Algorithms and Complexity

Lecture 6

Introduction to Exponential Time:

Clever Enumeration

Slides by Tom van der Zanden, partly based on those by Jesper Nederlof

Continuation of course

- 4 weeks remaining
  - These lectures
  - Lecture notes
  - Set of exercises for practice
  - Homework

  - Due to education commitments in Maastricht, the lectures on October 30 and November 6 will **only** be offered online. Sorry!

Exact Exponential & FPT Algorithms

- What to do in case of NP-completeness?
  1. Heuristic approaches: no guarantees
  2. Approximation algorithms: guarantee on time, loss of solution quality quantified
  3. Solvers (ILP,…): guarantee on solution quality, no meaningful guarantee on time
  4. Exact Exponential algorithms: guarantee on both quality and time
  5. Fixed-Parameter Tractable algorithms (FPT): more fine-grained analysis of running time

CNF-SAT

- Recall the definition of CNF-formula
  - **literal l_i:** expression of the type v_i or $\neg v_i$.
  - **clause C_i:** disjunction of literals (e.g. $v_i \vee \neg v_j$)
  - **CNF-formula:** conjunction of clauses (e.g. $C_i \wedge C_j$)
  - **k-CNF-formula:** all clauses of size at most k
- Examples:
  - $v_1 \wedge (\neg v_2 \vee v_3 \vee v_4)$
  - $(v_2 \vee v_1) \wedge (v_2 \vee \neg v_3) \wedge (\neg v_3 \vee v_5)$
- (k-)CNF-SAT: is the given (k-)CNF-formula satisfiable?
  - n denotes #vars, m denotes #clauses

CNF-SAT: *brute force*

- Easy $O(2^n nm)$ time algorithm:

Algorithm $\texttt{CNF}-\texttt{sat}(\varphi)$ — φ is a CNF-formula on variables $v_1, \ldots, v_n$.
Output: Whether there exists an assignment $(v_1, \ldots, v_n) \in \{true, false\}^n$ such that φ is true.

```
1: for all (v_1, ..., v_n) ∈ {true, false}^n do
2:     if φ is satisfied by v then return true
3: return false
```

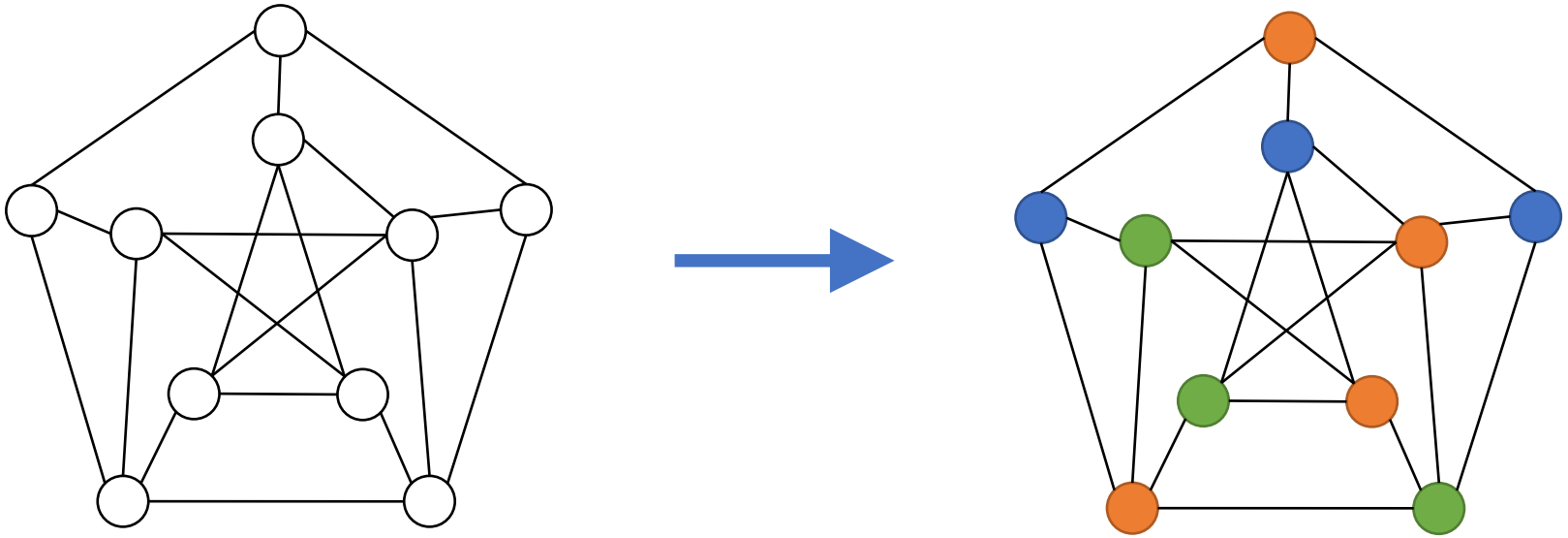
Check is easily done in time linear in formula size.

- Consider a 1GHz computer: can solve $n = 36$ in 1 hour

- Footnote: nothing substantially better is known!
  - (the `Strong Exponential Time Hypothesis' even states $O(1.99^n(n+m)^c)$ is not possible).

3-coloring

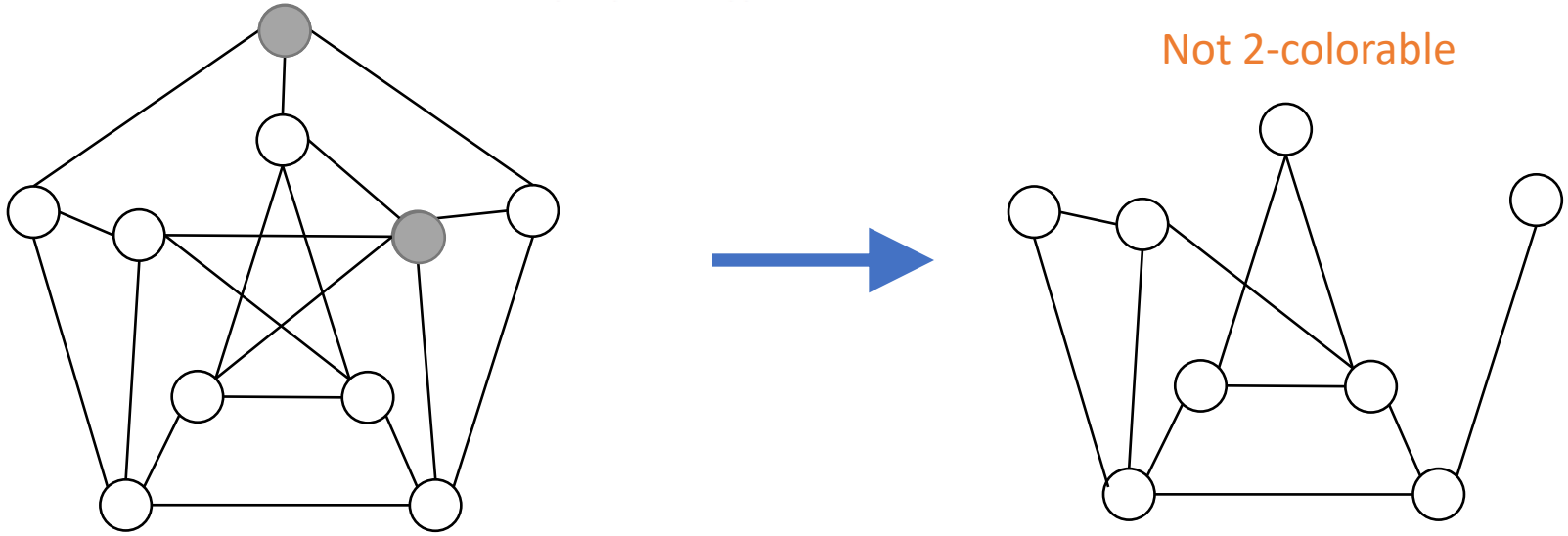
- k-coloring of G=(V,E): map $c: V \to \{1, \dots, k\}$ s.t. $c(u) \neq c(v)$, for every $(u, v) \in E$.
- Brute force algorithm? → 3^n



3-coloring smarter brute force in $O(2^n(n+m))$

Algorithm `3colv2`(G).
Output: Whether a 3-coloring exists.

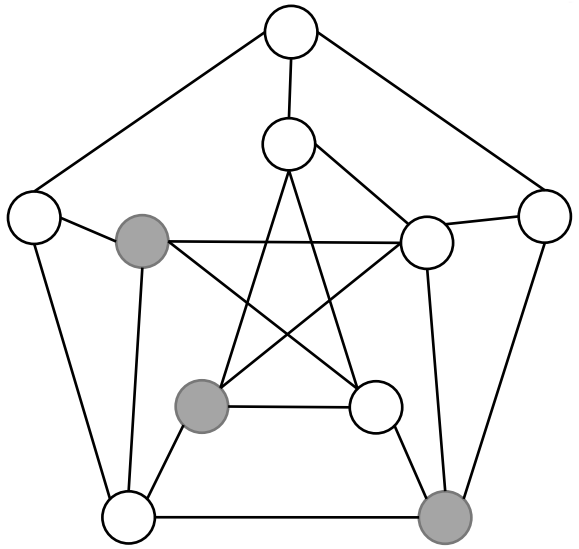
1: **for all** $X \subseteq V$ **do**
2: **if** X is an independent set of G **then**
For $G = (V, E)$ and $X \subseteq V$, $G[X]$ denotes the graph $(X, \{e \in E : e \subseteq X\})$.
3: **if** `2colorable`$(G[V \setminus X])$ **then return true**
4: **return false**



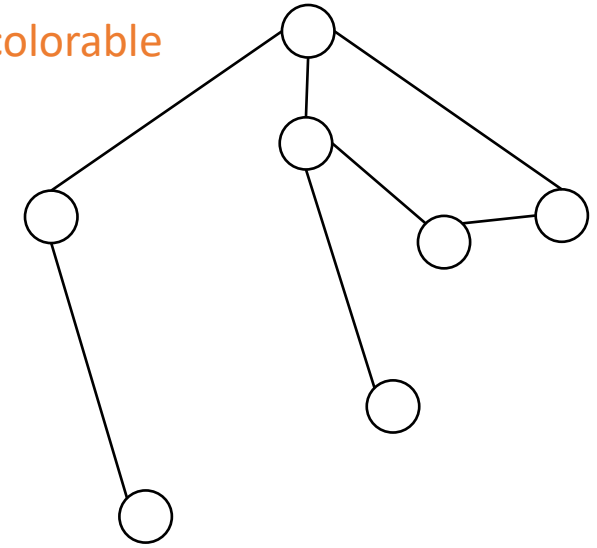
3-coloring smarter brute force in $O(2^n(n+m))$

Algorithm `3colv2`(G).
Output: Whether a 3-coloring exists.

1: **for all** $X \subseteq V$ **do**
2: **if** X is an independent set of G **then**
For $G = (V, E)$ and $X \subseteq V$, $G[X]$ denotes the graph $(X, \{e \in E : e \subseteq X\})$.
3: **if** `2colorable`$(G[V \setminus X])$ **then return true**
4: **return false**



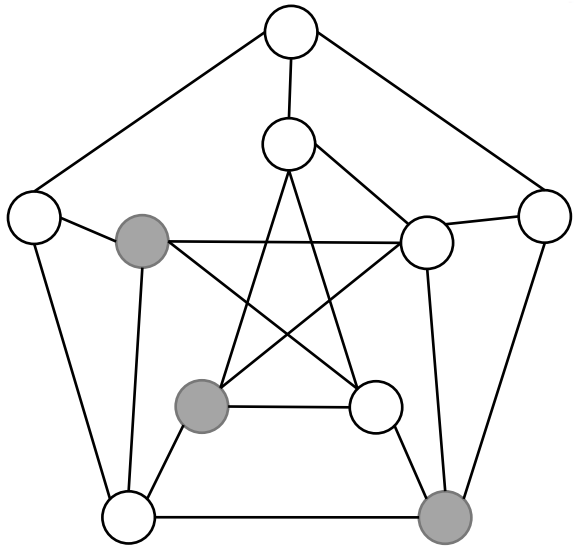
2-colorable

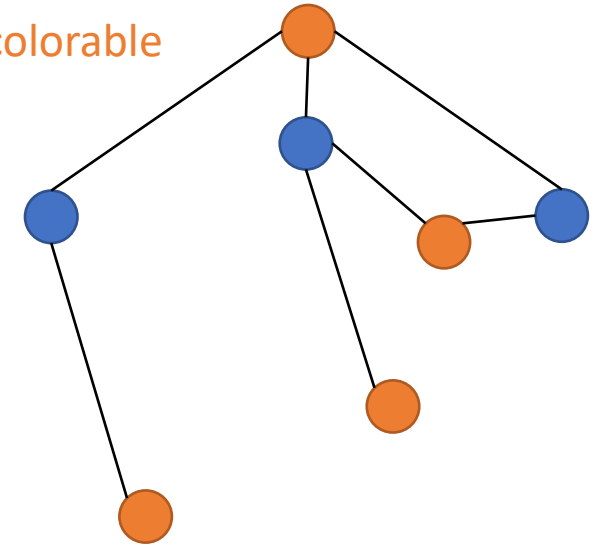


3-coloring smarter brute force in $O(2^n(n+m))$

Algorithm `3colv2`(G).
Output: Whether a 3-coloring exists.

1: **for all** $X \subseteq V$ **do**
2: **if** X is an independent set of G **then**
For $G = (V, E)$ and $X \subseteq V$, $G[X]$ denotes the graph $(X, \{e \in E : e \subseteq X\})$.
3: **if** `2colorable`$(G[V \setminus X])$ **then return true**
4: **return false**

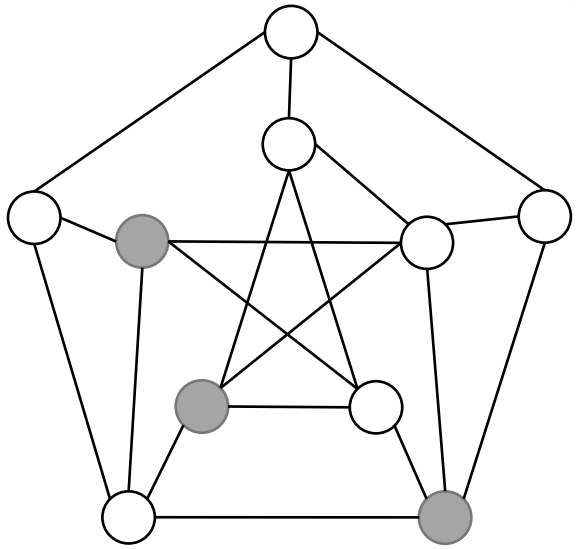




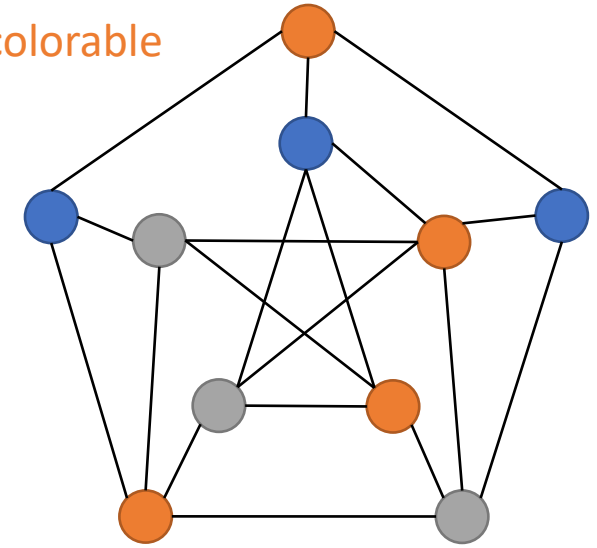
3-coloring smarter brute force in $O(2^n(n+m))$

Algorithm `3colv2`(G).
Output: Whether a 3-coloring exists.

1: **for all** $X \subseteq V$ **do**
2: **if** X is an independent set of G **then**
For $G = (V, E)$ and $X \subseteq V$, $G[X]$ denotes the graph $(X, \{e \in E : e \subseteq X\})$.
3: **if** `2colorable`$(G[V \setminus X])$ **then return true**
4: **return false**



2-colorable



3-SAT: *branching*

Algorithm $\texttt{CNF}-\texttt{sat}(\varphi, v')$

φ is a 3-CNF-formula on variables $v_1, \ldots, v_n$;

$(v_1, \ldots, v_n) \in \{true, false, unassigned\}^n$ is a partial assignment.

Output: Whether there exists an assignment $(v_1, \ldots, v_n) \in \{true, false\}^n$ such that φ is true.

```
1:  if φ does not contain any clause with 3 unassigned variables then
2:    if there is a satisfying assignment for the unassigned variables then
3:      return true                                  2-CNF is solvable in O(n + m)
4:    else
5:      return false
6:  else
7:    Let c be a clause with 3 unassigned variables v_i, v_j, v_k
8:    for all Extensions v'' of v' with assignments to v_i, v_j, v_k that satisfy c do
9:      if CNF − sat(φ, v'') then return true
10:   return false
```

3-SAT: *branching*

Algorithm $\texttt{CNF} - \texttt{sat}(\varphi, v')$

φ is a 3-CNF-formula on variables $v_1, \ldots, v_n$;

$(v_1, \ldots, v_n) \in \{true, false, unassigned\}^n$ is a partial assignment.

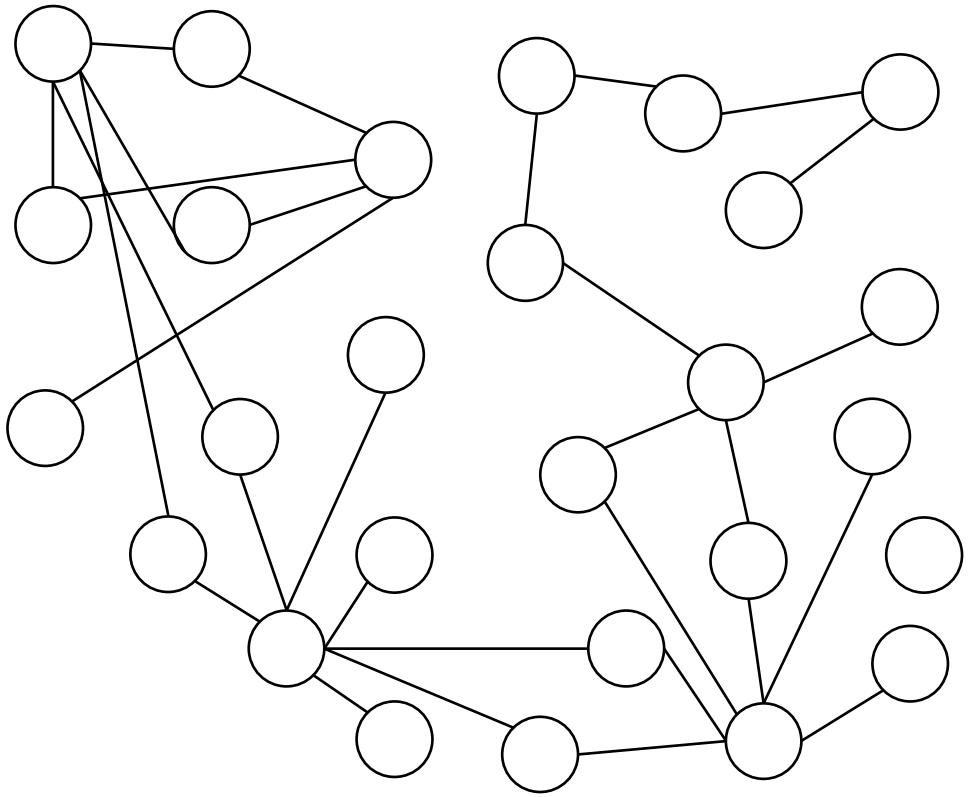
Output: Whether there exists an assignment $(v_1, \ldots, v_n) \in \{true, false\}^n$ such that φ is true.

```
1:  if φ does not contain any clause with 3 unassigned variables then
2:    if there is a satisfying assignment for the unassigned variables then
3:      return true                                   2-CNF is solvable in O(n + m)
4:    else
5:      return false
6:  else
7:    Let c be a clause with 3 unassigned variables v_i, v_j, v_k
8:    for all Extensions v'' of v' with assignments to v_i, v_j, v_k that satisfy c do
9:      if CNF − sat(φ, v'') then return true
10:   return false
```

- 7 branches per 3 variables: $7^{n/3} = \left(\sqrt[3]{7}\right)^n$
- $O(1.913^n)$ $n = 39$ on 1GHz computer

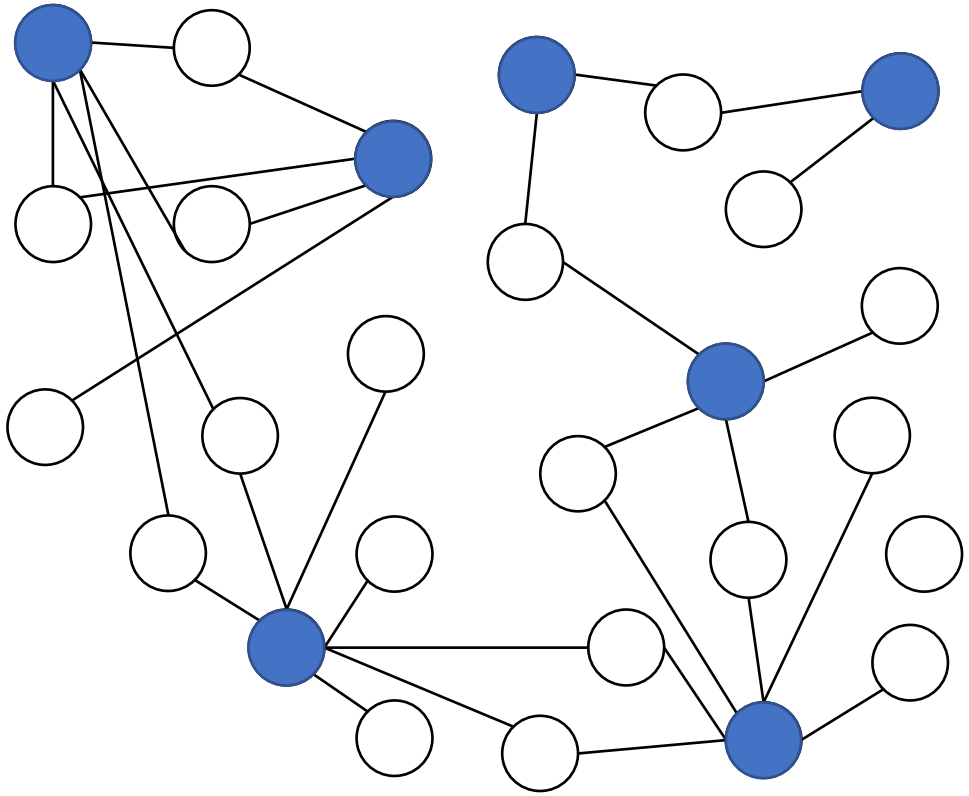
Vertex Cover

- A vertex cover of G=(V,E) is a subset $X \subseteq V$ such that for every edge $(u, v) \in E$, $u \in X$ or $v \in \mathrm{X}$.
- Can you find a vertex cover of size 7 in the graph below?



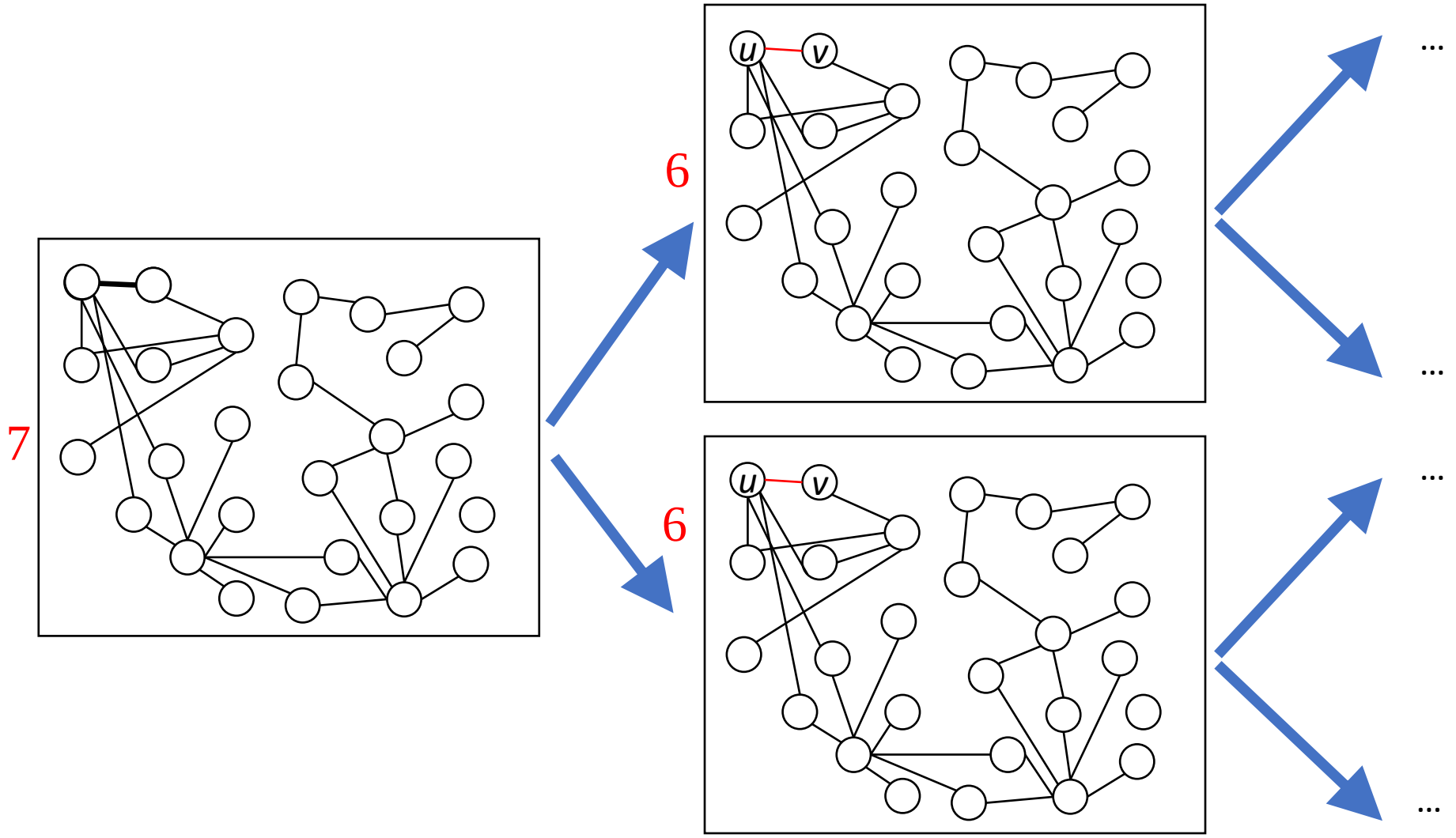
Vertex Cover

- A vertex cover of G=(V,E) is a subset $X \subseteq V$ such that for every edge $(u, v) \in E$, $u \in X$ or $v \in \mathrm{X}$.
- Can you find a vertex cover of size 7 in the graph below?



First algorithm for vertex cover

Algorithm $\texttt{vc}(G = (V, E), k)$
Output: Whether G has a vertex cover of size at most k.
1: **if** $E = \emptyset$ **and** $k \geq 0$ **then return true**
2: **if** $k \leq 0$ **then return false**
3: Let $(u, v) \in E$
4: **return** $\texttt{vc}(G[V \setminus u], k-1) \vee \texttt{vc}(G[V \setminus v], k-1)$

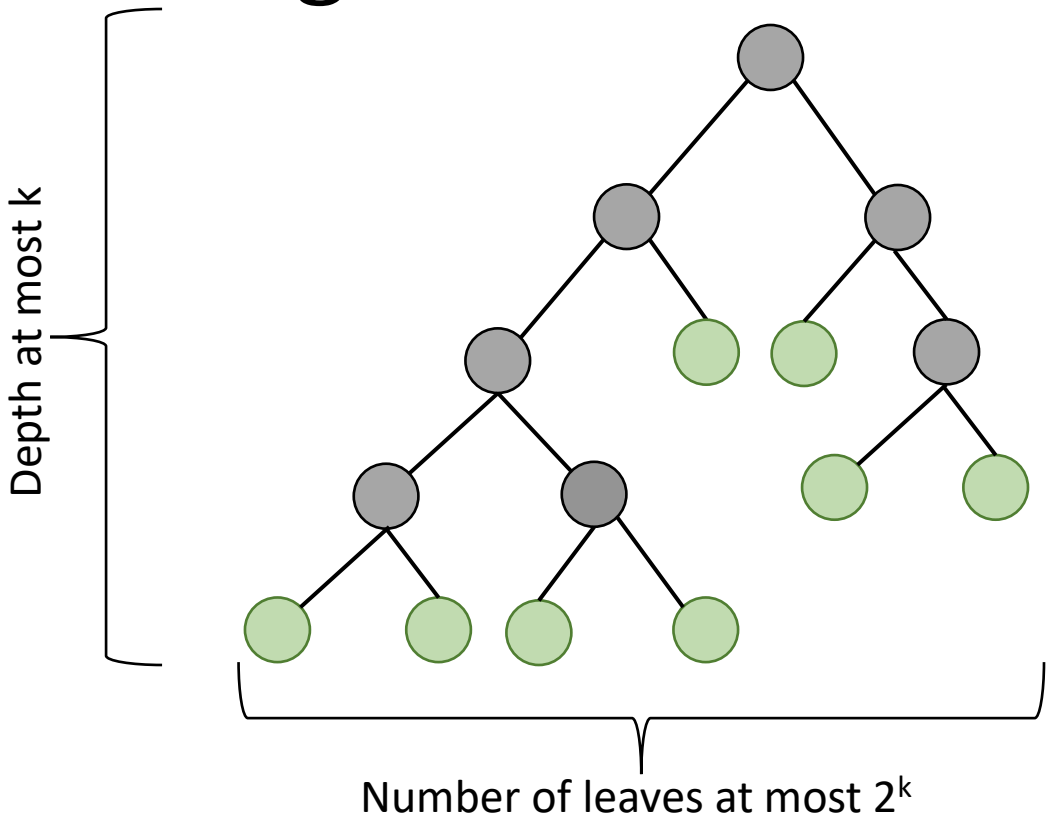


Algorithm $\texttt{vc}(G = (V, E), k)$
Output: Whether G has a vertex cover of size at most k.

1: **if** $E = \emptyset$ **and** $k \geq 0$ **then return true**
2: **if** $k \leq 0$ **then return false**
3: Let $(u, v) \in E$
4: **return** $\texttt{vc}(G[V \setminus u], k-1) \vee \texttt{vc}(G[V \setminus v], k-1)$

Time Bound and Branching tree

- We spend at most O(n+m) time per recursive call
- Recursion depth is at most k
- Thus, the number of recursive calls is at most k*#non-rec. calls.
- Let $T(k)$ be #non-rec calls
- $T(0) = 1$
- $T(k) \leq T(k-1) + T(k-1)$
- $T(k) \leq 2^k$
- Running time: $O((n+m)k2^k)$



Algorithm $\texttt{vc}(G = (V, E), k)$
Output: Whether G has a vertex cover of size at most k.
1: **if** $E = \emptyset$ **and** $k \geq 0$ **then return true**
2: **if** $k \leq 0$ **then return false**
3: Let $(u, v) \in E$
4: **return** $\texttt{vc}(G[V \setminus u], k-1) \vee \texttt{vc}(G[V \setminus v], k-1)$

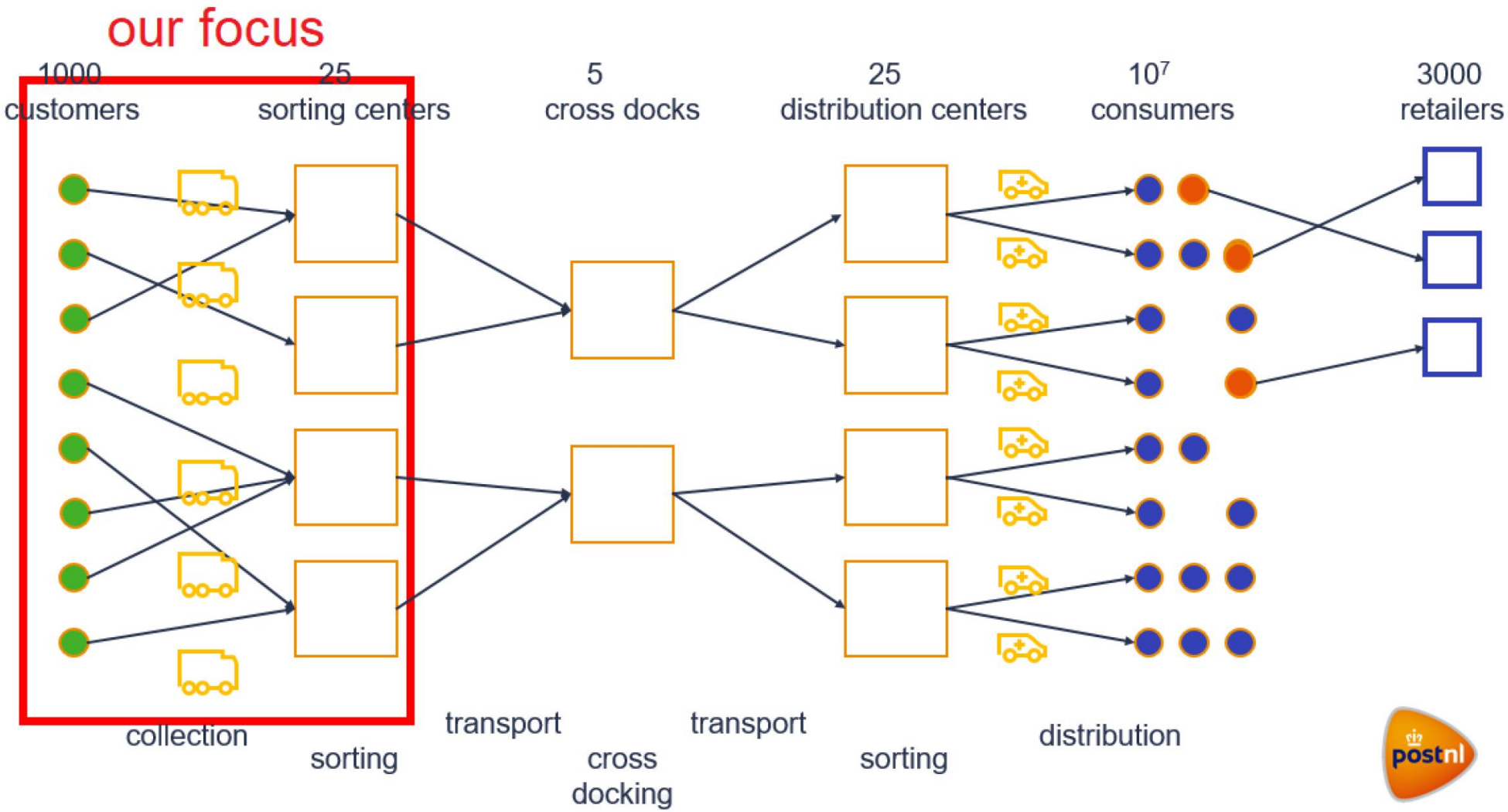
Parameterized Complexity

- As long as k is constant, $O\left((n+m)k2^k\right)$ is linear time.
- In our example n=27, k=7,
  - $2^{27} = 134217728$, $2^7 = 128$
- In general, we can often isolate the exponential dependency of the RT in a parameter that is often small.

A **parameterized problem** is a language $L \subseteq \{0,1\}^* \times \mathbb{N}$. For an instance $(x, k) \in \{0,1\}^*$, x is called the *input*, and k is called the *parameter*.
A parameterized problem is called **Fixed Parameter Tractable (FPT)** if there exists an algorithm for it that runs in time $f(k)|x|^c$, for some constant c and function $f(\cdot)$.

- So, vertex cover parameterized by k is FPT

Real-world example: PostNL



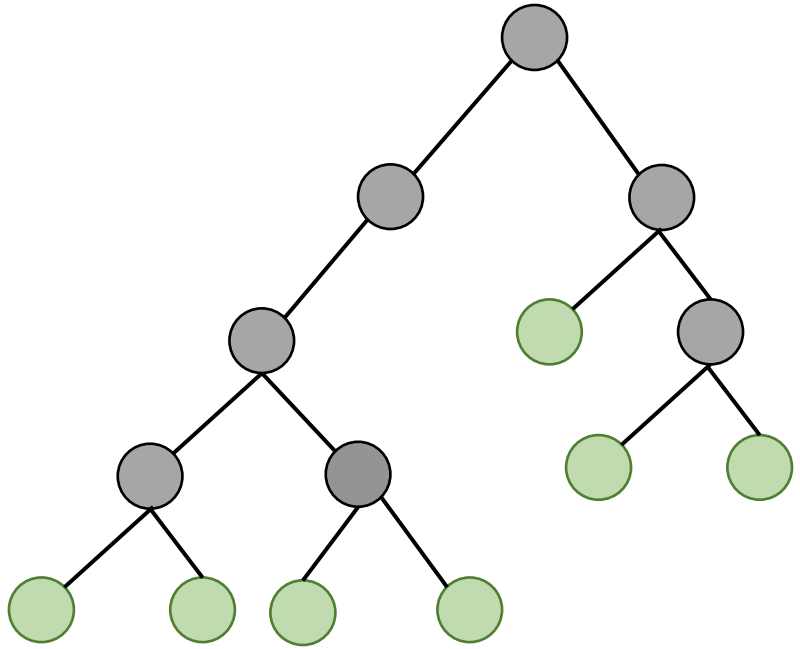
Second algorithm for vertex cover

Algorithm $\texttt{vc2}(G = (V, E), k)$
Output: Whether G has a vertex cover of size at most k

1: **if** $E = \emptyset$ and $k \geq 0$ **then return true**
2: **if** $k \leq 0$ **then return false**
3: **if** $\exists u \in V : \deg(u) \geq 2$ **then**
4: **return** $\texttt{vc2}(G[V \setminus u], k-1) \vee \texttt{vc2}(G[V \setminus N[u]], k - \deg(u))$
5: **else**
6: **if** $k \geq |E|$ **then return true else return false**

Time Bound and Branching tree

- We spend at most O(n+m) time per recursive call
- Recursion depth is at most k
- Thus, the number of recursive calls is at most k*#non-rec. calls.
- Let $T(k)$ be #non-rec calls
- $T(0) = 1$
- $T(k) \leq \max_{d \geq 2} T(k-1) + T(k-d)$
- $T(k) \leq 1.62^k$
- Running time: $O((n+m)k1.62^k)$



Algorithm vc2$(G = (V, E), k)$
Output: Whether G has a vertex cover of size at most k

1: **if** $E = \emptyset$ and $k \geq 0$ **then return true**
2: **if** $k \leq 0$ **then return false**
3: **if** $\exists u \in V : \deg(u) \geq 2$ **then**
4: **return** vc2$(G[V \setminus u], k-1) \vee$ vc2$(G[V \setminus N[u]], k - \deg(u))$
5: **else**
6: **if** $k \geq |E|$ **then return true else return false**

Time Bound and Branching tree

- We spend at most O(n+m) time per recursive call
- Recursion depth is at most k
- Thus, the number of recursive calls is at most k*#non-rec. calls.
- Let $T(k)$ be #non-rec calls
- $T(0) = 1$
- $T(k) \leq \max_{d \geq 2} T(k-1) + T(k-d)$
- $T(k) \leq 1.62^k$
- Running time: $O((n+m)k1.62^k)$

- How to guess $T(k) \leq 1.62^k$?
- Since it's a linear recurrence, $T(k) = c^k$
- $T(k) \leq \max_{d \geq 2} c^{k-1} + c^{k-d}$
  $\leq c^{k-1} + c^{k-2} \leq c^k$,
- Dividing both sides by c^k gives
  $c^{-1} + c^{-2} \leq 1$
- So for any c satisfying this, $T(k) \leq c^k$.
- Use educated guess or a numerical method for finding min c (there is no easy formula).

(want)

Algorithm vc2$(G = (V, E), k)$
Output: Whether G has a vertex cover of size at most k
1: **if** $E = \emptyset$ and $k \geq 0$ **then return true**
2: **if** $k \leq 0$ **then return false**
3: **if** $\exists u \in V : \deg(u) \geq 2$ **then**
4: **return** vc2$(G[V \setminus u], k-1) \vee$ vc2$(G[V \setminus N[u]], k - \deg(u))$
5: **else**
6: **if** $k \geq |E|$ **then return true else return false**

Table of FPT races

The results gradually keep improving. Please add your latest to the table.

NEWS: Edge clique cover does not have polynomial kernel (see 35). It was open for some time.
Very nice paper to read, so enjoy! — Saket

Problem	f(k)	vertices in kernel	Reference/Comments
Vertex Cover	1.2738^k	$2k$	1
Connected Vertex Cover	2^k	no $k^{O(1)}$	26, randomized algorithm
Multiway Cut	2^k	not known	21, 38: $O(k^{s+1})$-vertex kernel with s terminals
Directed Multiway Cut	$2^{O(k^3)}$	no $k^{O(1)}$	34
Almost-2-SAT (VC-PM)	2.3146^k	$O(k^6)$	37, 38, randomized kernel
Multicut	$2^{O(k^3)}$	no $k^{O(1)}$	22, 35
Pathwidth One Deletion Set	4.65^k	$O(k^2)$	28
Undirected Feedback Vertex Set	3.460^k	$4k^2$	36, deterministic algorithm
Undirected Feedback Vertex Set	2.7^k	$4k^2$	23, randomized algorithm
Subset Feedback Vertex Set	4^k	not known	39
Directed Feedback Vertex Set	$4^k k!$	not known	27
Odd Cycle Transversal	2.3146^k	$O(k^{4.5})$	37, 38, randomized kernel; kernel is for signed graph generalization
Edge Bipartization	2^k	$\tilde{O}(k^3)$	25, 38, randomized kernel; kernel is for signed graph generalization
Planar DS	$2^{11.98\sqrt{k}}$	$67k$	3
1-Sided Crossing Min	$2^{O(\sqrt{k}\log k)}$	$O(k^2)$	4
Max Leaf	3.72^k	$3.75k$	5
Directed Max Leaf	3.72^k	$O(k^2)$	6
Set Splitting	1.8213^k	k	7
Nonblocker	2.5154^k	$5k/3$	8
Edge Dominating Set	2.3147^k	$2k^2+2k$	10
k-Path	2.851^k	no $k^{O(1)}$	40, deterministic algorithm
k-Path	1.66^k	no $k^{O(1)}$	11b, randomized algorithm

FPT and W[1]-hardness

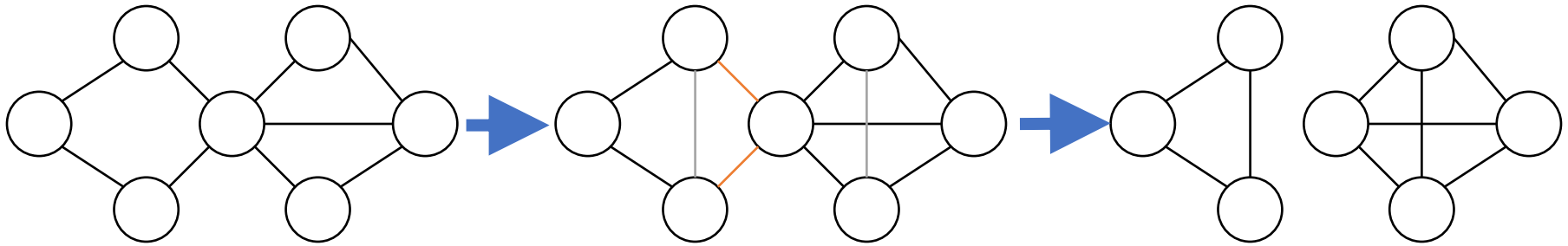
- Not all problems are known to have FPT algorithms
- E.g., for Independent Set parameterized by solution size k no FPT algorithm is known

- Independent Set is **W[1]-complete** with respect to k
- W[1]-hard is to FPT as NP-hard is to P

- Can show W[1]-hardness by parameterized reductions

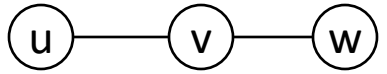
Cluster Editing

- Given graph G=(V,E), a cluster editing of size k is a set of k `modifications' to G, such that each connected component is a clique (cluster graph),
  - modification: addition of deletion of an edge.
- Models biological questions: partition species in families, where available data contains mistakes
- NP-complete.
- Example with k=4:



Cluster Editing via induced P_3's

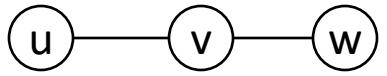
- G is a cluster graph if and only it does not contain an induced P_3



  - If cluster graph, clearly no induced P_3
  - If not cluster graph, there are non-adjacent u and x in same connected component. Let $u, v_1, \ldots, x$ be a shortest path from u to x.
    Then u, v_1, v_2 is an induced P_3.

Cluster Editing via induced P_3's

- G is a cluster graph if and only it does not contain an induced P_3



- G has cluster editing of size at most k if and only if at least one of the graphs

$$(V, E \setminus uv), (V, E \setminus vw), (V, E \cup uw)$$

has a cluster editing of size at most k-1.

Algorithm ce$(G = (V, E), k)$
Output: Whether G has a cluster editing of size at most k.

```
1: if k = 0 then return true if G is cluster graph, return false otherwise
2: if ∃ induced P_3 (u, v, w) in G then
3:     return ce((V, E ∪ uw), k − 1) ∨ ce((V, E \ uv), k − 1) ∨ ce((V, E \ vw), k − 1)
4: return true
```

- Runs in $O(3^k n^3)$ time.

$O^*()$ suppresses factors poly in input size

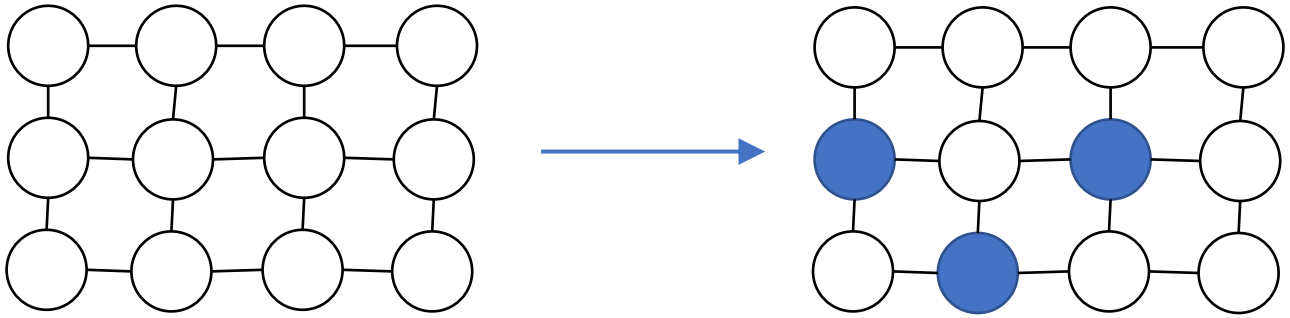
- Usually we focus on f(k): $O^*(3^k)$ time.

Parametrization by Structural Parameters

- So far, we have parameterized by the solution size k
- … but we can parameterize by anything that we want!
- E.g., **structural parameters**: parameterize by some measure of how complex the instance is (e.g.: maximum degree of a graph, number of distinct weights in a knapsack instance,…)

Feedback Vertex Set

- A feedback vertex set (FVS) of an undirected graph $G = (V, E)$ is a subset $X \subseteq V$ such that $G[V \setminus X]$ is a forest.



- In operating systems, feedback vertex sets play a prominent role in the study of deadlock recovery. In the wait-for graph of an operating system, each directed cycle corresponds to a deadlock situation. In order to resolve all deadlocks, some blocked processes have to be aborted.

FVS as structural parameter

- Many problems are easy to solve on forests/trees:
  - Independent Set
  - Vertex Cover
  - …
- This often leads to FPT algorithms
- E.g., Independent Set and Vertex Cover can be solved in $O^*\left(2^{|fvs(G)|}\right)$
- After branching on FVS vertices, we are left with a forest

FVS as structural parameter

- Many problems are easy to solve on forests/trees:
  - Independent Set
  - Vertex Cover
  - …
- This often leads to FPT algorithms
- E.g., Independent Set and Vertex Cover can be solved in $O^*\left(2^{|fvs(G)|}\right)$
- After branching on FVS vertices, we are left with a forest
- … requires first finding a FVS!

Feedback Vertex Set by Iterative Compression

- FVS is FPT parameterized by solution size k
- Idea: we can use the fact that the graph has a small FVS to speed up the search for one

- Given a FVS of size $k+1$, we can find (if it exists) a FVS of size k in FPT time (compression)

Feedback Vertex Set by Iterative Compression

- FVS is FPT parameterized by solution size k
- Idea: we can use the fact that the graph has a small FVS to speed up the search for one

- Given a FVS of size $k+1$, we can find (if it exists) a FVS of size k in FPT time (compression)
- …but we cannot start with FVS of size n, since compression would not be FPT

Iterative compression?

- Crux: it helps if we are given a FVS of size k+1
- Iterative compression allows us to assume this

Algorithm FVS$(G = (V, E), k)$
Output: Whether G has a feedback vertex set of size at most k

1: Let $V = \{v_1, \ldots, v_n\}$
2: Let $X = \{v_1, \ldots, v_k\}$
3: **for** $i = k + 1, \ldots, n$ **do**
4: $X \leftarrow X \cup v_i$ — X is a FVS of $G[\{v_1, \ldots, v_i\}]$ of size at most $k + 1$
start compression
5:
6:
7: Use X to find the minimum FVS of $G[\{v_1, \ldots, v_i\}]$. Write the found FVS to X again
8:
end compression
9: **if** $|X| = k + 1$ **then return false** — check whether the compression was successful
10: **return true**

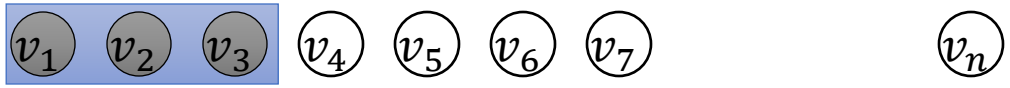
We can start with a FVS of size k of the graph induced by the first k vertices

Iterative compression?

- Crux: it helps if we are given a FVS of size k+1
- Iterative compression allows us to assume this

Algorithm FVS$(G = (V, E), k)$
Output: Whether G has a feedback vertex set of size at most k

1: Let $V = \{v_1, \ldots, v_n\}$
2: Let $X = \{v_1, \ldots, v_k\}$
3: **for** $i = k + 1, \ldots, n$ **do**
4: $X \leftarrow X \cup v_i$ — X is a FVS of $G[\{v_1, \ldots, v_i\}]$ of size at most $k + 1$
start compression
5:
6:
7: Use X to find the minimum FVS of $G[\{v_1, \ldots, v_i\}]$. Write the found FVS to X again
8:
end compression
9: **if** $|X| = k + 1$ **then return false** — check whether the compression was successful
10: **return true**

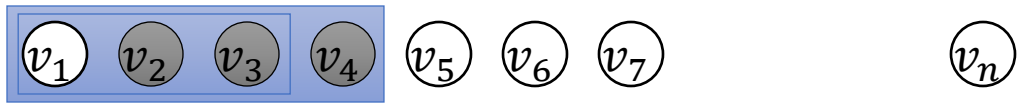


Iterative compression?

- Crux: it helps if we are given a FVS of size k+1
- Iterative compression allows us to assume this

Algorithm FVS$(G = (V, E), k)$
Output: Whether G has a feedback vertex set of size at most k

1: Let $V = \{v_1, \ldots, v_n\}$
2: Let $X = \{v_1, \ldots, v_k\}$
3: **for** $i = k + 1, \ldots, n$ **do**
4: $X \leftarrow X \cup v_i$ — X is a FVS of $G[\{v_1, \ldots, v_i\}]$ of size at most $k + 1$
start compression
5:
6:
7: Use X to find the minimum FVS of $G[\{v_1, \ldots, v_i\}]$. Write the found FVS to X again
8:
end compression
9: **if** $|X| = k + 1$ **then return false** — check whether the compression was successful
10: **return true**

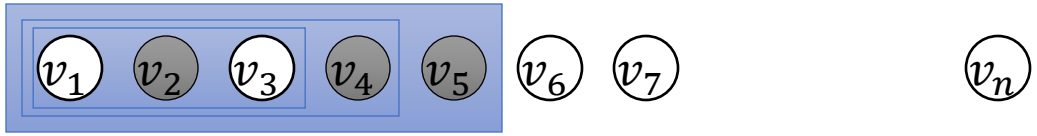


Iterative compression?

- Crux: it helps if we are given a FVS of size k+1
- Iterative compression allows us to assume this

Algorithm FVS$(G = (V, E), k)$
Output: Whether G has a feedback vertex set of size at most k

1: Let $V = \{v_1, \ldots, v_n\}$
2: Let $X = \{v_1, \ldots, v_k\}$
3: **for** $i = k + 1, \ldots, n$ **do**
4: $X \leftarrow X \cup v_i$ — X is a FVS of $G[\{v_1, \ldots, v_i\}]$ of size at most $k + 1$
start compression
5:
6:
7: Use X to find the minimum FVS of $G[\{v_1, \ldots, v_i\}]$. Write the found FVS to X again
8:
end compression
9: **if** $|X| = k + 1$ **then return false** — check whether the compression was successful
10: **return true**

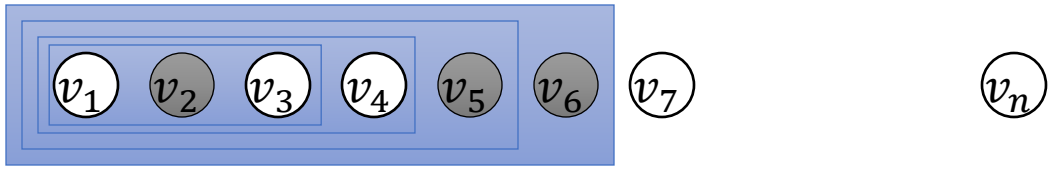


Iterative compression?

- Crux: it helps if we are given a FVS of size k+1
- Iterative compression allows us to assume this

Algorithm FVS$(G = (V, E), k)$
Output: Whether G has a feedback vertex set of size at most k

1: Let $V = \{v_1, \ldots, v_n\}$
2: Let $X = \{v_1, \ldots, v_k\}$
3: **for** $i = k + 1, \ldots, n$ **do**
4: $X \leftarrow X \cup v_i$ — X is a FVS of $G[\{v_1, \ldots, v_i\}]$ of size at most $k + 1$
start compression
5:
6:
7: Use X to find the minimum FVS of $G[\{v_1, \ldots, v_i\}]$. Write the found FVS to X again
8:
end compression
9: **if** $|X| = k + 1$ **then return false** — check whether the compression was successful
10: **return true**

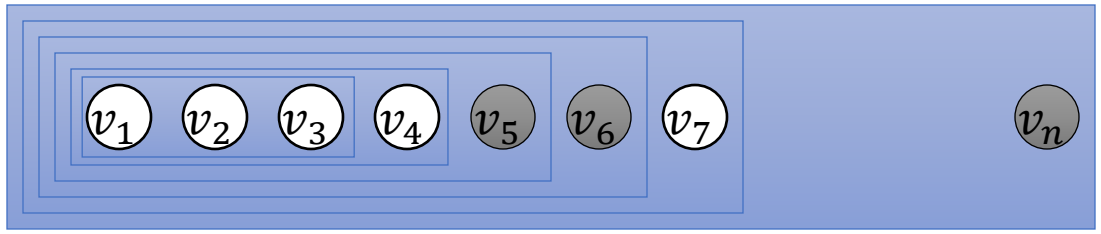


Iterative compression?

- Crux: it helps if we are given a FVS of size k+1
- Iterative compression allows us to assume this

Algorithm FVS$(G = (V, E), k)$
Output: Whether G has a feedback vertex set of size at most k

1: Let $V = \{v_1, \ldots, v_n\}$
2: Let $X = \{v_1, \ldots, v_k\}$
3: **for** $i = k + 1, \ldots, n$ **do**
4: $X \leftarrow X \cup v_i$ — X is a FVS of $G[\{v_1, \ldots, v_i\}]$ of size at most $k + 1$
start compression
5:
6:
7: Use X to find the minimum FVS of $G[\{v_1, \ldots, v_i\}]$. Write the found FVS to X again
8:
end compression
9: **if** $|X| = k + 1$ **then return false** — check whether the compression was successful
10: **return true**



Compression step for Feedback Vertex Set

- Given FVS X of size $k+1$, find FVS X' of size k
- Guess which part $Y = X \cap X'$ of X we "keep"

Algorithm FVS$(G = (V, E), k)$
Output: Whether G has a feedback vertex set of size at most k

1: Let $V = \{v_1, \ldots, v_n\}$
2: Let $X = \{v_1, \ldots, v_k\}$
3: **for** $i = k+1, \ldots, n$ **do**
4: $X \leftarrow X \cup v_i$ — X is a FVS of $G[\{v_1, \ldots, v_i\}]$ of size at most $k+1$

start compression

5: **for** $Y \subseteq X$ **do**
6: **if** disjFVS$(G[V \setminus Y], X \setminus Y, k - |Y|)$ **then**
7: Construct the FVS X' of $G[V \setminus Y]$ of size $k - |Y|$ using self-reduction
8: $Z \leftarrow X' \cup Y; X \leftarrow Z$ — in order to be able to refer to Z in the analysis

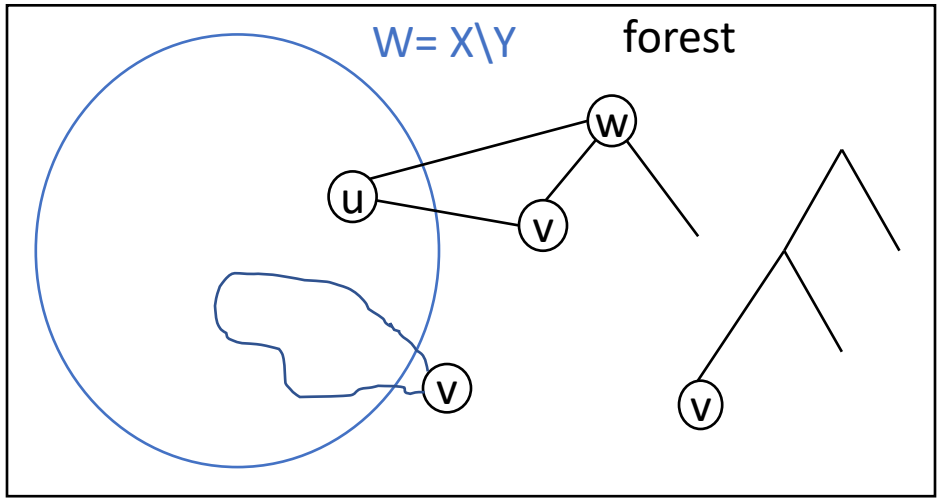
end compression

9: **if** $|X| = k+1$ **then return false** — check whether the compression was successful
10: **return true**

Algorithm `disjFVS`$(G = (V, E), W, k)$ W is a FVS of G
Output: Whether G has a feedback vertex set of size at most k disjoint from W

1: **if** $k < 0$ **then return false**
2: **if** $k = 0$ **then return true** if G is a forest, **return false** otherwise
3: **if** $\exists v \in V$ such that $\deg(v) \leq 1$ **then**
4: **return** `disjFVS`$(G \setminus v, W, k)$
5: **if** $\exists v \in V \setminus W$ such that $G[W \cup \{v\}]$ contains a cycle **then**
6: **return** `disjFVS`$(G \setminus v, W, k-1)$
7: **if** $\exists v \in V \setminus W$ such that $\deg(v) = 2$ and at least one neighbor from v in G is in $V \setminus W$ **then**
8: Let G' be obtained from G by adding the edge between both neighbors of v and removing v
Note: if there was such an edge already, there are multiple edges now!
9: **return** `disjFVS`(G', W, k)
10: Let $x \in V$ such that x has at most one neighbor in $V \setminus W$ and at least two neighbors in W.
11: **return** `disjFVS`$(G \setminus x, W, k-1) \vee$ `disjFVS`$(G, W \cup x, k)$

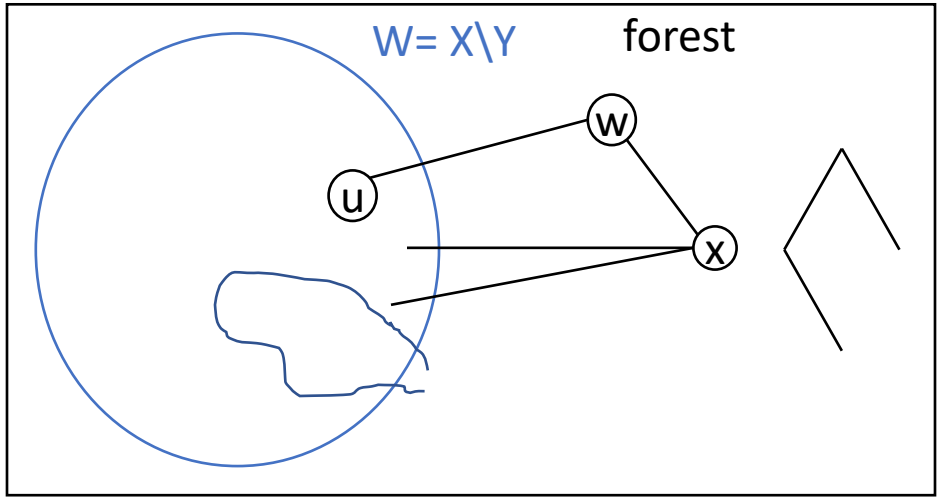
- L3: v is not on any cycle so not relevant
- L5: the only way to hit the cycle is to pick v
- L7: if $N(v) = \{u, w\}$, all cycles including v also include u and w.
  - In any FVS v can be replaced with u, w
  - Thus we may discard including v
  - Delete v, account for the connections via v by adding uw.



Algorithm `disjFVS`$(G = (V, E), W, k)$ W is a FVS of G
Output: Whether G has a feedback vertex set of size at most k disjoint from W

1: **if** $k < 0$ **then return false**
2: **if** $k = 0$ **then return true** if G is a forest, **return false** otherwise
3: **if** $\exists v \in V$ such that $\deg(v) \leq 1$ **then**
4: **return** `disjFVS`$(G \setminus v, W, k)$
5: **if** $\exists v \in V \setminus W$ such that $G[W \cup \{v\}]$ contains a cycle **then**
6: **return** `disjFVS`$(G \setminus v, W, k-1)$
7: **if** $\exists v \in V \setminus W$ such that $\deg(v) = 2$ and at least one neighbor from v in G is in $V \setminus W$ **then**
8: Let G' be obtained from G by adding the edge between both neighbors of v and removing v
Note: if there was such an edge already, there are multiple edges now!
9: **return** `disjFVS`(G', W, k)
10: Let $x \in V$ such that x has at most one neighbor in $V \setminus W$ and at least two neighbors in W.
11: **return** `disjFVS`$(G \setminus x, W, k-1) \vee$ `disjFVS`$(G, W \cup x, k)$

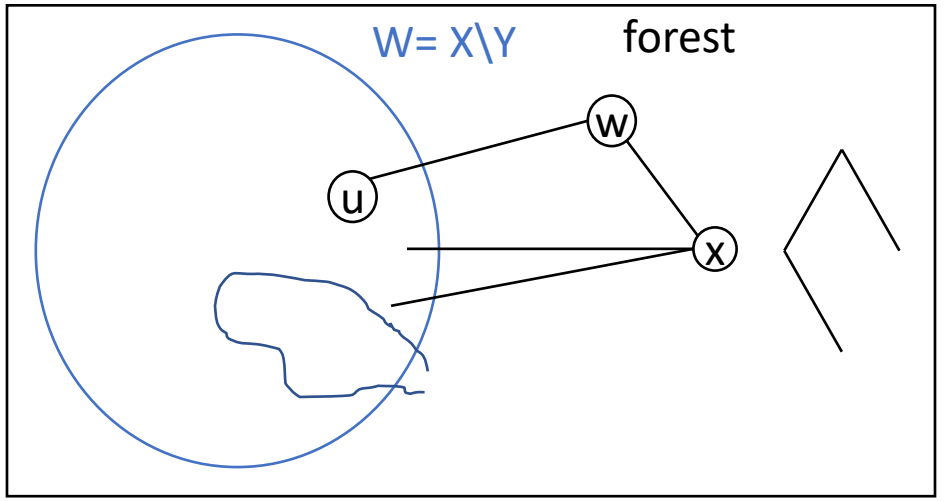
- Let x be leaf in forest with ≥ 2 nbs in W
  - if x has no such neighbor, L3 applies
  - if x has 1 such neighbor, L7 applies.
- Now decide whether x is in or not.
  - if it is, simply remove it
  - if it is not, discard by adding to W
- Finishes correctness



Algorithm `disjFVS`$(G = (V, E), W, k)$ W is a FVS of G
Output: Whether G has a feedback vertex set of size at most k disjoint from W

1: **if** $k < 0$ **then return false**
2: **if** $k = 0$ **then return true** if G is a forest, **return false** otherwise
3: **if** $\exists v \in V$ such that $\deg(v) \leq 1$ **then**
4: **return** `disjFVS`$(G \setminus v, W, k)$
5: **if** $\exists v \in V \setminus W$ such that $G[W \cup \{v\}]$ contains a cycle **then**
6: **return** `disjFVS`$(G \setminus v, W, k-1)$
7: **if** $\exists v \in V \setminus W$ such that $\deg(v) = 2$ and at least one neighbor from v in G is in $V \setminus W$ **then**
8: Let G' be obtained from G by adding the edge between both neighbors of v and removing v
Note: if there was such an edge already, there are multiple edges now!
9: **return** `disjFVS`(G', W, k)
10: Let $x \in V$ such that x has at most one neighbor in $V \setminus W$ and at least two neighbors in W.
11: **return** `disjFVS`$(G \setminus x, W, k-1) \vee$ `disjFVS`$(G, W \cup x, k)$

- Running time:
  - Does not decrease k every time.
  - But surely instance should get easier?!
- $k + \#\text{cc}(G[W])$ does decrease
  - k decreases in first branch
  - $\#\text{cc}(G[W])$ in second (L5)
- So as in previous analyses, algo runs in
  - $O^*\left(2^{k+\#cc(G[W])}\right) = O^*(4^k)$
- Total running time: $O^*(8^k)$.



Subset Sum

- Given integers $w_1, \ldots, w_n, t$ find $X \subseteq \{1, \ldots, n\}$ such that $\sum_{i \in X} w_i = t$
- {1 2 3 4 5 6 7 8 9 10 11 12}, t= 23
- We'll see $O^*(2^{n/2})$ time algorithm here.
- 2SUM: given $a_1, \ldots, a_m, b_1, \ldots, b_m, t$ find i, j such that $a_i + b_j = t$.
- We'll see: 2SUM in $O(m \lg m)$ time.

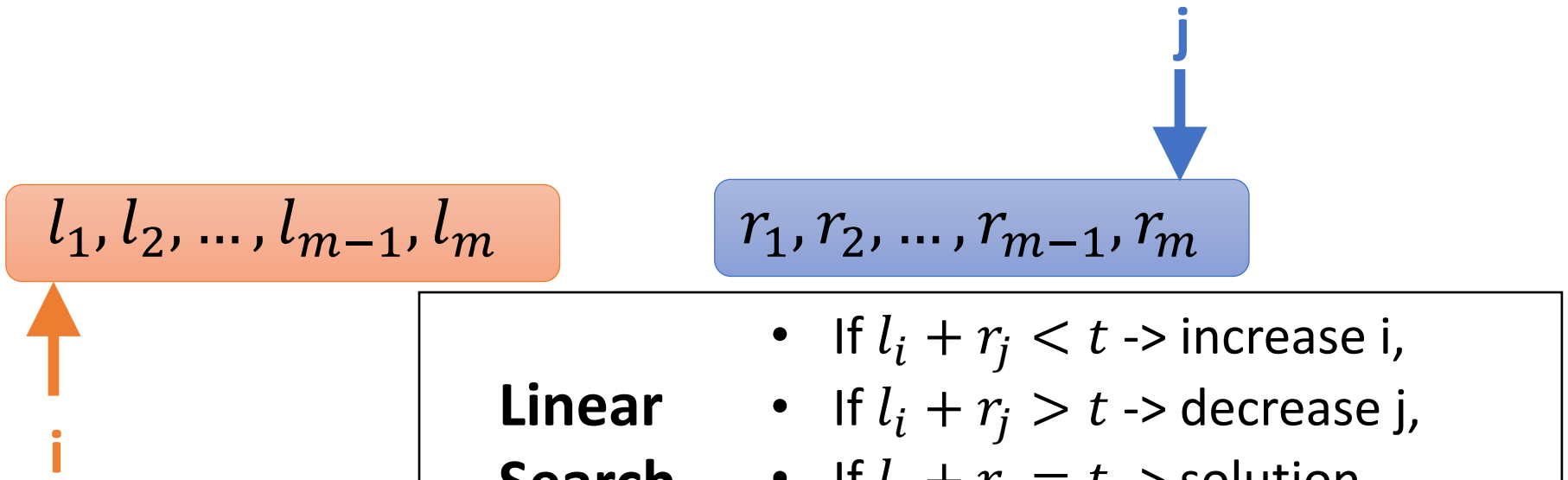
Subset Sum via 2SUM

- Use as follows; suppose n even.
  - Create an int $a_i = \sum_{e \in X} w_e$ for all $X \subseteq \{1, \ldots, n/2\}$
  - Create an int $b_i = \sum_{e \in Y} w_e$ for all $Y \subseteq \{n/2 + 1, \ldots, n\}$
- There exist i, j with $a_i + b_j = t$ iff there exists $X \subseteq \{1, \ldots, n/2\}$, $Y \subseteq \{n/2 + 1, \ldots, n\}$ such that $\sum_{e \in X} w_e + \sum_{e \in Y} w_e = t$ ($X \cup Y$ is a solution)
- 2SUM in $O(m \lg m)$ time, $m = 2^{n/2}$.

Linear Time for 2SUM

Algorithm 2SUM(L, R, t) L and R are lists of length m of integers, $t \in \mathbb{Z}$
Output: Whether there exist $l \in L$, $r \in R$ such that $l + r = t$

1: Sort L, R to obtain lists $l_1, \ldots, l_m$ and $r_1, \ldots, r_m$
2: $i \leftarrow 1$, $j \leftarrow m$.
3: **while** $i \leq n \wedge j \geq 1$ **do**
4: **if** $l_i + r_j = t$ **then return true**
5: **if** $l_i + r_j < t$ **then** $i \leftarrow i + 1$ **else** $j \leftarrow j - 1$
6: **return false**



Linear Search

- If $l_i + r_j < t$ -> increase i,
- If $l_i + r_j > t$ -> decrease j,
- If $l_i + r_j = t$ -> solution
- Declare NO if i or j out of range

Linear Time for 2SUM

Algorithm 2SUM(L, R, t) L and R are lists of length m of integers, $t \in \mathbb{Z}$
Output: Whether there exist $l \in L$, $r \in R$ such that $l + r = t$

```
1: Sort L, R to obtain lists l_1, ..., l_m and r_1, ..., r_m
2: i ← 1, j ← m.
3: while i ≤ n ∧ j ≥ 1 do
4:     if l_i + r_j = t then return true
5:     if l_i + r_j < t then i ← i + 1 else j ← j − 1
6: return false
```

- Clearly correct if it returns true
- If there exists x, y such that $l_x + r_y = t$,
  - consider the first moment where $i = x$ or $j = y$
  - say it is $i = x$, since $y < j$, j will be lowered until $l_x + r_j = t$. Case $j = y$ is similar
- Clearly runs in $O(m \lg m)$ time.

Subset Sum via 2SUM

- To solve subset sum:
  - Enumerate 2 lists of $2^{n/2}$ subset sums of $n/2$ elements
  - 2SUM in $O(m \lg m)$ time, $m = 2^{n/2}$
- $O^*\left(2^{n/2}\right) = O^*\left(\sqrt{2}^{n}\right) = O^*(1.412^n)$
- Roughly doubles the n we can handle in the same time

- "Meet in the middle"
- Useful as optimisation for many problems, e.g., A* pathfinding