Coping with NP-completeness: Special Cases

Alexander S. Kulikov

Steklov Institute of Mathematics at St. Petersburg
Russian Academy of Sciences

Advanced Algorithms and Complexity
Data Structures and Algorithms

The fact that a problem is **NP**-complete does not exclude an efficient algorithm for special cases of the problem.

Outline

1. 2-Satisfiability

2. Independent Sets in Trees

This part

- Striking connection between strongly connected components of a graph and formulas in 2-CNF
- A linear time algorithm for 2-SAT

2-Satisfiability (2-SAT)

Input: A set of clauses, each containing at most two literals (that is, a 2-CNF formula).

Output: Find a satisfying assignment (if exists).

Example

- $(x \vee y)(\overline{z})(z \vee \overline{x})$ is satisfied by $x = 0, y = 1, z = 0$

Example

- $(x \vee y)(\overline{z})(z \vee \overline{x})$ is satisfied by $x = 0, y = 1, z = 0$
- $(x \vee y)(\overline{z})(z \vee \overline{x})(\overline{y})$ is unsatisfiable

Example

- $(x \vee y)(\overline{z})(z \vee \overline{x})$ is satisfied by $x = 0, y = 1, z = 0$
- $(x \vee y)(\overline{z})(z \vee \overline{x})(\overline{y})$ is unsatisfiable
- $(x \vee y)(x \vee \overline{y})(\overline{x} \vee y)(\overline{x} \vee \overline{y})$ is unsatisfiable

- Consider a clause $(\ell_1 \vee \ell_2)$

- Consider a clause $(\ell_1 \vee \ell_2)$
- Essentially, it says that ℓ_1 and ℓ_2 cannot be both equal to 0

- Consider a clause $(\ell_1 \vee \ell_2)$
- Essentially, it says that ℓ_1 and ℓ_2 cannot be both equal to 0
- In other words, if $\ell_1 = 0$, then $\ell_2 = 1$ and if $\ell_2 = 0$, then $\ell_1 = 1$

Definition

Implication is a binary logical operation denoted by $\Rightarrow$ and defined by the following truth table:

x	y	$x \Rightarrow y$
0	0	1
0	1	1
1	0	0
1	1	1

Definition

For a 2-CNF formula, its implication graph is constructed as follows:

- for each variable x, introduce two vertices labeled by x and $\overline{x}$;
- for each 2-clause $(\ell_1 \vee \ell_2)$, introduce two directed edges $\overline{\ell}_1 \to \ell_2$ and $\overline{\ell}_2 \to \ell_1$
- for each 1-clause (ℓ), introduce an edge $\overline{\ell} \to \ell$

Definition

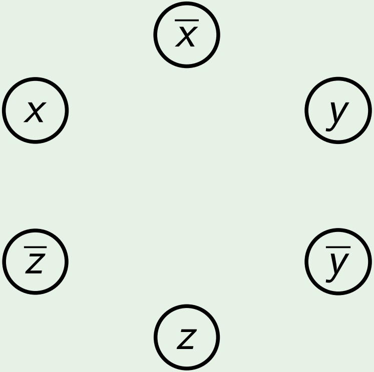
For a 2-CNF formula, its implication graph is constructed as follows:

- for each variable x, introduce two vertices labeled by x and $\overline{x}$;
- for each 2-clause $(\ell_1 \vee \ell_2)$, introduce two directed edges $\overline{\ell}_1 \to \ell_2$ and $\overline{\ell}_2 \to \ell_1$
- for each 1-clause (ℓ), introduce an edge $\overline{\ell} \to \ell$

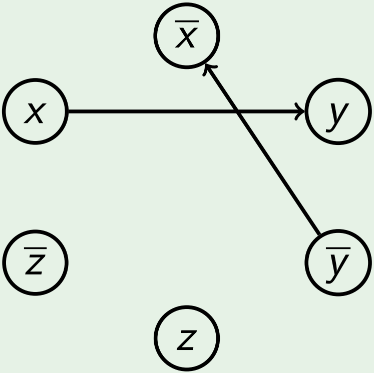
Encodes all implications imposed by the formula.

$$(\overline{x} \vee y)(\overline{y} \vee z)(x \vee \overline{z})(z \vee y)$$

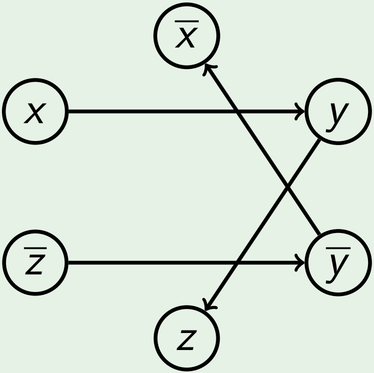
$$(\overline{x} \vee y)(\overline{y} \vee z)(x \vee \overline{z})(z \vee y)$$



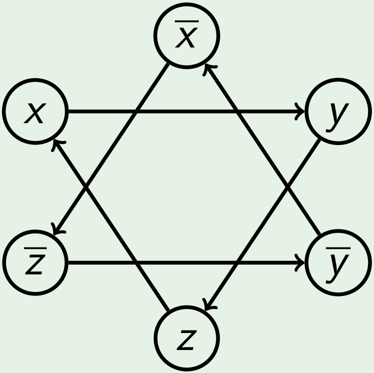
$(\overline{x} \vee y)(\overline{y} \vee z)(x \vee \overline{z})(z \vee y)$



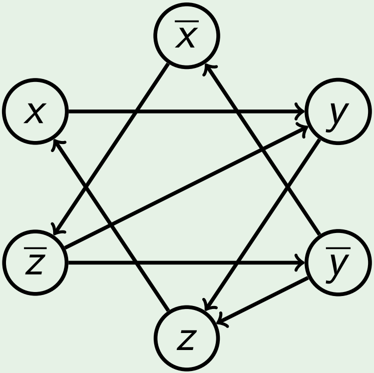
$$(\overline{x} \vee y)(\overline{y} \vee z)(x \vee \overline{z})(z \vee y)$$



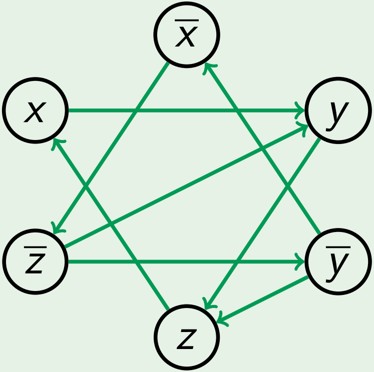
$$(\overline{x} \vee y)(\overline{y} \vee z)(x \vee \overline{z})(z \vee y)$$



$$(\overline{x} \vee y)(\overline{y} \vee z)(x \vee \overline{z})(z \vee y)$$

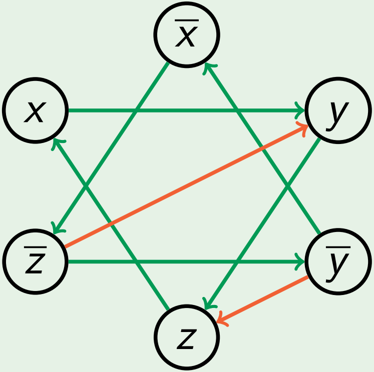


$(\overline{x} \vee y)(\overline{y} \vee z)(x \vee \overline{z})(z \vee y)$



$x = 1, y = 1, z = 1$

$(\overline{x} \vee y)(\overline{y} \vee z)(x \vee \overline{z})(z \vee y)$



$x = 0, y = 0, z = 0$

Thus, our goal is to assign truth values to the variables so that each edge in the implication graph is "satisfied", that is, there is no edge from 1 to 0.

Skew-Symmetry

- The graph is skew-symmetric: if there is an edge $\ell_1 \to \ell_2$, then there is an edge $\overline{\ell}_2 \to \overline{\ell}_1$

Skew-Symmetry

- The graph is skew-symmetric: if there is an edge $\ell_1 \to \ell_2$, then there is an edge $\overline{\ell}_2 \to \overline{\ell}_1$
- This generalizes to paths: if there is a path from ℓ_1 to ℓ_2, then there is a path from $\overline{\ell}_2$ to $\overline{\ell}_1$

Transitivity

Lemma

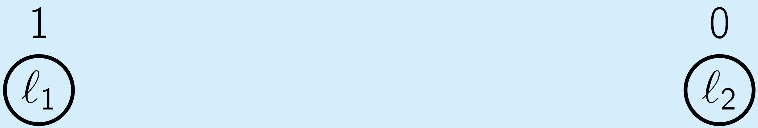
If all the edges are satisfied by an assignment and there is a path from ℓ_1 to ℓ_2, then it cannot be the case that $\ell_1 = 1$ and $\ell_2 = 0$.

Transitivity

Lemma

If all the edges are satisfied by an assignment and there is a path from ℓ_1 to ℓ_2, then it cannot be the case that $\ell_1 = 1$ and $\ell_2 = 0$.

Proof

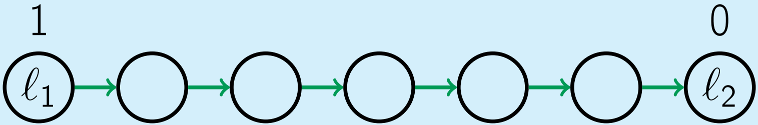


Transitivity

Lemma

If all the edges are satisfied by an assignment and there is a path from ℓ_1 to ℓ_2, then it cannot be the case that $\ell_1 = 1$ and $\ell_2 = 0$.

Proof

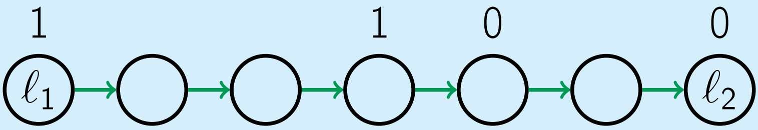


Transitivity

Lemma

If all the edges are satisfied by an assignment and there is a path from ℓ_1 to ℓ_2, then it cannot be the case that $\ell_1 = 1$ and $\ell_2 = 0$.

Proof

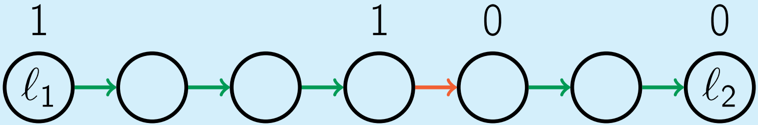


Transitivity

Lemma

If all the edges are satisfied by an assignment and there is a path from ℓ_1 to ℓ_2, then it cannot be the case that $\ell_1 = 1$ and $\ell_2 = 0$.

Proof



Strongly Connected Components

- All variables lying in the same SCC of the implication graph should be assigned the same value

Strongly Connected Components

- All variables lying in the same SCC of the implication graph should be assigned the same value
- In particular, if a SCC contains a variable together with its negation, then the formula is unsatisfiable

Strongly Connected Components

- All variables lying in the same SCC of the implication graph should be assigned the same value
- In particular, if a SCC contains a variable together with its negation, then the formula is unsatisfiable
- It turns out that otherwise the formula is satisfiable!

2SAT(2-CNF F)

```
construct the implication graph G
find SCC's of G
for all variables x:
  if x and x̄ lie in the same SCC of G:
    return "unsatisfiable"
find a topological ordering of SCC's
for all SCC's C in reverse order:
  if literals of C are not assigned yet:
    set all of them to 1
    set their negations to 0
return the satisfying assignment
```

2SAT(2-CNF F)

```
construct the implication graph G
find SCC's of G
for all variables x:
  if x and x̄ lie in the same SCC of G:
    return "unsatisfiable"
find a topological ordering of SCC's
for all SCC's C in reverse order:
  if literals of C are not assigned yet:
    set all of them to 1
    set their negations to 0
return the satisfying assignment
```

Running time: $O(|F|)$

Lemma

The algorithm `2SAT` is correct.

Proof

- When a literal is set to 1, all the literals that are reachable from it have already been set to 1 (since we process SCC's in reverse topological order).

Lemma

The algorithm `2SAT` is correct.

Proof

- When a literal is set to 1, all the literals that are reachable from it have already been set to 1 (since we process SCC's in reverse topological order).
- When a literal is set to 0, all the literals it is reachable from have already been set to 0 (by skew-symmetry). $\square$

Outline

1. 2-Satisfiability
2. Independent Sets in Trees

Planning a company party

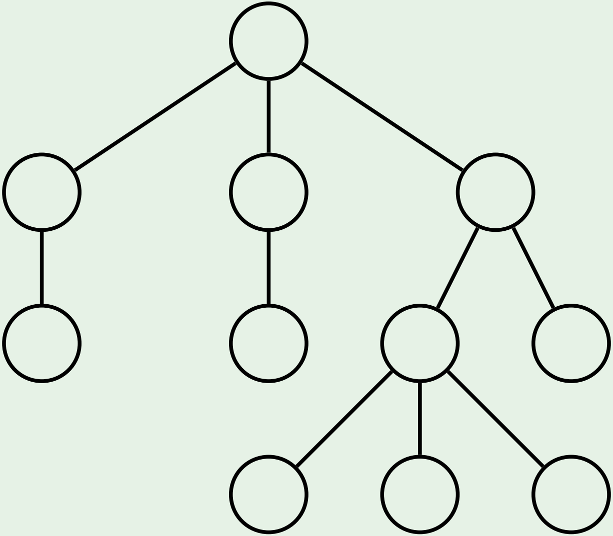
You are organizing a company party. You would like to invite as many people as possible with a single constraint: no person should attend a party with his or her direct boss.

Maximum independent set in a tree

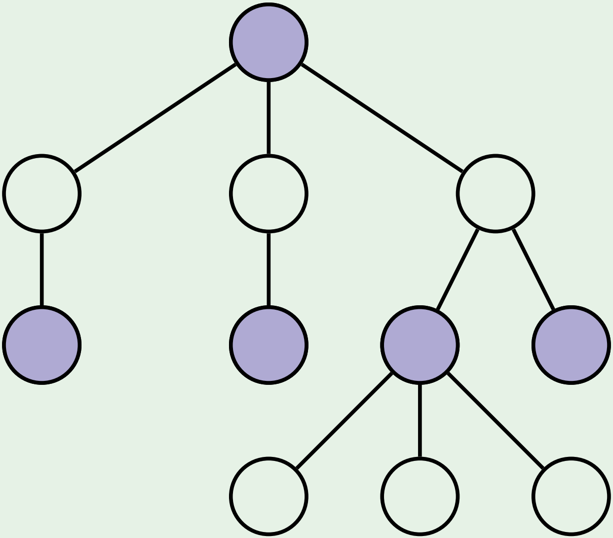
Input: A tree.

Output: An independent set (i.e., a subset of vertices no two of which are adjacent) of maximum size.

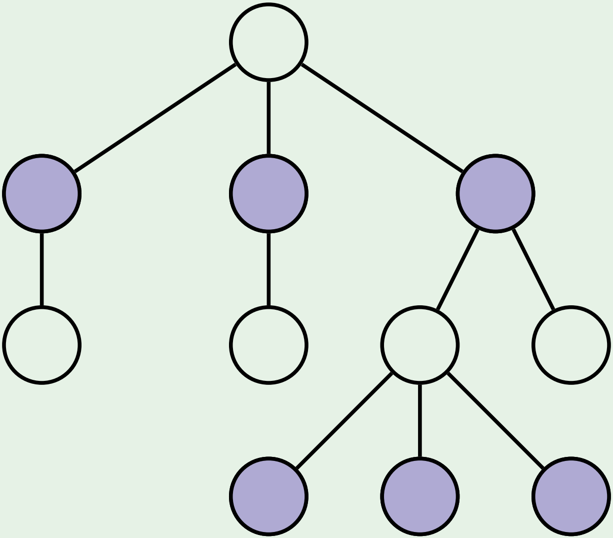
Example



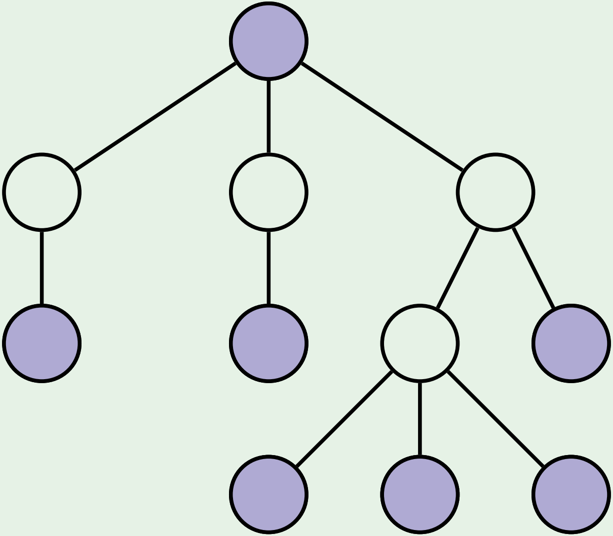
Example



Example

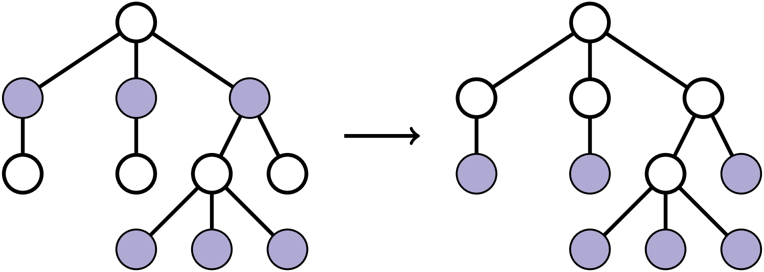


Example



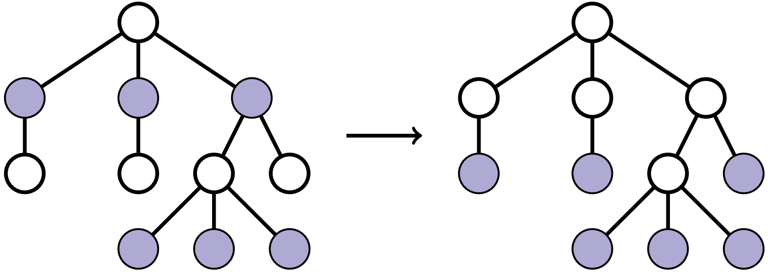
Safe move

For any leaf, there exists an optimal solution including this leaf.



Safe move

For any leaf, there exists an optimal solution including this leaf.



It is safe to take all the leaves.

PartyGreedy(T)

```
while T is not empty:
  take all the leaves to the solution
  remove them and their parents from T
return the constructed solution
```

PartyGreedy(T)

```
while T is not empty:
  take all the leaves to the solution
  remove them and their parents from T
return the constructed solution
```

Running time: $O(|T|)$ (for each vertex, maintain the number of its children).

Planning a company party

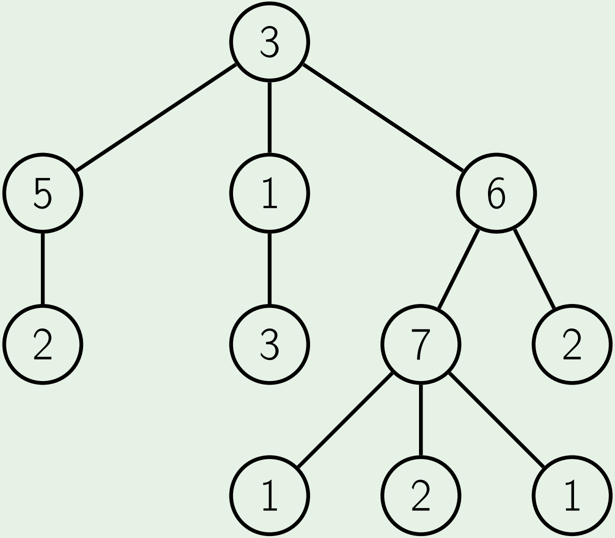
You are organizing a company party again. However this time, instead of maximizing the number of attendees, you would like to maximize the total fun factor.

Maximum weighted independent set in trees

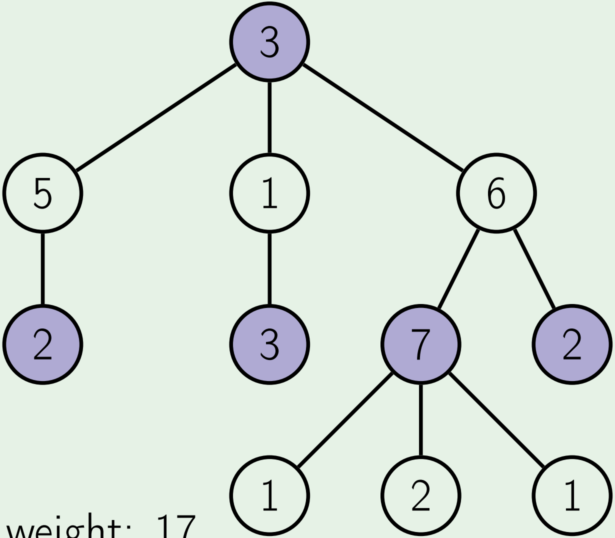
Input: A tree T with weights on vertices.

Output: An independent set (i.e., a subset of vertices no two of which are adjacent) of maximum total weight.

Example

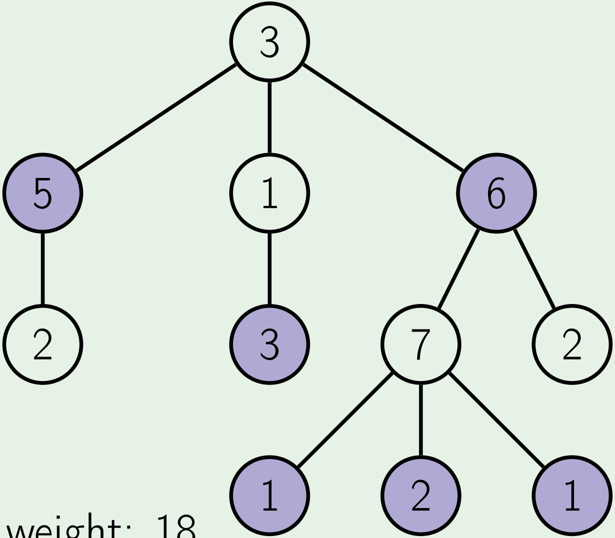


Example



total weight: 17

Example



total weight: 18

Subproblems

- $D(v)$ is the maximum weight of an independent set in a subtree rooted at v

Subproblems

- $D(v)$ is the maximum weight of an independent set in a subtree rooted at v
- Recurrence relation: $D(v)$ is

$$\max\left\{w(v) + \sum_{\substack{\text{grandchildren}\\ w \text{ of } v}} D(w), \sum_{\substack{\text{children}\\ w \text{ of } v}} D(w)\right\}$$

Function FunParty(v)

```
if D(v) = ∞:
  if v has no children:
    D(v) ← w(v)
```

Function FunParty(v)

```
if D(v) = ∞:
  if v has no children:
    D(v) ← w(v)
  else:
    m1 ← w(v)
    for all children u of v:
      for all children w of u:
        m1 ← m1 + FunParty(w)
```

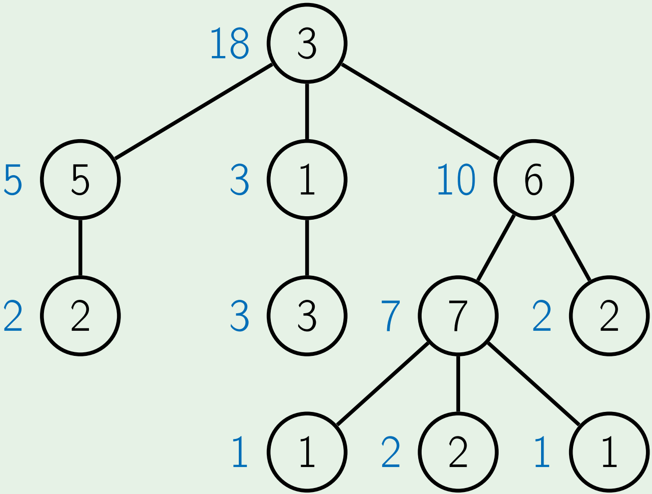
Function FunParty(v)

```
if D(v) = ∞:
  if v has no children:
    D(v) ← w(v)
  else:
    m_1 ← w(v)
    for all children u of v:
      for all children w of u:
        m_1 ← m_1 + FunParty(w)
    m_0 ← 0
    for all children u of v:
      m_0 ← m_0 + FunParty(u)
```

Function FunParty(v)

```
if D(v) = ∞:
  if v has no children:
    D(v) ← w(v)
  else:
    m_1 ← w(v)
    for all children u of v:
      for all children w of u:
        m_1 ← m_1 + FunParty(w)
    m_0 ← 0
    for all children u of v:
      m_0 ← m_0 + FunParty(u)
  D(v) ← max(m_1, m_0)
return D(v)
```

Example



Coping with NP-completeness: Introduction

Alexander S. Kulikov

Steklov Institute of Mathematics at St. Petersburg
Russian Academy of Sciences

Advanced Algorithms and Complexity
Data Structures and Algorithms

- Your boss asked you to implement a program that solves efficiently a certain search problem

- Your boss asked you to implement a program that solves efficiently a certain search problem
- If you are lucky enough, the problem can be solved by some known technique like dynamic programming, linear programming, flows (though it is usually still not immediate to notice this)

- Your boss asked you to implement a program that solves efficiently a certain search problem
- If you are lucky enough, the problem can be solved by some known technique like dynamic programming, linear programming, flows (though it is usually still not immediate to notice this)
- Alas, this happens rarely

After two weeks of unsuccessful attempts to implement an efficient program, you come to your boss' office.

After two weeks of unsuccessful attempts to implement an efficient program, you come to your boss' office. "I can't find an efficient algorithm, I guess I'm just too dumb."

Michael R. Garey and David S. Johnson.
Computers and Intractability: A Guide to the Theory of NP-Completeness. 1979.

Perhaps there is just no efficient algorithm for your search problem.

Perhaps there is just no efficient algorithm for your search problem. "I can't find an efficient algorithm, because no such algorithm is possible!"

Michael R. Garey and David S. Johnson.
Computers and Intractability: A Guide to the Theory of NP-Completeness. 1979.

- But currently, we don’t have a proof that a certain search problem has no efficient (that is, polynomial) algorithm

- But currently, we don't have a proof that a certain search problem has no efficient (that is, polynomial) algorithm
- Note that such a proof would resolve the **P** vs **NP** question

- But currently, we don't have a proof that a certain search problem has no efficient (that is, polynomial) algorithm
- Note that such a proof would resolve the **P** vs **NP** question
- Instead of showing that there is no efficient algorithm for your program, you show that it is one of the hardest search problems

- But currently, we don't have a proof that a certain search problem has no efficient (that is, polynomial) algorithm
- Note that such a proof would resolve the **P** vs **NP** question
- Instead of showing that there is no efficient algorithm for your program, you show that it is one of the hardest search problems
- That is, you show that your problem is **NP**-complete

"I can't find an efficient algorithm, but neither can all these famous people!"

Michael R. Garey and David S. Johnson.
Computers and Intractability: A Guide to the Theory of NP-Completeness. 1979.

OK, now you know that your problem is **NP**-complete meaning that it is unlikely that there exists an efficient algorithm for solving it. Should you give up?

OK, now you know that your problem is **NP**-complete meaning that it is unlikely that there exists an efficient algorithm for solving it. Should you give up?

Keep your head up!

It is just the beginning of a fascinating adventure!

Next Parts

If $\mathbf{P} \neq \mathbf{NP}$, then there is no polynomial time algorithm that finds an optimal solution to an $\mathbf{NP}$-complete problem in all cases.

Next Parts

If $\mathbf{P} \neq \mathbf{NP}$, then there is no polynomial time algorithm that finds an optimal solution to an **NP**-complete problem in all cases.

	poly time	optimal solution	all cases
special cases	✓	✓	✗
approximation algorithms	✓	✗	✓
exact algorithms	✗	✓	✓

Coping with NP-completeness: Exact Algorithms

Alexander S. Kulikov

Steklov Institute of Mathematics at St. Petersburg
Russian Academy of Sciences

Advanced Algorithms and Complexity
Data Structures and Algorithms

Exact algorithms or intelligent exhaustive search: finding an optimal solution without going through all candidate solutions

Outline

1. 3-Satisfiability
   - Backtracking
   - Local Search
2. Traveling Salesman Problem
   - Dynamic Programming
   - Branch-and-bound

3-Satisfiability (3-SAT)

Input: A set of clauses, each containing at most three literals (that is, a 3-CNF formula).

Output: Find a satisfying assignment (if exists).

Examples

- The formula

$$(x \vee y \vee z)(x \vee \overline{y})(y \vee \overline{z})$$

  is satisfiable: set $x = y = z = 1$ or $x = 1, y = z = 0$.
- The formula

$$(x \vee y \vee z)(x \vee \overline{y})(y \vee \overline{z})(z \vee \overline{x})(\overline{x} \vee \overline{y} \vee \overline{z})$$

  is unsatisfiable.

A brute force search algorithm checking satisfiability of a 3-CNF formula F with n variables, goes through all assignments and has running time $O(|F| \cdot 2^n)$.

A brute force search algorithm checking satisfiability of a 3-CNF formula F with $\boldsymbol{n}$ variables, goes through all assignments and has running time $O(|F| \cdot 2^n)$.

Goal

Avoid going through all 2^n assignments

Outline

1. 3-Satisfiability
   - Backtracking
   - Local Search
2. Traveling Salesman Problem
   - Dynamic Programming
   - Branch-and-bound

Main Idea of Backtracking

- Construct a solution piece by piece

Main Idea of Backtracking

- Construct a solution piece by piece
- Backtrack if the current partial solution cannot be extended to a valid solution

Example

$$\boxed{(x_1 \vee x_2 \vee x_3 \vee x_4)(\overline{x}_1)(x_1 \vee x_2 \vee \overline{x}_3)(x_1 \vee \overline{x}_2)(x_2 \vee \overline{x}_4)}$$

Example

$$\boxed{(x_1 \vee x_2 \vee x_3 \vee x_4)(\overline{x}_1)(x_1 \vee x_2 \vee \overline{x}_3)(x_1 \vee \overline{x}_2)(x_2 \vee \overline{x}_4)}$$

$$\Big\downarrow{\scriptstyle x_1 = 0}$$

$$\boxed{(x_2 \vee x_3 \vee x_4)(x_2 \vee \overline{x}_3)(\overline{x}_2)(x_2 \vee \overline{x}_4)}$$

Example

$$\begin{tikzcd}[column sep=small]
& \boxed{(x_1 \vee x_2 \vee x_3 \vee x_4)(\overline{x}_1)(x_1 \vee x_2 \vee \overline{x}_3)(x_1 \vee \overline{x}_2)(x_2 \vee \overline{x}_4)} \arrow[dl, "x_1 = 0"'] \\
\boxed{(x_2 \vee x_3 \vee x_4)(x_2 \vee \overline{x}_3)(\overline{x}_2)(x_2 \vee \overline{x}_4)} \arrow[d, "x_2 = 0"'] & \\
\boxed{(x_3 \vee x_4)(\overline{x}_3)(\overline{x}_4)} &
\end{tikzcd}$$

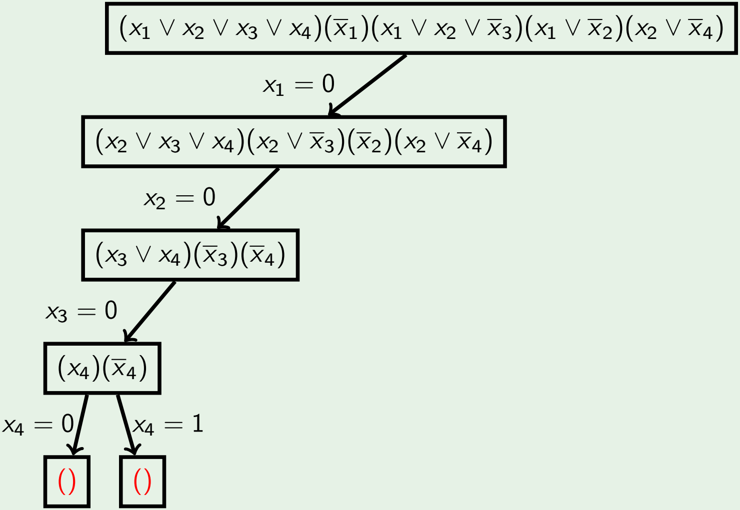
Example

$$
\begin{tikzcd}
\boxed{(x_1 \vee x_2 \vee x_3 \vee x_4)(\overline{x}_1)(x_1 \vee x_2 \vee \overline{x}_3)(x_1 \vee \overline{x}_2)(x_2 \vee \overline{x}_4)} \arrow[d, "x_1 = 0"'] \\
\boxed{(x_2 \vee x_3 \vee x_4)(x_2 \vee \overline{x}_3)(\overline{x}_2)(x_2 \vee \overline{x}_4)} \arrow[d, "x_2 = 0"'] \\
\boxed{(x_3 \vee x_4)(\overline{x}_3)(\overline{x}_4)} \arrow[d, "x_3 = 0"'] \\
\boxed{(x_4)(\overline{x}_4)}
\end{tikzcd}
$$

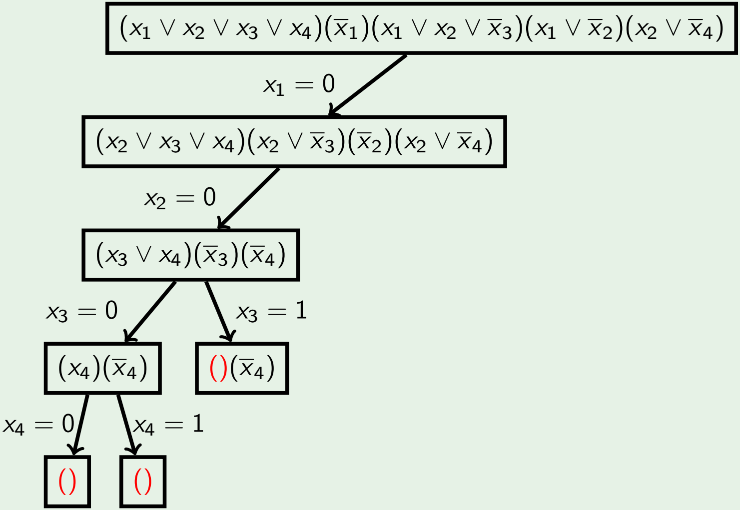
Example

$$
\begin{tikzcd}
\boxed{(x_1 \vee x_2 \vee x_3 \vee x_4)(\overline{x}_1)(x_1 \vee x_2 \vee \overline{x}_3)(x_1 \vee \overline{x}_2)(x_2 \vee \overline{x}_4)} \arrow[d, "x_1 = 0"'] \\
\boxed{(x_2 \vee x_3 \vee x_4)(x_2 \vee \overline{x}_3)(\overline{x}_2)(x_2 \vee \overline{x}_4)} \arrow[d, "x_2 = 0"'] \\
\boxed{(x_3 \vee x_4)(\overline{x}_3)(\overline{x}_4)} \arrow[d, "x_3 = 0"'] \\
\boxed{(x_4)(\overline{x}_4)} \arrow[d, "x_4 = 0"'] \\
\boxed{\color{red}{()}}
\end{tikzcd}
$$

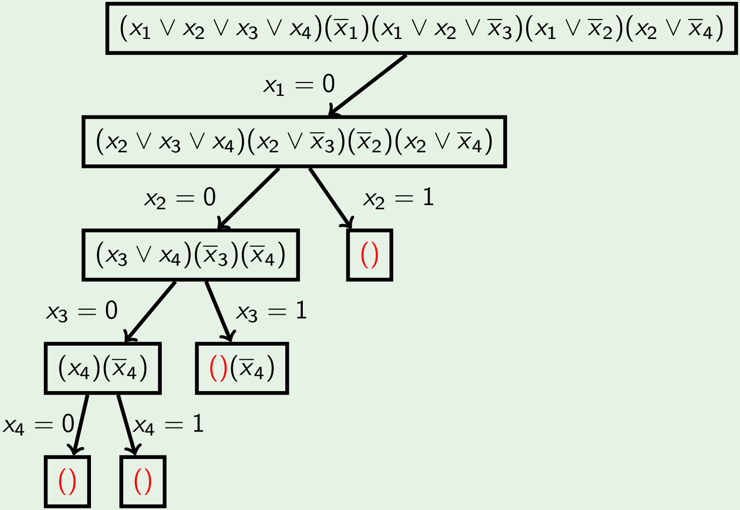
Example


$(x_1 \vee x_2 \vee x_3 \vee x_4)(\overline{x}_1)(x_1 \vee x_2 \vee \overline{x}_3)(x_1 \vee \overline{x}_2)(x_2 \vee \overline{x}_4)$
$x_1 = 0$
$(x_2 \vee x_3 \vee x_4)(x_2 \vee \overline{x}_3)(\overline{x}_2)(x_2 \vee \overline{x}_4)$
$x_2 = 0$
$(x_3 \vee x_4)(\overline{x}_3)(\overline{x}_4)$
$x_3 = 0$
$(x_4)(\overline{x}_4)$
$x_4 = 0$
$x_4 = 1$
$()$
$()$

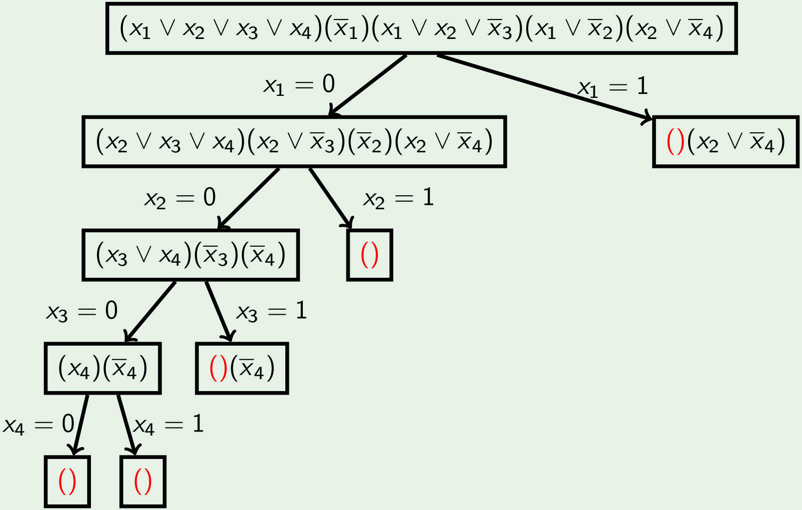
Example


$(x_1 \vee x_2 \vee x_3 \vee x_4)(\overline{x}_1)(x_1 \vee x_2 \vee \overline{x}_3)(x_1 \vee \overline{x}_2)(x_2 \vee \overline{x}_4)$
$x_1 = 0$
$(x_2 \vee x_3 \vee x_4)(x_2 \vee \overline{x}_3)(\overline{x}_2)(x_2 \vee \overline{x}_4)$
$x_2 = 0$
$(x_3 \vee x_4)(\overline{x}_3)(\overline{x}_4)$
$x_3 = 0$
$x_3 = 1$
$(x_4)(\overline{x}_4)$
$()(\overline{x}_4)$
$x_4 = 0$
$x_4 = 1$
$()$
$()$

Example


$(x_1 \vee x_2 \vee x_3 \vee x_4)(\overline{x}_1)(x_1 \vee x_2 \vee \overline{x}_3)(x_1 \vee \overline{x}_2)(x_2 \vee \overline{x}_4)$
$x_1 = 0$
$(x_2 \vee x_3 \vee x_4)(x_2 \vee \overline{x}_3)(\overline{x}_2)(x_2 \vee \overline{x}_4)$
$x_2 = 0$
$x_2 = 1$
$(x_3 \vee x_4)(\overline{x}_3)(\overline{x}_4)$
$()$
$x_3 = 0$
$x_3 = 1$
$(x_4)(\overline{x}_4)$
$()(\overline{x}_4)$
$x_4 = 0$
$x_4 = 1$
$()$
$()$

Example


$(x_1 \vee x_2 \vee x_3 \vee x_4)(\overline{x}_1)(x_1 \vee x_2 \vee \overline{x}_3)(x_1 \vee \overline{x}_2)(x_2 \vee \overline{x}_4)$
$x_1 = 0$
$x_1 = 1$
$(x_2 \vee x_3 \vee x_4)(x_2 \vee \overline{x}_3)(\overline{x}_2)(x_2 \vee \overline{x}_4)$
$()(x_2 \vee \overline{x}_4)$
$x_2 = 0$
$x_2 = 1$
$(x_3 \vee x_4)(\overline{x}_3)(\overline{x}_4)$
$()$
$x_3 = 0$
$x_3 = 1$
$(x_4)(\overline{x}_4)$
$()(\overline{x}_4)$
$x_4 = 0$
$x_4 = 1$
$()$
$()$

```
SolveSAT(F)

if F has no clauses:
  return "sat"
```

SolveSAT(F)

```
if F has no clauses:
  return "sat"
if F contains an empty clause:
  return "unsat"
```

SolveSAT(F)

```
if F has no clauses:
  return "sat"
if F contains an empty clause:
  return "unsat"
x ← unassigned variable of F
```

SolveSAT(F)

```
if F has no clauses:
  return "sat"
if F contains an empty clause:
  return "unsat"
x ← unassigned variable of F
if SolveSAT(F[x ← 0]) = "sat":
  return "sat"
```

SolveSAT(F)

```
if F has no clauses:
  return "sat"
if F contains an empty clause:
  return "unsat"
x ← unassigned variable of F
if SolveSAT(F[x ← 0]) = "sat":
  return "sat"
if SolveSAT(F[x ← 1]) = "sat":
  return "sat"
```

SolveSAT(F)

```
if F has no clauses:
  return "sat"
if F contains an empty clause:
  return "unsat"
x ← unassigned variable of F
if SolveSAT(F[x ← 0]) = "sat":
  return "sat"
if SolveSAT(F[x ← 1]) = "sat":
  return "sat"
return "unsat"
```

- Thus, instead of considering all 2^n branches of the recursion tree, we track carefully each branch

- Thus, instead of considering all 2^n branches of the recursion tree, we track carefully each branch
- When we realize that a branch is dead (cannot be extended to a solution), we immediately cut it

- Backtracking is used in many state-of-the-art SAT-solvers

- Backtracking is used in many state-of-the-art SAT-solvers
- SAT-solvers use tricky heuristics to choose a variable to branch on and to simplify a formula before branching

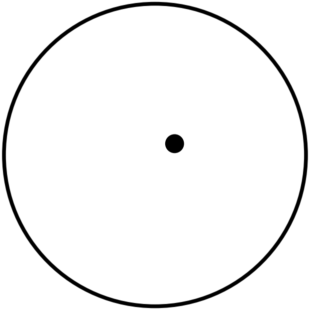
- Backtracking is used in many state-of-the-art SAT-solvers
- SAT-solvers use tricky heuristics to choose a variable to branch on and to simplify a formula before branching
- Another commonly used technique is local search — will consider it in the next part

Outline

1. 3-Satisfiability
   - Backtracking
   - Local Search
2. Traveling Salesman Problem
   - Dynamic Programming
   - Branch-and-bound

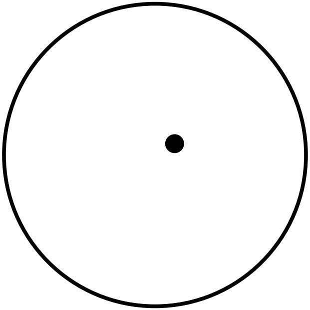
Main Idea of Local Search

- Start with a candidate solution



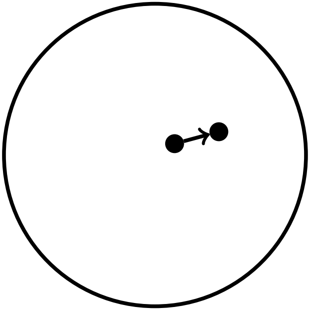
Main Idea of Local Search

- Start with a candidate solution
- Iteratively move from the current candidate to its neighbor trying to improve the candidate



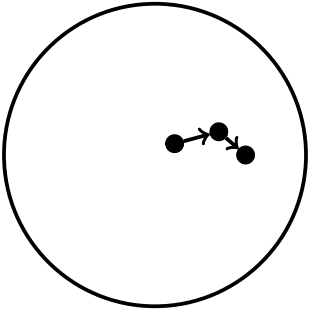
Main Idea of Local Search

- Start with a candidate solution
- Iteratively move from the current candidate to its neighbor trying to improve the candidate



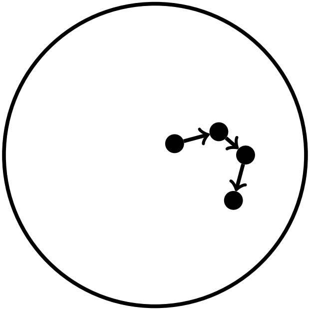
Main Idea of Local Search

- Start with a candidate solution
- Iteratively move from the current candidate to its neighbor trying to improve the candidate



Main Idea of Local Search

- Start with a candidate solution
- Iteratively move from the current candidate to its neighbor trying to improve the candidate



- Let F be a 3-CNF formula over variables $x_1, x_2, \ldots, x_n$

- Let F be a 3-CNF formula over variables $x_1, x_2, \ldots, x_n$
- A candidate solution is a truth assignment to these variables, that is, a vector from $\{0, 1\}^n$

Definition

Hamming distance (or just distance) between two assignments $\alpha, \beta \in \{0,1\}^n$ is the number of bits where they differ:
$\mathrm{dist}(\alpha, \beta) = |\{i \colon \alpha_i \neq \beta_i\}|$.

Definition

Hamming distance (or just distance) between two assignments $\alpha, \beta \in \{0,1\}^n$ is the number of bits where they differ: $\mathrm{dist}(\alpha, \beta) = |\{i \colon \alpha_i \neq \beta_i\}|$.

Definition

Hamming ball with center $\alpha \in \{0,1\}^n$ and radius r, denoted by $\mathcal{H}(\alpha, r)$, is the set of all truth assignments from $\{0,1\}^n$ at distance at most r from α.

Example

- $\mathcal{H}(1011, 0) = \{1011\}$

Example

- $\mathcal{H}(1011, 0) = \{1011\}$
- $\mathcal{H}(1011, 1) = \{1011, 0011, 1111, 1001, 1010\}$

Example

- $\mathcal{H}(1011, 0) = \{1011\}$
- $\mathcal{H}(1011, 1) =$
  $\{1011, 0011, 1111, 1001, 1010\}$
- $\mathcal{H}(1011, 2) =$
  $\{1011, 0011, 1111, 1001, 1010,$
  $0111, 0001, 0010, 1101, 1110, 1000\}$

Searching a Ball for a Solution

Lemma

Assume that $\mathcal{H}(\alpha, r)$ contains a satisfying assignment β for F. We can then find a (possibly different) satisfying assignment in time $O(|F| \cdot 3^r)$.

Proof

- If α satisfies F, return α

Proof

- If α satisfies F, return α
- Otherwise, take an unsatisfied clause — say, $(x_i \vee \overline{x}_j \vee x_k)$

Proof

- If α satisfies F, return α
- Otherwise, take an unsatisfied clause — say, $(x_i \vee \overline{x}_j \vee x_k)$
- α assigns $x_i = 0, x_j = 1, x_k = 0$

Proof

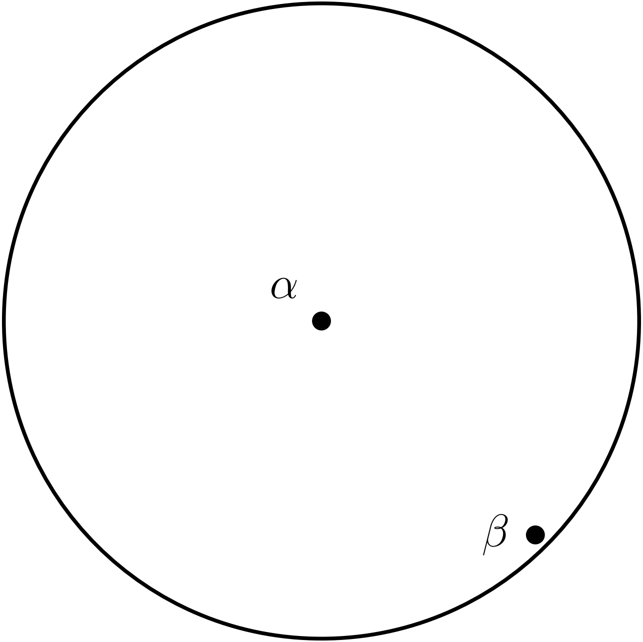
- If α satisfies F, return α
- Otherwise, take an unsatisfied clause — say, $(x_i \vee \overline{x}_j \vee x_k)$
- α assigns $x_i = 0, x_j = 1, x_k = 0$
- Let $\alpha^i, \alpha^j, \alpha^k$ be assignments resulting from α by flipping the i-th, j-th, k-th bit, respectively

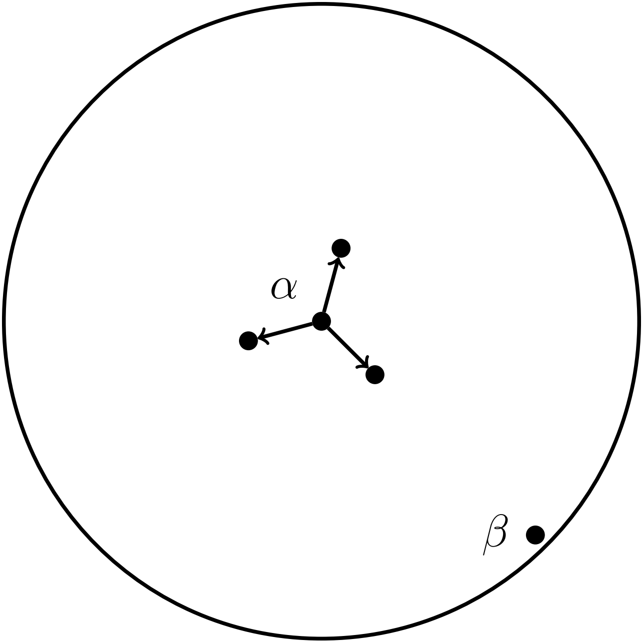
Proof

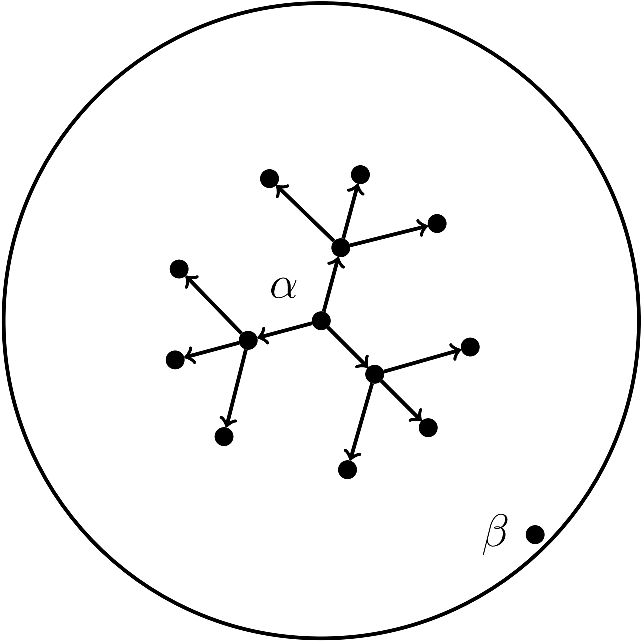
- If α satisfies F, return α
- Otherwise, take an unsatisfied clause — say, $(x_i \vee \overline{x}_j \vee x_k)$
- α assigns $x_i = 0, x_j = 1, x_k = 0$
- Let $\alpha^i, \alpha^j, \alpha^k$ be assignments resulting from α by flipping the i-th, j-th, k-th bit, respectively
- Crucial observation: at least one of them is closer to β than α

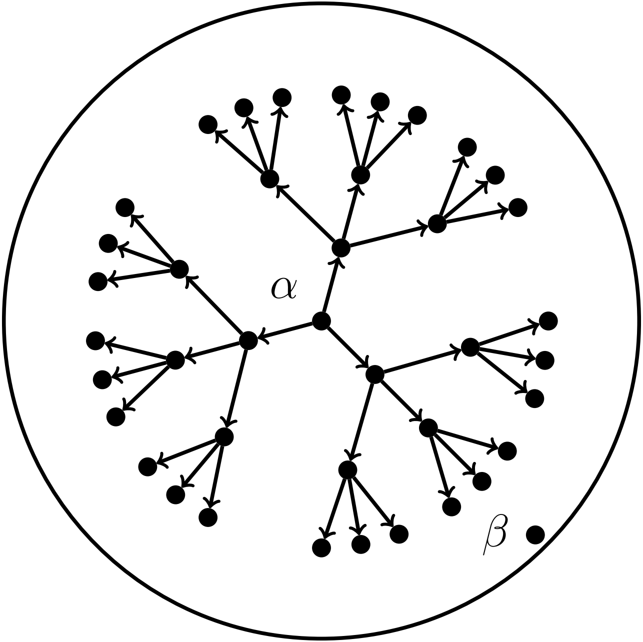
Proof

- If α satisfies F, return α
- Otherwise, take an unsatisfied clause — say, $(x_i \vee \overline{x}_j \vee x_k)$
- α assigns $x_i = 0, x_j = 1, x_k = 0$
- Let $\alpha^i, \alpha^j, \alpha^k$ be assignments resulting from α by flipping the i-th, j-th, k-th bit, respectively
- Crucial observation: at least one of them is closer to β than α
- Hence there are at most 3^r recursive calls $\square$


α
β


α
β


α
β


α
β

CheckBall(F, α, r)

```
if α satisfies F:
  return α
```

`CheckBall`(F, α, r)

```
if α satisfies F:
  return α
if r = 0:
  return "not found"
```

CheckBall(F, α, r)

```
if α satisfies F:
  return α
if r = 0:
  return "not found"
```

$x_i, x_j, x_k \leftarrow$ variables of unsatisfied clause

$\alpha^i, \alpha^j, \alpha^k \leftarrow \alpha$ with bits i, j, k flipped

CheckBall(F, α, r)

```
if α satisfies F:
  return α
if r = 0:
  return "not found"
x_i, x_j, x_k ← variables of unsatisfied clause
α^i, α^j, α^k ← α with bits i, j, k flipped
CheckBall(F, α^i, r − 1)
CheckBall(F, α^j, r − 1)
CheckBall(F, α^k, r − 1)
```

$\texttt{CheckBall}(F, \alpha, r)$

```
if α satisfies F:
  return α
if r = 0:
  return ‘‘not found’’
x_i, x_j, x_k ← variables of unsatisfied clause
α^i, α^j, α^k ← α with bits i, j, k flipped
CheckBall(F, α^i, r − 1)
CheckBall(F, α^j, r − 1)
CheckBall(F, α^k, r − 1)
if a satisfying assignment is found:
  return it
else:
  return ‘‘not found’’
```

- Assume that F has a satisfying assignment β

- Assume that F has a satisfying assignment β
- If it has more 1's than 0's then it has distance at most $n/2$ from all-1's assignment

- Assume that F has a satisfying assignment β
- If it has more 1's than 0's then it has distance at most $n/2$ from all-1's assignment
- Otherwise it has distance at most $n/2$ from all-0's assignment

- Assume that F has a satisfying assignment β
- If it has more 1's than 0's then it has distance at most $n/2$ from all-1's assignment
- Otherwise it has distance at most $n/2$ from all-0's assignment
- Thus, it suffices to make two calls: $\texttt{CheckBall}(F, 11\dots1, n/2)$ and $\texttt{CheckBall}(F, 00\dots0, n/2)$

Running Time

- The running time of the resulting algorithm is $O(|F| \cdot 3^{n/2}) \approx O(|F| \cdot 1.733^n)$

Running Time

- The running time of the resulting algorithm is $O(|F| \cdot 3^{n/2}) \approx O(|F| \cdot 1.733^n)$
- On one hand, this is still exponential

Running Time

- The running time of the resulting algorithm is $O(|F| \cdot 3^{n/2}) \approx O(|F| \cdot 1.733^n)$
- On one hand, this is still exponential
- On the other hand, it is exponentially faster than a brute force search algorithm that goes through all 2^n truth assignments!

Outline

1. 3-Satisfiability
   - Backtracking
   - Local Search
2. Traveling Salesman Problem
   - Dynamic Programming
   - Branch-and-bound

Traveling salesman problem (TSP)

Input: A complete graph with weights on edges and a budget b.

Output: A cycle that visits each vertex exactly once and has total weight at most b.

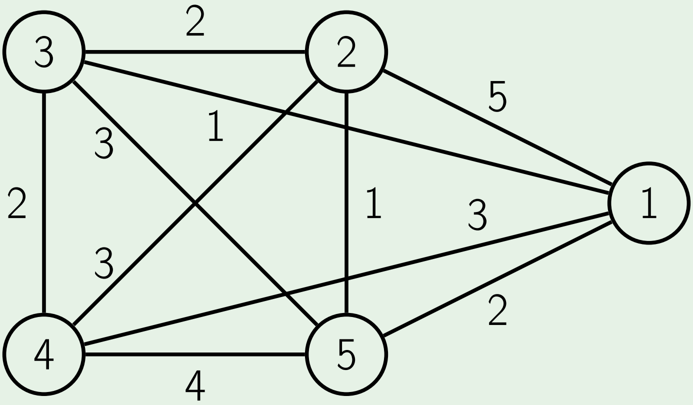
Traveling salesman problem (TSP)

Input: A complete graph with weights on edges and a budget b.

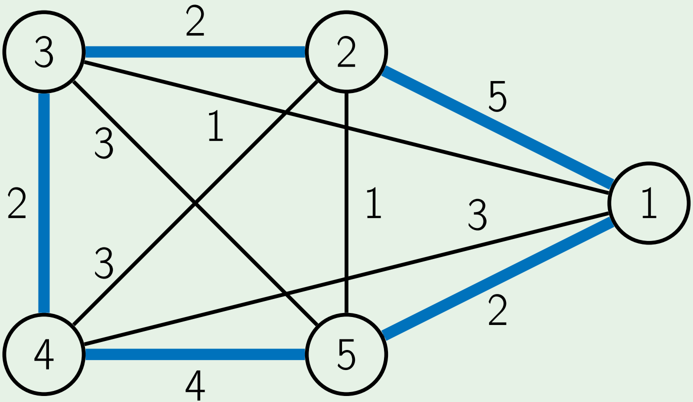
Output: A cycle that visits each vertex exactly once and has total weight at most b.

It will be convenient to assume that vertices are integers from 1 to n and that the salesman starts his trip in (and also returns back to) vertex 1.

Example

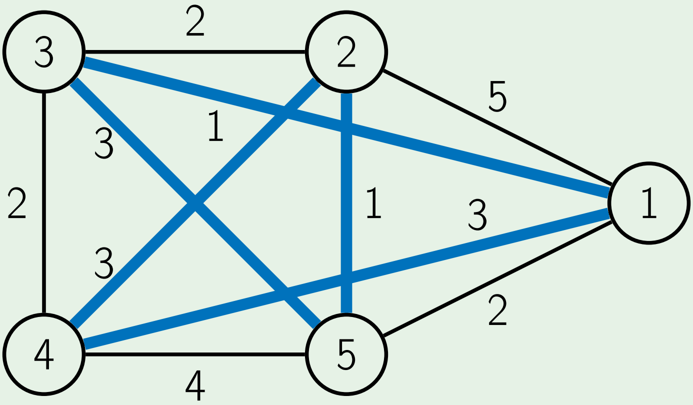


Example



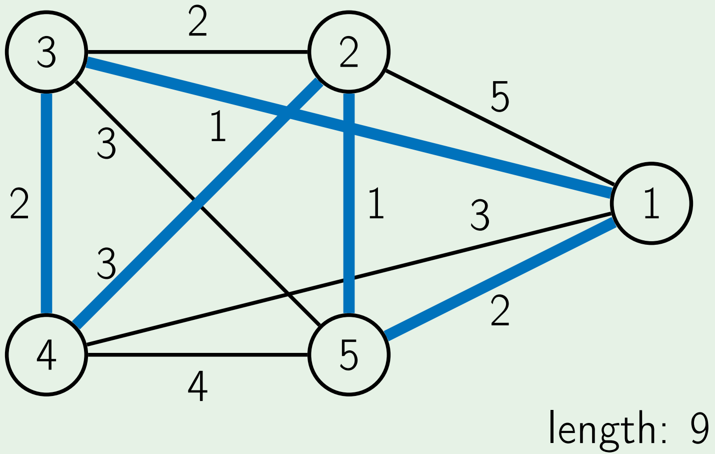
length: 15

Example



length: 11

Example



Brute Force Solution

A naive algorithm just checks all possible $(n-1)!$ cycles.

Brute Force Solution

A naive algorithm just checks all possible $(n-1)!$ cycles.

This part

- Use dynamic programming to solve TSP in $O(n^2 \cdot 2^n)$

Brute Force Solution

A naive algorithm just checks all possible $(n-1)!$ cycles.

This part

- Use dynamic programming to solve TSP in $O(n^2 \cdot 2^n)$
- The running time is exponential, but is much better than $(n-1)!$.

Outline

1. 3-Satisfiability
   - Backtracking
   - Local Search
2. Traveling Salesman Problem
   - Dynamic Programming
   - Branch-and-bound

Dynamic Programming

- We are going to use dynamic programming: instead of solving one problem we will solve a collection of (overlapping) subproblems

Dynamic Programming

- We are going to use dynamic programming: instead of solving one problem we will solve a collection of (overlapping) subproblems
- A subproblem refers to a partial solution

Dynamic Programming

- We are going to use dynamic programming: instead of solving one problem we will solve a collection of (overlapping) subproblems
- A subproblem refers to a partial solution
- A reasonable partial solution in case of TSP is the initial part of a cycle

Dynamic Programming

- We are going to use dynamic programming: instead of solving one problem we will solve a collection of (overlapping) subproblems
- A subproblem refers to a partial solution
- A reasonable partial solution in case of TSP is the initial part of a cycle
- To continue building a cycle, we need to know the last vertex as well as the set of already visited vertices

Subproblems

- For a subset of vertices $S \subseteq \{1, \ldots, n\}$ containing the vertex 1 and a vertex $i \in S$, let $C(S, i)$ be the length of the shortest path that starts at 1, ends at i and visits all vertices from S exactly once

Subproblems

- For a subset of vertices $S \subseteq \{1, \dots, n\}$ containing the vertex 1 and a vertex $i \in S$, let $C(S, i)$ be the length of the shortest path that starts at 1, ends at i and visits all vertices from S exactly once
- $C(\{1\}, 1) = 0$ and $C(S, 1) = +\infty$ when $|S| > 1$

Recurrence Relation

- Consider the second-to-last vertex j on the required shortest path from 1 to i visiting all vertices from S

Recurrence Relation

- Consider the second-to-last vertex j on the required shortest path from 1 to i visiting all vertices from S
- The subpath from 1 to j is the shortest one visiting all vertices from $S - \{i\}$ exactly once

Recurrence Relation

- Consider the second-to-last vertex j on the required shortest path from 1 to i visiting all vertices from S
- The subpath from 1 to j is the shortest one visiting all vertices from $S - \{i\}$ exactly once
- Hence
  $C(S, i) = \min\{C(S - \{i\}, j) + d_{ji}\}$,
  where the minimum is over all $j \in S$ such that $j \neq i$

Order of Subproblems

- Need to process all subsets $S \subseteq \{1, \dots, n\}$ in an order that guarantees that when computing the value of $C(S, i)$, the values of $C(S - \{i\}, j)$ have already been computed

Order of Subproblems

- Need to process all subsets $S \subseteq \{1, \ldots, n\}$ in an order that guarantees that when computing the value of $C(S, i)$, the values of $C(S - \{i\}, j)$ have already been computed
- For example, we can process subsets in order of increasing size

$\mathrm{TSP}(G)$

$C(\{1\}, 1) \leftarrow 0$

TSP(G)

```
C({1}, 1) ← 0
for s from 2 to n:
  for all 1 ∈ S ⊆ {1, ..., n} of size s:
    C(S, 1) ← +∞
```

TSP(G)

```
C({1},1) ← 0
for s from 2 to n:
  for all 1 ∈ S ⊆ {1,...,n} of size s:
    C(S,1) ← +∞
    for all i ∈ S, i ≠ 1:
      for all j ∈ S, j ≠ i:
        C(S,i) ← min{C(S,i), C(S−{i},j)+d_ji}
```

TSP(G)

```
C({1}, 1) ← 0
for s from 2 to n:
  for all 1 ∈ S ⊆ {1, ..., n} of size s:
    C(S, 1) ← +∞
    for all i ∈ S, i ≠ 1:
      for all j ∈ S, j ≠ i:
        C(S, i) ← min{C(S, i), C(S − {i}, j) + d_ji}
return min_i {C({1, ..., n}, i) + d_i,1}
```

Implementation Remark

- How to iterate through all subsets of $\{1, \ldots, n\}$?

Implementation Remark

- How to iterate through all subsets of $\{1, \ldots, n\}$?
- There is a natural one-to-one correspondence between integers in the range from 0 and $2^n - 1$ and subsets of $\{0, \ldots, n-1\}$:

$$k \leftrightarrow \{i \colon i\text{-th bit of } k \text{ is } 1\}$$

Example

k	$\mathrm{bin}(k)$	$\{i : i\text{-th bit of } k \text{ is } 1\}$
0	000	$\emptyset$
1	001	$\{0\}$
2	010	$\{1\}$
3	011	$\{0,1\}$
4	100	$\{2\}$
5	101	$\{0,2\}$
6	110	$\{1,2\}$
7	111	$\{0,1,2\}$

- If k corresponds to S, how to find out the integer corresponding to $S - \{j\}$ (for $j \in S$)?

- If k corresponds to S, how to find out the integer corresponding to $S - \{j\}$ (for $j \in S$)?
- For this, we need to flip the j-th bit of k (from 1 to 0)

- If k corresponds to S, how to find out the integer corresponding to $S - \{j\}$ (for $j \in S$)?
- For this, we need to flip the j-th bit of k (from 1 to 0)
- For this, in turn, we compute a bitwise XOR of k and 2^j (that has 1 only in j-th position)

- If k corresponds to S, how to find out the integer corresponding to $S - \{j\}$ (for $j \in S$)?
- For this, we need to flip the j-th bit of k (from 1 to 0)
- For this, in turn, we compute a bitwise XOR of k and 2^j (that has 1 only in j-th position)
- In `C/C++`, `Java`, `Python`:
  `k ^ (1 << j)`

Outline

1. 3-Satisfiability
   - Backtracking
   - Local Search

2. Traveling Salesman Problem
   - Dynamic Programming
   - Branch-and-bound

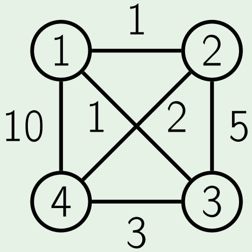
- The branch-and-bound technique can be viewed as a generalization of backtracking for optimization problems

- The branch-and-bound technique can be viewed as a generalization of backtracking for optimization problems
- We grow a tree of partial solutions

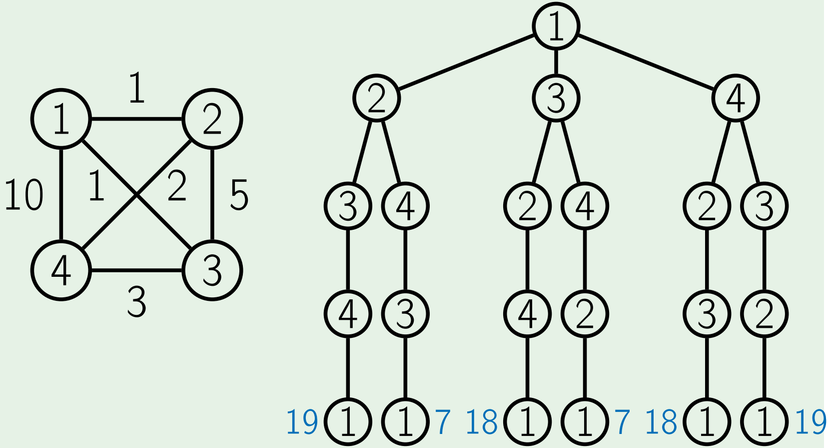
- The branch-and-bound technique can be viewed as a generalization of backtracking for optimization problems
- We grow a tree of partial solutions
- At each node of the recursion tree we check whether the current partial solution can be extended to a solution which is better than the best solution found so far

- The branch-and-bound technique can be viewed as a generalization of backtracking for optimization problems
- We grow a tree of partial solutions
- At each node of the recursion tree we check whether the current partial solution can be extended to a solution which is better than the best solution found so far
- If not, we don't continue this branch

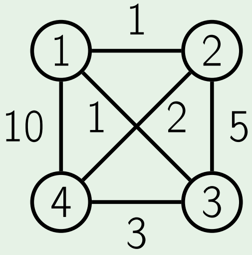
Example: brute force search



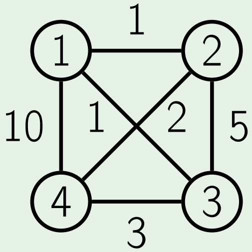
Example: brute force search

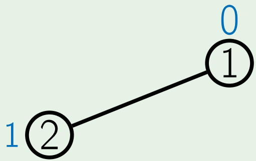


Example: pruned search

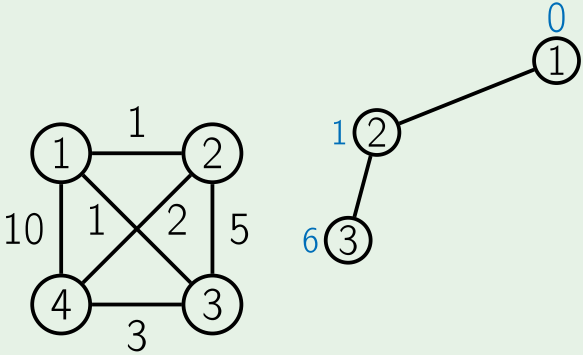


Example: pruned search

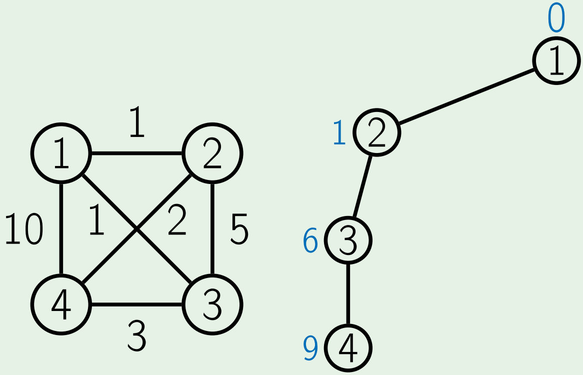




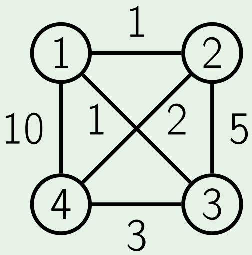
Example: pruned search

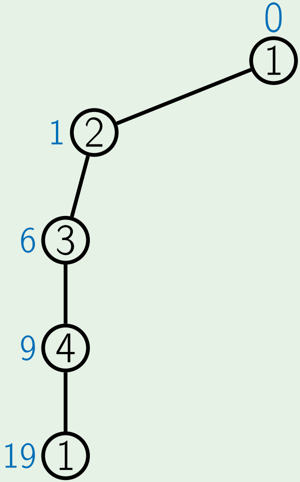


Example: pruned search



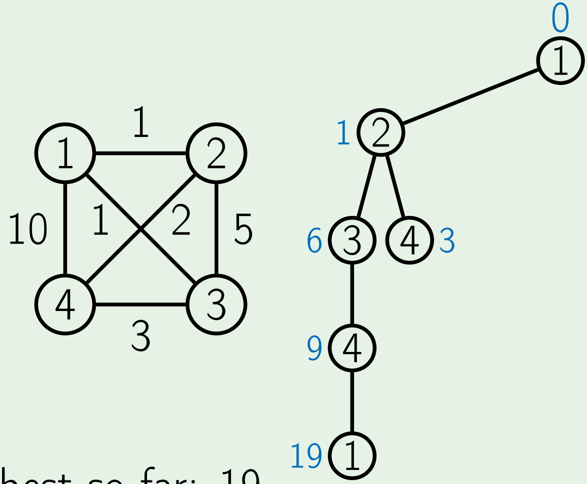
Example: pruned search





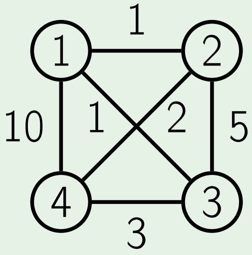
best so far: 19

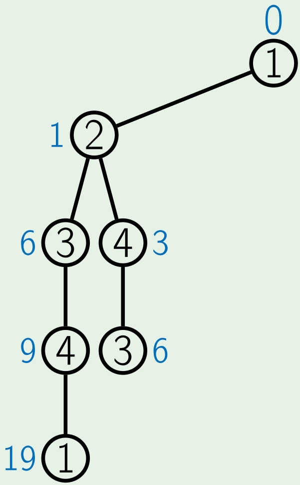
Example: pruned search



best so far: 19

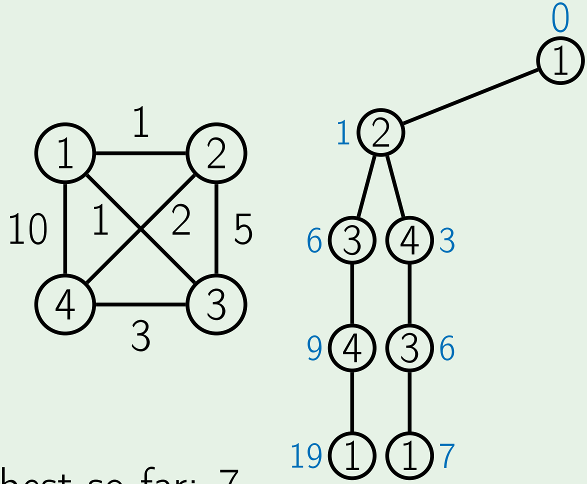
Example: pruned search





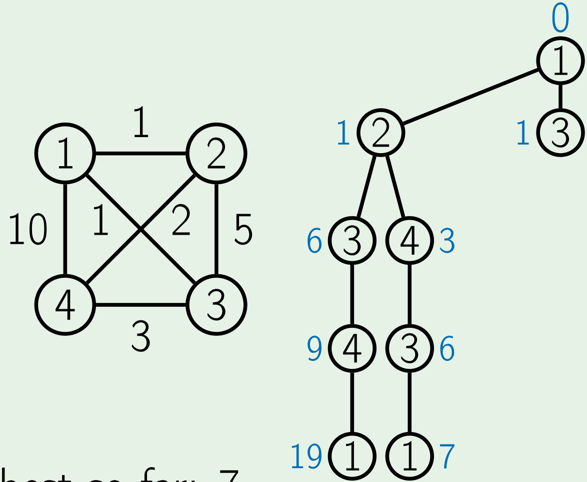
best so far: 19

Example: pruned search



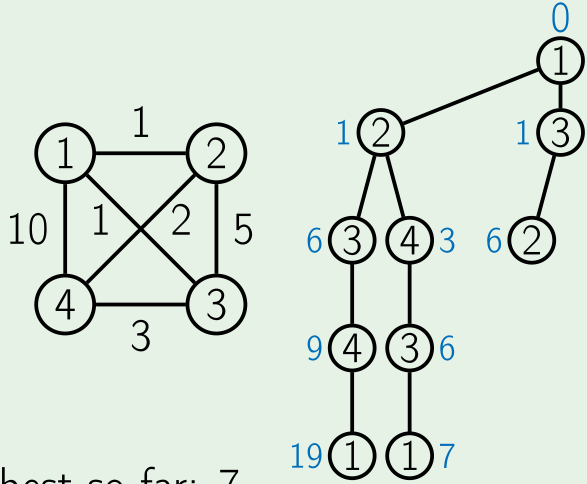
best so far: 7

Example: pruned search



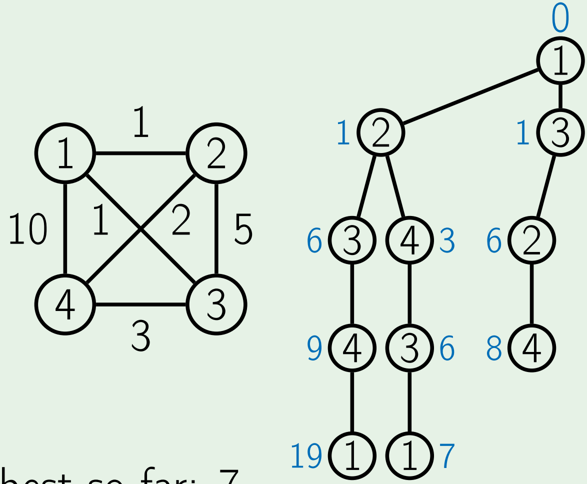
best so far: 7

Example: pruned search



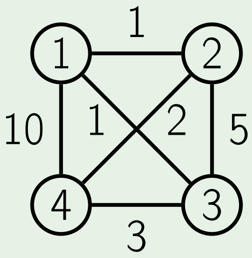
best so far: 7

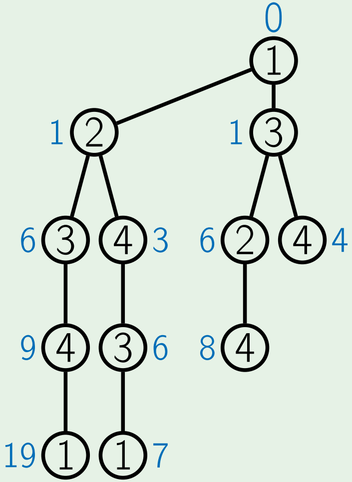
Example: pruned search



best so far: 7

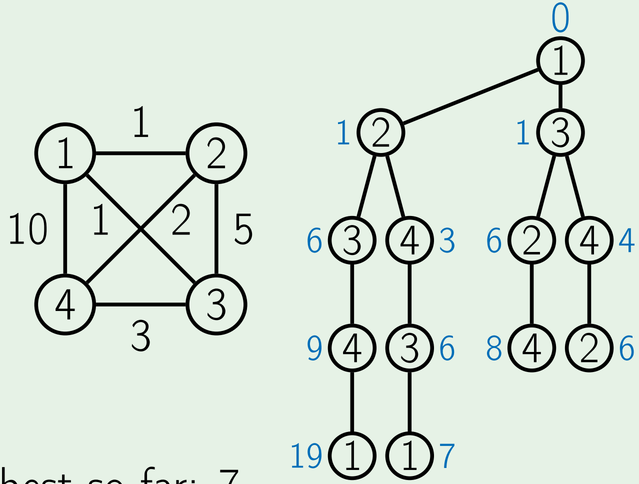
Example: pruned search





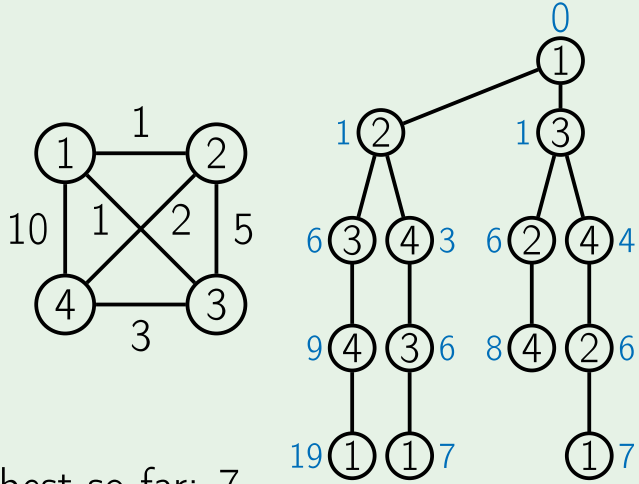
best so far: 7

Example: pruned search



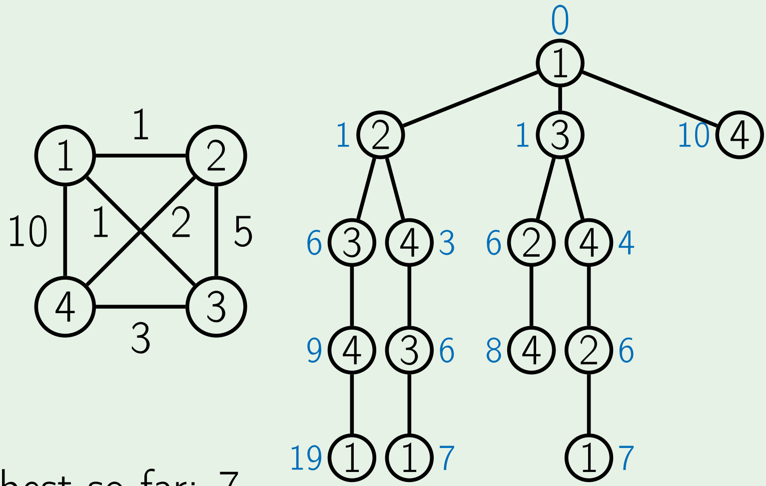
best so far: 7

Example: pruned search



best so far: 7

Example: pruned search



best so far: 7

- We used the simplest possible lower bound: any extension of a path has length at least the length of the path

- We used the simplest possible lower bound: any extension of a path has length at least the length of the path
- Modern TSP-solvers use smarter lower bounds to solve instances with thousands of vertices

Example: lower bounds (still simple)

The length of an optimal TSP cycle is at least

- $\frac{1}{2}\sum_{v \in V}$(two min length edges adjacent to v)

Example: lower bounds (still simple)

The length of an optimal TSP cycle is at least

- $\frac{1}{2}\sum_{v \in V}$(two min length edges adjacent to v)
- the length of a minimum spanning tree

Next time

Approximation algorithms: polynomial algorithms that find a solution that is not much worse than an optimal solution

Coping with NP-completeness: Approximation Algorithms

Alexander S. Kulikov

Steklov Institute of Mathematics at St. Petersburg
Russian Academy of Sciences

Advanced Algorithms and Complexity
Data Structures and Algorithms

Outline

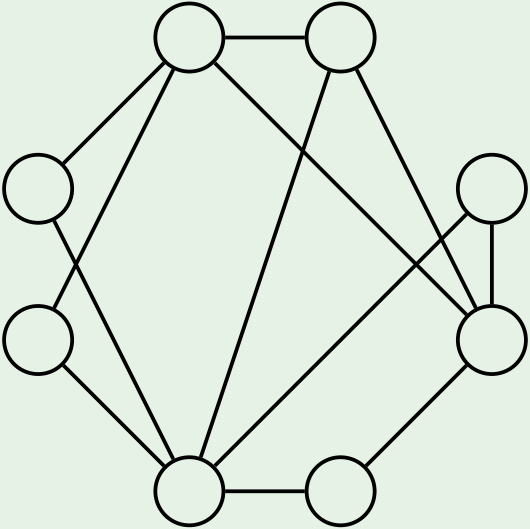
1. Vertex cover

2. Traveling salesman
    - Metric TSP
    - Local search

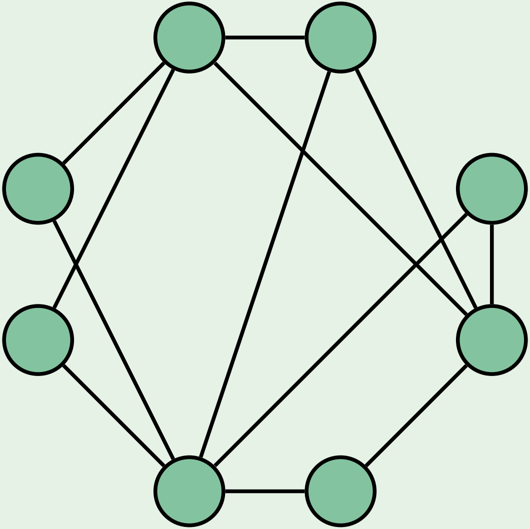
Vertex cover (optimization version)

Input:	A graph.
Output:	A subset of vertices of minimum size that touches every edge.

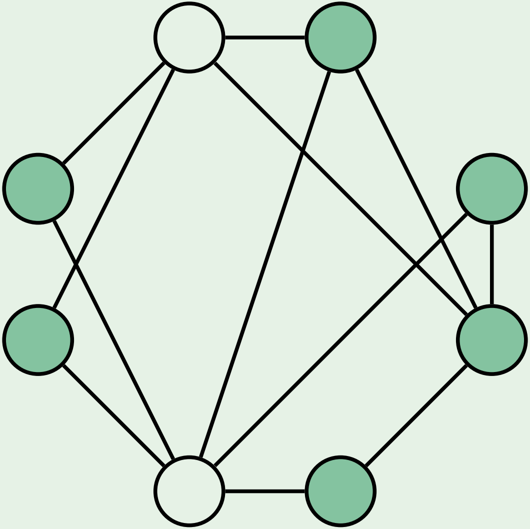
Example



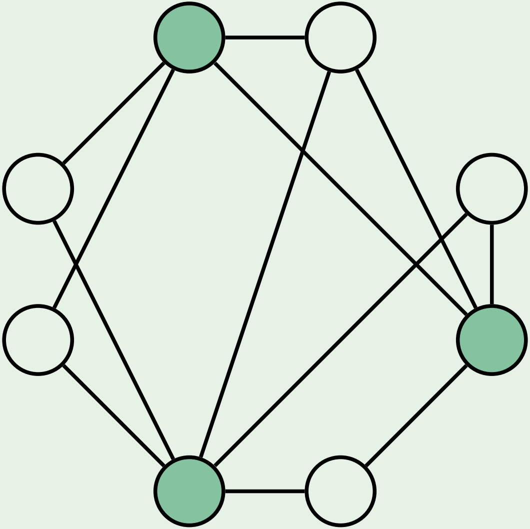
Example



Example



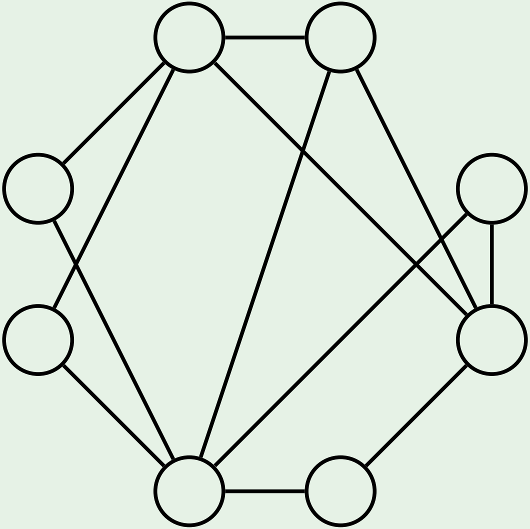
Example



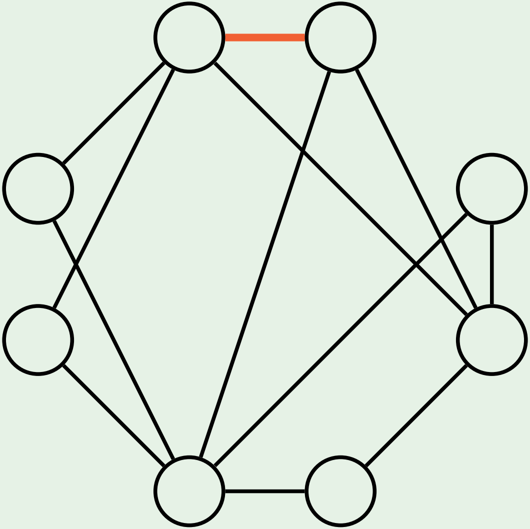
`ApproxVertexCover`$(G(V,E))$

$C \leftarrow$ `empty set`
`while` E `is not empty:`
  $\{u,v\} \leftarrow$ `any edge from` E
  `add` u, v `to` C
  `remove from` E `all edges incident to` u, v
`return` C

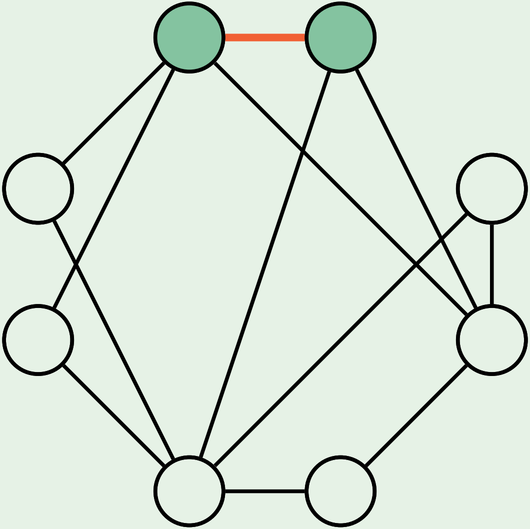
Example



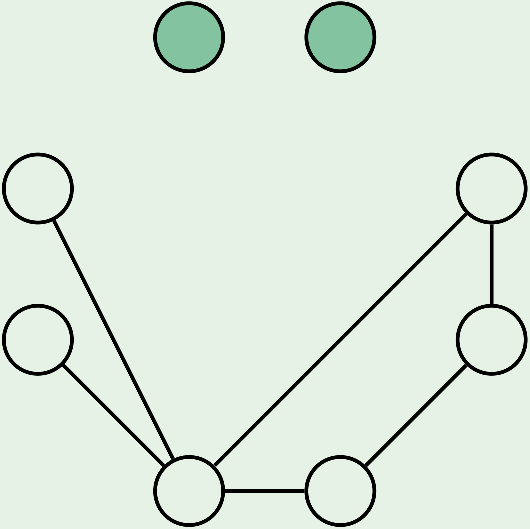
Example



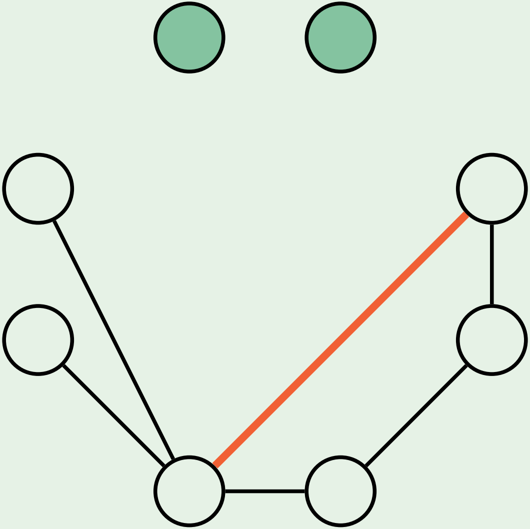
Example



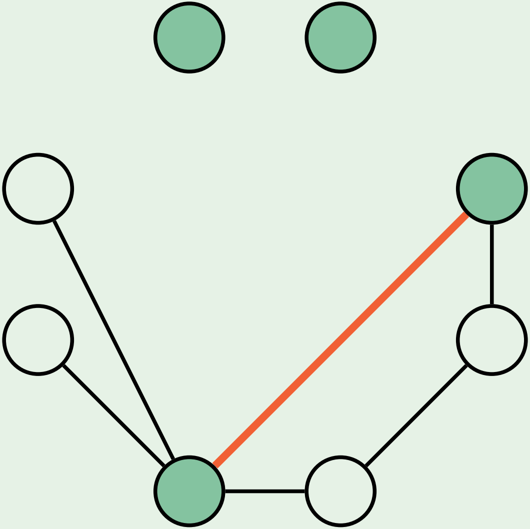
Example



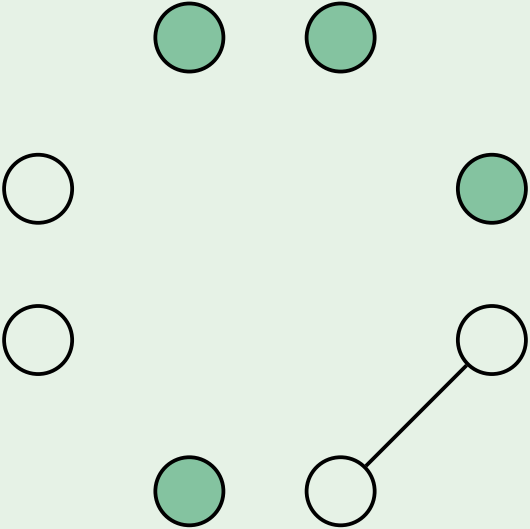
Example



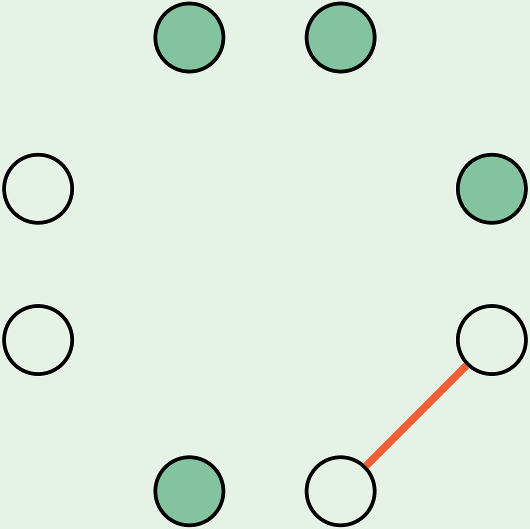
Example



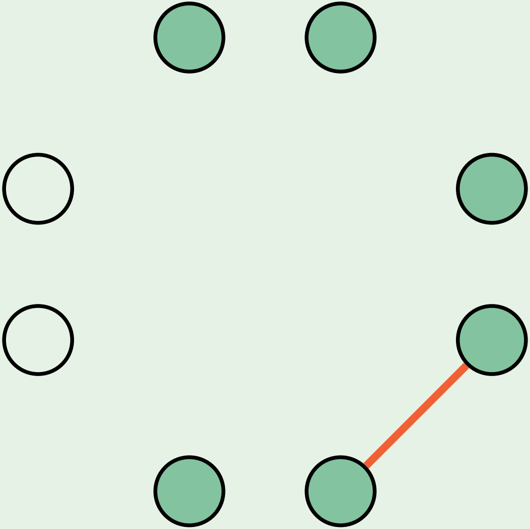
Example



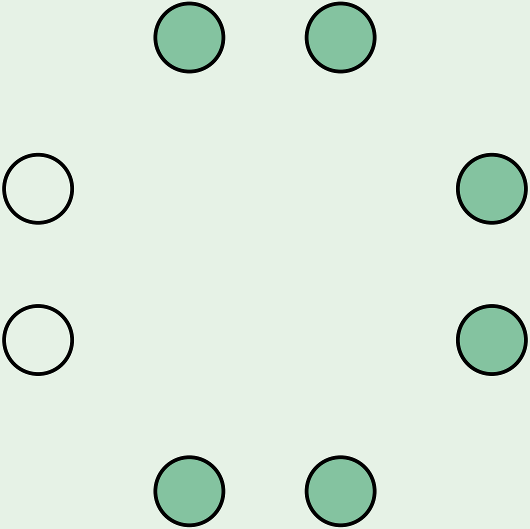
Example



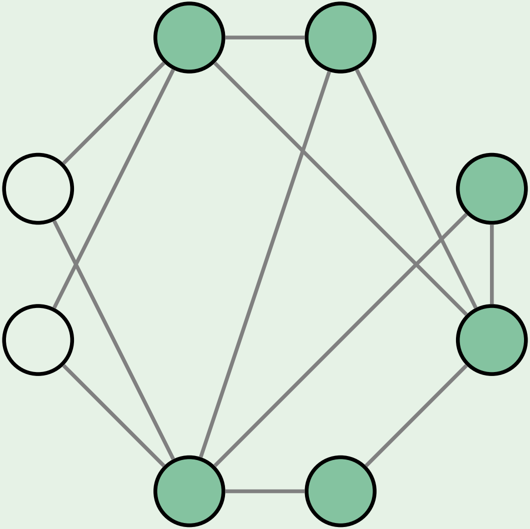
Example



Example



Example



Lemma

The algorithm `ApproxVertexCover` is 2-approximate: it returns a vertex cover that is at most twice as large as an optimal one and runs in polynomial time.

Proof

- The set M of all edges selected by the algorithm forms a matching

Proof

- The set M of all edges selected by the algorithm forms a matching
- Any vertex cover of the graph has size at least $|M|$

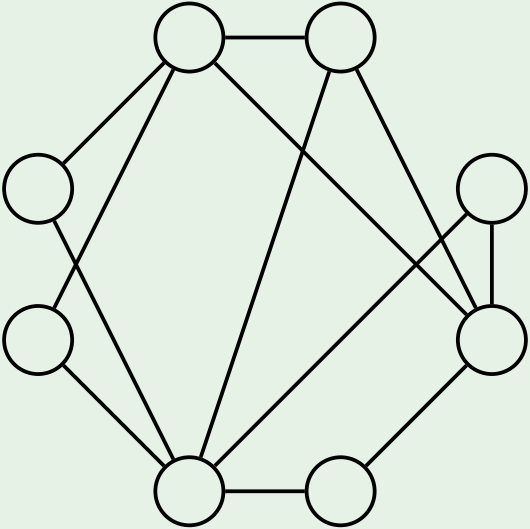
Proof

- The set M of all edges selected by the algorithm forms a matching
- Any vertex cover of the graph has size at least $|M|$
- The algorithm returns a vertex cover C of size $2|M|$, hence

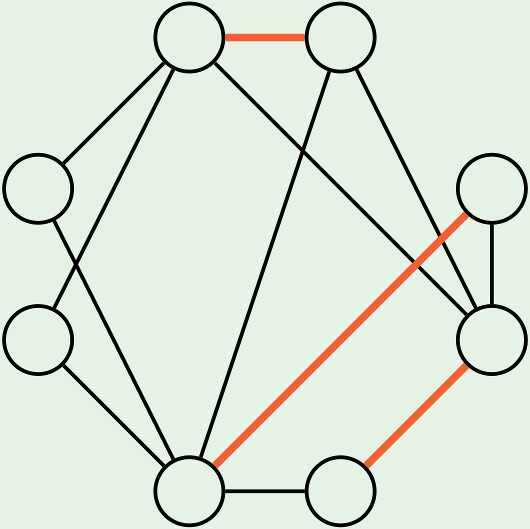
$$|C| = 2 \cdot |M| \leq 2 \cdot \mathrm{OPT}$$

$\square$

Example



Example



Summary

- We don't know the value of OPT, but we've managed to prove that

$$|C| \leq 2 \cdot \mathrm{OPT}$$

Summary

- We don't know the value of OPT, but we've managed to prove that

$$|C| \leq 2 \cdot \mathrm{OPT}$$

- This is because we know a lower bound on OPT: it is at least the size of any matching

$$|C| = 2 \cdot |M| \leq 2 \cdot \mathrm{OPT}$$

Final Remarks

- The bound is tight: there are graphs for which the algorithm returns a vertex cover of size twice the minimum size.

Final Remarks

- The bound is tight: there are graphs for which the algorithm returns a vertex cover of size twice the minimum size.
- No 1.99-approximation algorithm is known.

Outline

1. Vertex cover
2. Traveling salesman
   - Metric TSP
   - Local search

Outline

1. Vertex cover

2. Traveling salesman
   - Metric TSP
   - Local search

Metric TSP (optimization version)

Input: An undirected graph $G(V, E)$ with non-negative edge weights satisfying the triangle inequality: for all $u, v, w \in V$, $d(u, v) + d(v, w) \geq d(u, w)$.

Output: A cycle of minimum total length visiting each vertex exactly once .

Lower Bound

- We are going to design a 2-approximation algorithm: it returns a cycle that is at most twice as long as an optimal cycle: $C \leq 2 \cdot \mathrm{OPT}$

Lower Bound

- We are going to design a 2-approximation algorithm: it returns a cycle that is at most twice as long as an optimal cycle: $C \leq 2 \cdot \mathrm{OPT}$
- Since we don't know the value of OPT, we need a good lower bound L on OPT:

$$C \leq 2 \cdot L \leq 2 \cdot \mathrm{OPT}$$

Minimum Spanning Trees

Lemma

Let G be an undirected graph with non-negative edge weights. Then $\mathsf{MST}(G) \leq \mathsf{TSP}(G)$.

Minimum Spanning Trees

Lemma

Let G be an undirected graph with non-negative edge weights. Then $\mathrm{MST}(G) \leq \mathrm{TSP}(G)$.

Proof

By removing any edge from an optimum TSP cycle one gets a spanning tree of G. $\square$

ApproxMetricTSP(G)

$T \leftarrow$ `minimum spanning tree of` G

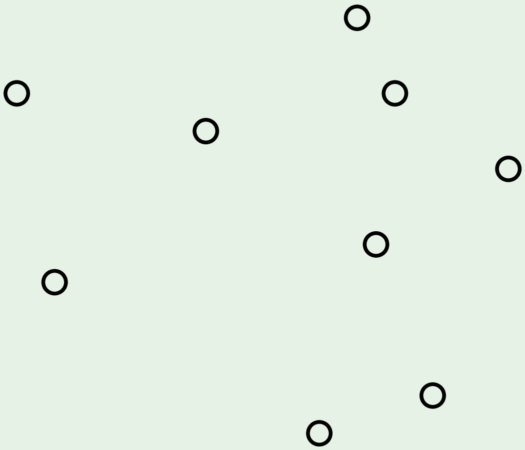
ApproxMetricTSP(G)

$T \leftarrow$ `minimum spanning tree of` G
$D \leftarrow T$ `with each edge doubled`

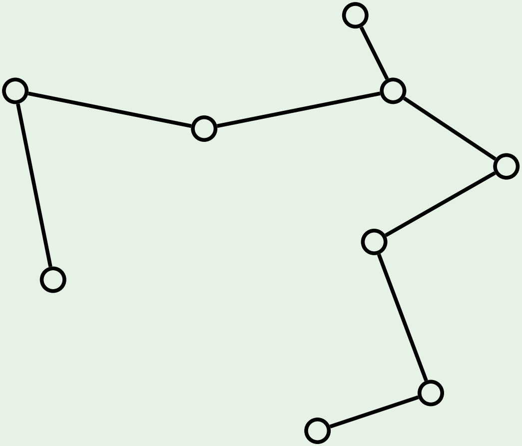
```
ApproxMetricTSP(G)

T ← minimum spanning tree of G
D ← T with each edge doubled
find an Eulerian cycle C in D
```

```
ApproxMetricTSP(G)
T ← minimum spanning tree of G
D ← T with each edge doubled
find an Eulerian cycle C in D
return a cycle that visits vertices in
  the order of their first appearance in C
```

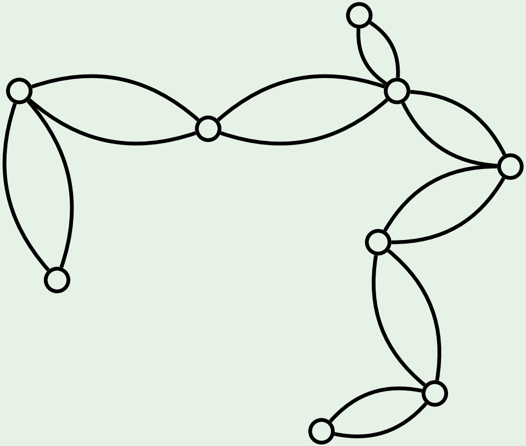
Example: points on a plane



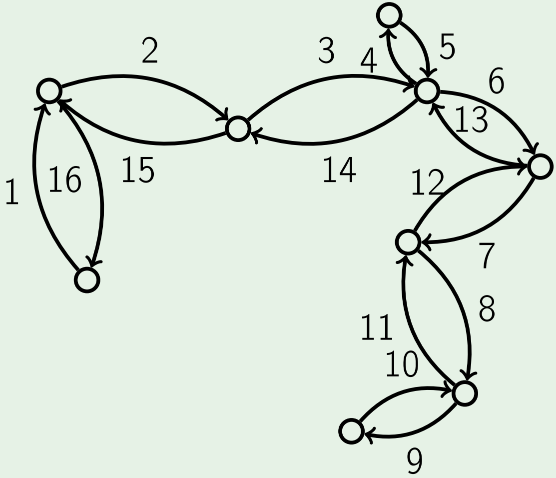
Example: points on a plane



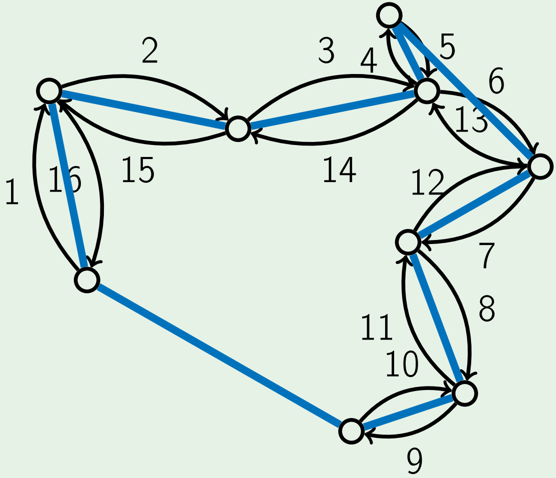
Example: points on a plane



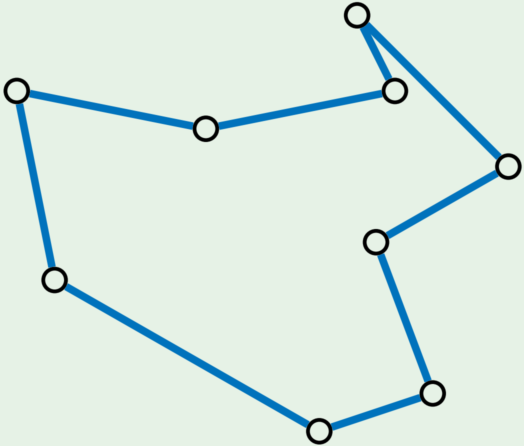
Example: points on a plane



Example: points on a plane



Example: points on a plane



Lemma

The algorithm `ApproxMetricTSP` is 2-approximate.

Lemma

The algorithm `ApproxMetricTSP` is 2-approximate.

Proof

- The total length of the MST T is at most OPT.

Lemma

The algorithm `ApproxMetricTSP` is 2-approximate.

Proof

- The total length of the MST T is at most OPT.
- Bypasses can only decrease the total length. $\square$

Final Remarks

- The currently best known approximation algorithm for metric TSP is Christofides' algorithm that achieves a factor of 1.5

Final Remarks

- The currently best known approximation algorithm for metric TSP is Christofides' algorithm that achieves a factor of 1.5
- If $\mathbf{P} \neq \mathbf{NP}$, then there is no α-approximation algorithm for the general version of TSP for any polynomial time computable function α

Outline

1. Vertex cover

2. Traveling salesman
   - Metric TSP
   - Local search

LocalSearch

$s \leftarrow$ some initial solution
while there is a solution s' in the neighborhood of s which is better than s:
    $s \leftarrow s'$
return s

LocalSearch

```
s ← some initial solution
while there is a solution s′ in the
neighborhood of s which is better than s:
  s ← s′
return s
```

- Computes a local optimum instead of a global optimum

LocalSearch

```
s ← some initial solution
while there is a solution s′ in the
neighborhood of s which is better than s:
  s ← s′
return s
```

- Computes a local optimum instead of a global optimum
- The larger is the neighborhood, the better is the resulting solution and the higher is the running time

Local Search for TSP

- Let s and s' be two cycles visiting each vertex of the graph exactly once

Local Search for TSP

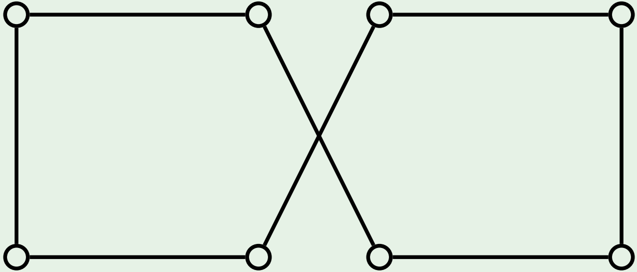
- Let s and s' be two cycles visiting each vertex of the graph exactly once
- The distance between s and s' is at most d, if one can get s' by deleting d edges from s and adding other d edges

Local Search for TSP

- Let s and s' be two cycles visiting each vertex of the graph exactly once
- The distance between s and s' is at most d, if one can get s' by deleting d edges from s and adding other d edges
- Neighborhood $N(s, r)$ with center s and radius r: all cycles with distance at most r from s

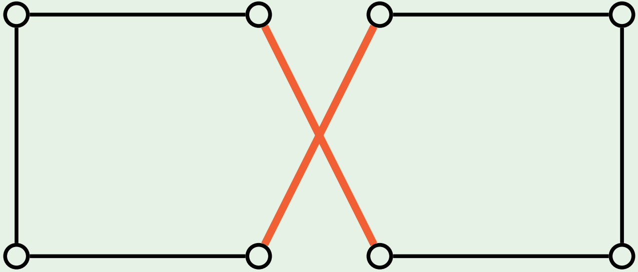
Example

Changing two edges in a suboptimal solution:



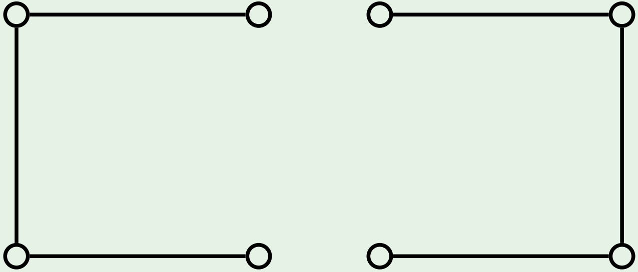
Example

Changing two edges in a suboptimal solution:



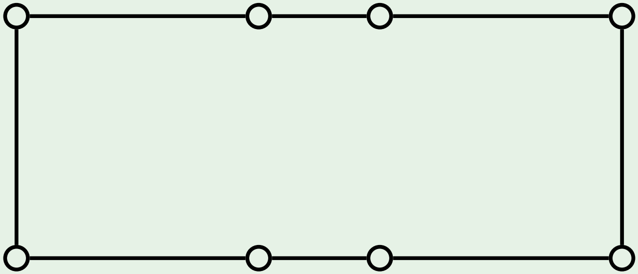
Example

Changing two edges in a suboptimal solution:



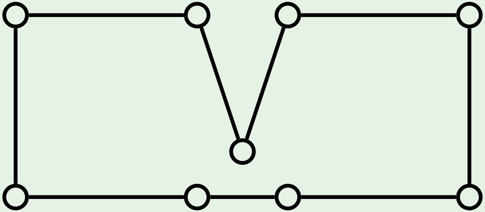
Example

Changing two edges in a suboptimal solution:



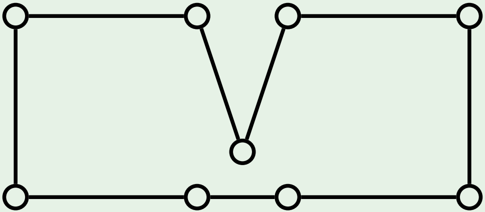
Example

A suboptimal solution that cannot be improved by changing two edges:



Example

A suboptimal solution that cannot be improved by changing two edges:



Need to allow changing three edges to improve this solution

Performance

- Trade-off between quality and running time of a single iteration

Performance

- Trade-off between quality and running time of a single iteration
- Still, the number of iterations may be exponential and the quality of the found cycle may be poor

Performance

- Trade-off between quality and running time of a single iteration
- Still, the number of iterations may be exponential and the quality of the found cycle may be poor
- But works well in practice

Coping with NP-completeness

- special cases
- intelligent exhaustive search
- approximation algorithms