Exact Optimization in Spin-Glass Physics

Frauke Liers

Department of Computer Science, University of Cologne

Spring School Leipzig 2008

Outline

Outline

1. Spin Glasses
2. Complexity Theory in a Nutshell
3. Branch-and-Bound
4. Branch-and-Cut for Ising Spin Glasses
5. Branch-and-Cut for Potts Spin Glasses

Outline

Outline

1. Spin Glasses
2. Complexity Theory in a Nutshell
3. Branch-and-Bound
4. Branch-and-Cut for Ising Spin Glasses
5. Branch-and-Cut for Potts Spin Glasses

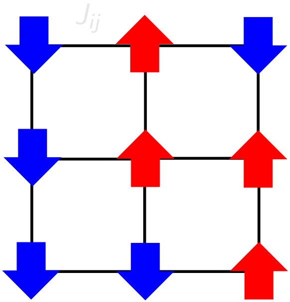
Outline

Spin Glasses

e.g. $Rb_2Cu_{1-x}Co_xF_4$
experiments (Cannella & Mydosh 1972) reveal:
at low temperatures: $\rightarrow$ phase transition spin glass state

Edwards Anderson Model (1975)

- short-range model
- interactions randomly chosen
  - $J_{ij} \in \{+1, -1\}$ or
  - Gaussian distributed
- $H(S) = -\sum_{<i,j>} J_{ij} S_i S_j$, with spin variables S_i

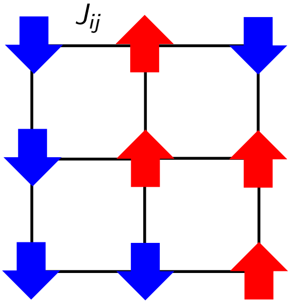


ground state: $\min\{H(\underline{S}) \mid \underline{S}$ is spin configuration$\}$

Spin Glasses

e.g. $Rb_2Cu_{1-x}Co_xF_4$
experiments (Cannella & Mydosh 1972) reveal:
at low temperatures: $\rightarrow$ phase transition spin glass state
Edwards Anderson Model (1975)

- short-range model
- interactions randomly chosen
  - $J_{ij} \in \{+1, -1\}$ or
  - Gaussian distributed
- $H(S) = -\sum_{<i,j>} J_{ij} S_i S_j$, with spin variables S_i



ground state: $\min\{H(\underline{S}) \mid \underline{S}$ is spin configuration$\}$

Why Exact Ground States?

in contrast to Monte-Carlo Simulations or genetic algorithms:
we want to compute **exact solutions**
disadvantages of heuristic methods:

- might become stuck in local minima and do not know anything about the quality of the solution $\longrightarrow$ physical analysis might be biased.
- configurations with almost the same energy might be very different $\longrightarrow$ not clear whether results of spin-spin correlation functions can yield relevant information.

Why Exact Ground States?

in contrast to Monte-Carlo Simulations or genetic algorithms: we want to compute **exact solutions**
disadvantages of heuristic methods:

- might become stuck in local minima and do not know anything about the quality of the solution → physical analysis might be biased.
- configurations with almost the same energy might be very different → not clear whether results of spin-spin correlation functions can yield relevant information.

Why Exact Ground States?

in contrast to Monte-Carlo Simulations or genetic algorithms: we want to compute **exact solutions**
disadvantages of heuristic methods:

- might become stuck in local minima and do not know anything about the quality of the solution $\longrightarrow$ physical analysis might be biased.
- configurations with almost the same energy might be very different $\longrightarrow$ not clear whether results of spin-spin correlation functions can yield relevant information.

Cologne Spin-Glass Ground-State Server

- An exact ground state can be computed via our server on the web
  `www.informatik.uni-koeln.de`
- submit jobs via a command-line client (or via www-interface) and get exact results via email. (will be extended in the future.)
- current focus: compute exact results for **hard** instances of the problem.
  - 2d spin glasses
  - 3d spin glasses
  - SK spin glasses

Cologne Spin-Glass Ground-State Server

- An exact ground state can be computed via our server on the web
  `www.informatik.uni-koeln.de`
- submit jobs via a command-line client (or via www-interface) and get exact results via email. (will be extended in the future.)
- current focus: compute exact results for **hard** instances of the problem.
  - 2d spin glasses
  - 3d spin glasses
  - SK spin glasses

‘This is a Hard Problem’ means...

- NP-hard, i.e. we cannot expect to find an algorithm that solves it in time growing polynomial in the size of the input
- e.g., $2d$ Ising spin glasses with an external field or $3d$ lattices
- whereas $2d$, no field, free boundaries: ‘easy’

more details later.

'This is a Hard Problem' means...

- NP-hard, i.e. we cannot expect to find an algorithm that solves it in time growing polynomial in the size of the input
- e.g., $2d$ Ising spin glasses with an external field or $3d$ lattices
- whereas $2d$, no field, free boundaries: 'easy'

more details later.

‘This is a Hard Problem’ means...

- NP-hard, i.e. we cannot expect to find an algorithm that solves it in time growing polynomial in the size of the input
- e.g., $2d$ Ising spin glasses with an external field or $3d$ lattices
- whereas $2d$, no field, free boundaries: ‘easy’

more details later.

'This is a Hard Problem' means...

- NP-hard, i.e. we cannot expect to find an algorithm that solves it in time growing polynomial in the size of the input
- e.g., $2d$ Ising spin glasses with an external field or $3d$ lattices
- whereas $2d$, no field, free boundaries: 'easy'

more details later.

Outline

General Concepts

many problems can be formulated on **graphs** $G = (V, E)$ with node set V and edge set E.

Edge weights $c_e \in \mathcal{R}$ might be present.

Examples (from ABC^2's TSPbook):

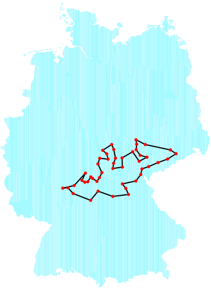


Figure 1.3 The Commis-Voyageur tour in Germany.

many problems can be formulated on **graphs** $G = (V, E)$ with node set V and edge set E.
Edge weights $c_e \in \mathcal{R}$ might be present.
Examples (from ABC^2's TSPbook):

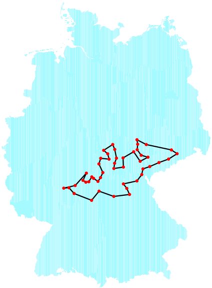


Figure 1.3 The Commis-Voyageur tour in Germany.

General Concepts

many problems can be formulated on **graphs** $G = (V, E)$ with node set V and edge set E.

Edge weights $c_e \in \mathcal{R}$ might be present.

Examples (from ABC^2's TSPbook):

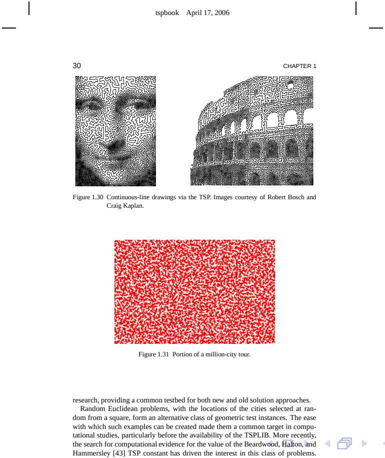


Figure 1.30 Continuous-line drawings via the TSP. Images courtesy of Robert Bosch and Craig Kaplan.

Figure 1.31 Portion of a million-city tour.

research, providing a common testbed for both new and old solution approaches.

Random Euclidean problems, with the locations of the cities selected at random from a square, form an alternative class of geometric test instances. The ease with which such examples can be created made them a common target in computational studies, particularly before the availability of the TSPLIB. More recently, the search for computational evidence for the value of the Beardwood, Halton, and Hammersley [43] TSP constant has driven the interest in this class of problems.

General Concepts

many problems can be formulated on **graphs** $G = (V, E)$ with node set V and edge set E.
Edge weights $c_e \in \mathcal{R}$ might be present.
Examples (from ABC^2's TSPbook):

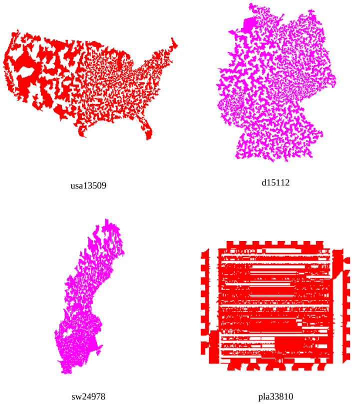


Figure 1.47 Optimal tours.

General Concepts

many problems can be formulated on **graphs** $G = (V, E)$ with node set V and edge set E.
Edge weights $c_e \in \mathcal{R}$ might be present.
Examples:

- shortest paths in networks
- shortest traveling salesman tour (TSP)
- etc.

Some of them are 'easy', some 'hard'.
...in more detail:

Complexity of Decision Problems

A problem $\mathcal{P}$ is a **decision problem**, if the set of all instances $I_{\mathcal{P}}$ of $\mathcal{P}$ is partitioned into the 'yes' and the 'no'-instances. For each instance we ask: Is it a 'yes'- or a 'no'-instance?

Complexity of Decision Problems

A problem $\mathcal{P}$ is a **decision problem**, if the set of all instances $I_{\mathcal{P}}$ of $\mathcal{P}$ is partitioned into the 'yes' and the 'no'-instances. For each instance we ask: Is it a 'yes'- or a 'no'-instance?

Complexity of Decision Problems

Example

1. instance: $n \in \mathcal{N}$
   question: Is n a prime number?
2. instance: graph $G = (V, E)$
   question: Is G connected?

A decision problem is solved by an algorithm $\mathcal{A}$, if $\mathcal{A}$ for each instance terminates and gives the correct 'yes' or 'no' answer.

Complexity of Decision Problems

Example

1. instance: $n \in \mathcal{N}$
   question: Is n a prime number?
2. instance: graph $G = (V, E)$
   question: Is G connected?

A decision problem is solved by an algorithm $\mathcal{A}$, if $\mathcal{A}$ for each instance terminates and gives the correct 'yes' or 'no' answer.

Complexity of Decision Problems

Example

1. instance: $n \in \mathcal{N}$
   question: Is n a prime number?
2. instance: graph $G = (V, E)$
   question: Is G connected?

A decision problem is solved by an algorithm $\mathcal{A}$, if $\mathcal{A}$ for each instance terminates and gives the correct 'yes' or 'no' answer.

Complexity Class P

Contains all decision problems $\mathcal{P}$ for which there exists a solution algorithm that solves $\mathcal{P}$ within a time polynomially bounded in the size needed to store the input, i.e., number of bits.

polynomially bounded?

- assume: each elementary algorithmic step (summation, assignment, etc.) has cost 1.
- Let input for some problem be stored by m bits. (e.g., for $n \in \mathcal{N}$, $m = \log n$ bits are needed.)
- A decision problem is in P, if the number of steps in the solution algorithm asymptotically only grows as fast as m^k, with $k \in \mathcal{N}$.

Complexity Class P

Contains all decision problems $\mathcal{P}$ for which there exists a solution algorithm that solves $\mathcal{P}$ within a time polynomially bounded in the size needed to store the input, i.e., number of bits.

polynomially bounded?

- assume: each elementary algorithmic step (summation, assignment, etc.) has cost 1.
- Let input for some problem be stored by m bits. (e.g., for $n \in \mathcal{N}$, $m = \log n$ bits are needed.)
- A decision problem is in P, if the number of steps in the solution algorithm asymptotically only grows as fast as m^k, with $k \in \mathcal{N}$.

Example

1. instance: $n \in \mathcal{N}$
   question: Is n a prime number? Is in P (recent result by Agrawal et al. from 2002).
2. instance: graph $G = (V, E)$
   question: Is G connected?. Is in P. (breadth-first search)

Example

1. instance: $n \in \mathcal{N}$
   question: Is n a prime number? Is in P (recent result by Agrawal et al. from 2002).
2. instance: graph $G = (V, E)$
   question: Is G connected?. Is in P. (breadth-first search)

Complexity Class NP

NP: nondeterministic polynomial
A decision problem is in NP, if for an arbitrary instance and for a given (possible) solution it can be **verified** in polynomial time whether the solution yields a 'yes' or a 'no' answer.
Example:

- for boolean variables $x_1, \ldots, x_n$: given a clause $x_1 \vee \neg x_2 \ldots x_k \wedge x_2 \vee x_4 \ldots x_l$. Is there a true/false assignment of values to the variables satisfying the clause? (SAT)
- given a graph $G = (V, E)$ with edge weights c_e, and $k \in \mathcal{N}$. Is there a tour that visits every node exactly once with length $\leq k$? (TSP)

Here, a candidate solution can be verified by insertion.

Complexity Class NP

NP: nondeterministic polynomial

A decision problem is in NP, if for an arbitrary instance and for a given (possible) solution it can be **verified** in polynomial time whether the solution yields a 'yes' or a 'no' answer.

Example:

- for boolean variables $x_1, \ldots, x_n$: given a clause $x_1 \vee \neg x_2 \ldots x_k \wedge x_2 \vee x_4 \ldots x_l$. Is there a true/false assignment of values to the variables satisfying the clause? (SAT)
- given a graph $G = (V, E)$ with edge weights c_e, and $k \in \mathcal{N}$. Is there a tour that visits every node exactly once with length $\leq k$? (TSP)

Here, a candidate solution can be verified by insertion.

Complexity Class NP

NP: nondeterministic polynomial
A decision problem is in NP, if for an arbitrary instance and for a given (possible) solution it can be **verified** in polynomial time whether the solution yields a 'yes' or a 'no' answer.
Example:

- for boolean variables $x_1, \ldots, x_n$: given a clause $x_1 \vee \neg x_2 \ldots x_k \wedge x_2 \vee x_4 \ldots x_l$. Is there a true/false assignment of values to the variables satisfying the clause? (SAT)
- given a graph $G = (V, E)$ with edge weights c_e, and $k \in \mathcal{N}$. Is there a tour that visits every node exactly once with length $\leq k$? (TSP)

Here, a candidate solution can be verified by insertion.

Complexity Class NP

NP: nondeterministic polynomial
A decision problem is in NP, if for an arbitrary instance and for a given (possible) solution it can be **verified** in polynomial time whether the solution yields a 'yes' or a 'no' answer.
Example:

- for boolean variables $x_1, \ldots, x_n$: given a clause $x_1 \vee \neg x_2 \ldots x_k \wedge x_2 \vee x_4 \ldots x_l$. Is there a true/false assignment of values to the variables satisfying the clause? (SAT)
- given a graph $G = (V, E)$ with edge weights c_e, and $k \in \mathcal{N}$. Is there a tour that visits every node exactly once with length $\leq k$? (TSP)

Here, a candidate solution can be verified by insertion.

Complexity Class *NP*

NP: nondeterministic polynomial
A decision problem is in *NP*, if for an arbitrary instance and for a given (possible) solution it can be **verified** in polynomial time whether the solution yields a 'yes' or a 'no' answer.
Example:

- for boolean variables $x_1, \ldots, x_n$: given a clause $x_1 \vee \neg x_2 \ldots x_k \wedge x_2 \vee x_4 \ldots x_l$. Is there a true/false assignment of values to the variables satisfying the clause? (SAT)
- given a graph $G = (V, E)$ with edge weights c_e, and $k \in \mathcal{N}$. Is there a tour that visits every node exactly once with length $\leq k$? (TSP)

Here, a candidate solution can be verified by insertion.

Complexity Class *NP*

name *NP*: alternative classification can be given in which constructive nondetermininistic algorithms need to have polynomial running time. In the latter, guessing steps are allowed.

- obviously: $P \subseteq NP$.
- core problem in theoretical computer science: $P \stackrel{?}{=} NP$.

Complexity Class NP

name NP: alternative classification can be given in which constructive nondetermininistic algorithms need to have polynomial running time. In the latter, guessing steps are allowed.

- obviously: $P \subseteq NP$.
- core problem in theoretical computer science: $P \stackrel{?}{=} NP$.

Complexity Class NP

name NP: alternative classification can be given in which constructive nondeterminininistic algorithms need to have polynomial running time. In the latter, guessing steps are allowed.

- obviously: $P \subseteq NP$.
- core problem in theoretical computer science: $P \stackrel{?}{=} NP$.

Complexity Class *NP*

A problem is NP-complete, if it belongs to the 'hardest' problems in NP. Knowing a polynomial-time algorithm for one NP-complete problem would immediately yield polynomial-time algorithms for **all** NP-complete problems.

- SAT was the first problem to be proven to be NP-complete (Cook (1971))
- For many other problems *NP*-completeness was proven since then by reduction to and from other *NP*-complete problems

Examples

$\{0,1\}$-integral solution for inequality system ($\exists 01IP$)

instance: $A \in \mathcal{Z}^{m\times n}$, $b \in \mathcal{Z}^m$

question: Does there exist an $x \in \{0,1\}^n$ with $Ax \geq b$?

$\exists 01IP$ is *NP*-complete.

Complexity Class *NP*

A problem is NP-complete, if it belongs to the 'hardest' problems in NP. Knowing a polynomial-time algorithm for one NP-complete problem would immediately yield polynomial-time algorithms for **all** NP-complete problems.

- SAT was the first problem to be proven to be NP-complete (Cook (1971))
- For many other problems *NP*-completeness was proven since then by reduction to and from other *NP*-complete problems

Examples

$\{0,1\}$-integral solution for inequality system ($\exists 01IP$)

instance: $A \in \mathcal{Z}^{m\times n}$, $b \in \mathcal{Z}^m$

question: Does there exist an $x \in \{0,1\}^n$ with $Ax \geq b$?

$\exists 01IP$ is *NP*-complete.

Complexity Class *NP*

A problem is NP-complete, if it belongs to the 'hardest' problems in NP. Knowing a polynomial-time algorithm for one NP-complete problem would immediately yield polynomial-time algorithms for **all** NP-complete problems.

- SAT was the first problem to be proven to be NP-complete (Cook (1971))
- For many other problems *NP*-completeness was proven since then by reduction to and from other *NP*-complete problems

Examples
$\{0,1\}$-integral solution for inequality system ($\exists 01IP$)
instance: $A \in \mathcal{Z}^{m\times n}$, $b \in \mathcal{Z}^m$
question: Does there exist an $x \in \{0,1\}^n$ with $Ax \geq b$?
$\exists 01IP$ is NP-complete.

Optimization Problems

an optimization problem is characterized by

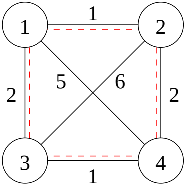
- the set of instances
- information whether we should maximize or minimize
- the set of feasible solutions
- for each instance and each feasible solution, the objective function value of the solution

Example

MINIMUM TRAVELING SALESMAN (TSP)

instance: set of cities $\{1, 2, \ldots, n\}$, distance matrix $D \in \mathcal{Z}_+^{n \times n}$
solution: permutation $\{i_1, i_2, \ldots, i_n\}$ of $\{1, 2, \ldots, n\}$ ('tour' through all cities)
objective function value: $\left(\sum_{k=1}^{n-1} d_{i_k, i_{k+1}}\right) + d_{i_n, i_1}$

$$\begin{array}{c|cccc|} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 1 & 2 & 5 \\ 2 & 1 & 0 & 6 & 2 \\ 3 & 2 & 6 & 0 & 1 \\ 4 & 5 & 2 & 1 & 0 \\ \hline \end{array}$$



optimal tour $\langle 1, 2, 4, 3 \rangle$, cost 6.
The associated decision problems is *NP*-complete.

Example

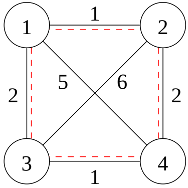
MINIMUM TRAVELING SALESMAN (TSP)

instance: set of cities $\{1, 2, \ldots, n\}$, distance matrix $D \in \mathcal{Z}_+^{n \times n}$

solution: permutation $\{i_1, i_2, \ldots, i_n\}$ of $\{1, 2, \ldots, n\}$ ('tour' through all cities)

objective function value: $\left(\sum_{k=1}^{n-1} d_{i_k, i_{k+1}}\right) + d_{i_n, i_1}$

$$
\begin{array}{c|cccc|}
 & 1 & 2 & 3 & 4 \\
\hline
1 & 0 & 1 & 2 & 5 \\
2 & 1 & 0 & 6 & 2 \\
3 & 2 & 6 & 0 & 1 \\
4 & 5 & 2 & 1 & 0 \\
\hline
\end{array}
$$



optimal tour $\langle 1, 2, 4, 3 \rangle$, cost 6.

The associated decision problems is *NP*-complete.

The Classes NP and NPO

An optimization problem is in *PO*, if there exists a polynomial-time algorithm that for each instance of the problem determines an optimum solution and returns its value.

Example:

- shortest paths
- minimum spanning trees
- matching
- etc.

An optimization problem is in *NPO*, if

- the instances can be recognized in polynomial time
- for all instances the size of a feasible solution is polynomially bounded in the size of the input
- for all y that are polynomially bounded it can be verified in polynomial time whether y is a feasible solution
- the objective function value can be determined in polynomial time for each instance and feasible solution

The Classes NP and NPO

An optimization problem is in *PO*, if there exists a polynomial-time algorithm that for each instance of the problem determines an optimum solution and returns its value.

Example:

- shortest paths
- minimum spanning trees
- matching
- etc.

An optimization problem is in *NPO*, if

- the instances can be recognized in polynomial time
- for all instances the size of a feasible solution is polynomially bounded in the size of the input
- for all y that are polynomially bounded it can be verified in polynomial time whether y is a feasible solution
- the objective function value can be determined in polynomial time for each instance and feasible solution

The Classes NP and NPO

An optimization problem is in *PO*, if there exists a polynomial-time algorithm that for each instance of the problem determines an optimum solution and returns its value.
Example:

- shortest paths
- minimum spanning trees
- matching
- etc.

An optimization problem is in *NPO*, if

- the instances can be recognized in polynomial time
- for all instances the size of a feasible solution is polynomially bounded in the size of the input
- for all y that are polynomially bounded it can be verified in polynomial time whether y is a feasible solution
- the objective function value can be determined in polynomial time for each instance and feasible solution

The Classes NP and NPO

An optimization problem is in *PO*, if there exists a polynomial-time algorithm that for each instance of the problem determines an optimum solution and returns its value.
Example:

- shortest paths
- minimum spanning trees
- matching
- etc.

An optimization problem is in *NPO*, if

- the instances can be recognized in polynomial time
- for all instances the size of a feasible solution is polynomially bounded in the size of the input
- for all y that are polynomially bounded it can be verified in polynomial time whether y is a feasible solution
- the objective function value can be determined in polynomial time for each instance and feasible solution

The Classes NP and NPO

An optimization problem is in *PO*, if there exists a polynomial-time algorithm that for each instance of the problem determines an optimum solution and returns its value.
Example:

- shortest paths
- minimum spanning trees
- matching
- etc.

An optimization problem is in *NPO*, if

- the instances can be recognized in polynomial time
- for all instances the size of a feasible solution is polynomially bounded in the size of the input
- for all y that are polynomially bounded it can be verified in polynomial time whether y is a feasible solution
- the objective function value can be determined in polynomial time for each instance and feasible solution

The Classes NP and NPO

An optimization problem is in *PO*, if there exists a polynomial-time algorithm that for each instance of the problem determines an optimum solution and returns its value.
Example:

- shortest paths
- minimum spanning trees
- matching
- etc.

An optimization problem is in *NPO*, if

- the instances can be recognized in polynomial time
- for all instances the size of a feasible solution is polynomially bounded in the size of the input
- for all y that are polynomially bounded it can be verified in polynomial time whether y is a feasible solution
- the objective function value can be determined in polynomial time for each instance and feasible solution

If the corresponding decision version is NP-complete, the optimization version is called NP-hard.

Example:

- TSP
- minimum SAT
- etc.

If the corresponding decision version is *NP*-complete, the optimization version is called *NP*-hard.
Example:

- TSP
- minimum SAT
- etc.

Our Focus

Martin Weigel has already pointed you to several relevant polynomial optimization problems with applications in physics.

- matching
- flows
- etc.

In the following: we focus on the NP-hard variants.

Our Focus

Martin Weigel has already pointed you to several relevant polynomial optimization problems with applications in physics.

- matching
- flows
- etc.

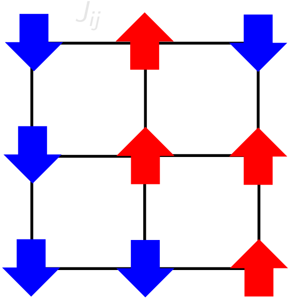
In the following: we focus on the NP-hard variants.

Spin Glasses

e.g. $Rb_2Cu_{1-x}Co_xF_4$
experiments (Cannella & Mydosh 1972) reveal:
at low temperatures: $\rightarrow$ phase transition spin glass state

Edwards Anderson Model (1975)

- short-range model
- interactions randomly chosen
  - $J_{ij} \in \{+1, -1\}$ or
  - Gaussian distributed
- $H(S) = -\sum_{<i,j>} J_{ij} S_i S_j$, with spin variables S_i

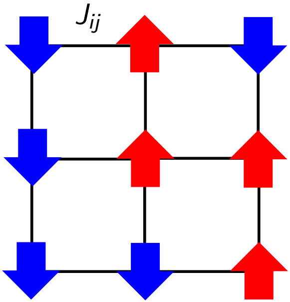


ground state: $\min\{H(\underline{S}) \mid \underline{S}$ is spin configuration$\}$

Spin Glasses

e.g. $Rb_2Cu_{1-x}Co_xF_4$
experiments (Cannella & Mydosh 1972) reveal:
at low temperatures: $\rightarrow$ phase transition spin glass state
Edwards Anderson Model (1975)

- short-range model
- interactions randomly chosen
  - $J_{ij} \in \{+1, -1\}$ or
  - Gaussian distributed
- $H(S) = -\sum_{<i,j>} J_{ij} S_i S_j$, with spin variables S_i

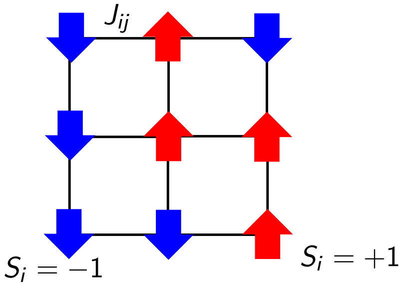


ground state: $\min\{H(\underline{S}) \mid \underline{S}$ is spin configuration$\}$

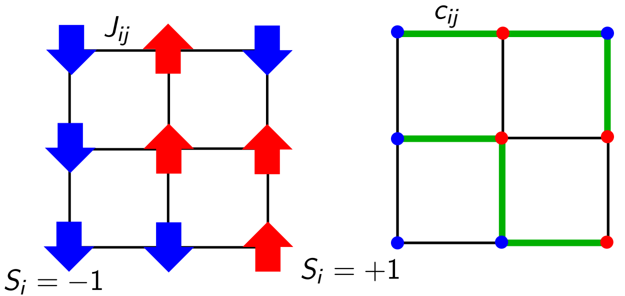
Exact Ground States of Hard Instances

Spinglass	Graph $G=(V, E)$
↑	● node of G
↑–↓	●–● edge of G
coupling J_{ij}	edge weight c_{ij}
configuration	node partition V^+, V^-

Exact Ground States of Hard Instances



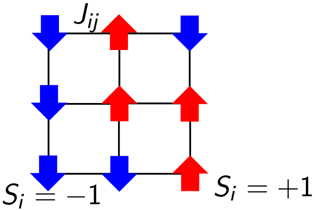
Exact Ground States of Hard Instances

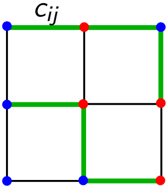


$G = (V, E)$

$H = -\sum_{e \in E} J_{ij} S_i S_j$

Computing Exact Ground States



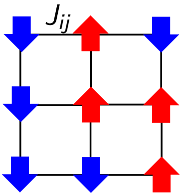


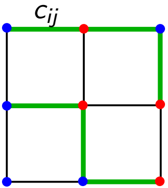
$$H(\underline{S}) + \sum_{(i,j)\in E} J_{ij} = \sum_{(i,j)\in E} J_{ij} \underbrace{(1 - S_i S_j)}$$

$$= \begin{cases} 2 & \text{, if } S_i \neq S_j \\ 0 & \text{, otherwise} \end{cases}$$

$$= 2 \sum_{S_i \neq S_j} J_{ij}$$

Computing Exact Ground States



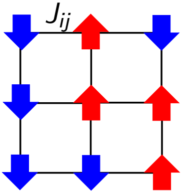


$$H(\underline{S}) + \text{const}$$
$$= 2 \sum_{S_i \neq S_j} J_{ij}$$

$$\text{cut} = \{(i,j) \in E \mid (i,j) = \}$$

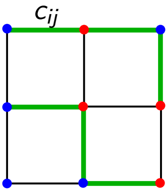
its weight: $\sum_{(i,j)\in \text{cut}} c_{ij}$

Computing Exact Ground States



$$H(\underline{S}) + \text{const} = 2 \sum_{S_i \neq S_j} J_{ij}$$

$\boxed{\text{ground state } \min H(\underline{S})}$



$\text{cut} = \{(i,j) \in E \mid (i,j) = \}$

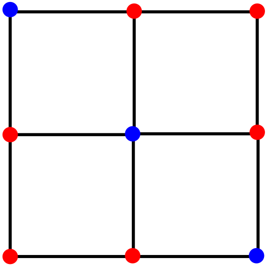
weight $\sum_{(i,j)\in \text{cut}} c_{ij}$

with $c_{ij} = -J_{ij}$:

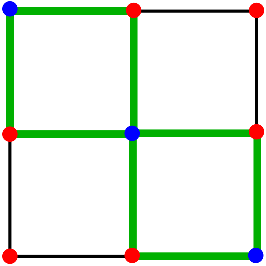
$\boxed{\text{maximum cut in } G}$

NP-hard in general

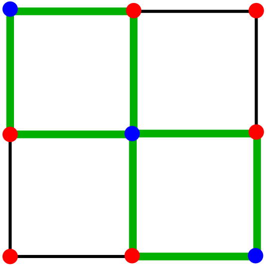
Example

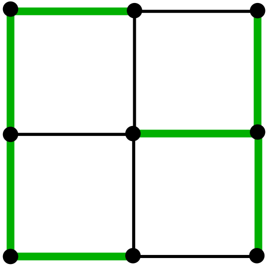


Example

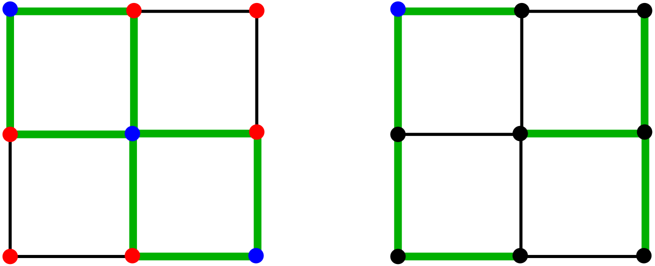


Example

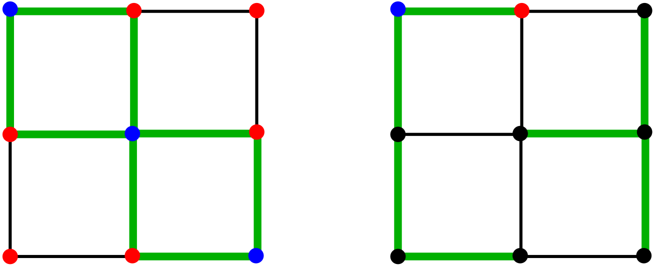




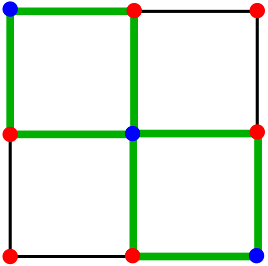
Example

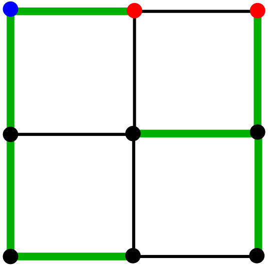


Example

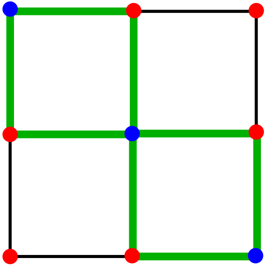


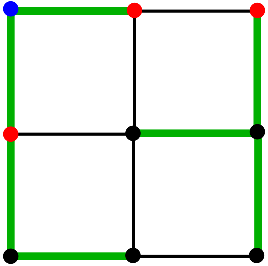
Example



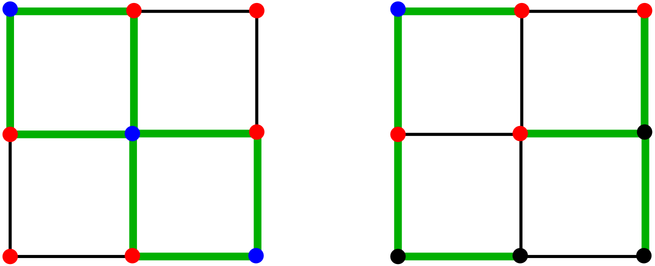


Example

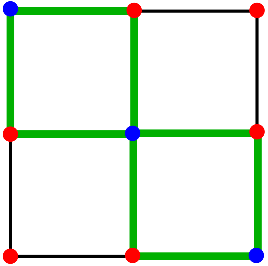


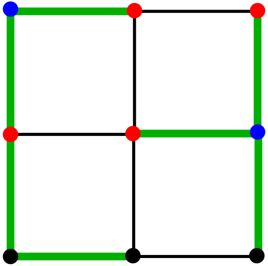


Example

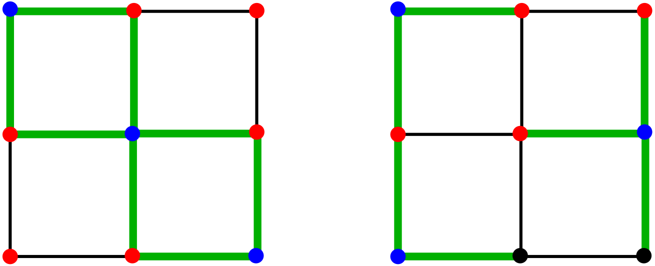


Example

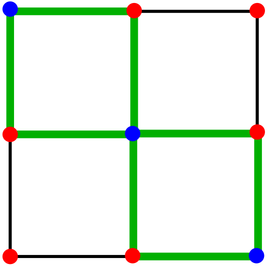


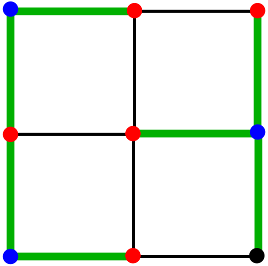


Example

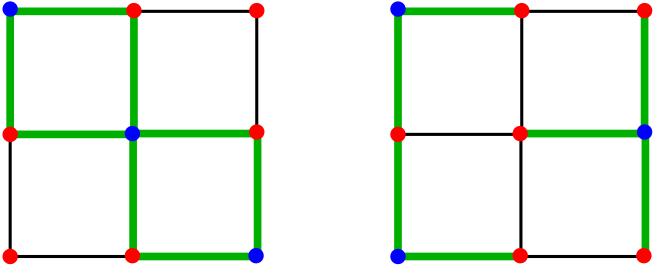


Example





Example



Complexity Status of Maximum Cut

- for general instances NP-hard, i.e. we cannot expect to find an algorithm that solves it in time growing polynomial in the size of the input
- NP-hard for, e.g., $2d$ Ising spin glasses with an external field or $3d$ lattices. more general: on general graphs or on almost planar graphs
- $2d$, no field, free boundaries: polynomial solvable (see M. Weigel's talk)
- Goemans und Williamson found a 0.878-approximation algorithm, i.e.,. a polynomial algorithm, in which the computed solution has a value of at least 0.878 times the value of an optimum cut. However: If P$\neq$NP, there does not exist a polynomial algorithm that computes a solution with value at least 98% of the value of an maximum cut.

Complexity Status of Maximum Cut

- for general instances NP-hard, i.e. we cannot expect to find an algorithm that solves it in time growing polynomial in the size of the input
- NP-hard for, e.g., 2*d* Ising spin glasses with an external field or 3*d* lattices. more general: on general graphs or on almost planar graphs
- 2*d*, no field, free boundaries: polynomial solvable (see M. Weigel's talk)
- Goemans und Williamson found a 0.878-approximation algorithm, i.e.,. a polynomial algorithm, in which the computed solution has a value of at least 0.878 times the value of an optimum cut. However: If P$\neq$NP, there does not exist a polynomial algorithm that computes a solution with value at least 98% of the value of an maximum cut.

Complexity Status of Maximum Cut

- for general instances NP-hard, i.e. we cannot expect to find an algorithm that solves it in time growing polynomial in the size of the input
- NP-hard for, e.g., $2d$ Ising spin glasses with an external field or $3d$ lattices. more general: on general graphs or on almost planar graphs
- $2d$, no field, free boundaries: polynomial solvable (see M. Weigel's talk)
- Goemans und Williamson found a 0.878-approximation algorithm, i.e.,. a polynomial algorithm, in which the computed solution has a value of at least 0.878 times the value of an optimum cut. However: If P$\neq$NP, there does not exist a polynomial algorithm that computes a solution with value at least 98% of the value of an maximum cut.

Complexity Status of Maximum Cut

- for general instances NP-hard, i.e. we cannot expect to find an algorithm that solves it in time growing polynomial in the size of the input
- NP-hard for, e.g., 2d Ising spin glasses with an external field or 3d lattices. more general: on general graphs or on almost planar graphs
- 2d, no field, free boundaries: polynomial solvable (see M. Weigel's talk)
- Goemans und Williamson found a 0.878-approximation algorithm, i.e.,. a polynomial algorithm, in which the computed solution has a value of at least 0.878 times the value of an optimum cut. However: If P$\neq$NP, there does not exist a polynomial algorithm that computes a solution with value at least 98% of the value of an maximum cut.

Applications of Maximum Cut

- quadratic 0-1 optimization: Given $A \in \mathcal{R}^{n \times n}$ and $a \in \mathcal{R}^n$, compute $\min\{x^T A x + a^T x \mid x \in \{0,1\}^n\}$
- separation can also be used in practice for quadratic optimization with additional side constraints
- layout of electronic circuits
- scheduling of sports leagues
- etc.

Applications of Maximum Cut

- quadratic 0-1 optimization: Given $A \in \mathcal{R}^{n \times n}$ and $a \in \mathcal{R}^n$, compute $\min\{x^T A x + a^T x \mid x \in \{0,1\}^n\}$
- separation can also be used in practice for quadratic optimization with additional side constraints
- layout of electronic circuits
- scheduling of sports leagues
- etc.

Applications of Maximum Cut

- quadratic 0-1 optimization: Given $A \in \mathcal{R}^{n \times n}$ and $a \in \mathcal{R}^n$, compute $\min\{x^T A x + a^T x \mid x \in \{0,1\}^n\}$
- separation can also be used in practice for quadratic optimization with additional side constraints
- layout of electronic circuits
- scheduling of sports leagues
- etc.

Applications of Maximum Cut

- quadratic 0-1 optimization: Given $A \in \mathcal{R}^{n \times n}$ and $a \in \mathcal{R}^n$, compute $\min\{x^T Ax + a^T x \mid x \in \{0,1\}^n\}$
- separation can also be used in practice for quadratic optimization with additional side constraints
- layout of electronic circuits
- scheduling of sports leagues
- etc.

Outline

Prerequisites: Optimization Terminology

- **linear optimization problem**: matrix A, vector b

$$\min_x \{c^\top x \mid Ax \leq b, x \geq 0\}$$

- **feasible solutions:** $x \in \mathcal{R}^n$, s.t. $Ax \leq b, x \geq 0$
- X is a **relaxation**, if $\{x \mid Ax \leq b, x \geq 0\} \subseteq X$

- linear optimization problems can be solved within polynomial time (ellipsoid method)
- and fast in practice (simplex algorithm)
- software: CPLEX (ILOG, commercial) or CLP (open source)
- in the following we consider linear optimization problems as a black box

Prerequisites: Optimization Terminology

- **linear optimization problem**: matrix A, vector b

$$\min_{x}\{c^\top x \mid Ax \leq b, x \geq 0\}$$

- **feasible solutions:** $x \in \mathcal{R}^n$, s.t. $Ax \leq b, x \geq 0$
- X is a **relaxation**, if $\{x \mid Ax \leq b, x \geq 0\} \subseteq X$
- linear optimization problems can be solved within polynomial time (ellipsoid method)
- and fast in practice (simplex algorithm)
- software: CPLEX (ILOG, commercial) or CLP (open source)
- in the following we consider linear optimization problems as a black box

Prerequisites: Optimization Terminology

- **linear optimization problem**: matrix A, vector b

$$\min_x \{c^\top x \mid Ax \le b, x \ge 0\}$$

- **feasible solutions:** $x \in \mathcal{R}^n$, s.t. $Ax \le b, x \ge 0$
- X is a **relaxation**, if $\{x \mid Ax \le b, x \ge 0\} \subseteq X$
- linear optimization problems can be solved within polynomial time (ellipsoid method)
- and fast in practice (simplex algorithm)
- software: CPLEX (ILOG, commercial) or CLP (open source)
- in the following we consider linear optimization problems as a black box

Prerequisites: Optimization Terminology

- **linear optimization problem**: matrix A, vector b

$$\min_x \{c^\top x \mid Ax \leq b, x \geq 0\}$$

- **feasible solutions:** $x \in \mathcal{R}^n$, s.t. $Ax \leq b, x \geq 0$
- X is a **relaxation**, if $\{x \mid Ax \leq b, x \geq 0\} \subseteq X$

- linear optimization problems can be solved within polynomial time (ellipsoid method)
- and fast in practice (simplex algorithm)
- software: CPLEX (ILOG, commercial) or CLP (open source)
- in the following we consider linear optimization problems as a black box

Prerequisites: Optimization Terminology

- **linear optimization problem**: matrix A, vector b

$$\min_x \{c^\top x \mid Ax \le b, x \ge 0\}$$

- **feasible solutions:** $x \in \mathcal{R}^n$, s.t. $Ax \le b, x \ge 0$
- X is a **relaxation**, if $\{x \mid Ax \le b, x \ge 0\} \subseteq X$

- linear optimization problems can be solved within polynomial time (ellipsoid method)
- and fast in practice (simplex algorithm)
- software: CPLEX (ILOG, commercial) or CLP (open source)
- in the following we consider linear optimization problems as a black box

Branch & Bound

- standard approach for the solution of NP-hard integer and mixed-integer optimization problems
- can be used for a wide class of problems
- basic idea is very simple
- however: practical usefulness depends strongly on good data structures, clever implementation, etc.

known names for Branch & Bound:

- implicit enumeration
- divide & conquer
- backtracking

ingredients:

- strategy for dividing a problem into sub problems.
- method for calculating upper and lower bounds.

Branch & Bound

- standard approach for the solution of NP-hard integer and mixed-integer optimization problems
- can be used for a wide class of problems
- basic idea is very simple
- however: practical usefulness depends strongly on good data structures, clever implementation, etc.

known names for Branch & Bound:

- implicit enumeration
- divide & conquer
- backtracking

ingredients:

- strategy for dividing a problem into sub problems.
- method for calculating upper and lower bounds.

Branch & Bound

- standard approach for the solution of NP-hard integer and mixed-integer optimization problems
- can be used for a wide class of problems
- basic idea is very simple
- however: practical usefulness depends strongly on good data structures, clever implementation, etc.

known names for Branch & Bound:

- implicit enumeration
- divide & conquer
- backtracking

ingredients:

- strategy for dividing a problem into sub problems.
- method for calculating upper and lower bounds.

Branch & Bound

- standard approach for the solution of NP-hard integer and mixed-integer optimization problems
- can be used for a wide class of problems
- basic idea is very simple
- however: practical usefulness depends strongly on good data structures, clever implementation, etc.

known names for Branch & Bound:

- implicit enumeration
- divide & conquer
- backtracking

ingredients:

- strategy for dividing a problem into sub problems.
- method for calculating upper and lower bounds.

Basic Idea of Branch & Bound

In the following wlog: consider maximization problems.

- start solving the original problem
- bounds through feasible solutions and through relaxations
- in case bounds are equal: optimality proven
- otherwise: divide the problem into subproblems so that the combination of the solutions in the sub problems can be combined to the solutions of the original problem
- solve sub problem through
  1. determination of an optimum solution, or
  2. proof of its infeasibility, or
  3. calculation of an upper bound that is not better than the currently best known solution, or
  4. subdividing the problem into further sub problems.

obviously:

- method terminates correctly in case splitting is done in a reasonable way.

Basic Idea of Branch & Bound

In the following wlog: consider maximization problems.

- start solving the original problem
- bounds through feasible solutions and through relaxations
- in case bounds are equal: optimality proven
- otherwise: divide the problem into subproblems so that the combination of the solutions in the sub problems can be combined to the solutions of the original problem
- solve sub problem through
    1. determination of an optimum solution, or
    2. proof of its infeasibility, or
    3. calculation of an upper bound that is not better than the currently best known solution, or
    4. subdividing the problem into further sub problems.

obviously:

- method terminates correctly in case splitting is done in a reasonable way.

Basic Idea of Branch & Bound

In the following wlog: consider maximization problems.

- start solving the original problem
- bounds through feasible solutions and through relaxations
- in case bounds are equal: optimality proven
- otherwise: divide the problem into subproblems so that the combination of the solutions in the sub problems can be combined to the solutions of the original problem
- solve sub problem through
    1. determination of an optimum solution, or
    2. proof of its infeasibility, or
    3. calculation of an upper bound that is not better than the currently best known solution, or
    4. subdividing the problem into further sub problems.

obviously:

- method terminates correctly in case splitting is done in a reasonable way.

Basic Idea of Branch & Bound

In the following wlog: consider maximization problems.

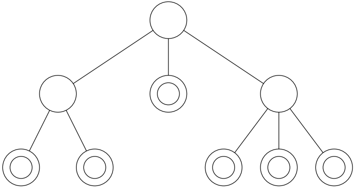
- start solving the original problem
- bounds through feasible solutions and through relaxations
- in case bounds are equal: optimality proven
- otherwise: divide the problem into subproblems so that the combination of the solutions in the sub problems can be combined to the solutions of the original problem
- solve sub problem through
  1. determination of an optimum solution, or
  2. proof of its infeasibility, or
  3. calculation of an upper bound that is not better than the currently best known solution, or
  4. subdividing the problem into further sub problems.

obviously:

- method terminates correctly in case splitting is done in a reasonable way.

Basic Idea of Branch & Bound

associate to the solution process in a natural way a branch & bound-tree:



- the root is the original problem
- a node represents some sub problem
- a direct child of a node u represents a sub problem of u
- tree leafs represent 'solved' problems

BRANCH & BOUND(A, b, c, N_1) (Dakin) for MIP

Input:
mixed-integer problem (MIP) with rational data

$$
(MIP^{=}) \qquad \begin{array}{rrcl} \max & c^T & & \\ & Ax & = & b \\ & x & \geq & 0 \\ & x_i & & \text{integer } \forall i \in N_1 \end{array}
$$

Output: solution of the problem or proof of infeasibility.

BRANCH & BOUND(A, b, c, N_1) (Dakin) for (MIP)

1. initialize the list of active sub problems with the original problem. opt $= -\infty$
2. **while** list of active sub problems not empty **do**
   begin
3. choose from the list of active problems one. 'Solve' it by:
   1. find optimal solution for the sub problem, or
   2. prove that the sub problem does not have a feasible solution, or
   3. prove by using a relaxation (bound) that there does not exist a feasible solution for the sub problem with a higher objective function value than the up to now best known solution (**fathoming**)
   4. if above not possible: **branch**, i.e., divide the problem into further sub problems, add them to list of active problems.

   end
4. if (opt $> -\infty$) **return** best known feasible solution as optimum. otherwise: return 'problem infeasible'.

BRANCH & BOUND(A, b, c, N_1) (Dakin) for (MIP)

1. initialize the list of active sub problems with the original problem. opt $= -\infty$
2. **while** list of active sub problems not empty **do**
   begin
3. choose from the list of active problems one. 'Solve' it by:
   1. find optimal solution for the sub problem, or
   2. prove that the sub problem does not have a feasible solution, or
   3. prove by using a relaxation (bound) that there does not exist a feasible solution for the sub problem with a higher objective function value than the up to now best known solution (**fathoming**)
   4. if above not possible: **branch**, i.e., divide the problem into further sub problems, add them to list of active problems.

   end
4. if (opt $> -\infty$) **return** best known feasible solution as optimum. otherwise: return 'problem infeasible'.

BRANCH & BOUND(A, b, c, N_1) (Dakin) for (MIP)

1. initialize the list of active sub problems with the original problem. opt $= -\infty$
2. **while** list of active sub problems not empty **do**
   begin
3. choose from the list of active problems one. 'Solve' it by:
   1. find optimal solution for the sub problem, or
   2. prove that the sub problem does not have a feasible solution, or
   3. prove by using a relaxation (bound) that there does not exist a feasible solution for the sub problem with a higher objective function value than the up to now best known solution (**fathoming**)
   4. if above not possible: **branch**, i.e., divide the problem into further sub problems, add them to list of active problems.

   end
4. if (opt $> -\infty$) **return** best known feasible solution as optimum. otherwise: return 'problem infeasible'.

Discussion

- important: keep the size of the tree 'small' $\Rightarrow$ need good bounds.
- to 3 (2): if a relaxation of a sub problem is infeasible $\longrightarrow$ sub problem itself is infeasible.
- For fathoming a sub problem in 3 (3): need good upper and lower bounds. Lower bounds: given by feasible solutions calculated by heuristics, or by an optimal solution of a sub problem.
- easiest branching: choose some x_e that needs to be integer, however the optimum in the LP is $x_e^\star \notin \mathbb{Z}$. Replace the current sub problem by two, in one of which one adds the inequality $x_e \leq \lfloor x_e^\star \rfloor$, and in the other $x_e \geq \lceil x_e^\star \rceil$.
- upper bounds given by the values of the **LP-relaxations.**

Discussion

- important: keep the size of the tree 'small' $\Rightarrow$ need good bounds.
- to 3 (2): if a relaxation of a sub problem is infeasible $\rightarrow$ sub problem itself is infeasible.
- For fathoming a sub problem in 3 (3): need good upper and lower bounds. Lower bounds: given by feasible solutions calculated by heuristics, or by an optimal solution of a sub problem.
- easiest branching: choose some x_e that needs to be integer, however the optimum in the LP is $x_e^\star \notin \mathbb{Z}$. Replace the current sub problem by two, in one of which one adds the inequality $x_e \leq \lfloor x_e^\star \rfloor$, and in the other $x_e \geq \lceil x_e^\star \rceil$.
- upper bounds given by the values of the **LP-relaxations.**

Discussion

- important: keep the size of the tree 'small' $\Rightarrow$ need good bounds.
- to 3 (2): if a relaxation of a sub problem is infeasible $\rightarrow$ sub problem itself is infeasible.
- For fathoming a sub problem in 3 (3): need good upper and lower bounds. Lower bounds: given by feasible solutions calculated by heuristics, or by an optimal solution of a sub problem.
- easiest branching: choose some x_e that needs to be integer, however the optimum in the LP is $x_e^\star \notin \mathcal{Z}$. Replace the current sub problem by two, in one of which one adds the inequality $x_e \leq \lfloor x_e^\star \rfloor$, and in the other $x_e \geq \lceil x_e^\star \rceil$.
- upper bounds given by the values of the **LP-relaxations.**

Discussion

- important: keep the size of the tree 'small' $\Rightarrow$ need good bounds.
- to 3 (2): if a relaxation of a sub problem is infeasible $\rightarrow$ sub problem itself is infeasible.
- For fathoming a sub problem in 3 (3): need good upper and lower bounds. Lower bounds: given by feasible solutions calculated by heuristics, or by an optimal solution of a sub problem.
- easiest branching: choose some x_e that needs to be integer, however the optimum in the LP is $x_e^\star \notin \mathcal{Z}$. Replace the current sub problem by two, in one of which one adds the inequality $x_e \leq \lfloor x_e^\star \rfloor$, and in the other $x_e \geq \lceil x_e^\star \rceil$.
- upper bounds given by the values of the **LP-relaxations.**

Discussion

- important: keep the size of the tree 'small' $\Rightarrow$ need good bounds.
- to 3 (2): if a relaxation of a sub problem is infeasible $\rightarrow$ sub problem itself is infeasible.
- For fathoming a sub problem in 3 (3): need good upper and lower bounds. Lower bounds: given by feasible solutions calculated by heuristics, or by an optimal solution of a sub problem.
- easiest branching: choose some x_e that needs to be integer, however the optimum in the LP is $x_e^\star \notin \mathcal{Z}$. Replace the current sub problem by two, in one of which one adds the inequality $x_e \leq \lfloor x_e^\star \rfloor$, and in the other $x_e \geq \lceil x_e^\star \rceil$.
- upper bounds given by the values of the **LP-relaxations.**

Example

Consider

$$\begin{array}{rrrrrrrrrrrl}
\max & -7x_1 & - & 3x_2 & - & 4x_3 & & & & & & \\
 & x_1 & + & 2x_2 & + & 3x_3 & - & x_4 & & & = & 8 \\
 & 3x_1 & + & x_2 & + & x_3 & & & - & x_5 & = & 5 \\
 & & & & & & & & & x_1, x_2, x_3, x_4, x_5 & \geq & 0 \\
 & & & & & & & & & x_1, x_2, x_3, x_4, x_5 & \in & \mathcal{Z}
\end{array}$$

LP-Optimum

$$x_3 = x_4 = x_5 = 0 \ , \ x_1 = \frac{2}{5} \ , \ x_2 = \frac{19}{5}$$

value $c^* = -\frac{71}{5}$ $(= -14.2)$. upper bound: -15
branch on x_2

$$P_1 = P_0 \cap \{x \mid x_2 \leq 3\}$$
$$P_2 = P_0 \cap \{x \mid x_2 \geq 4\}$$

choose P_1 as next problem.

Example

optimum solution of LP-relaxation LP_1 is

$$x_4 = x_5 = 0\ ,\ x_1 = \frac{1}{2}\ ,\ x_2 = 2\ ,\ x_3 = \frac{1}{2}$$

and $c^* = -\frac{29}{2}$ (upper bound -15).
subdivide P_1, get:

$$P_3 = P_1 \cap \{x \mid x_1 \leq 0\}$$
$$P_4 = P_1 \cap \{x \mid x_1 \geq 1\}$$

active problems are $K = \{P_2, P_3, P_4\}$. solving LP_3 gives

$$x_1 = x_5 = 0\ ,\ x_2 = 3\ ,\ x_3 = 2\ ,\ x_4 = 4$$

and $c^* = -17$. then P_3 is solved , best solution (global lower bound) has value -17.

Example

solve P_4, get:

$$x_4 = 0 \ , \ x_1 = 1 \ , \ x_2 = 3 \ , \ x_3 = \frac{1}{3} \ , \ x_5 = \frac{4}{3}$$

and $c^* = -\frac{52}{3} = -17\frac{1}{3}$. The found upper bound -18 is worse than best solution, and P_4 is fathomed.
Need to solve P_2. We get as a solution of the LP-relaxation

$$x_3 = x_5 = 0 \ , \ x_1 = \frac{1}{3} \ , \ x_2 = 4 \ , \ x_4 = \frac{1}{3}$$

and $c^* = -\frac{43}{3}$. P_2 is not yet solved. Branch on x_1

$$P_5 = P_2 \cap \{x \mid x_1 \leq 0\}$$
$$P_6 = P_2 \cap \{x \mid x_1 \geq 1\}$$

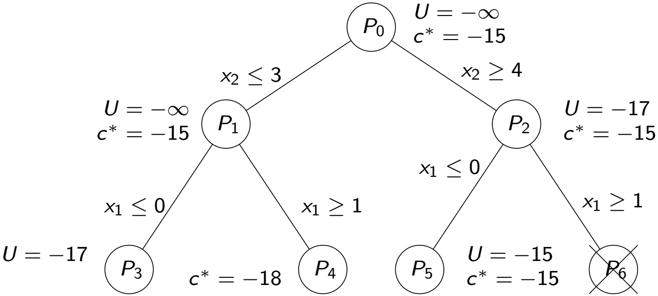
solving LP_5 yields

$$x_1 = x_3 = x_5 = 0 \ , \ x_2 = 5 \ , \ x_4 = 2$$

and $c^* = -15$. This is the new best solution with value -15.
P_5 is then solved.

Example

No need to consider P_6 further, as because of LP_2 no better solution is possible.



Implementation Details

- **branching** can also divide into more than 2 sub problems, or by more complicated inequalities. Good choice of branching variable is important!
- Exploit **logical implications**. E.g. for constraints $\sum_{i \in S} x_i = 1$, $x_i \in \{0, 1\}$: If some $x_j = 1 \rightarrow$ all other variables in S have value 0. $\Rightarrow$ branch with $\sum_{i \in S_1} x_i = 0$ and $\sum_{i \in S_2} x_i = 0$ with $S_1 \cup S_2 = S$
- **primal heuristics**: sometimes it takes long until good feasible solutions are found. $\rightarrow$ additional heuristics
- good strategies for **sub problem selection** strongly influence the total number of sub problems to be solved. It is not easy to devise a strategy that works well for any problem. often used: "'Best first search"'
- natural **parallelisation** possible, as all active nodes can be solved simultaneously.

Implementation Details

- **branching** can also divide into more than 2 sub problems, or by more complicated inequalities. Good choice of branching variable is important!
- Exploit **logical implications**. E.g. for constraints $\sum_{i \in S} x_i = 1$, $x_i \in \{0, 1\}$: If some $x_j = 1 \rightarrow$ all other variables in S have value 0. $\Rightarrow$ branch with $\sum_{i \in S_1} x_i = 0$ and $\sum_{i \in S_2} x_i = 0$ with $S_1 \cup S_2 = S$
- **primal heuristics**: sometimes it takes long until good feasible solutions are found. $\rightarrow$ additional heuristics
- good strategies for **sub problem selection** strongly influence the total number of sub problems to be solved. It is not easy to devise a strategy that works well for any problem. often used: "'Best first search"'
- natural **parallelisation** possible, as all active nodes can be solved simultaneously.

Implementation Details

- **branching** can also divide into more than 2 sub problems, or by more complicated inequalities. Good choice of branching variable is important!
- Exploit **logical implications**. E.g. for constraints $\sum_{i \in S} x_i = 1$, $x_i \in \{0, 1\}$: If some $x_j = 1 \rightarrow$ all other variables in S have value 0. $\Rightarrow$ branch with $\sum_{i \in S_1} x_i = 0$ and $\sum_{i \in S_2} x_i = 0$ with $S_1 \cup S_2 = S$
- **primal heuristics**: sometimes it takes long until good feasible solutions are found. $\rightarrow$ additional heuristics
- good strategies for **sub problem selection** strongly influence the total number of sub problems to be solved. It is not easy to devise a strategy that works well for any problem. often used: "'Best first search"'
- natural **parallelisation** possible, as all active nodes can be solved simultaneously.

Implementation Details

- **branching** can also divide into more than 2 sub problems, or by more complicated inequalities. Good choice of branching variable is important!
- Exploit **logical implications**. E.g. for constraints $\sum_{i \in S} x_i = 1$, $x_i \in \{0, 1\}$: If some $x_j = 1 \rightarrow$ all other variables in S have value 0. $\Rightarrow$ branch with $\sum_{i \in S_1} x_i = 0$ and $\sum_{i \in S_2} x_i = 0$ with $S_1 \cup S_2 = S$
- **primal heuristics**: sometimes it takes long until good feasible solutions are found. $\rightarrow$ additional heuristics
- good strategies for **sub problem selection** strongly influence the total number of sub problems to be solved. It is not easy to devise a strategy that works well for any problem. often used: "'Best first search"'
- natural **parallelisation** possible, as all active nodes can be solved simultaneously.

Implementation Details

- **branching** can also divide into more than 2 sub problems, or by more complicated inequalities. Good choice of branching variable is important!
- Exploit **logical implications**. E.g. for constraints $\sum_{i \in S} x_i = 1$, $x_i \in \{0, 1\}$: If some $x_j = 1 \rightarrow$ all other variables in S have value 0. $\Rightarrow$ branch with $\sum_{i \in S_1} x_i = 0$ and $\sum_{i \in S_2} x_i = 0$ with $S_1 \cup S_2 = S$
- **primal heuristics**: sometimes it takes long until good feasible solutions are found. $\rightarrow$ additional heuristics
- good strategies for **sub problem selection** strongly influence the total number of sub problems to be solved. It is not easy to devise a strategy that works well for any problem. often used: "'Best first search"'
- natural **parallelisation** possible, as all active nodes can be solved simultaneously.

Conclusions

- for hard integer or combinatorial optimization problems, typically, some variant of Branch&Bound is used.
- Modern IP-Solvers, however, can drastically reduce the size of the branch-and-bound tree and therefore go to large system sizes. (e.g., as contained in the Mixed Integer Problem Library MIPLIB that yields a test-bed for solution algorithms)

Conclusions

- for hard integer or combinatorial optimization problems, typically, some variant of Branch&Bound is used.
- Modern IP-Solvers, however, can drastically reduce the size of the branch-and-bound tree and therefore go to large system sizes. (e.g., as contained in the Mixed Integer Problem Library MIPLIB that yields a test-bed for solution algorithms)