually subcube) is a subset of $\Omega = \{0,1\}^n$, where the values of some bits are fixed and other are free to vary. A hyperplane may be represented by a *schema* $H \in \{0,1,*\}^n$.
- E.g. schema '$0*1**$' represents the 3-dimensional hyperplane (subcube) of $\{0,1\}^5$ where bit 1 is fixed to 0, bit 3 is fixed to 1, and bits 2, 4, and 5 vary.
- Individual $s \in \{0,1\}^n$ *samples* hyperplane H, or *matches* the corresponding schema if the fixed bits of H match the corresponding bits in s. (Denoted $s \in H$.)
- *Note*: given individual generally samples many hyperplanes simultaneously, e.g. individual '101' samples '$10*$', '$1*1$', etc.

Hyperplane sampling

Consider e.g. the following utility function and partition of Ω into hyperplanes (in this case, intervals) of order 3:



Here the current population of 21 individuals samples the hyperplanes so that e.g. '$000**$' and '$010**$' are sampled by three individuals each, and '$100**$' and '$101**$' by two individuals each. Hyperplane '$010**$' has a rather low average fitness in this population, whereas '$111**$' has a rather high average fitness.

Hyperplane sampling—cont'd

- *order* of hyperplane H:

$$o(H) \;=\; \text{number of fixed bits in } H \;=\; n - \dim H$$

- $m(H,p) =$
  number of individuals in population p that sample hyperplane H.
- *average fitness* of hyperplane H in population p:

$$f(H,p) = \frac{1}{m(H,p)} \sum_{s \in H \cap p} f(s,p)$$

Heuristic claim: selection drives the search towards hyperplanes of higher average fitness.

Holland's schema theorem

- By making simplifying assumptions (very large population size) one can make predictions on the short-term evolution of the population (a.k.a. Holland's schema theorem).
- The formula leads to so-called *"Building Block Hypothesis"*:

  > In a genetic search, short, above-average-fitness schemata of low order (*"building blocks"*) receive an exponentially increasing representation in the population.

- Please see more details on the schema theorem in the slides (those marked with "(*)" are not exam relevant).

The effect of crossover on schemata (*)

- Consider a schema such as

$$H = **\underbrace{11**01*1}_{\Delta(H)=7}**$$

  and assume that it is represented in the current population by some $s \in H$.
- If s participates in a crossover operation and the crossover point is located between bit positions 3 and 10, then with large probability the offspring are no longer in H (H is *disrupted*).
- On the other hand, if the crossover point is elsewhere, then one of the offspring stays in H (H is *retained*).

The effect of crossover on schemata—cont'd (*)

- Generally, the probability that in 1-point crossover a schema $H = \{0, 1, *\}^n$ is retained, is (ignoring the possibility of "lucky combinations")

$$\Pr(\text{retain } H) \approx 1 - \frac{\Delta(H)}{n-1},$$

  where $\Delta(H)$ is the *defining length* of H, i.e. the distance between the first and last fixed bit in H.

- More precisely, if H has $m(H, p)$ representatives in population p of total size m:

$$\Pr(\text{retain } H) \geq 1 - \frac{\Delta(H)}{n-1} P_{\text{diff}}, \quad P_{\text{diff}} \leq 1 - \frac{m(H,p)}{m}$$

The Schema "Theorem" (*)

Heuristic estimate of the changes in representation of a given schema H from one generation to the next. Proposed by J. Holland (1975).

Denote:

$$m(H,t) = \text{number of individuals in population at generation } t \text{ that sample } H.$$

Then:

Recall: Selection $\rightarrow$ Recombination $\rightarrow$ Mutation

(i) Effect of selection:

$$m(H,t') \approx m(H,t) \cdot f(H)$$

The Schema "Theorem"—cont'd (*)

(ii) Effect of recombination:

$$m(H,t'') \approx (1-p_\rho)m(H,t') + p_\rho \left(m(H,t')\Pr(\text{retain } H) + \underbrace{m \cdot \Pr(\text{luck})}_{\geq 0} \right)$$

$$\geq (1-p_\rho)m(H,t') + p_\rho m(H,t') \left(1 - \frac{\Delta(H)}{n-1}\left(1 - \frac{m(H,t')}{m}\right)\right)$$

$$= m(H,t') \left(1 - p_\rho \frac{\Delta(H)}{n-1}\left(1 - \frac{m(H,t')}{m}\right)\right)$$

(iii) Effect of mutation:

$$m(H,t+1) \approx m(H,t'') \cdot (1-p_\mu)^{o(H)}$$

The Schema "Theorem"—cont'd (*)

In summary, then:

$$m(H,t+1) \gtrsim m(H,t) \cdot f(H) \cdot \left(1 - p_\rho \frac{\Delta(H)}{n-1}\left(1 - \frac{m(H,t')}{m}\right)\right) \cdot (1-p_\mu)^{o(H)}$$

The formula leads to so called *"Building Block Hypothesis"*:

- In a genetic search, short, above-average fitness schemata of low order ("building blocks") receive an exponentially increasing representation in the population.

The Schema "Theorem": Criticisms (*)

- Many of the approximations used in deriving the "Schema Theorem" implicitly assume that the population is very large. In particular, it is assumed that all the relevant schemata are well sampled. This is clearly not possible in practice, because there are 3^n schemata of length n.
- The result cannot be used to predict the development of the population for much more than one generation:
  1. the long-term development depends on the coevolution of the schemata, and the "theorem" considers only one schema in isolation;
  2. an "exponential growth" cannot continue for long in a finite population.
- Proper treatment: analyze the genetic search as a stochastic process (Markov chain). This is unfortunately very difficult.