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

Brute force search algorithm finds an optimal solution enumerating through all possible candidate solutions and selecting the best one

Outline

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 $O(|F| \cdot 2^n)$.

It goes through all the variables of a formula and checks whether any of them satisfies the input formula . Running time is proportional to length of F times 2^n because for 2^n assignments we have to scan the formula to check whether all of the clauses are satisfied or not

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

Goal

Avoid going through all 2^n assignments

Outline

Main Idea of Backtracking

- Construct a solution piece by piece

Main Idea of Backtracking

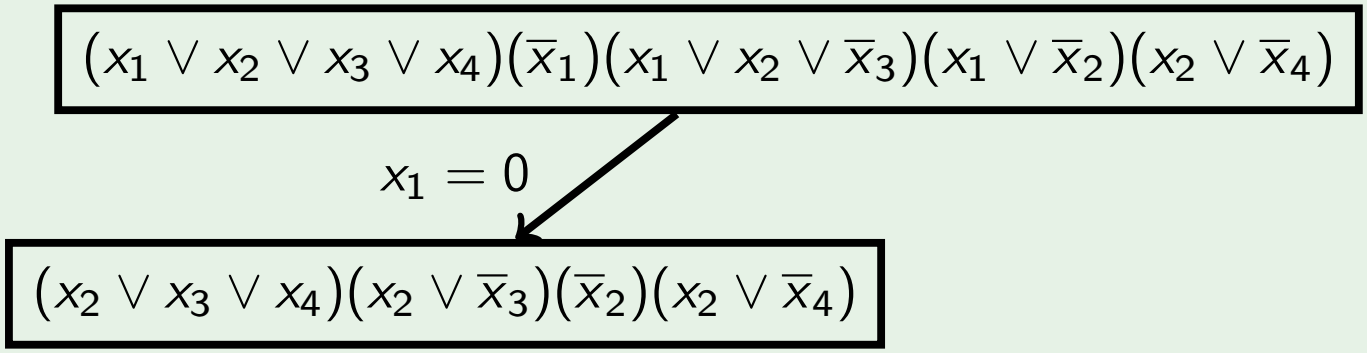
- Construct a solution piece by piece
- Backtrack if the current partial solution cannot be extended to a valid solution

Similar idea to that of playing game to solve a maze

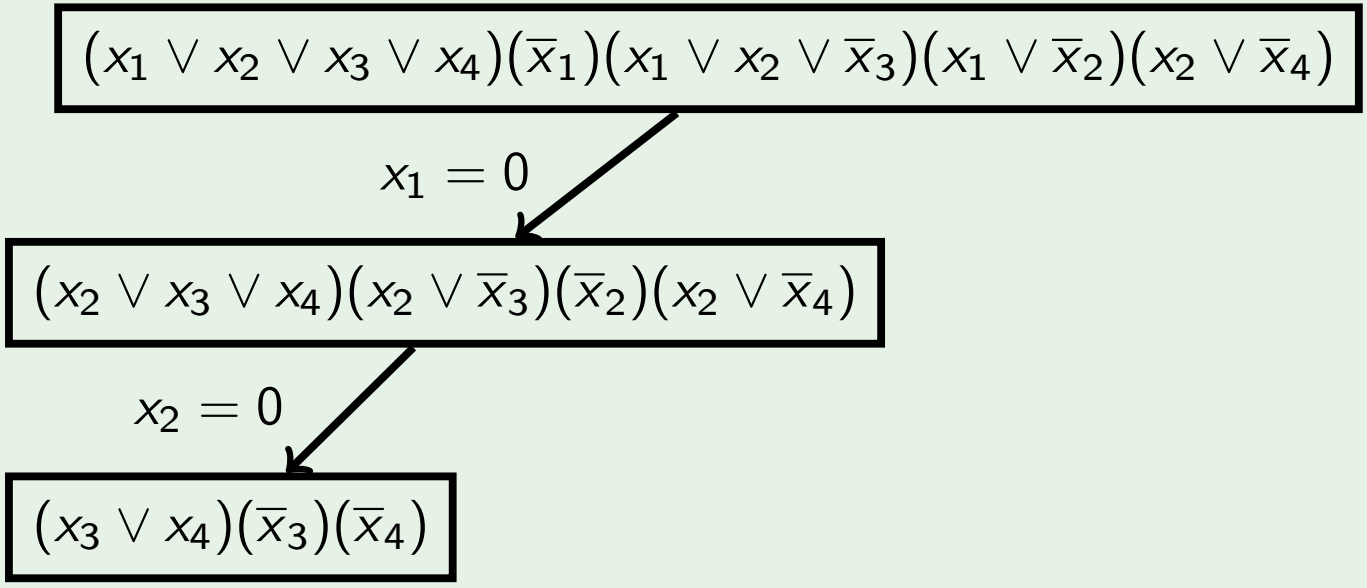
Example

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

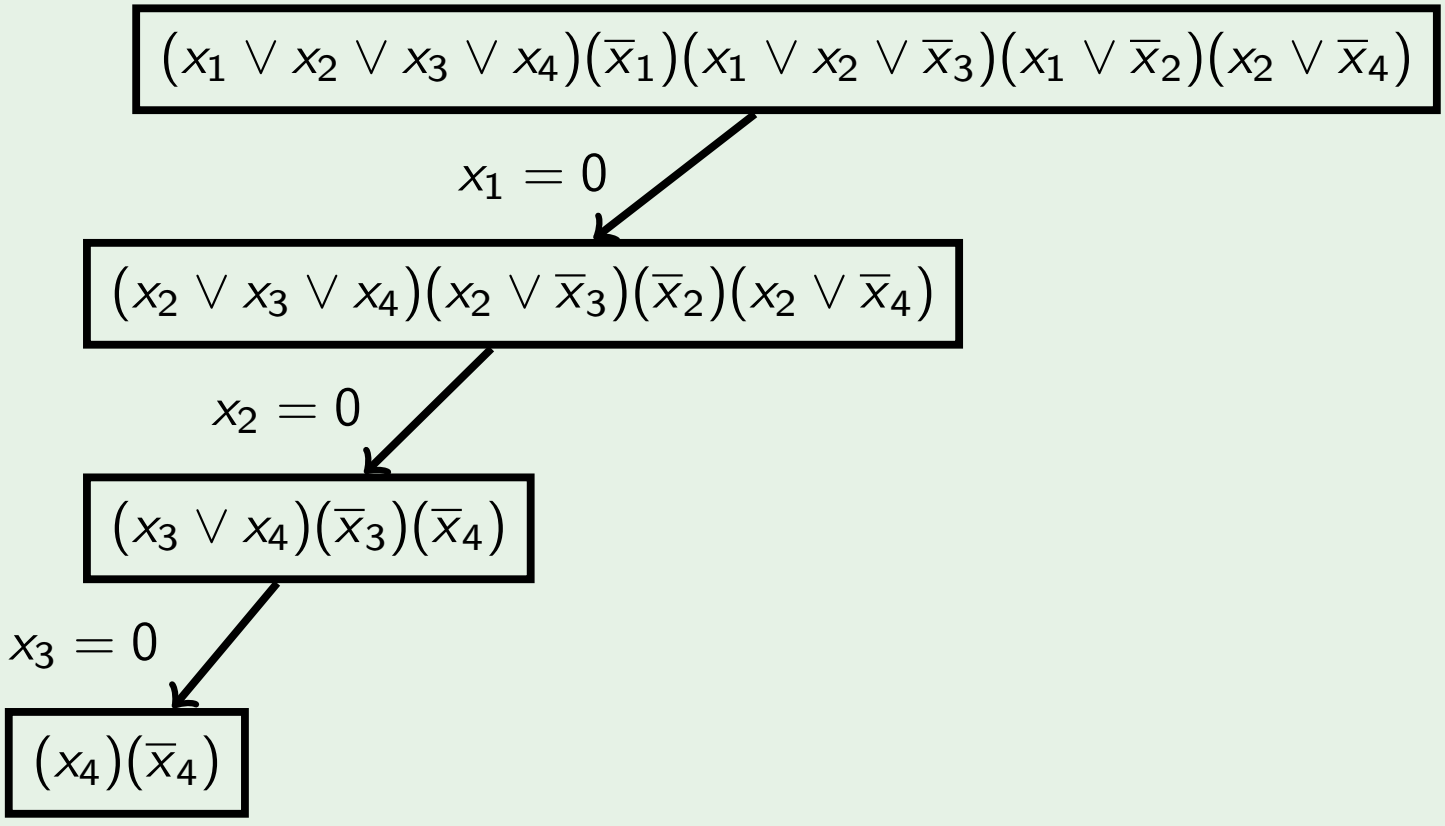
Example


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

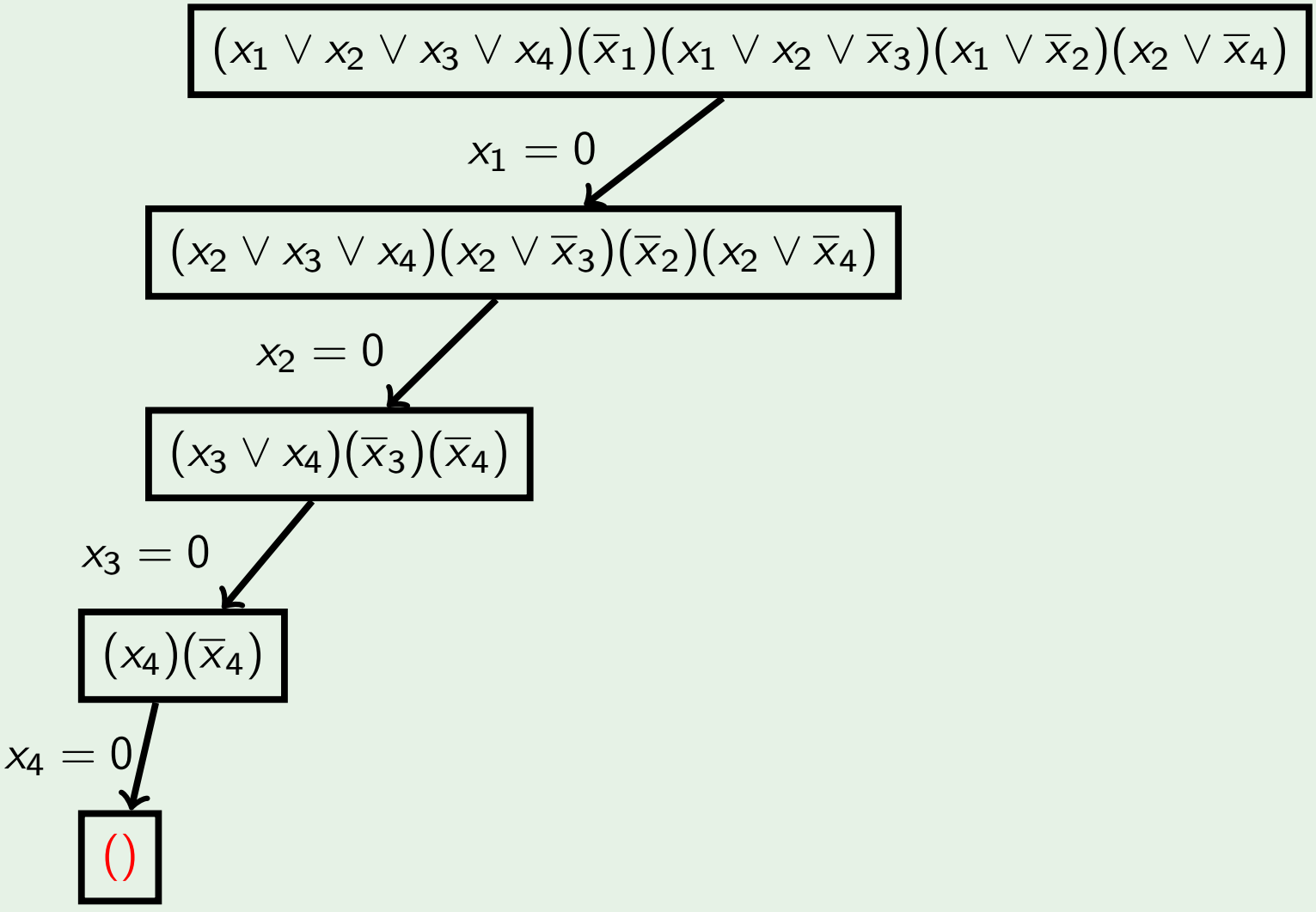
Example


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

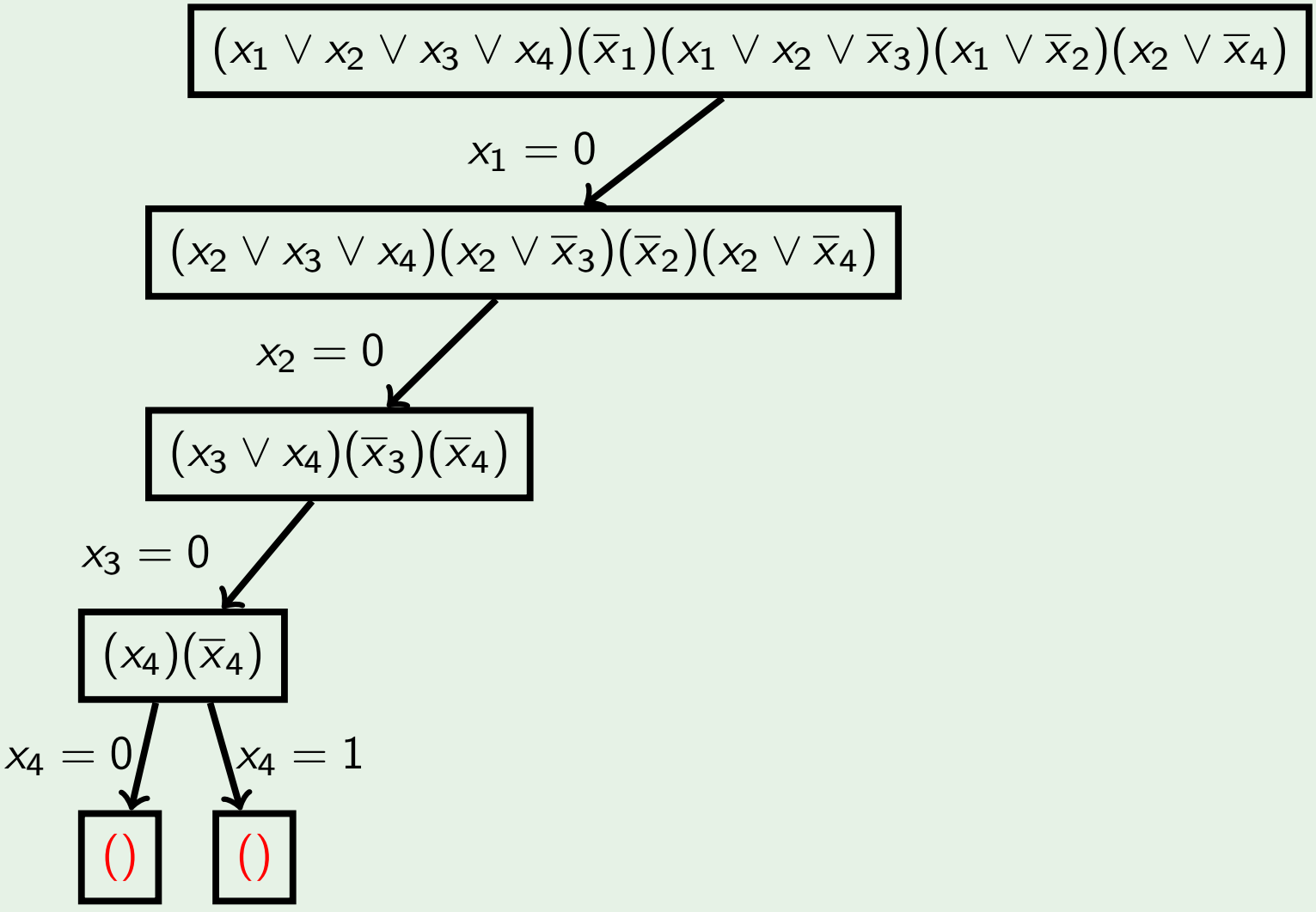
Example


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

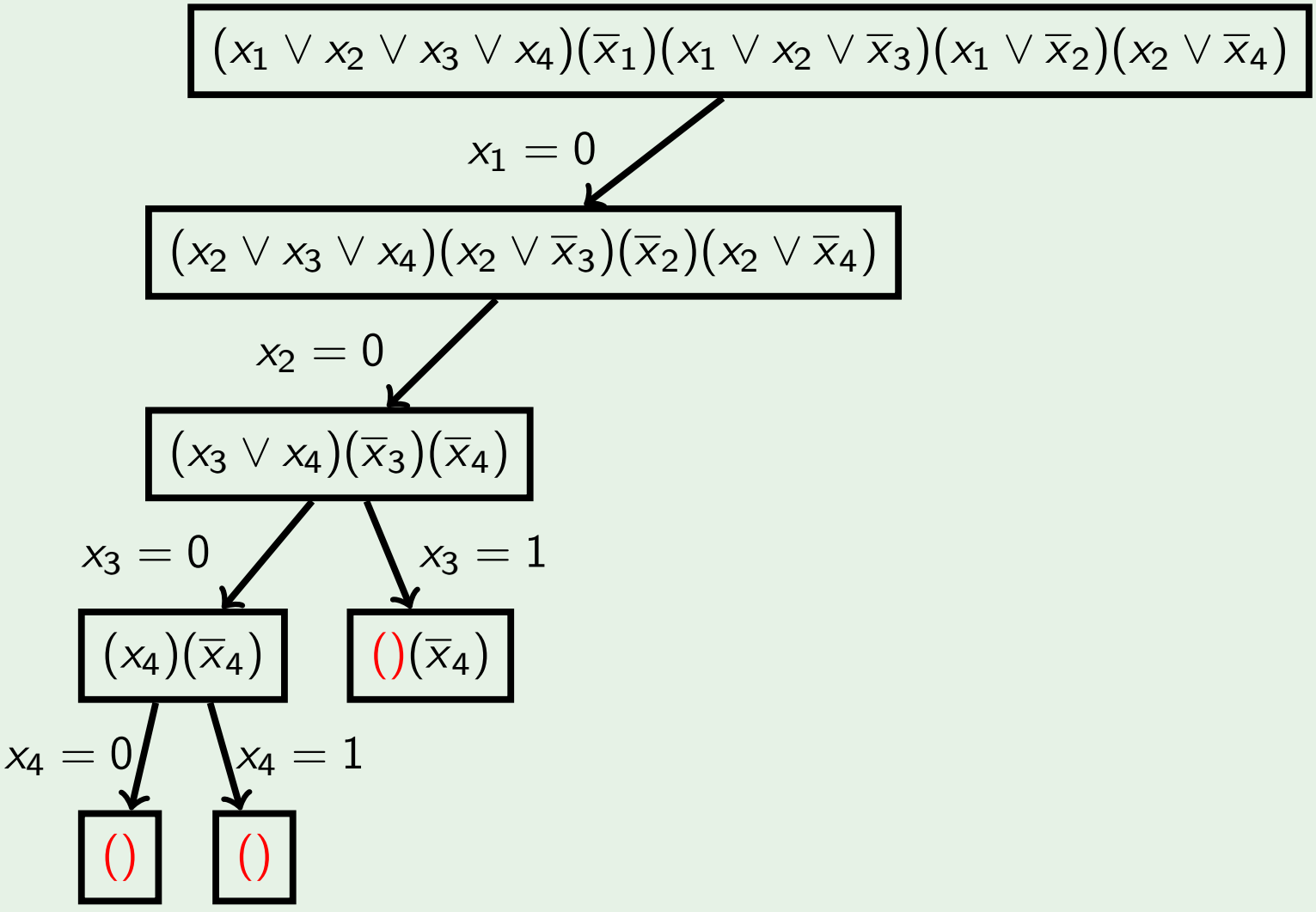
Example


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

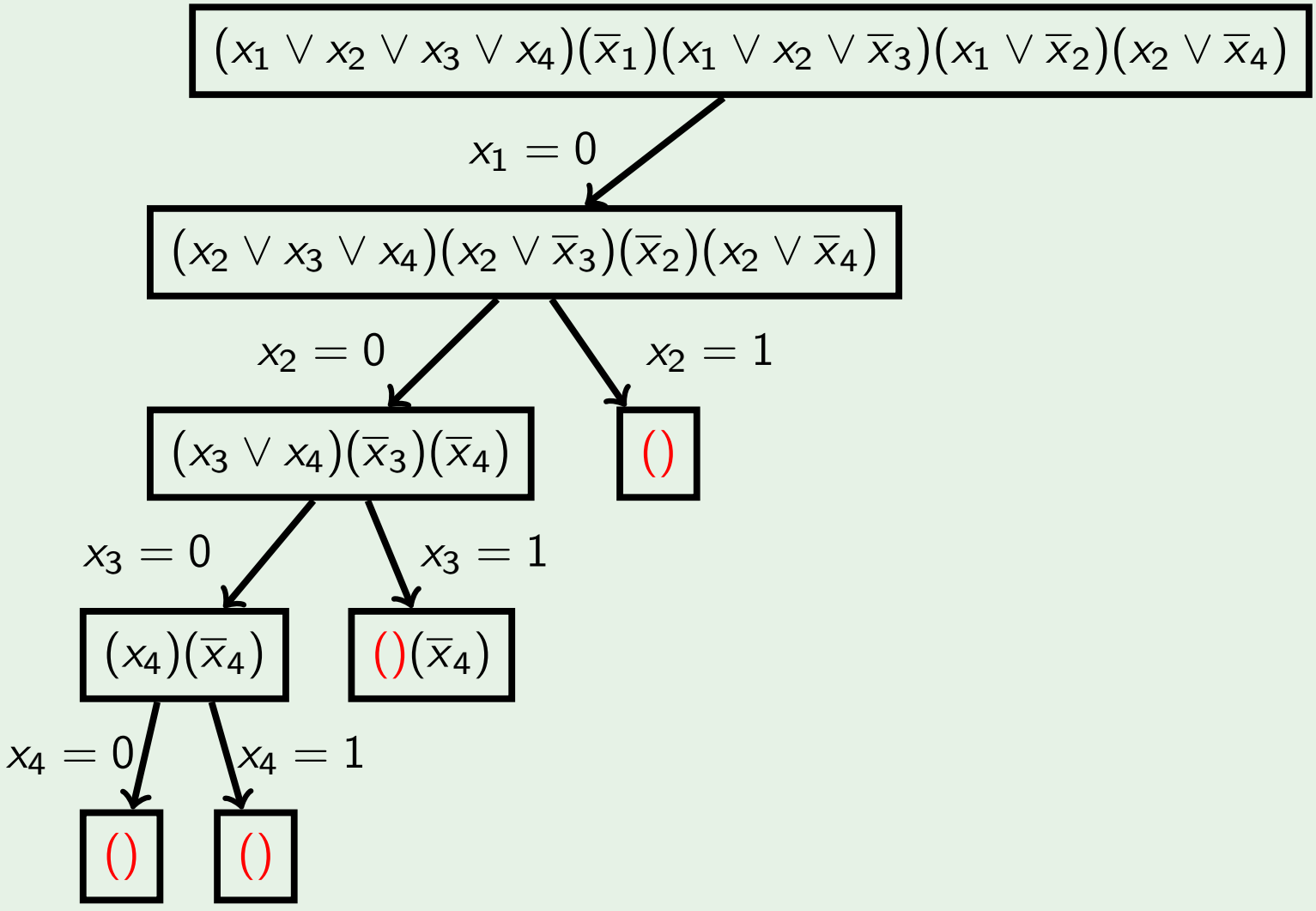
Example


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

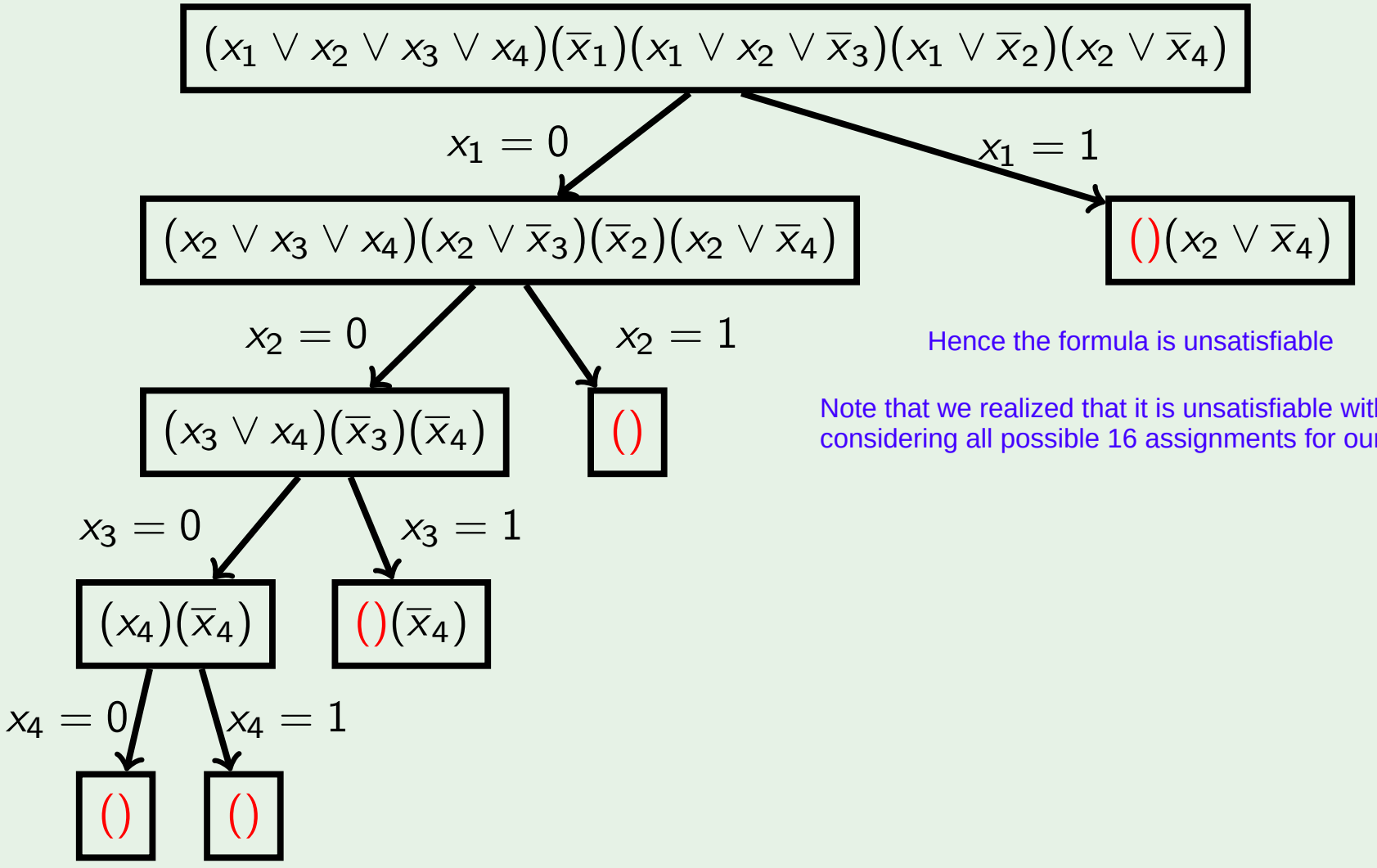
Example


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

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



Hence the formula is unsatisfiable

Note that we realized that it is unsatisfiable without considering all possible 16 assignments for our 4 variables

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"
```

Empty clause is a clause which cannot be satisfied meaning all the literals has already been falsified AND THE ANSWER TURNS to 0

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"
```

Assigning 0 to x. This means all x! are satisfied. After this step we remove all occurrences of x! from the formula. Also we remove all occurrences of x from all other clauses because x is falsified and now we are left with SAT formula without x

SolveSAT(F)

if F has no clauses:
  return ‘‘sat’’
if F contains an empty clause:
  return ‘‘unsat’’
$x \leftarrow$ unassigned variable of F
if SolveSAT($F[x \leftarrow 0]$) = ‘‘sat’’:
  return ‘‘sat’’
if SolveSAT($F[x \leftarrow 1]$) = ‘‘sat’’:
  return ‘‘sat’’

Otherwise we backtrack and assign x = 1

SolveSAT(F)

if F has no clauses:

  return "sat"

if F contains an empty clause:

  return "unsat"

$x \leftarrow$ unassigned variable of F

if SolveSAT($F[x \leftarrow 0]$) = "sat":

  return "sat"

if SolveSAT($F[x \leftarrow 1]$) = "sat":

  return "sat"

return "unsat"

If the CNF is satisfied with x=1 or x=0 then the next formula will have x removed from it as shown in the previous slide

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

Question

Assume that a CNF formula with n variables has k satisfying assignments. What is the maximum possible number of leaves in the recursion tree of a backtracking algorithm solving satisfiability of this formula?

- ○ $2^n - k$
- ◉ $2^n - k + 1$
- ○ 2^k
- ○ k

✓ **Correct**

That's right! Each leaf of the recursion tree corresponds to a partial assignment that either satisfies the formula or falsifies it. The algorithm stops if it finds a satisfying assignment. Hence, in the worst case it will first go through all $2^n - k$ falsifying assignments, then will find a satisfying assignment and halt.

- 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

We didn't prove the upper bound on the running time of this algorithm. As expected it must be exponential because if it is polynomial then it is no longer a NP problem

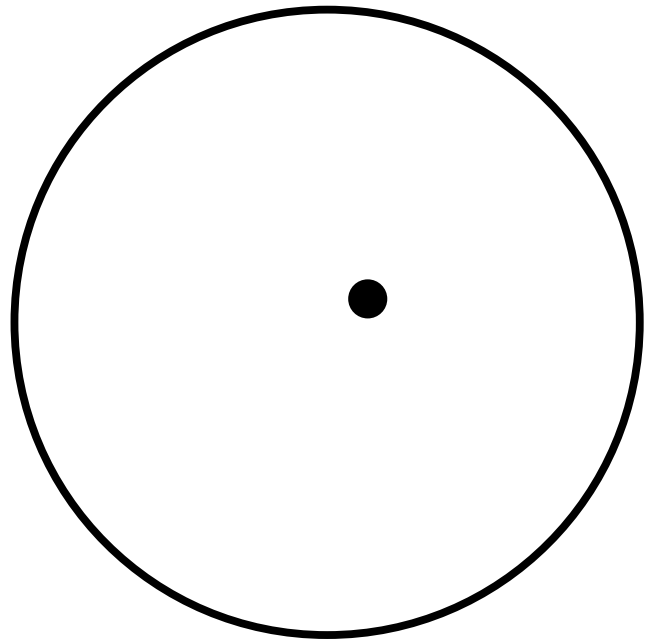
- Backtracking is used in many state-of-the-art SAT-solvers
- SAT-solvers use tricky heuristics to choose a variable to branch on and to simplify a formula before branching
- Another commonly used technique is local search — will consider it in the next part

Note that the upper bound on the running time of the algorithm might be exponential

Outline

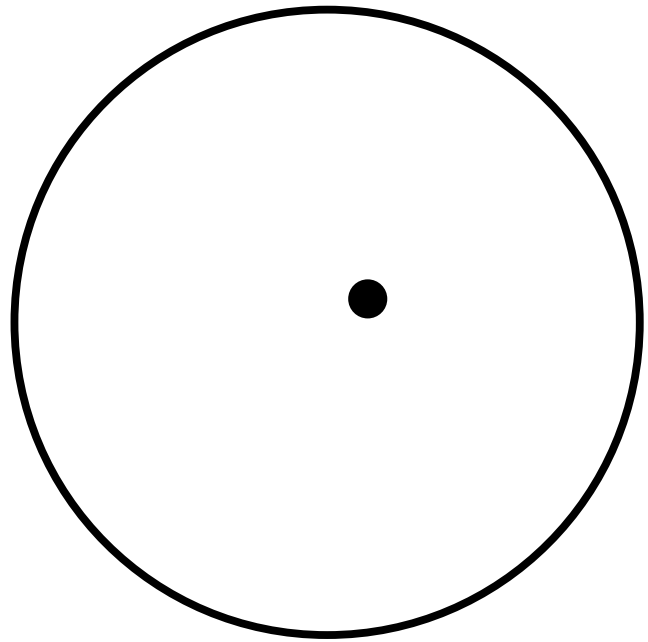
Main Idea of Local Search

- Start with a candidate solution



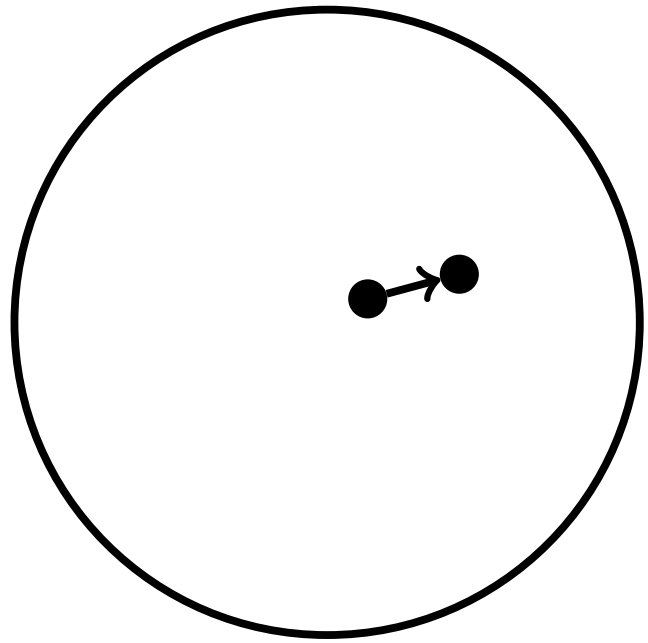
Main Idea of Local Search

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



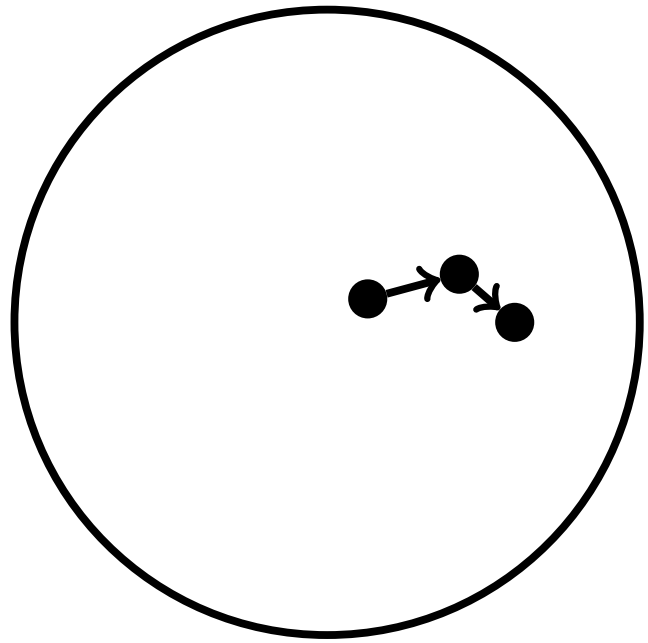
Main Idea of Local Search

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



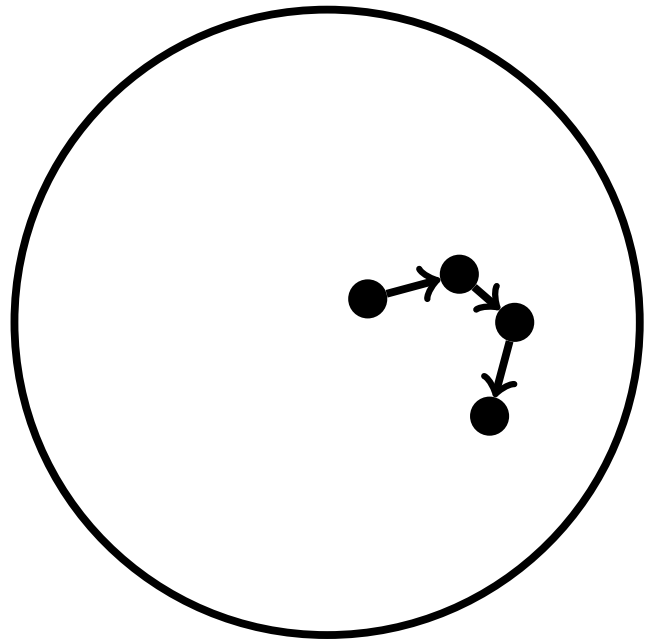
Main Idea of Local Search

- 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:
$\text{dist}(\alpha, \beta) = |\{i : \alpha_i \neq \beta_i\}|$.

Definition

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

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

We already know that it contains a satisfying assignment

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

The satisfying assignment that we are going to find might coincide with B or might be a completely different one but if we know that it contains satisfying assignment then we can find it in this running time

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, $(x_i \vee \overline{x}_j \vee x_k)$

take unsatisfied clause from the CNF Formula

- 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

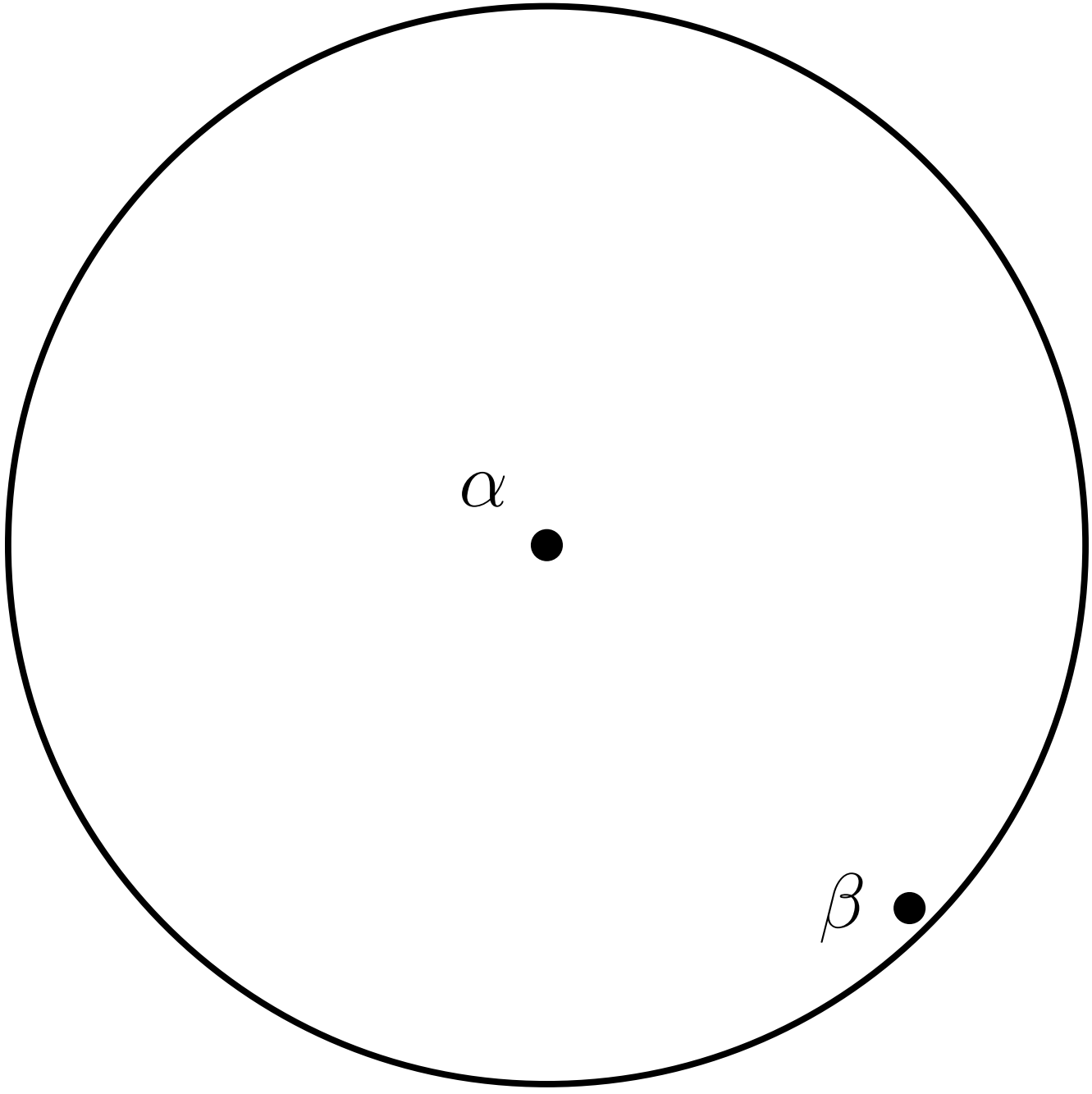
Proof

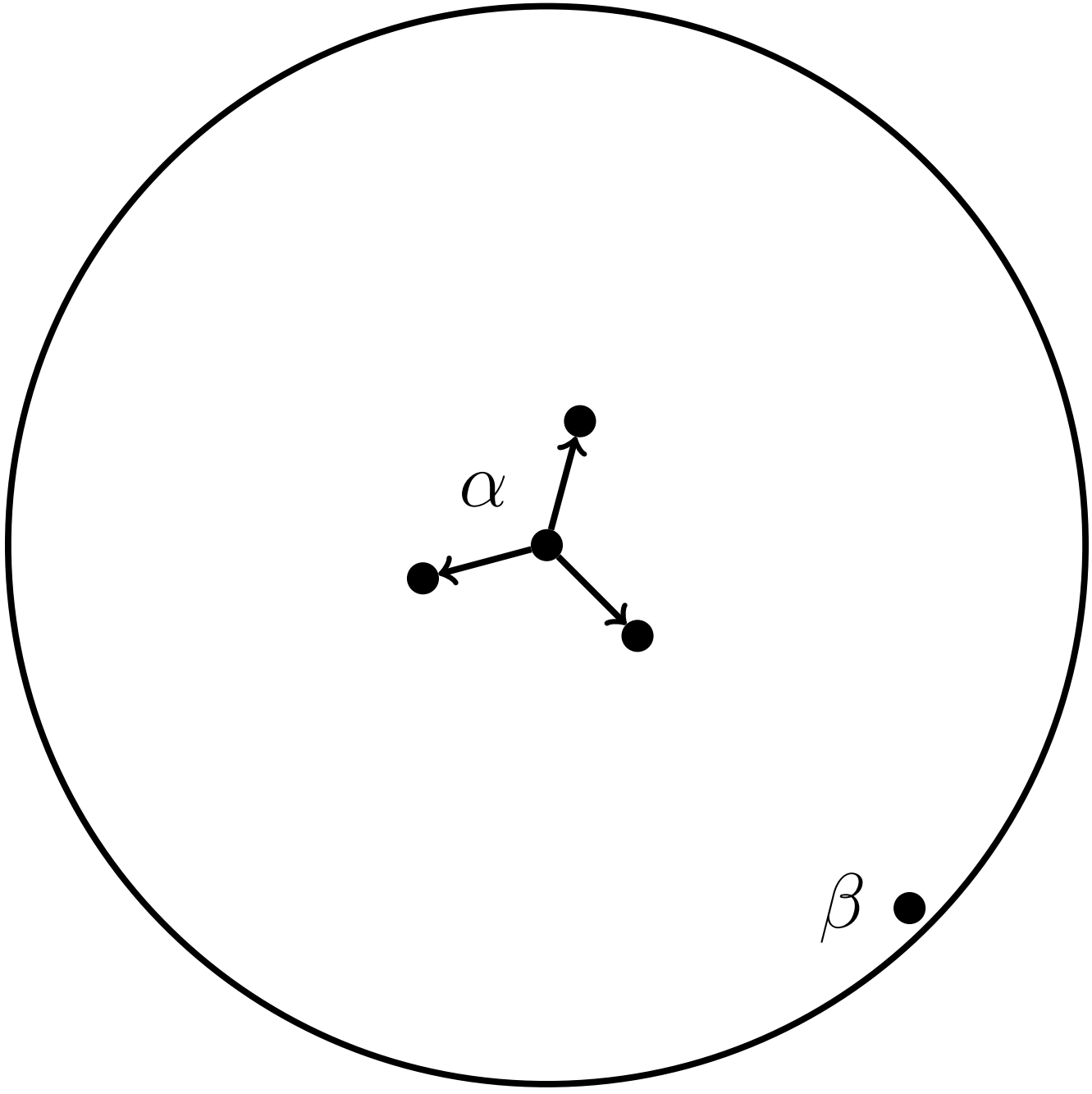
- If α satisfies F, return α
- Otherwise, take an unsatisfied clause — say, $(x_i \vee \overline{x}_j \vee x_k)$
- α assigns $x_i = 0, x_j = 1, x_k = 0$
- Let $\alpha^i, \alpha^j, \alpha^k$ be assignments resulting from α by flipping the i-th, j-th, k-th bit, respectively
- Crucial observation: at least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

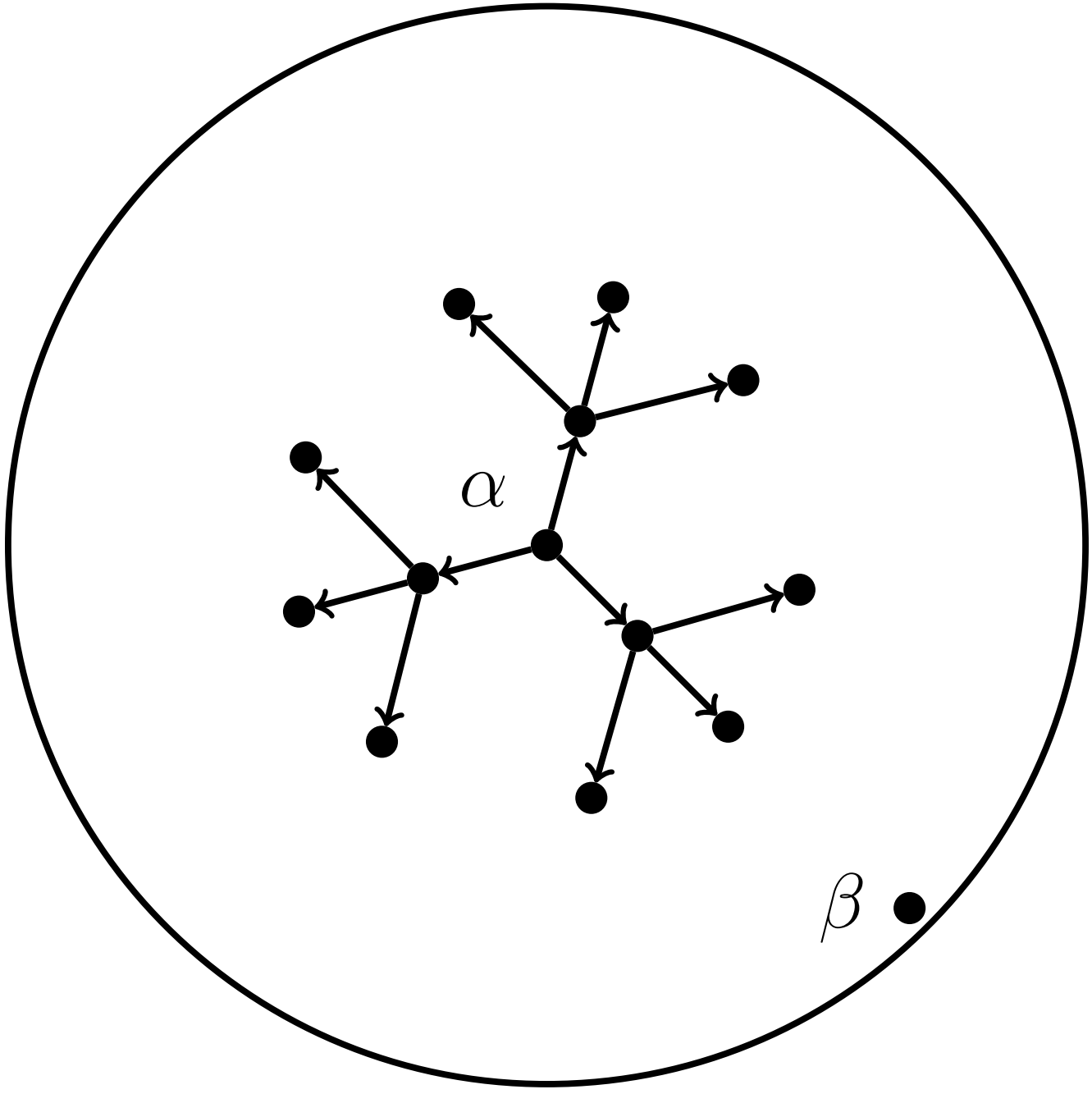
Proof

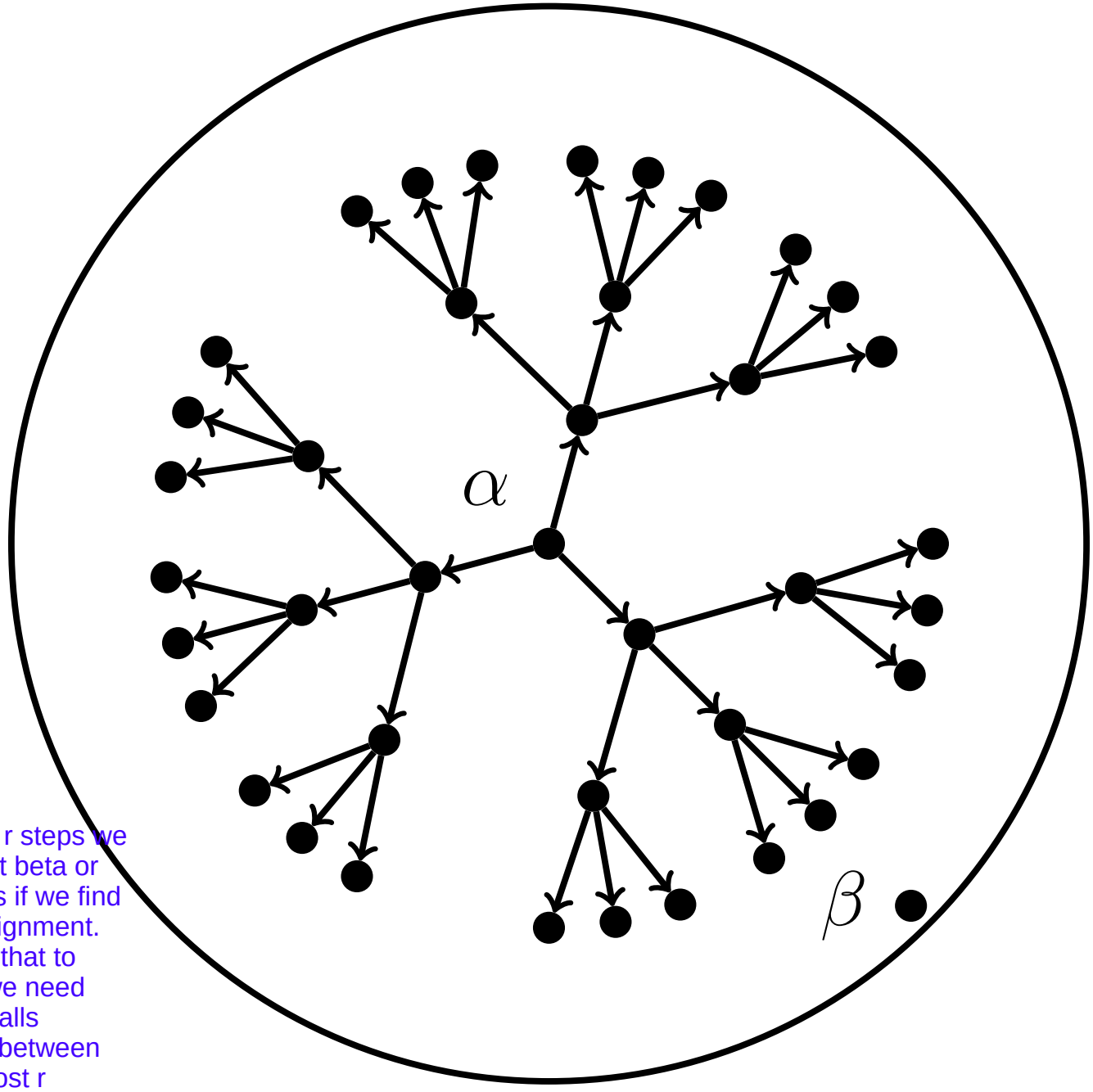
- If α satisfies F, return α
- Otherwise, take an unsatisfied clause — say, $(x_i \vee \overline{x}_j \vee x_k)$
- α assigns $x_i = 0, x_j = 1, x_k = 0$
- Let $\alpha^i, \alpha^j, \alpha^k$ be assignments resulting from α by flipping the i-th, j-th, k-th bit, respectively
- Crucial observation: at least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

because there are 3 variables and the difference between alpha and beta is atmost r


α
β


α
β


α
β



Eventually if we make r steps we will find an assignment beta or we will stop before this if we find another satisfying assignment. Crucial observation is that to eventually find beta, we need atmost 3^r recursive calls because the distance between alpha and beta is atmost r

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"

take any unsatisfied clause from the CNF Formula

$x_i, x_j, x_k \leftarrow$ 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"

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

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

CheckBall$(F, \alpha^i, r-1)$ If Satisfying assignment is found with alpha i return immediately and so on

CheckBall$(F, \alpha^j, r-1)$

CheckBall$(F, \alpha^k, r-1)$

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)
if a satisfying assignment is found:
  return it
else:
  return "not found"
```

- Assume that F has a satisfying assignment β

Now we need to understand HOW TO USE this procedure which checks whether a ball contains a satisfying 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

- 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

Make only these two calls to the original CheckBall Algorithm

- Thus, it suffices to make two calls: `CheckBall`$(F, 11\ldots1, n/2)$ and `CheckBall`$(F, 00\ldots0, n/2)$

Thus total running time will be 3^r + 3^r = 2(3^r) =2(3^(n/2)) ~= 3^(n/2)

Running Time

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

Question

The branch-and-bound algorithm from the previous video can be used for general CNF formulas, not only for 3-CNF formulas. Is it true that the local search algorithm considered in this video also finds a satisfying assignment for any CNF formula with n variables in time roughly $3^{n/2}$?

- ○ Yes, this is true.
- ◉ No, this is not true.

✓ **Correct**
That's right! It is essential for the algorithm that each clause has at most three literals. This ensures that the recursion tree has branching factor 3. This, in turn, implies a $3^{n/2}$ upper bound on its running time.

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

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.

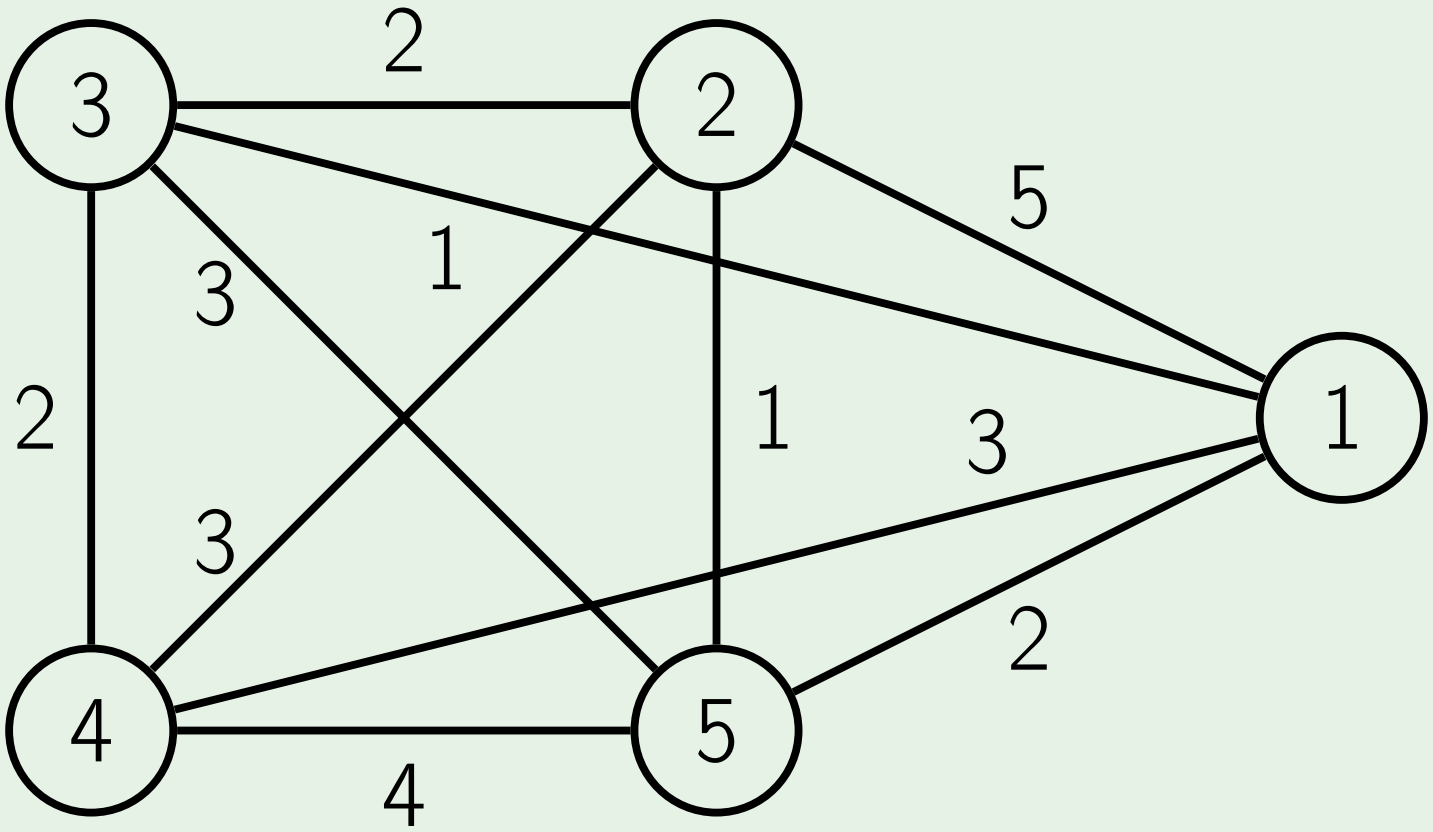
Traveling salesman problem (TSP)

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

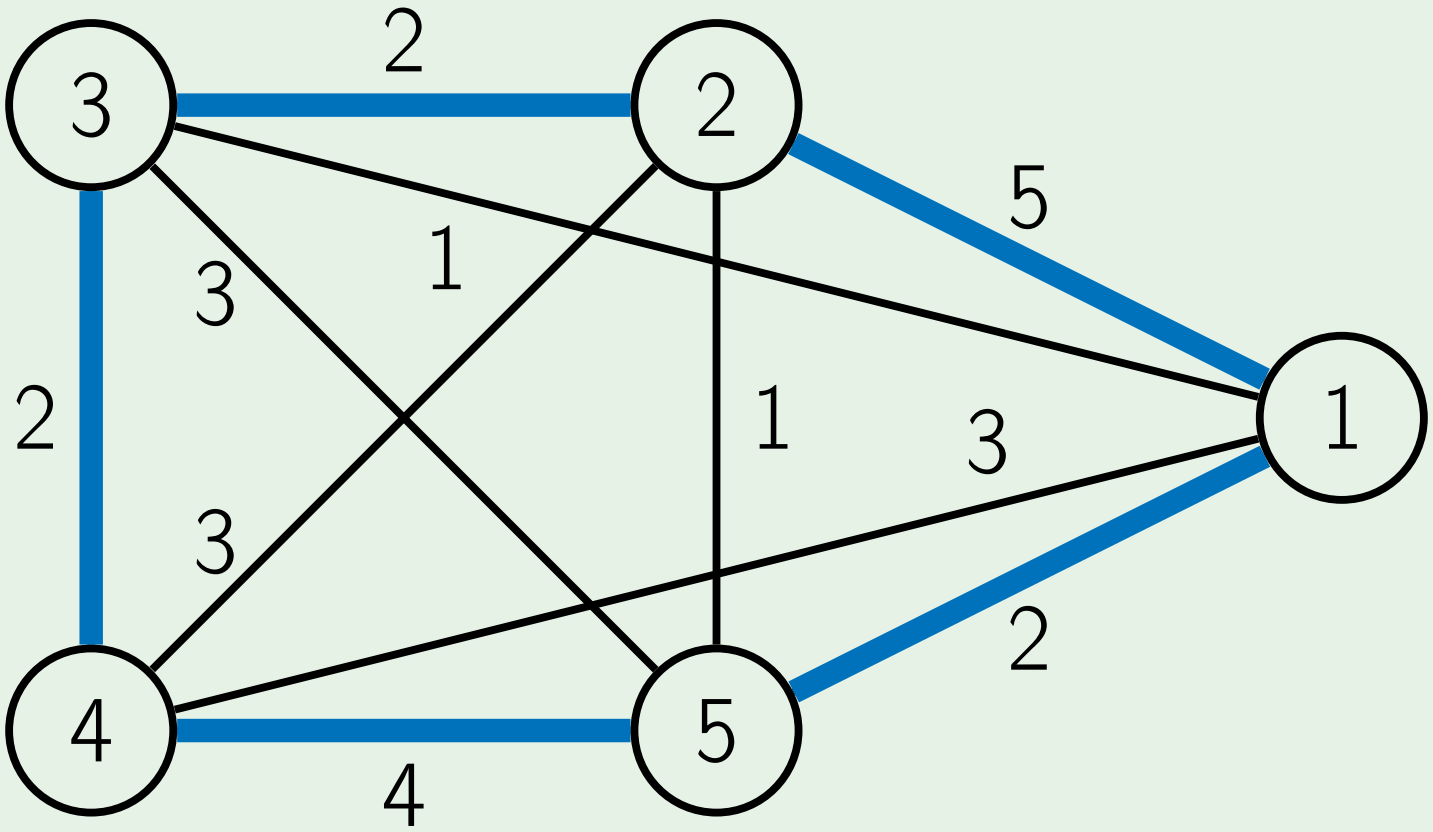
Output: A cycle that visits each vertex exactly once and has total weight at most b.

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

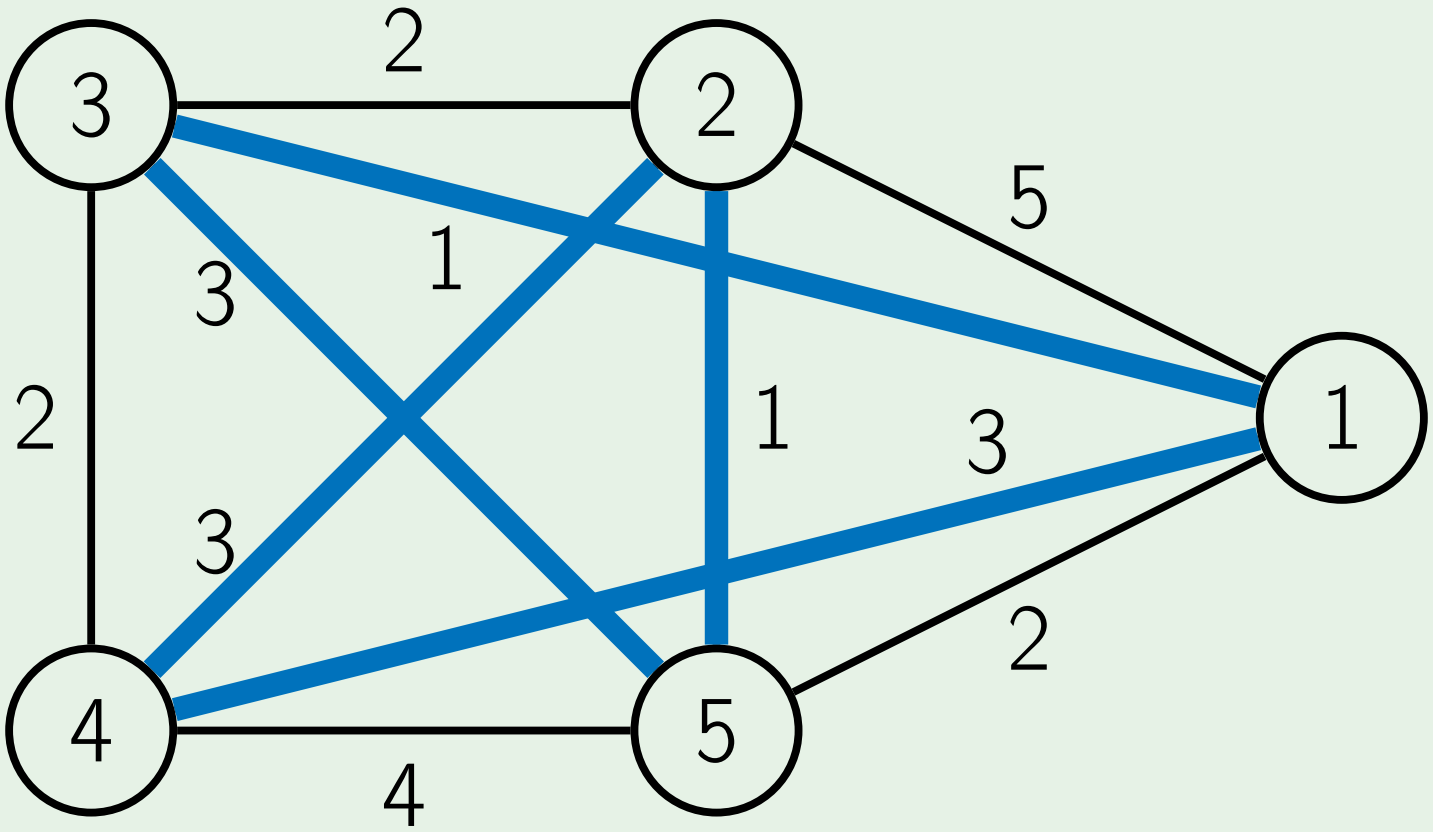


Example



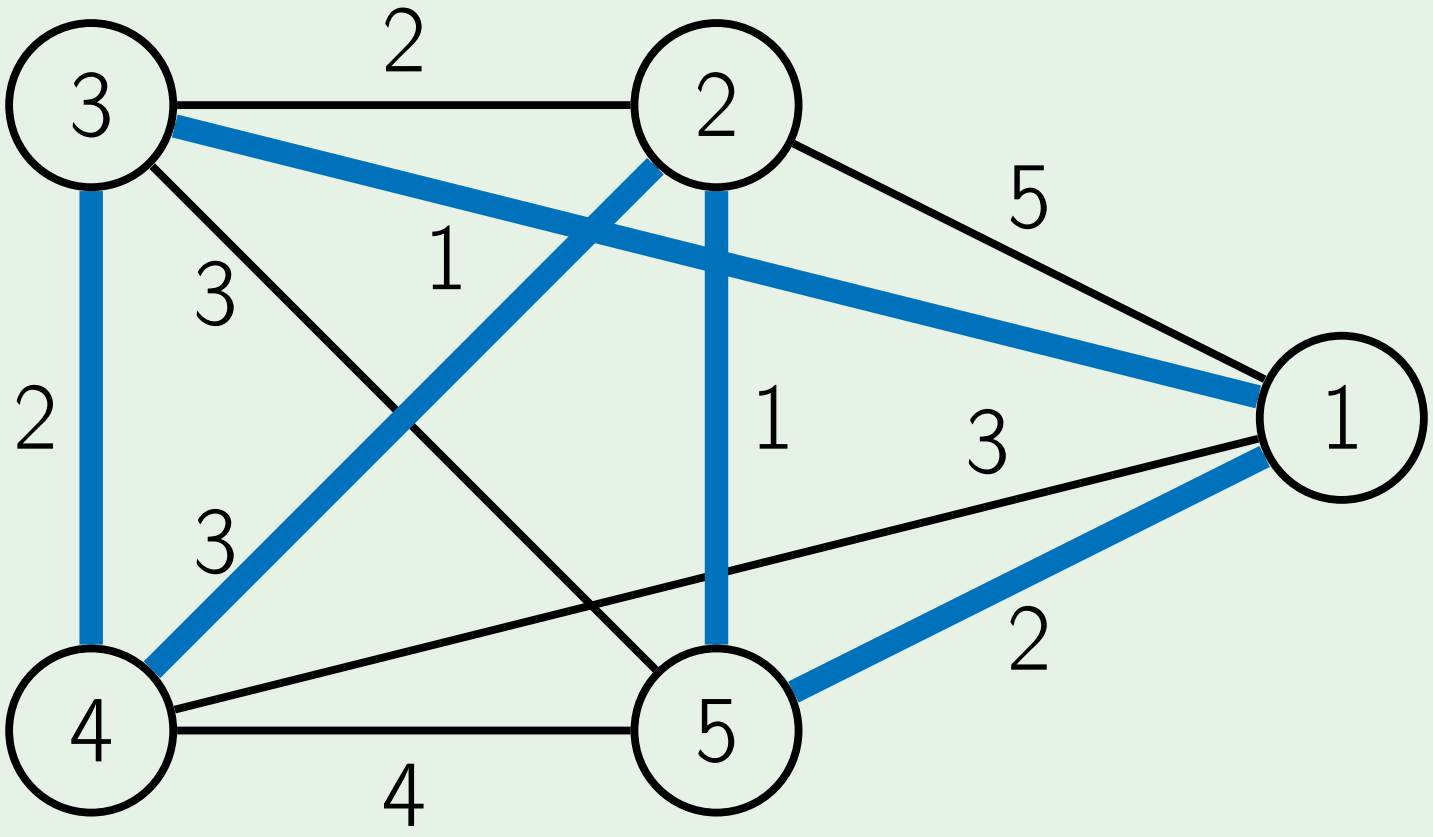
length: 15

Example



length: 11

Example



length: 9

Brute Force Solution

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

because initially we start with vertex 1 and then we have (n-1)! choices. Of these (n-1)! vertices we have (n-2)! choices and so on.

Brute Force Solution

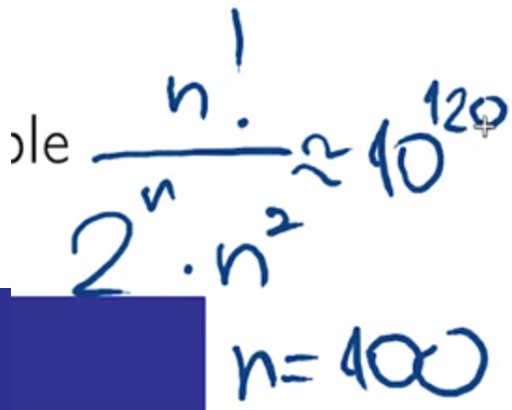
A naive algorithm just checks all possible $(n-1)!$ cycles.

This part

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

Brute Force Solution

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

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 $S \subseteq \{1, \ldots, n\}$ 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
- $C(\{1\}, 1) = 0$ and $C(S, 1) = +\infty$ when $|S| > 1$

For any other vertices in S the path cannot end in vertex 1 hence the distance is infinite

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 $S \subseteq \{1, \ldots, n\}$ 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

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

  Sp if we just go through all sub sets in order of increase in size then we are guaranteed to have the values of C(S-{i},j) when computing the value of C(S,i)

- For example, we can process subsets in order of increasing size

$\mathrm{TSP}(G)$

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

TSP(G)

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

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

TSP(G)

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

```
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)

$C(\{1\}, 1) \leftarrow 0$
for s from 2 to n:
  for all $1 \in S \subseteq \{1, \ldots, n\}$ of size s:
    $C(S, 1) \leftarrow +\infty$
    for all $i \in S,\ i \neq 1$:
      for all $j \in S,\ j \neq i$:
        $C(S, i) \leftarrow \min\{C(S, i), C(S - \{i\}, j) + d_{ji}\}$
return $\min_i \{C(\{1, \ldots, n\}, i) + d_{i,1}\}$

Question

What is the space requirement of this algorithm?

- ○ $\Theta(n)$
- ○ $\Theta(n^2)$
- ○ $\Theta(2^n)$
- ◉ $\Theta(n \cdot 2^n)$
- ○ $\Theta(n^2 \cdot 2^n)$

✓ **Correct**
That's right! The dynamic programming table has $n \cdot 2^n$ cells.

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 $\{0, \ldots, n-1\}$:

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

Example

k	bin(k)	$\{i : i$-th bit of k 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	$\{0,1,2\}$

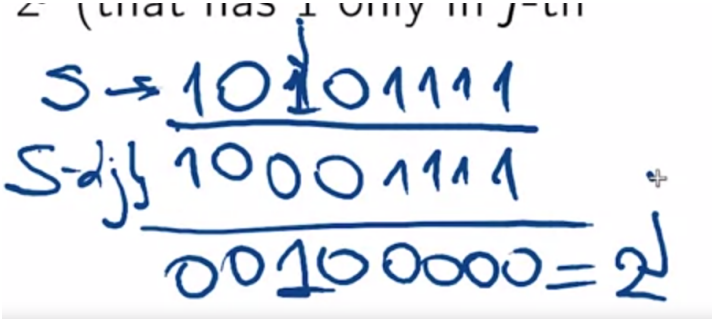
- If k corresponds to S, how to find out the integer corresponding to $S - \{j\}$ (for $j \in S$)?

- 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)`

This computes 2^j



k is 1010111 in the example shown

Outline

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

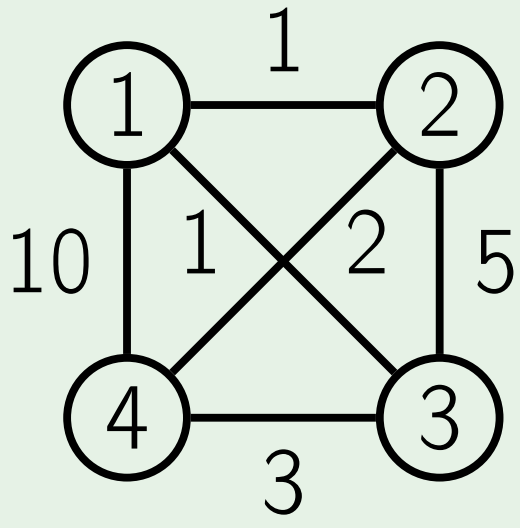
Backtracking is usually used for solving decision problems while branch and bound is used to solve optimization problems

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

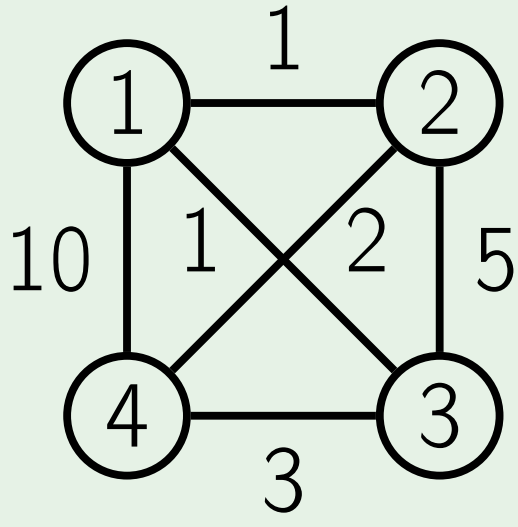
- The branch-and-bound technique can be viewed as a generalization of backtracking for optimization problems
- We grow a tree of partial solutions
- At each node of the recursion tree we check whether the current partial solution can be extended to a solution which is better than the best solution found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

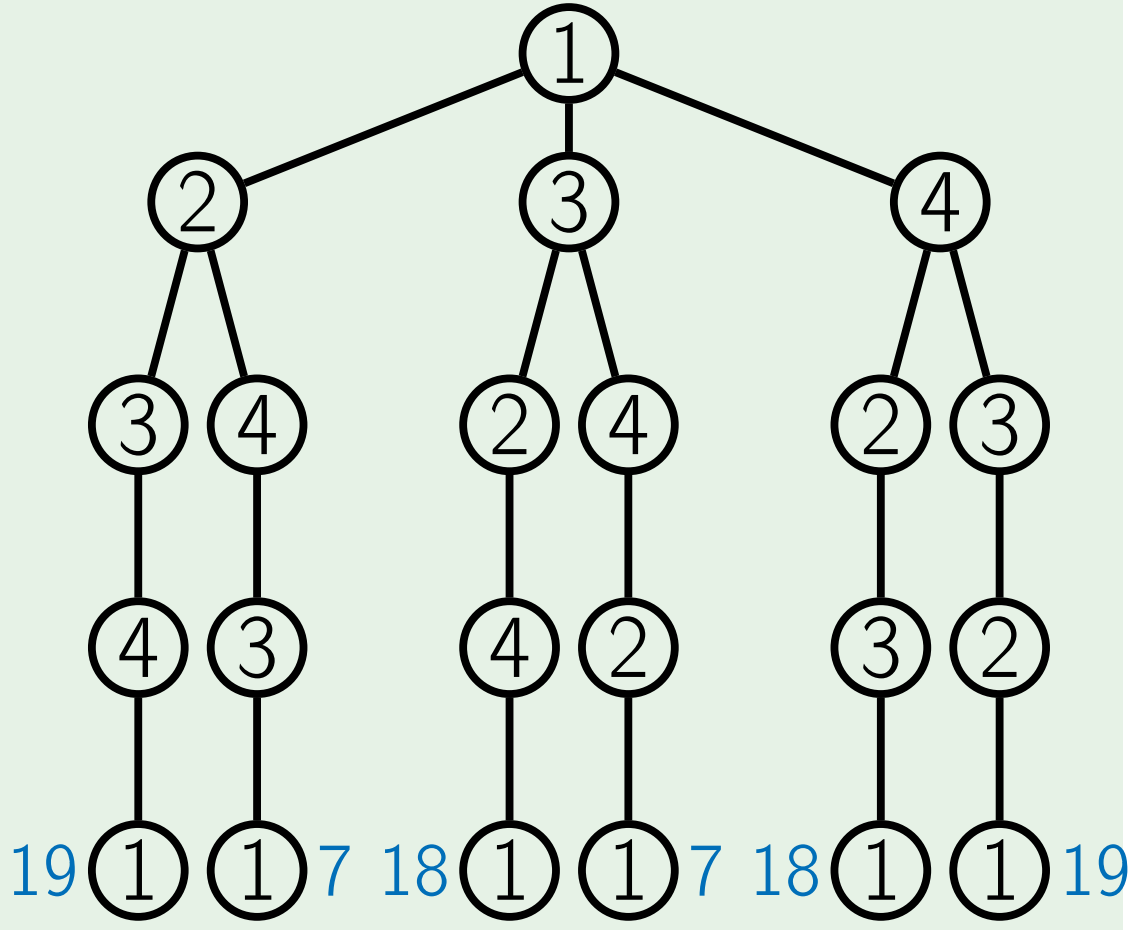
- The branch-and-bound technique can be viewed as a generalization of backtracking for optimization problems
- We grow a tree of partial solutions
- At each node of the recursion tree we check whether the current partial solution can be extended to a solution which is better than the best solution found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

Example: brute force search

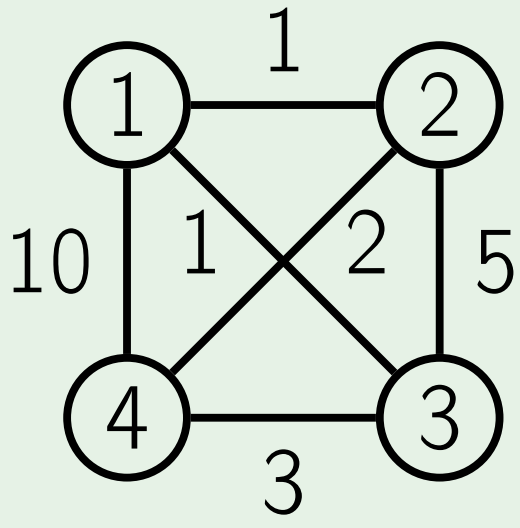


Example: brute force search

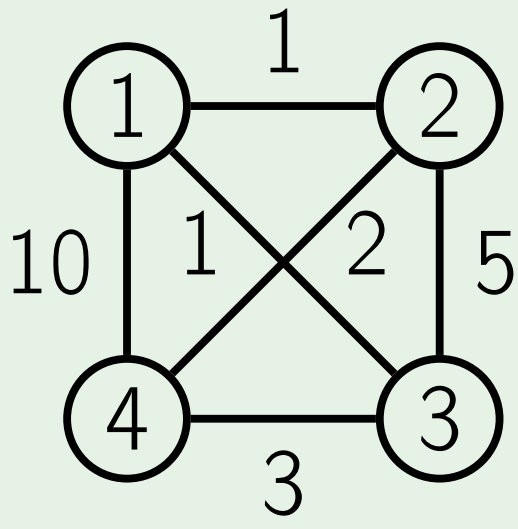


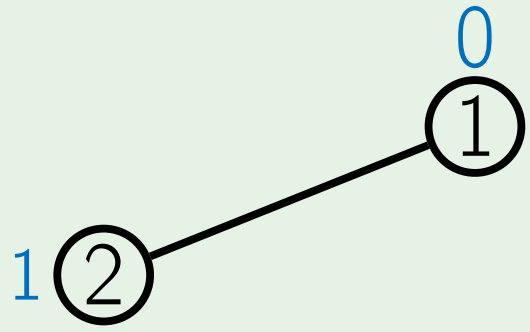


Example: pruned search

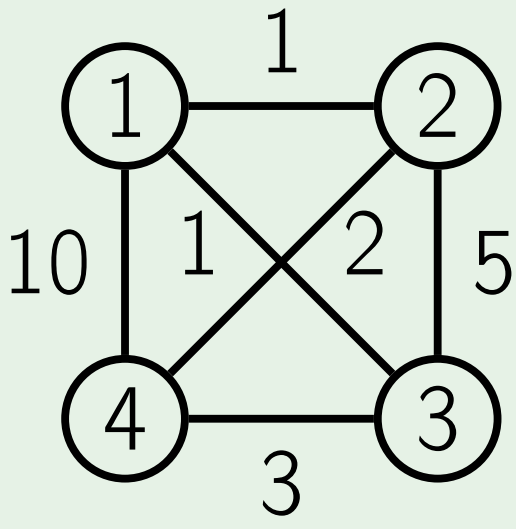


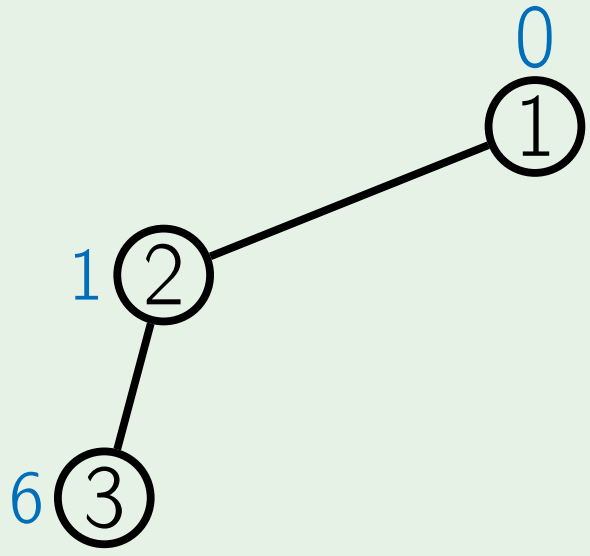
Example: pruned search



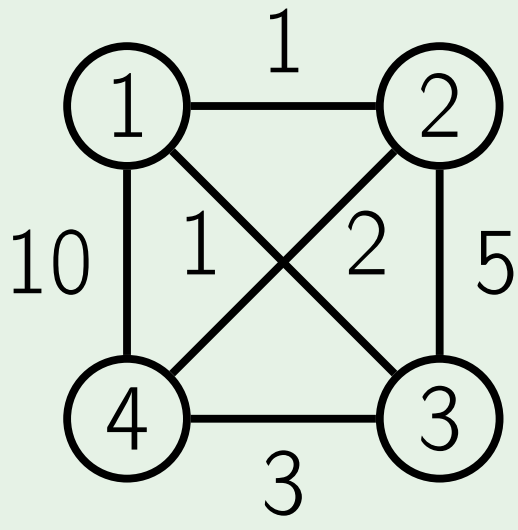


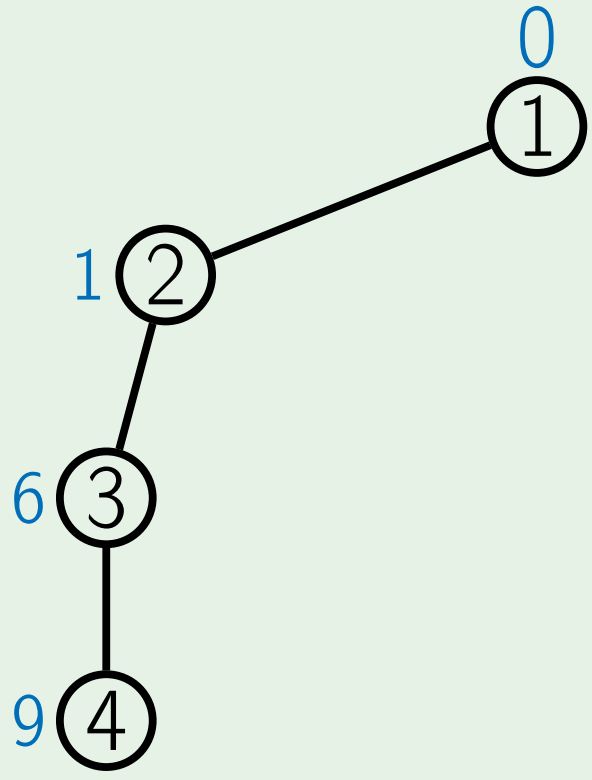
Example: pruned search



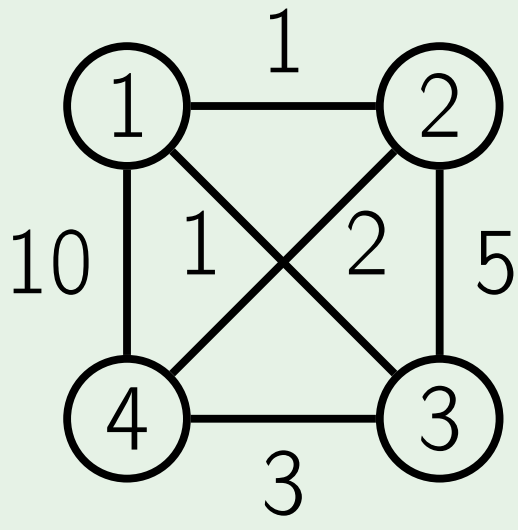


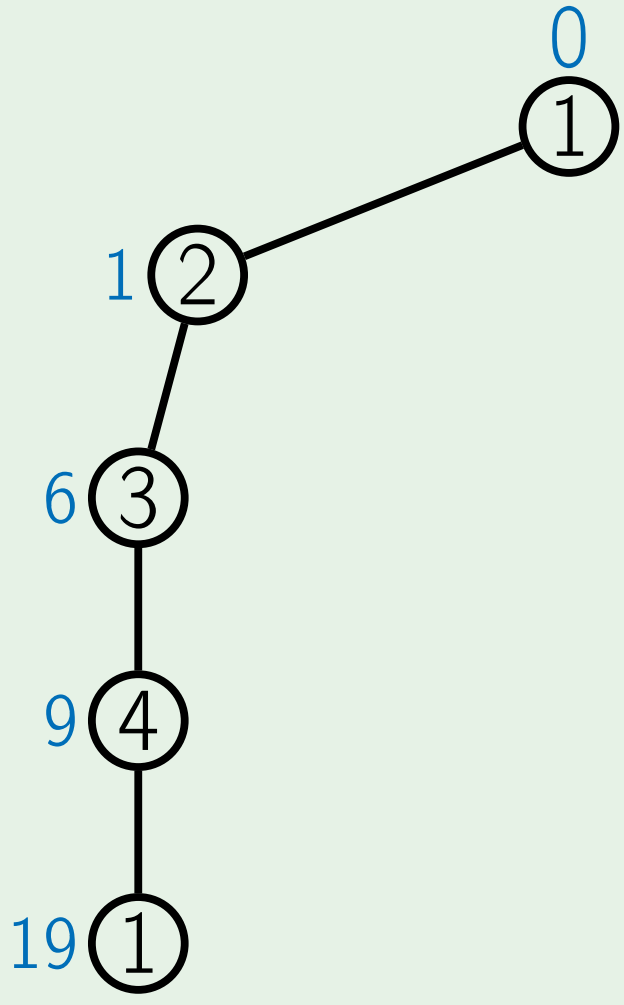
Example: pruned search





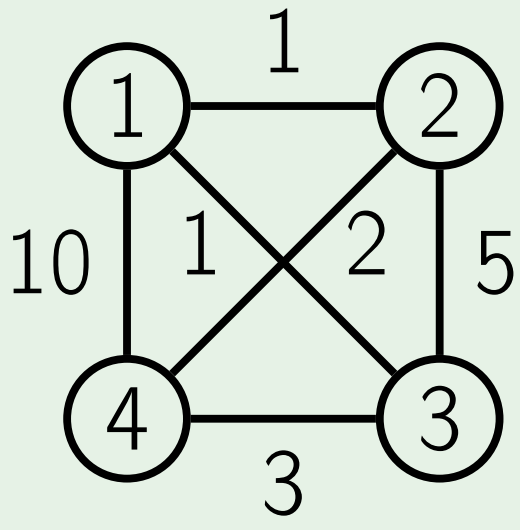
Example: pruned search

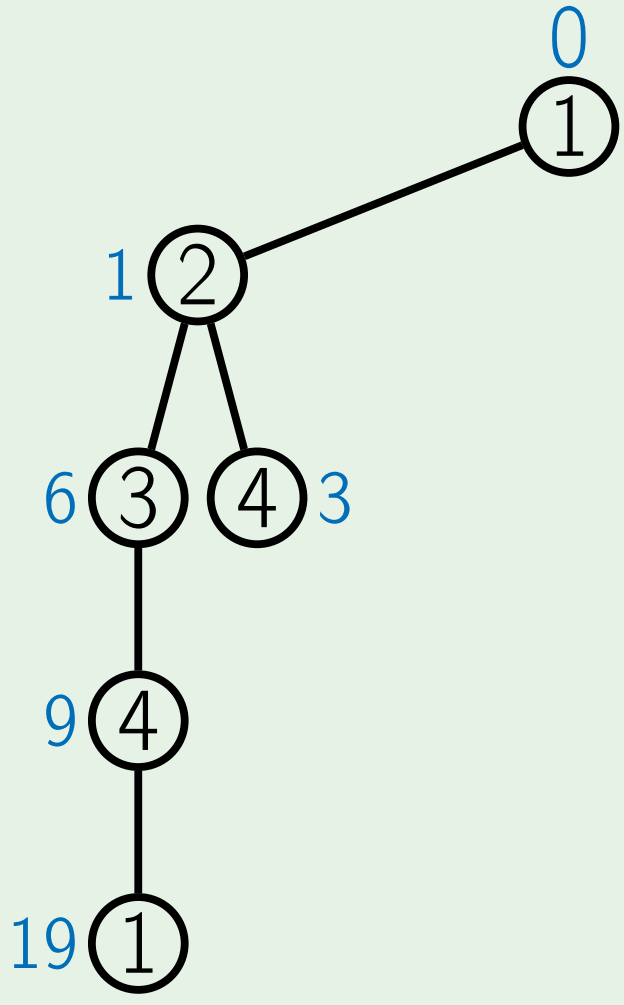




best so far: 19

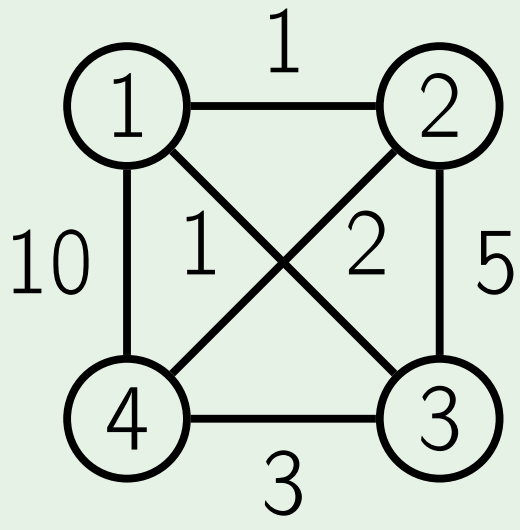
Example: pruned search

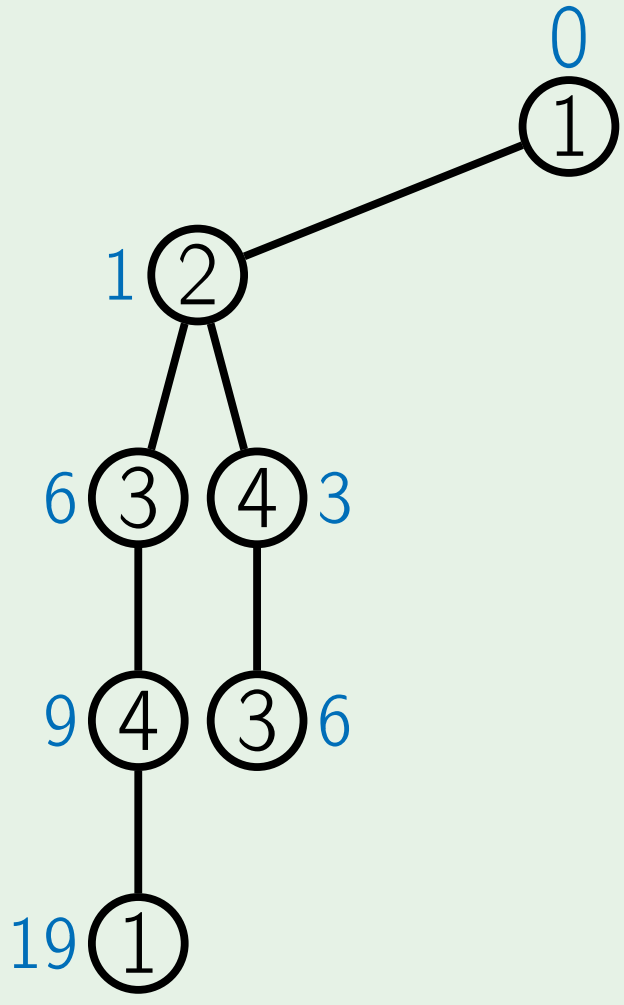




best so far: 19

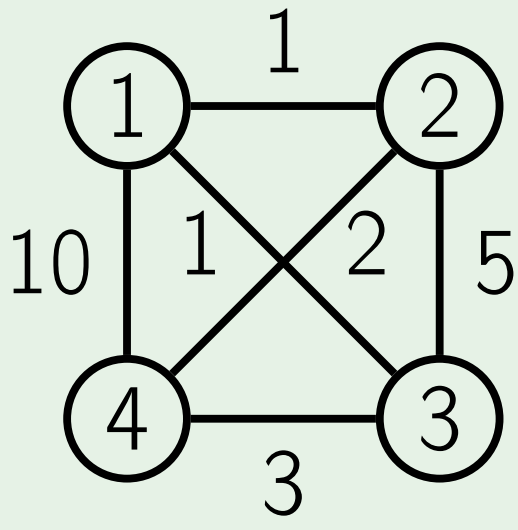
Example: pruned search

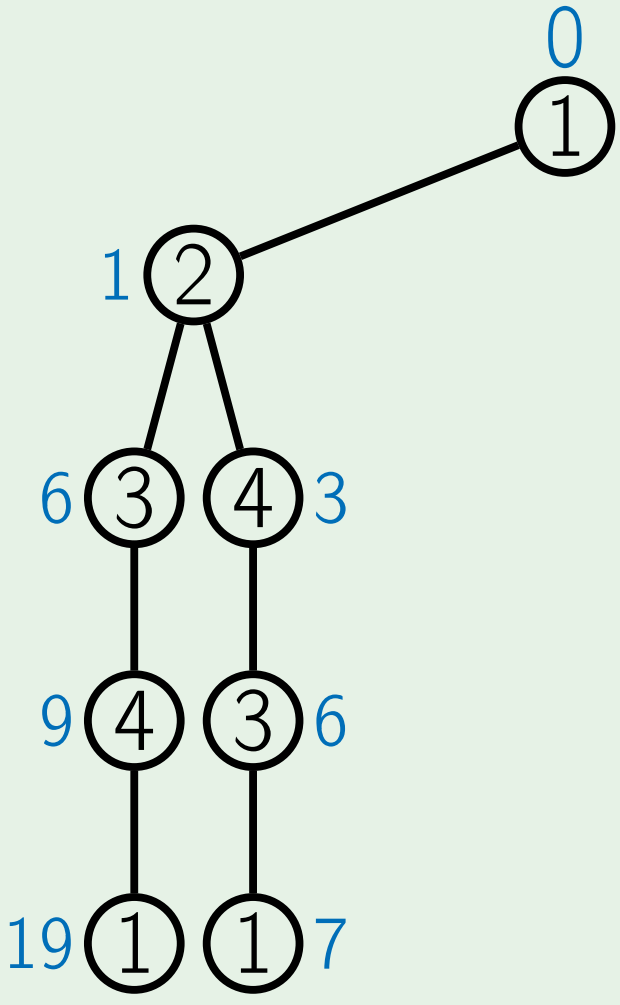




best so far: 19

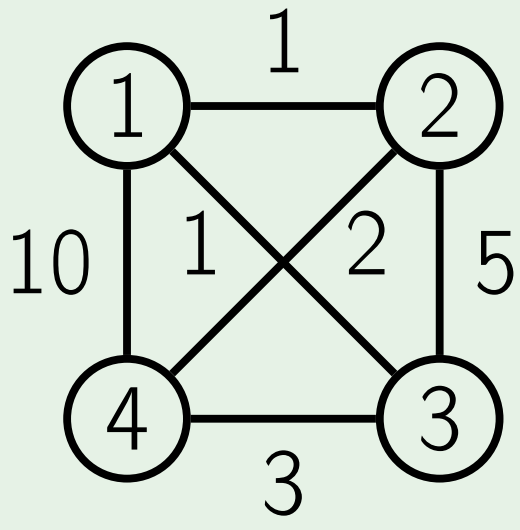
Example: pruned search

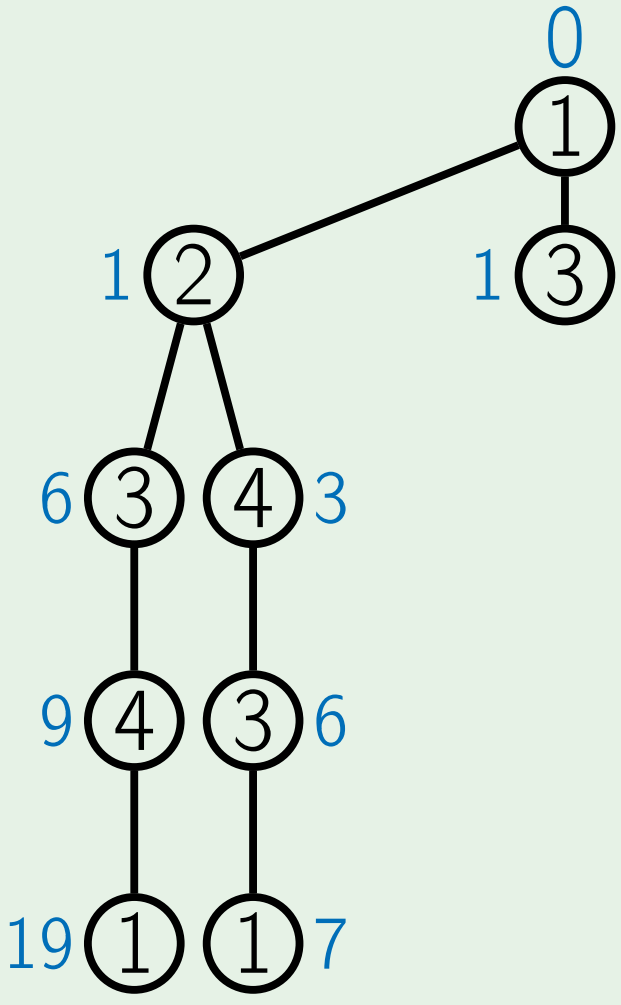




best so far: 7

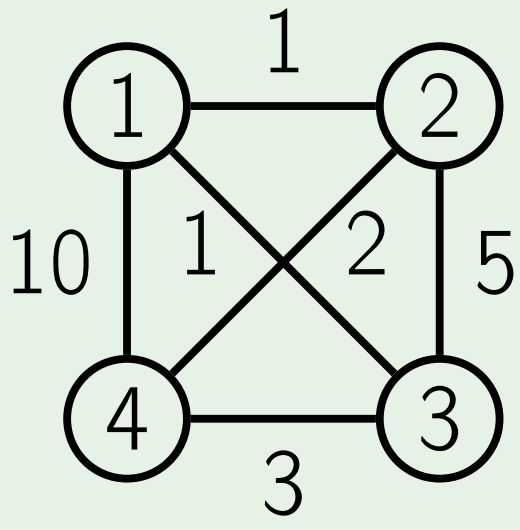
Example: pruned search

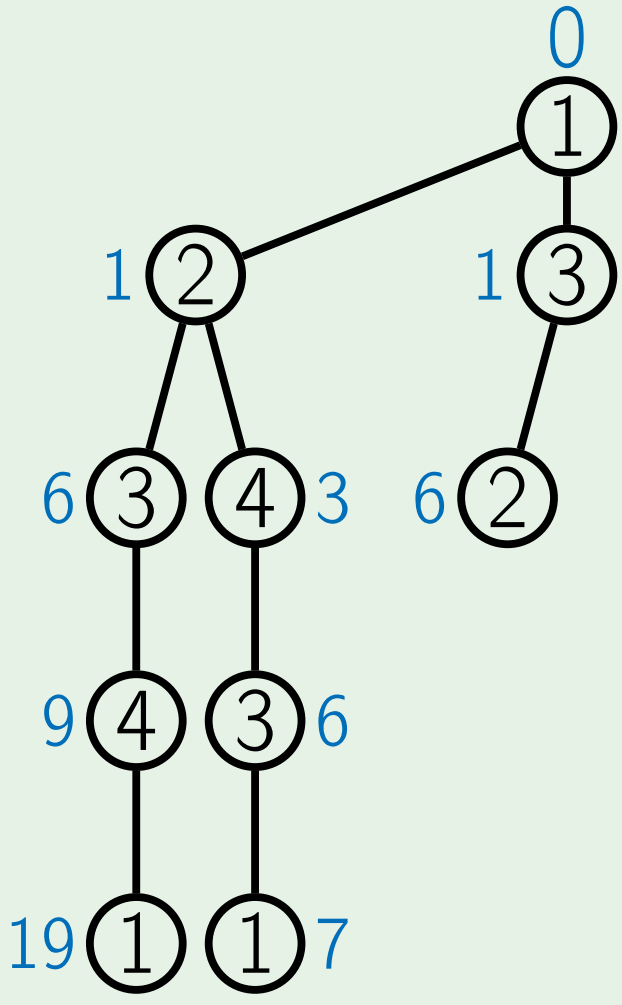




best so far: 7

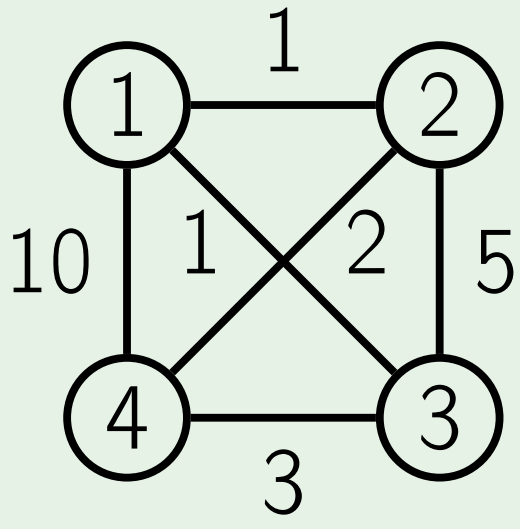
Example: pruned search

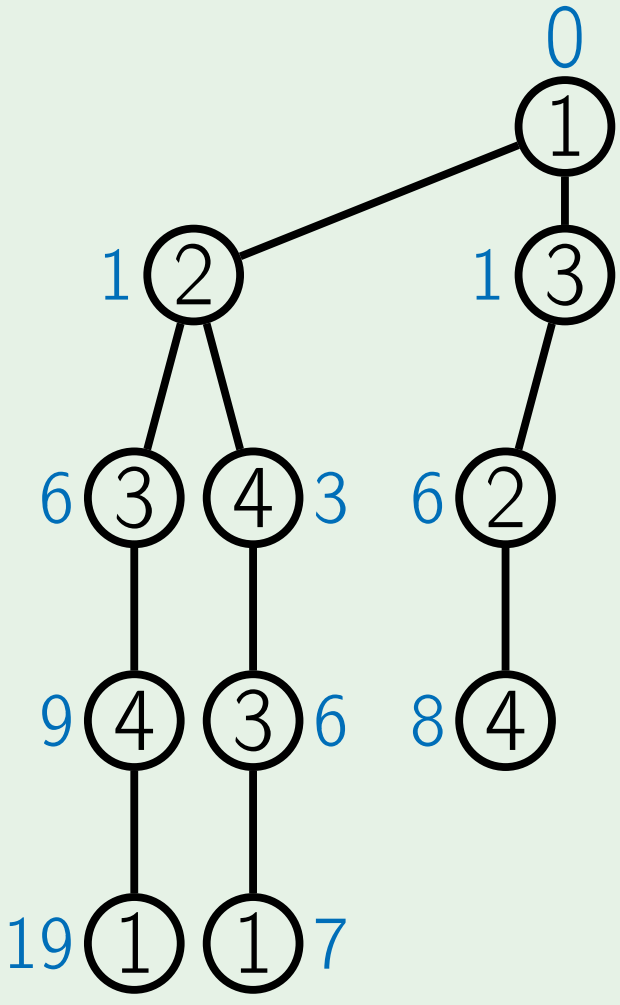




best so far: 7

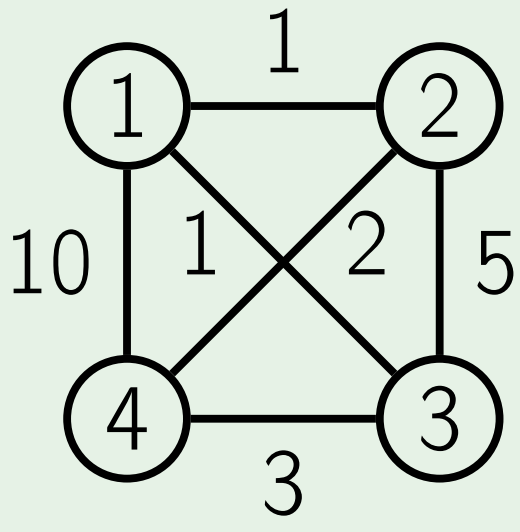
Example: pruned search

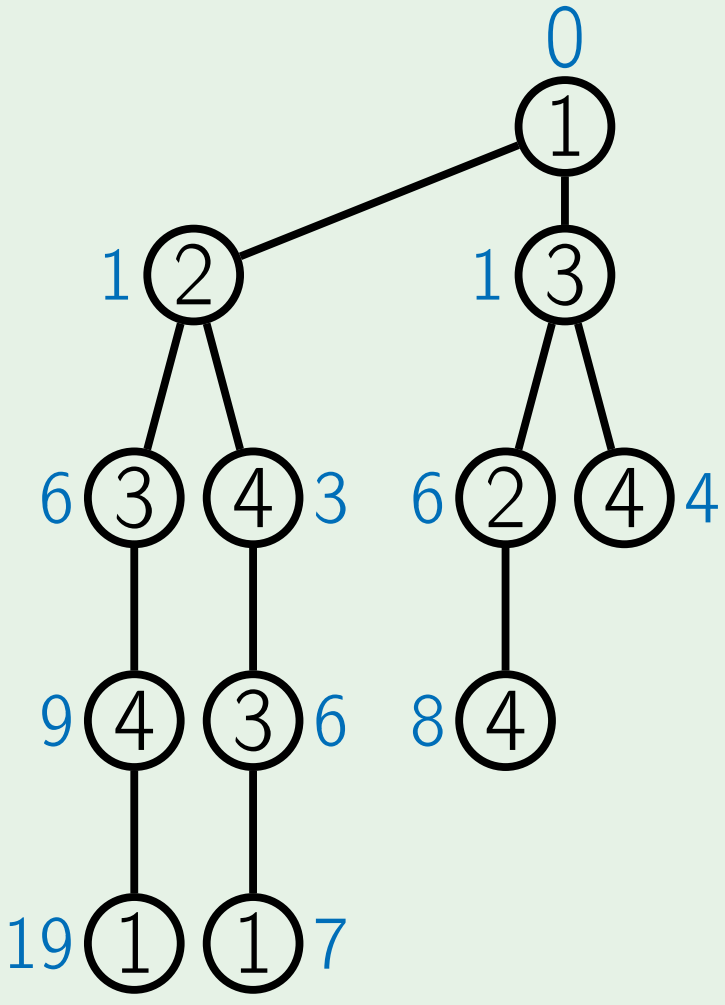




best so far: 7

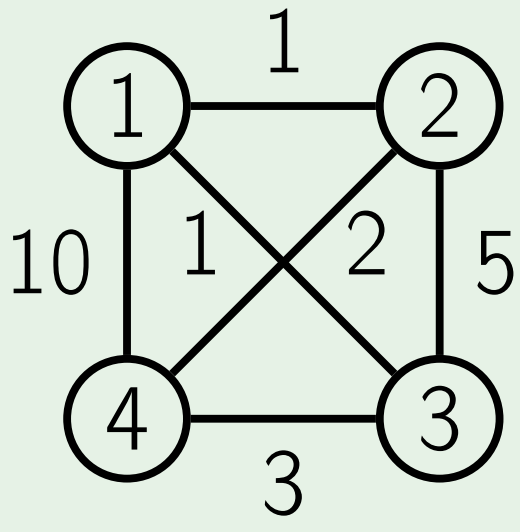
Example: pruned search

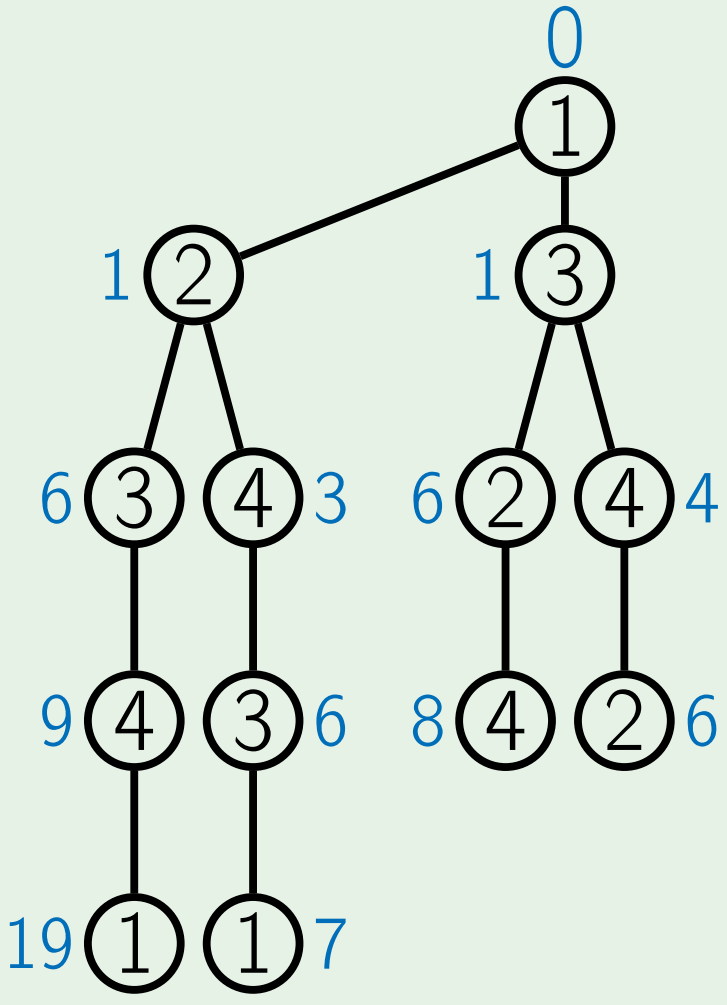




best so far: 7

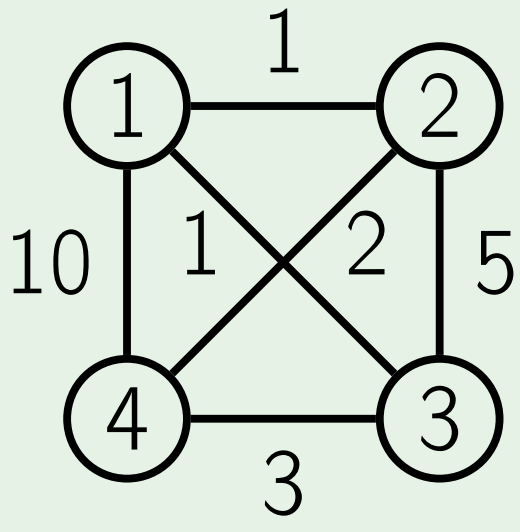
Example: pruned search

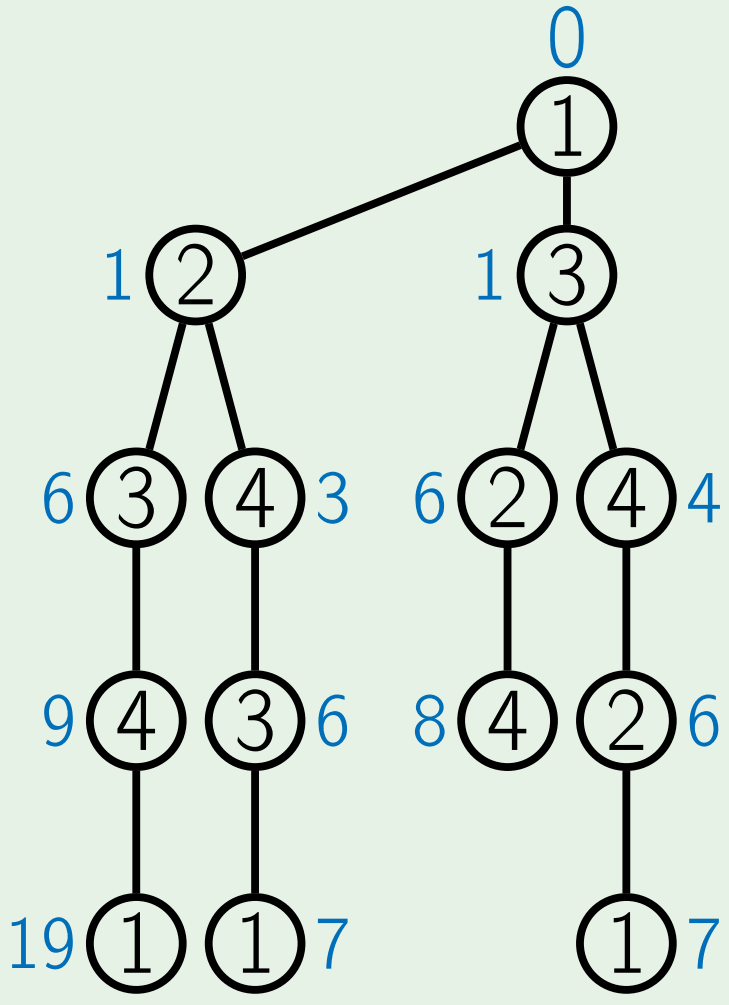




best so far: 7

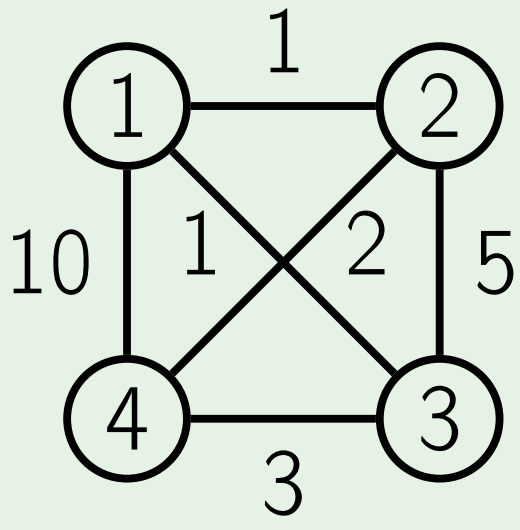
Example: pruned search

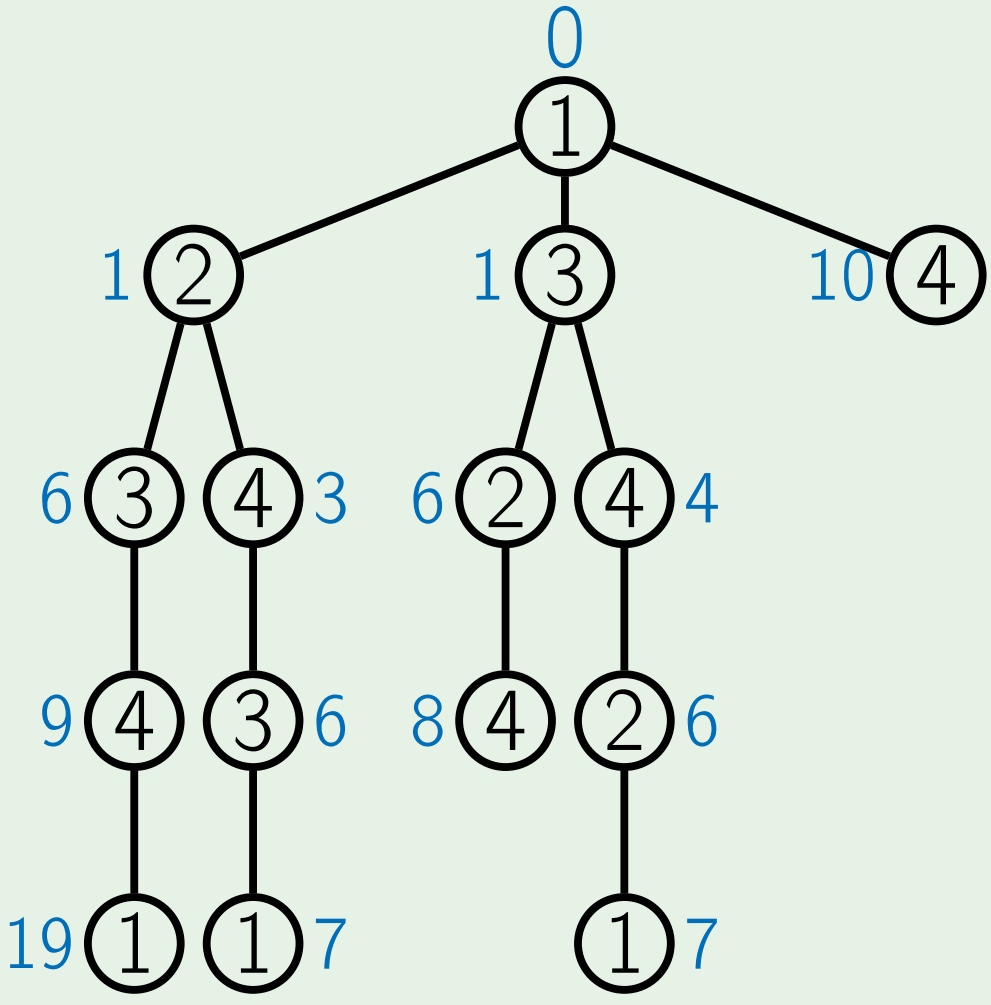




best so far: 7

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