Exact Algorithms for NP-hard problems

IN4344 Advanced Algorithms: Part 2, Lecture 1

Today

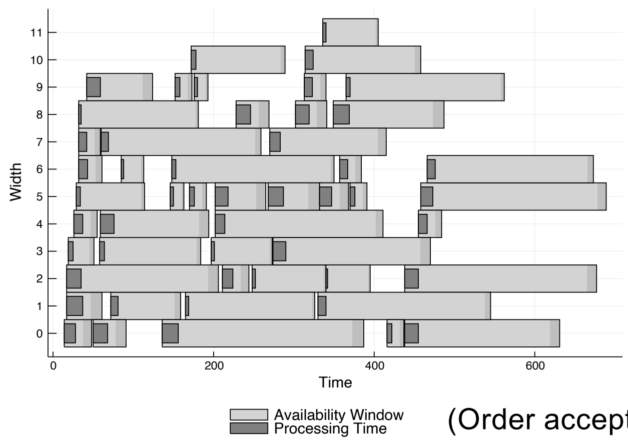
- Intro Part 2
- Search Trees
- Bounded Search Trees
- Fixed Parameter Tractability

Mathijs de Weerdt

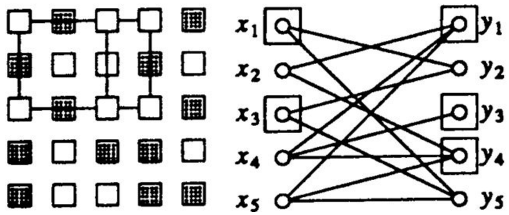
Benefits of following part 2 on exact algorithms

This part of the course will help to design *problem-specific algorithms* that
- if they complete, return the optimal (exact) solution (guaranteed!)
- provide/use deep(er) understanding of the problem
- provide a solid basis for fast heuristics or approximations

For problems like:
- traveling salesperson – e.g. routing & delivery of packages
- single-machine scheduling – e.g. for manufacturing planning
- vertex cover – e.g. for classification of proteins
- independent set – e.g. for VLSI circuit design



(Order acceptance and scheduling problem)



(from Low & Leong (1997), "On the reconfiguration of degradable VLSI/WSI arrays" IEEE)

TU Delft

Exact algorithms for NP-hard problems

Learning objective part 2:

Mastering techniques to solve NP-hard problems exactly

using
1. (complete/bounded) search trees (Ch.10.1+[2,3])
2. dynamic programming (Ch.10.3+[2])
3. tree decomposition (Ch.10.2-10.4 + [2])
4. decision diagrams
5. preprocessing & kernelization [2, 3]

Exam: November 6 (don't forget to register in Osiris **15** days in advance & always check the schedule for time & location)

Exact algorithms for NP-hard problems

Course material part 2:

1. John Kleinberg and Éva Tardos, *Algorithm Design*, Addison Wesley, 2005. Ch.10.
2. Gerhard Woeginger, Exact algorithms for NP-hard problems: A survey, *Combinatorial Optimization*, LNCS 3570, pp 187-207, 2003.
3. Falk Hueffner, Rold Niedermeier and Sebastian Wernicke, Techniques for Practical Fixed-Parameter Algorithms, *The Computer Journal*, 51(1):7–25, 2008.
4. Literature on "Decision Diagrams" (see respective folder on BrightSpace)

Important warning:

We're going to see algorithms that take exponential time!

TU Delft

Why Runtime Matters

Table 2.1 The running times (rounded up) of different algorithms on inputs of increasing size, for a processor performing a million high-level instructions per second. In cases where the running time exceeds 10^{25} years, we simply record the algorithm as taking a very long time.

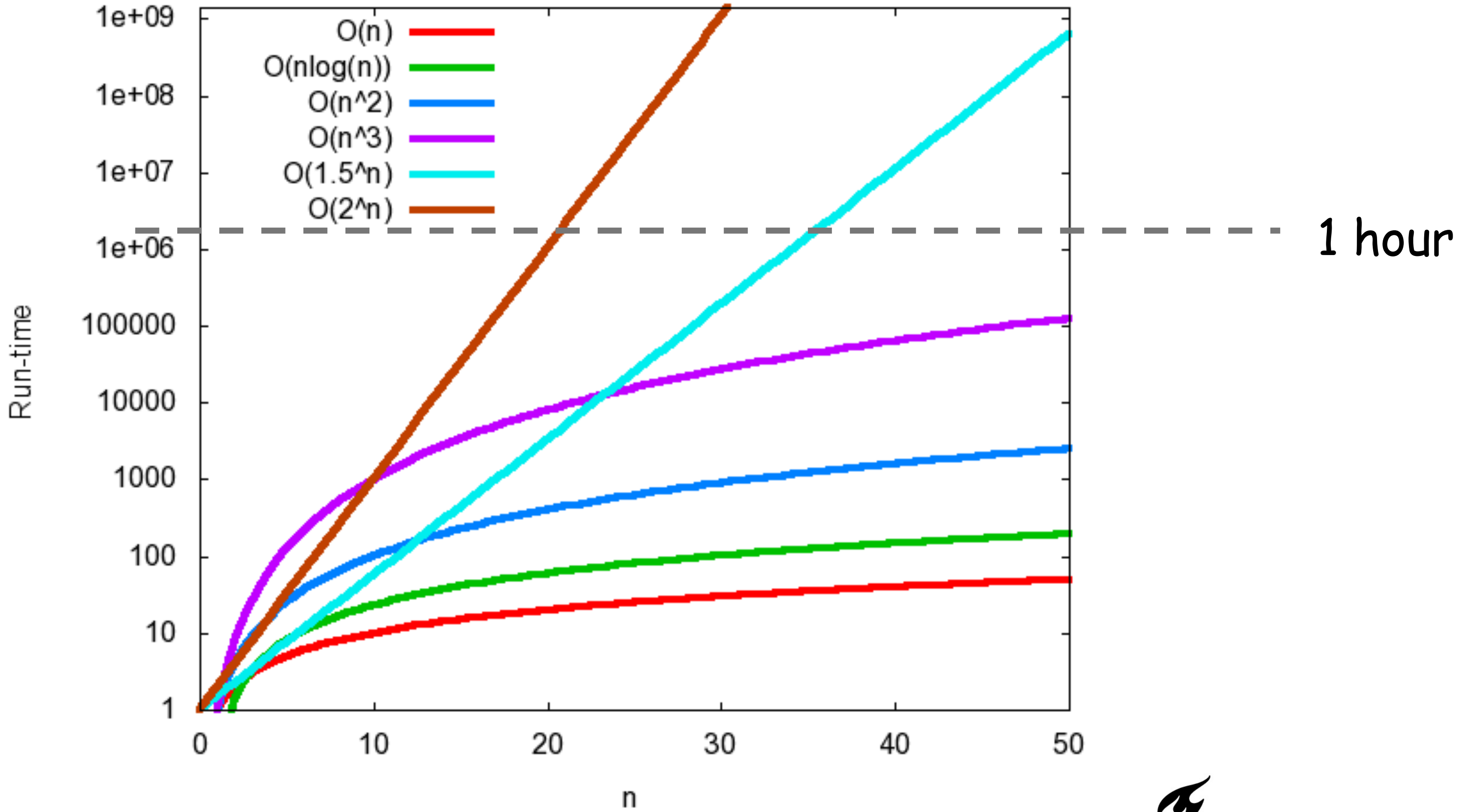
	n	$n \log_2 n$	n^2	n^3	1.5^n	2^n	$n!$
$n = 10$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	4 sec
$n = 30$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	18 min	10^{25} years
$n = 50$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	11 min	36 years	very long
$n = 100$	< 1 sec	< 1 sec	< 1 sec	1 sec	12,892 years	10^{17} years	very long
$n = 1{,}000$	< 1 sec	< 1 sec	1 sec	18 min	very long	very long	very long
$n = 10{,}000$	< 1 sec	< 1 sec	2 min	12 days	very long	very long	very long
$n = 100{,}000$	< 1 sec	2 sec	3 hours	32 years	very long	very long	very long
$n = 1{,}000{,}000$	1 sec	20 sec	12 days	31,710 years	very long	very long	very long

From: John Kleinberg and Éva Tardos, *Algorithm Design*, Addison Wesley, 2005. Ch.2
(Or actually: More Programming Pearls, p. 82, 400MhZ Pentium II scaled up)

TU Delft

Runtimes in logarithmic scale

log scale
(ms)



Analyzing exponential-time algorithms

Upper bound on the runtime

Ex. $T(n) = 32n^2 + 17n + 32$.

Upper bound on run time is… $O(n^2)$ (but also $O(n^3)$, $O(n^4)$, etc.)

New notation omitting polynomial part: $O^*(2^n)$ versus $O^*(1.9^n)$

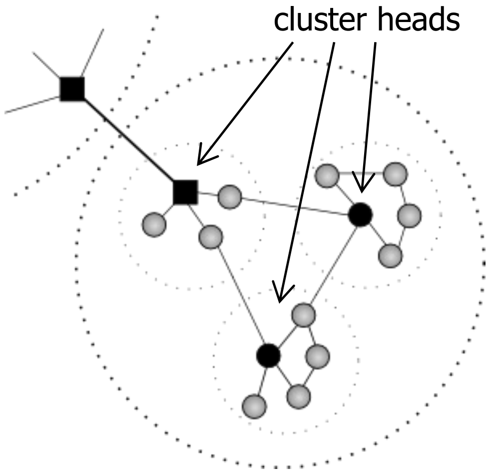
Def. An algorithm A has a runtime bounded by $O^*(T(n))$ iff there exists a polynomial function p such that the run-time of A has an upper bound of $O(p(n) \cdot T(n))$.

Complete search trees

by example

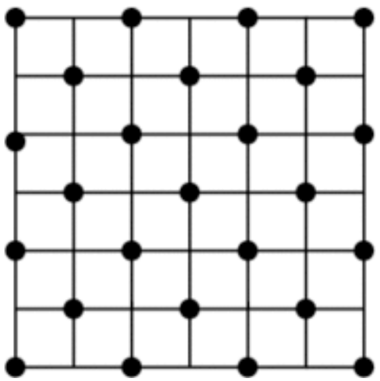
- routing in ad-hoc wireless networks
- 3-satisfiability

Routing in ad-hoc wireless networks



- Nodes can communicate directly to neighbours, but this interferes with other close nodes.
- Common approach is to create clusters

Goal. find *at least k cluster heads that do not interfere* during simultaneous transmissions



Similar to

Given. a set of potential locations for e.g. a Starbucks and connections if they interfere.

Goal. select at least k locations that do not interfere.

Q. How can we model such problems?

Independent set

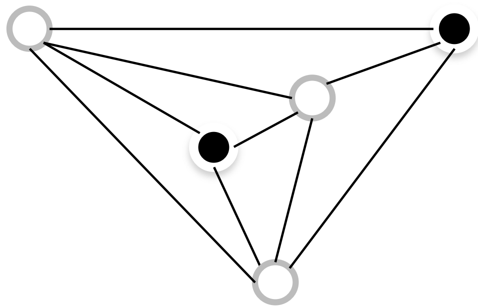
Independent Set (decision problem in NP)

Given

- a graph G=(V,E) with n vertices, and
- an integer k

Decide

- whether there exist a subset S of V of size at least k such that no two vertices in S are neighbors



Q. Give a trivial algorithm for the independent set problem.

Independent set with brute force (enumeration)

Independent set. Given a graph, is there an independent set of size at least k?

Enumerate all subsets:

```
S* ← φ
foreach subset S out of n nodes {
   return true when S is an independent set and
        S is of size at least k
}
```

Q. What is the runtime complexity written using O()?
A. $O(n^2 2^n)$, (or $O(\binom{n}{k} \cdot kn)$ if considering all subsets of size k)

Q. What is the run-time complexity written using $O^*()$?
A. $O^*(2^n)$

TU Delft

Pruning the search tree: Independent set

Given
- a graph G=(V,E) with n vertices, and
- an integer k

Decide
- whether there exists a subset S of V of size at least k such that no two vertices in S are neighbors

General idea of a search tree:
- *recursive* definition of the solution
- *prevent* repeated subproblems as much as possible

Q. Recursion for Independent Set: what is a basic decision?

TU Delft

Pruning the search tree: Independent set

Given

- a graph G=(V,E) with n vertices, and
- an integer k

Decide

- whether there exists a subset S of V of size at least k such that no two vertices in S are neighbors

Proof: there is an IS iff there is one with or without v
Directly from definition

Construct a search tree for independent set (IS)

Branch on each vertex v in turn:

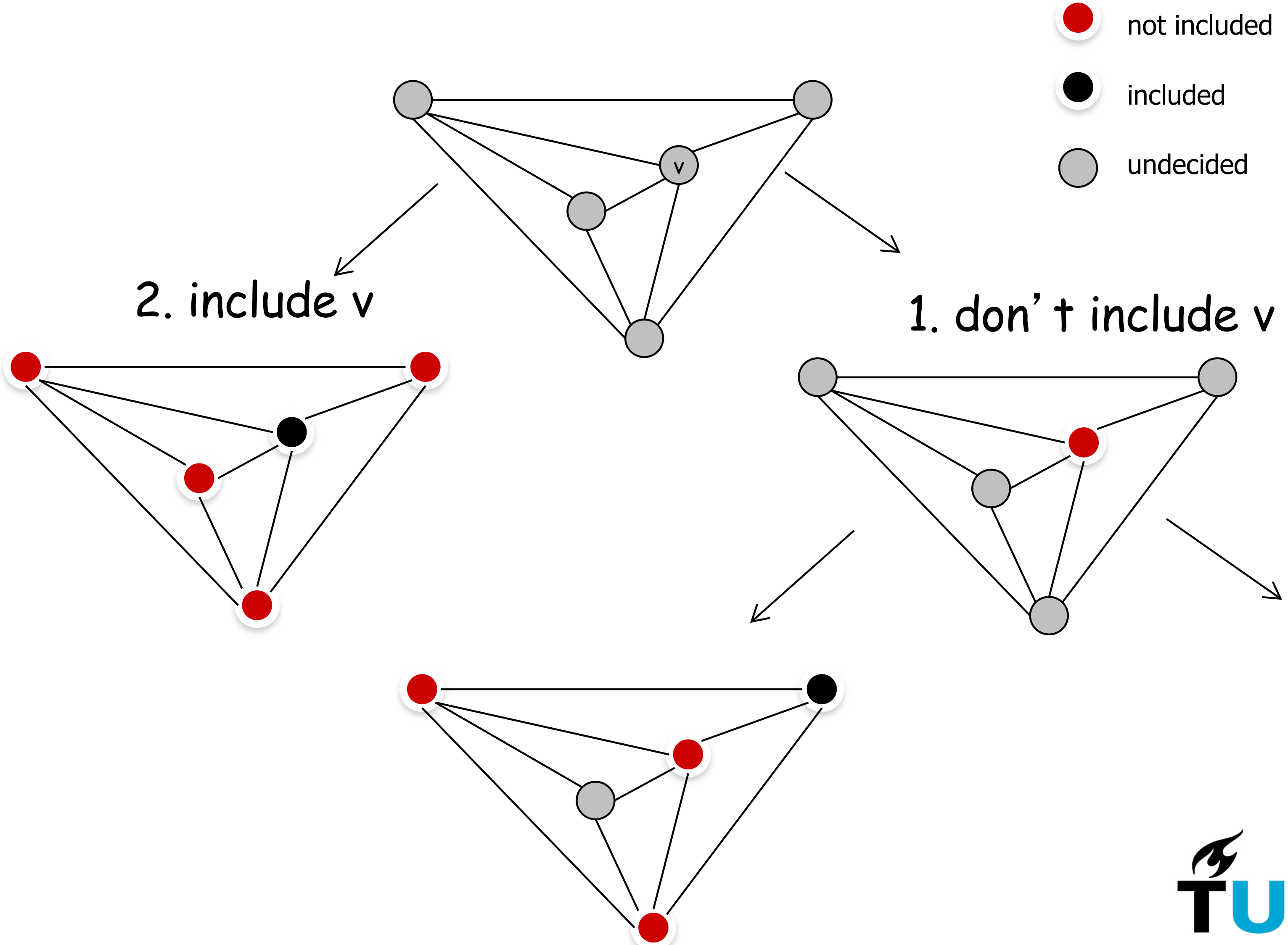
1. don't include v in the independent set
2. include v in the independent set, but don't include its neighbors

Decision problem: stop if independent set found of size k

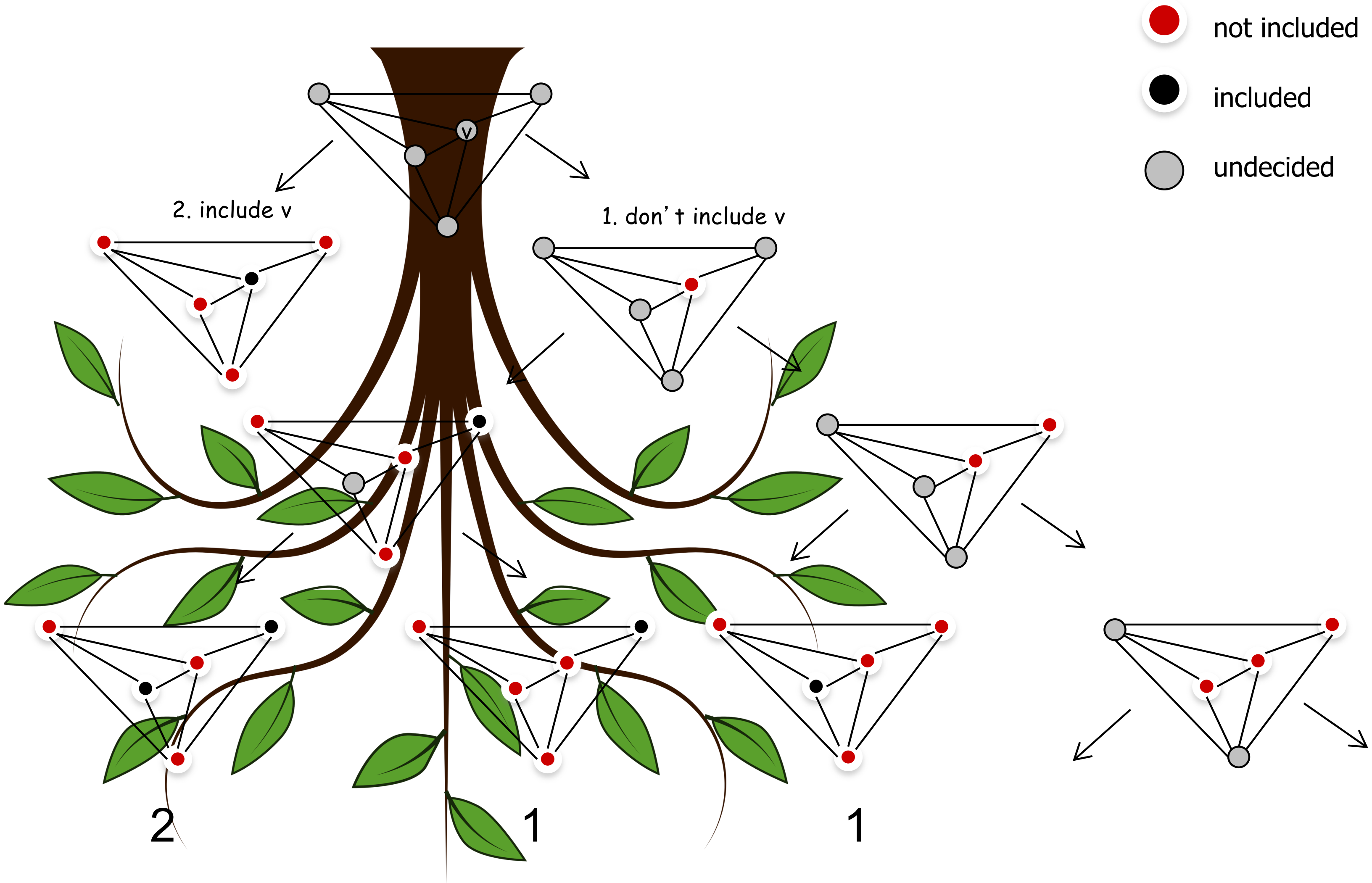
Optimization problem: search complete tree for maximum independent set

TU Delft

Pruning the search tree: Independent set



Pruning the search tree: Independent set



NB: tree is not a *complete* binary tree… maybe not $O^*(2^n)$?

Analyzing the worst-case runtime

Possibilities to consider:
1. don't include v in the independent set
2. include v in the independent set, but don't include its neighbors

Analyze the size of the recursive sub-problems for each case
1. n-1, and
2. n-1 - degree(v)

Q. What leads to the *worst-case runtime bound* for degree(v)?
We decide which v to select: vertex with many undecided neighbors
Worst case then is when all undecided vertices have only 1 or 2 neighbors.
However, in such a case, it is easy to find an independent set!

Q. How?

Independent Set for isolated points

Rule 0. if degree(v) = 0 then: select v.

Claim. A vertex without neighbors should always be included in a maximum independent set.
Pf. (by contradiction)
Let us a maximum independent set S be given.
Let’s look at a vertex v without neighbors.
Suppose, for a contradiction that v is not included in S.
v can be added to S without violating the independence property.
$S \cup \{v\}$ is larger than S and also an independent set.
Contradiction. So v should always be included.

Independent Set for one or two neighbors

Rule 1. if degree(v) = 1 then: select v, don't select the neighbor (u).

Claim. Rule 1 does not remove any solutions with a larger independent set.
Pf. Let any solution S be given without v. We show that S is never larger.

- case 1: neighbor u is included in S;
  - Then remove u and add v
  - This is also an independent set, because v has no other neighbors than u.
  - This new set is of the same size.
- case 2: neighbor u is also not included in S;
  - Then add v
  - This is also an independent set, because v has no other neighbors than u and u is not included.
  - This new set is of a larger size.
- In all cases, the solution S without v is not larger than the solution with v, which is included because of Rule 1.

Independent Set for one or two neighbors

First apply Rule 1 and deal with vertices v with degree(v) > 2, until this is not possible anymore.

Rule 2. if degree(v) = 2 for all vertices then for each cycle: alternatingly select a vertex to include.

Claim. Rule 2 does not remove any solutions with a larger independent set.
Pf.
For any even-length cycle of length c, there at most c/2 vertices to include (otherwise edge with two vertices), which is what rule 2 accomplishes.

For any odd-length cycle of length c, there at most (c-1)/2 vertices to include, which is what rule 2 accomplishes.

Pruning the Search Trees: Independent Set Algorithm

Thm. The following algorithm determines if G has an independent set of size at least k in $O^*(1.3803^n)$ time.

```
boolean Independent-Set(G, k) {
   if (k = 0)                          return true
   if (G contains no vertices)    return false
   apply Rule 0 (for all vertices of degree 0)
   apply Rule 1 (for all vertices of degree 1)
   if there is a vertex v of degree > 2 then
      a = Independent-Set(G – {v}, k)
      b = Independent-Set(G – {v} - N(v), k-1)
      return a or b
   else
      run Rule 2 for vertices of degree 2
}
```

where **N(v)** is the set of neighbors of v

Analyzing the worst-case run-time

Possibilities to consider:

1. don't include v in the independent set
2. include v in the independent set, but don't include its neighbors

Analyze the size of the recursive sub-problems for each case

1. n-1, and
2. n-1 - degree(v)

Determine the worst case for degree(v)

We branch only vertex with at least 3 undecided neighbors.

Worst-case thus is when all undecided vertices have 3 neighbors.

Recurrence relation describing the run time T(n)

$T(n) \leq T(n-1) + T(n-4) + O(n+m)$

O(n+m) for updating graph and, dealing with rules 1 and 2, and selecting vertex with three or more neighbors

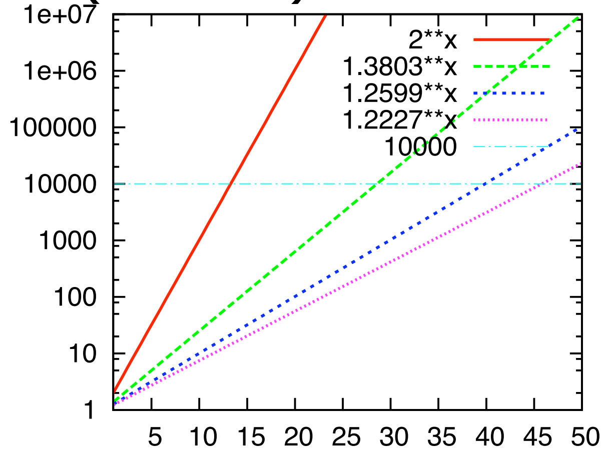
Analyzing the worst-case run-time

Recurrence relation describing the run time T(n):
$T(n) \leq T(n-1) + T(n-4) + O(n+m)$

O(n+m) for updating graph and selecting vertex with three or more neighbors

Evaluate this recurrence (to compare to $O^*(2^n)$)
Idea. Find α such that $T(n) \leq O^*(\alpha^n)$

1. ignore polynomial part of O(n+m)
2. rewrite T(n) to α^n
3. so solve α from $\alpha^n = \alpha^{n-1} + \alpha^{n-4}$, or equivalently (divide by α^{n-4})
4. $\alpha^4 = \alpha^3 + 1$
5. use e.g. Matlab to find out that $\alpha \approx 1.3803$, so T(n) is $O^*(1.3803^n)$



Later improvements

By careful analysis of special (sub)cases:

- $O^*(1.2599^n)$ in 1977
- $O^*(1.2108^n)$ in 1986 (using exponential space)
- $O^*(1.2227^n)$ in 1999 (using polynomial space)

General idea

Search tree: efficient representation of essential *feasible* solutions

- root is complete problem
- children are smaller subproblems: alternatives for a single decision (mutually exclusive, all need to be investigated)
- the smaller subcases the better (using worst case analysis)!
- resolve recurrence by assuming exponential runtime

E.g. if both children have size n-1:

- $T(n) = T(n-1) + T(n-1) + O(n)$
- Solve from from $\alpha^n = \alpha^{n-1} + \alpha^{n-1} = 2\alpha^{n-1}$, so $\alpha=2$.
  (i.e. complete binary tree)

Pruning the search tree: k-satisfiability

Given

- Set of logical (Boolean) variables $X=\{x_1, \ldots, x_n\}$
- A Boolean formula F in k-CNF (i.e. each clause is a disjunction with at most k literals) with m clauses (with $m \leq n^k$)

Decide

- whether there exists a satisfying assignment of $x_1, \ldots, x_n$ for all m clauses

Q. Given $F= \{x_1 \vee \neg x_2,\ x_1 \vee x_2\}$, is F satisfiable?
A. Yes, $x_1=1$, and x_2 can be both 0 or 1.

First consider 2-satisfiability
Q. How can we simplify $\{x_1 \vee \neg x_2,\ x_3 \vee x_2\}$? What if there is no pair of clauses, one with x_2, the other with $\neg x_2$?
A. To $\{x_1 \vee x_3\}$. Repeat until either no such situation exists or we have both $\{x_1 \vee x_1\}$ and $\{\neg x_1 \vee \neg x_1\}$.
However, 3-satisfiability is NP-complete.

Pruning the search tree: 3-satisfiability

Given

- Set of logical (boolean) variables $X=\{x_1, \ldots, x_n\}$
- A Boolean formula F in 3-CNF (i.e. each clause is a disjunction with at most 3 literals) with m clauses (with $m \leq n^3$)

Decide

- whether there exists a satisfying assignment of $x_1, \ldots, x_n$ for F

Q. What is the run-time of a trivial algorithm for 3-satisfiability?
A. $O^*(2^n)$ for trying all assignments.
Q*. How to construct a search tree for 3-satisfiability? (3 min)

Pruning the search tree: 3-satisfiability

A failed attempt…: Suppose we branch on variables v

1. v is true
2. v is false

for each of these solve the remaining sub-problem of size n-1

Recurrence relation describing the runtime T(n)

$T(n) \leq 2T(n-1) + O(n)$

Evaluate this recurrence

Find α such that $T(n) \leq O^*(\alpha^n)$

1. ignore polynomial part of O(n)
2. rewrite T(n) to α^n
3. so solve α from $\alpha^n = 2\alpha^{n-1}$, or equivalently (divide by α^{n-1})
4. $\alpha = 2$

So, T(n) is $O^*(2^n)$… same as trivial algorithm!

TU Delft

Pruning the search tree: 3-satisfiability

Given
- Set of logical (Boolean) variables $X=\{x_1, \ldots, x_n\}$
- A Boolean formula F in 3-CNF (i.e. each clause is a disjunction with at most 3 literals) with m clauses (with $m \leq n^3$)

Decide
- whether there exists a satisfying assignment of $x_1, \ldots, x_n$ for F

Q. What is the runtime of a trivial algorithm for 3-satisfiability?
A. $O^*(2^n)$ for trying all assignments.
Q*. How to construct a search tree for 3-satisfiability?
Hint. Construct a search tree by considering each clause in turn.
Q. Given a clause $L_1 \vee L_2 \vee L_3$ which sub-cases lead to a smaller search tree?

1. L_1 is true
2. L_1 is false, and L_2 is true
3. L_1 and L_2 are false and L_3 is true

TU Delft

Pruning the search tree: 3-satisfiability

Sub-cases to consider:

1. L_1 is true
2. L_1 is false, and L_2 is true
3. L_1 and L_2 are false and L_3 is true

Idea. In each case, make the assignments accordingly and see if we can find a satisfying assignment for the smaller problem.

Q. How large is the recursive problem in each sub-case (in terms of n, the number of unfixed Boolean variables)?

1. n-1
2. n-2
3. n-3

Q. What is the recurrence relation describing the run time T(n) for this search tree?

A. $T(n) \leq T(n-1) + T(n-2) + T(n-3) + O(n+m)$

O(n+m) for updating clauses and selecting clause with three unknowns

Pruning the search tree: 3-satisfiability

Recurrence relation describing the run time T(n):

$T(n) \leq T(n-1) + T(n-2) + T(n-3) + O(n+m)$

O(n+m) for updating clauses and selecting clause with three unknowns (or less if these are all processed)

Q*. How evaluate this recurrence to compare to $O^*(2^n)$?

Idea. Find α such that $T(n) \leq O^*(\alpha^n)$

1. ignore polynomial part of O(n+m)
2. rewrite T(n) to α^n
3. so solve α from $\alpha^n = \alpha^{n-1} + \alpha^{n-2} + \alpha^{n-3}$, or equivalently
4. $\alpha^3 = \alpha^2 + \alpha + 1$ (dividing by α^{n-3})
5. use Matlab to find out that $\alpha \approx 1.8393$, so T(n) is $O^*(1.8393^n)$

Later improvements

By careful analysis of special (sub)cases:

- $O^*(1.6181^n)$ in 1985 [1]
- $O^*(1.5783^n)$ in 1992 [2]
- $O^*(1.4963^n)$ in 1999 [3]

[1] B. Monien and E. Speckenmeyer [1985]. Solving satisfiability in less than 2n steps. *Discrete Applied Mathematics* 10, 287–295.
[2] I. Schiermeyer [1992]. Solving 3-satisfiability in less than O(1.579n) steps. Selected papers from *Computer Science Logic* (CSL'1992), Springer, LNCS 702, 379–394.
[3] O. Kullmann [1999]. New methods for 3-SAT decision and worst-case analysis. Theoretical Computer Science 223, 1–72.

Applications of 3-satisfiability

http://www.satcompetition.org

The International SAT Competition Web Page

Current Competition

SAT 2023 Competition	
Organizers	Marijn Heule, Matti Järvisalo, Martin Suda, Markus Iser, Tomáš Balyo

Past Competitions, Races and Evaluations

SAT 2022 Competition	
Organizers	Marijn Heule, Matti Järvisalo, Martin Suda, Markus Iser, Tomáš Balyo

SAT 2021 Competition	
Organizers	Marijn Heule, Matti Järvisalo, Martin Suda, Markus Iser, Tomáš Balyo Nils Froleyks

SAT 2020 Competition	
Organizers	Marijn Heule, Matti Järvisalo, Martin Suda, Markus Iser, Tomáš Balyo Nils Froleyks

SAT 2019 Race	
Organizers	Marijn Heule, Matti Järvisalo, Martin Suda

SAT 2018 Competition	
Organizers	Marijn Heule, Matti Järvisalo, Martin Suda
Slides	Slides used at SAT 2018
Proceedings	Descriptions of the solvers and benchmarks
Benchmarks	Available here
Solvers	Available here

- The Eclipse open platform uses SAT for solving dependencies between components [Le Berre and Rapicault, IWOCE 2009]
- Intel core I7 processor designed with the help of SAT solvers [Kaivola et al, CAV 2009]
- Windows 7 device drivers verified using SAT related technology (Z3, SMT solver) [De Moura and Bjorner, IJCAR 2010]

TU Delft

Bounded Search Trees

… and fixed parameter tractability

Bounded Search Trees (FPT)

Idea. Bound size of search tree exponential in parameter k, but *polynomially in n*

some property of input significantly smaller than n

In particular, the *depth* should depend on k and not on n.

Examples

- optimal placement of museum cameras
- dominating set in planar graphs (in paper, not in lecture)

Optimal placement of museum cameras

Given. museum consisting of corridors

Goal. Can we place at most k cameras in corners
such that all corridors are watched over?

Q. How is the (abstract) decision problem called?

Vertex cover

Given

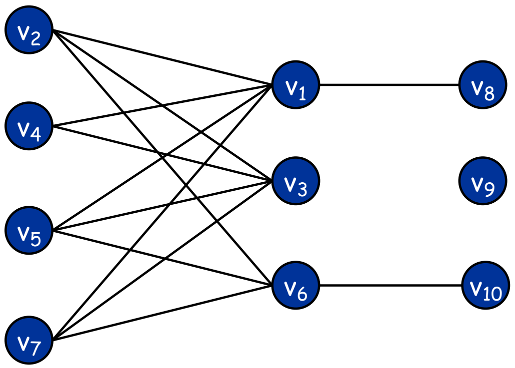
- an undirected graph $G=(V,E)$
- a nonnegative integer k

Decide

- is there a subset of vertices $S \subseteq V$ with k or fewer vertices such that each edge in E has one endpoint in S?

Bad news. Vertex cover is NP-complete

Q. Does the graph below (10 vertices) have a vertex cover of size 3 or less?



Vertex Cover

Given
- an undirected graph G=(V,E)
- a nonnegative integer k

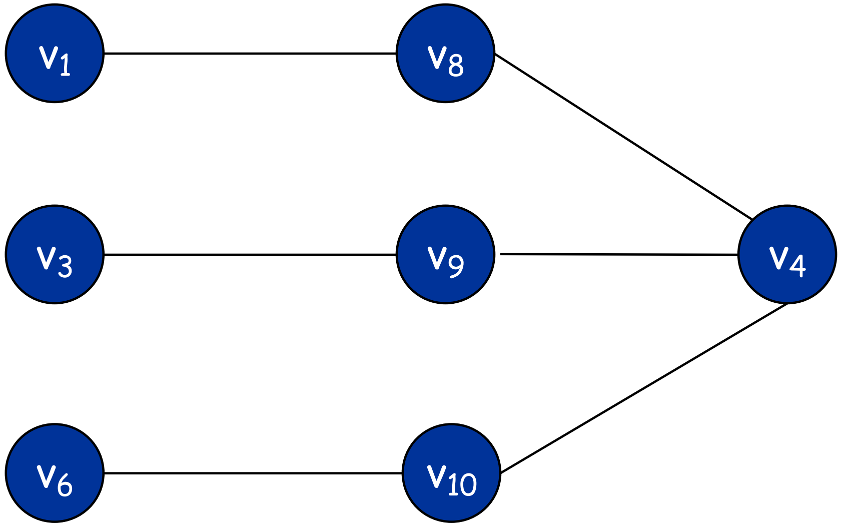
Decide
- is there a subset of vertices $S \subseteq V$ with k or fewer vertices such that each edge in E has one endpoint in S?

Q. What is a trivial algorithm?
A. $O^*(2^n)$ to try all subsets.

Q. What is a trivial algorithm if k is small?
A. Try all $\binom{n}{k} = O^*(n^k)$ subsets of size k.

Q. What seems a likely greedy strategy here to select vertices?

Counter-example to greedily including vertices with high degree



Bounded Search Trees: Vertex Cover

First let's get rid of a border case

Q. How many edges can be covered at most with k vertices?

What happens when there are more edges than that?

A. Each vertex covers at most n-1 edges. Thus at most k(n-1) edges can be covered. If graph contains more edges -> no.

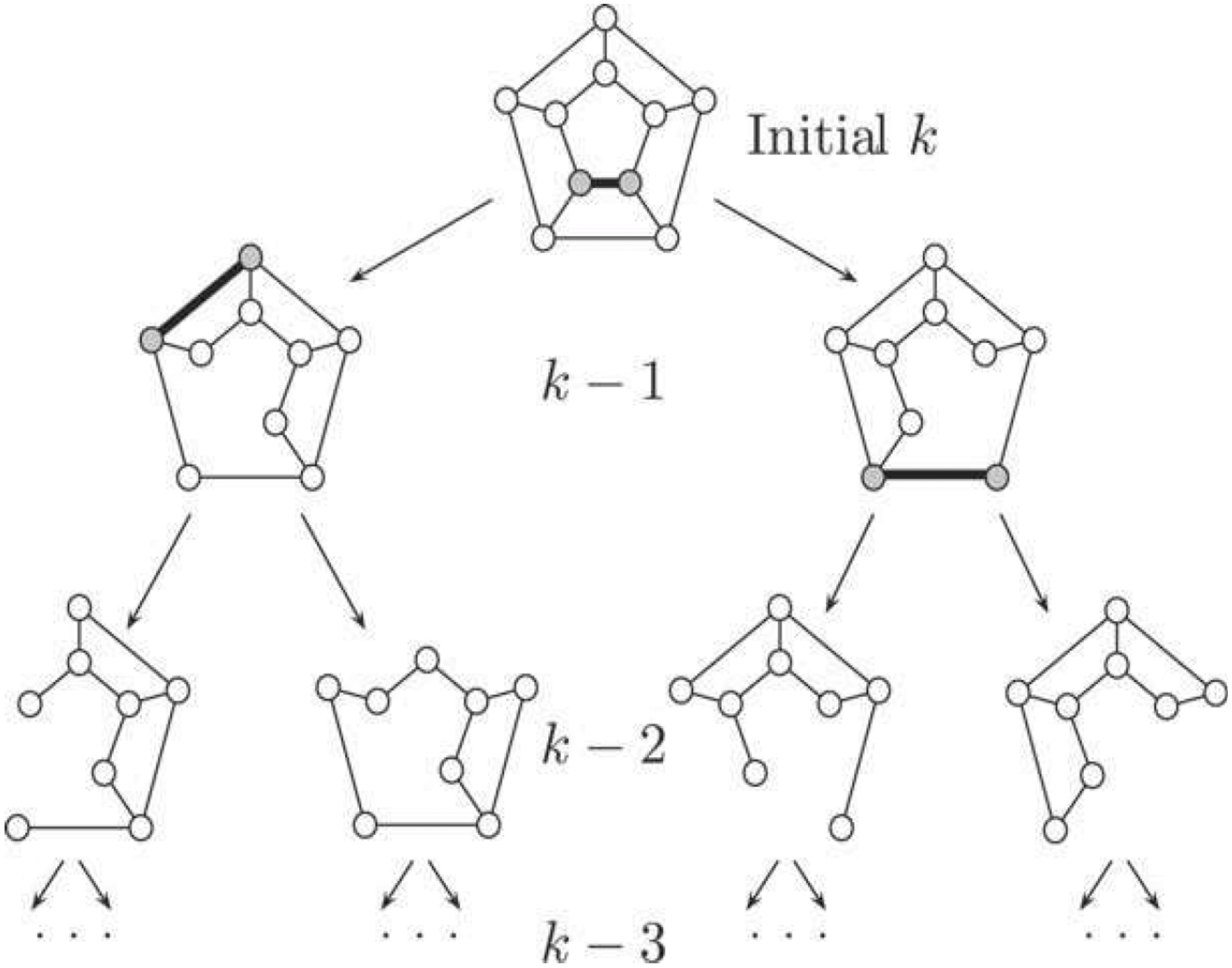
Bounded Search Trees: Vertex Cover

Construct a search tree for vertex cover. Q. How?
Consider each edge in turn.
Considering vertices simply in turn may lead to depth > k. (However, more about considering vertices in turn later.)

Given edge (u,v) consider the following possibilities
1. include u in the vertex cover; find cover of G-{u} of size k-1, or
2. include v in the vertex cover; find cover of G-{v} of size k-1

delete respectively u or v and all incident edges

Bounded Search Trees: Vertex Cover Recursion Tree



Bounded Search Trees: Vertex Cover Algorithm

Thm. The following algorithm determines if G has a vertex cover of size $\leq$ k in $O^*(2^k)$ time.

```
boolean Vertex-Cover(G, k) {
   if (G contains no edges)    return true
   if (G contains ≥ kn edges) return false

   let (u, v) be any edge of G
   a = Vertex-Cover(G - {u}, k-1)
   b = Vertex-Cover(G - {v}, k-1)
   return a or b
}
```

Bounded Search Trees: Vertex Cover: Correctness

Claim. Let (u,v) be an edge of G. G has a vertex cover of size $\leq$ k iff at least one of G – { u } and G – { v } has a vertex cover of size $\leq$ k-1.

Pf. $\Leftarrow$

- Suppose S is a vertex cover of G – { u } of size $\leq$ k-1.
- Then S $\cup$ { u } is a vertex cover of G (since all added edges are covered by u) and is of size $\leq$ k. Analog for v and elimination of "or" ▪

Pf. $\Rightarrow$

- Suppose G has a vertex cover S of size $\leq$ k.
- S must contain either u or v (or both). W.l.o.g. assume it contains u.
- S – { u } is a vertex cover of G – { u }. S – { u } is of size $\leq$ k-1.

Lemma. The BST algorithm for Vertex Cover is correct.
Pf. (with induction over k)
Base. Trivially, in case k=0, but there are 0 edges left to be covered.
Step. Follows immediately from the claim.▪

Bounded Search Trees: Vertex Cover Algorithm

Thm. The following algorithm determines if G has a vertex cover of size $\leq$ k in $O^*(2^k)$ time.

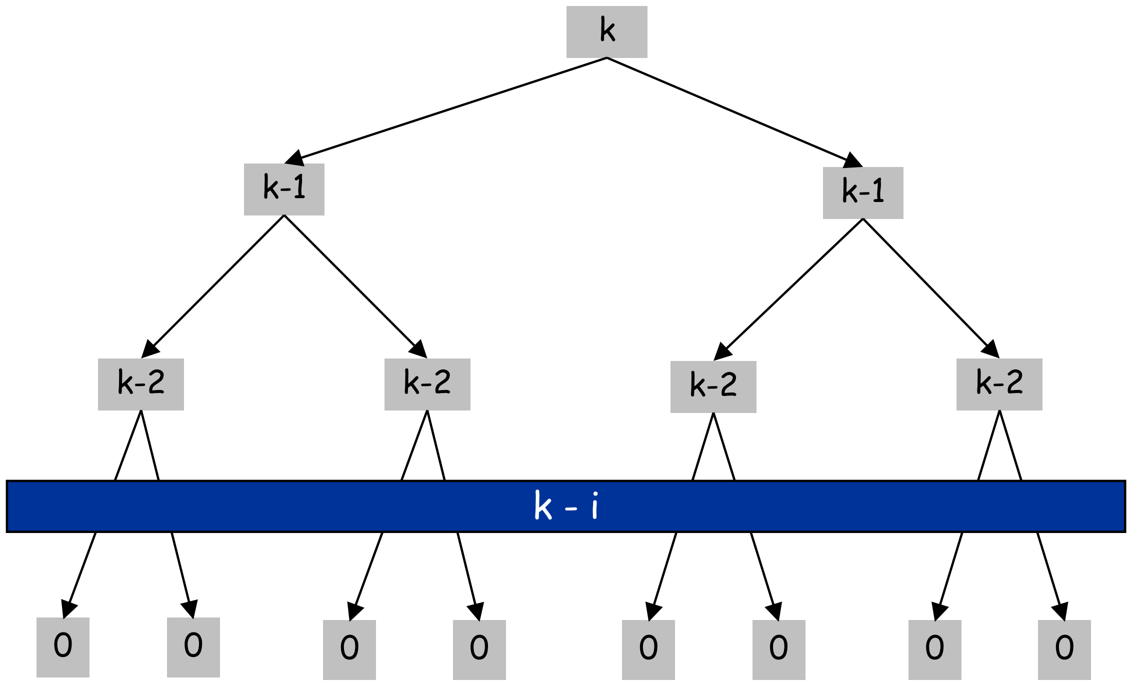
```
boolean Vertex-Cover(G, k) {
   if (G contains no edges)    return true
   if (G contains ≥ kn edges) return false

   let (u, v) be any edge of G
   a = Vertex-Cover(G - {u}, k-1)
   b = Vertex-Cover(G - {v}, k-1)
   return a or b
}
```

Q. Why $O^*(2^k)$?

Bounded Search Trees: Vertex Cover Recursion Tree

$$T(n,k) \le \begin{cases} cn & \text{if } k = 1 \\ 2T(n,k-1) + ckn & \text{if } k > 1 \end{cases} \quad \Rightarrow \quad T(n,k) \le 2^k c\,k\,n$$

So search tree: $O^*(2^k)$



Bounded Search Trees: Vertex Cover Algorithm

Thm. The following algorithm determines if G has a vertex cover of size $\leq$ k in $O^*(2^k)$ time.

```
boolean Vertex-Cover(G, k) {
   if (G contains no edges)    return true
   if (G contains ≥ kn edges) return false

   let (u, v) be any edge of G
   a = Vertex-Cover(G - {u}, k-1)
   b = Vertex-Cover(G - {v}, k-1)
   return a or b
}
```

Pf.

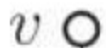
- Correctness follows from previous claim and observation that cover can cover at most k(n-1) edges.
- There are $\leq 2^{k+1}$-1 nodes in the recursion tree, so $O^*(2^k)$. ▪

Not 2^n, not n^k, but 2^k, which is much more efficient!

With the two claims, the theorem is proven.

Bounded Search Trees: Improving Vertex Cover – Branch on Vertices

Branch on vertices – no neighbors

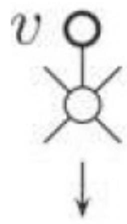


Q. Why if there is an optimal solution, there is one that does not include v?

Proof. v is never included in a minimal vertex cover: suppose it is; remove it, and the result is still a valid vertex cover, but of smaller size.

Bounded Search Trees: Improving Vertex Cover

Branch on vertices – 1 neighbor

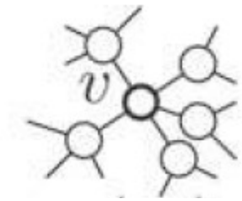


Q. Why if there is an optimal solution, there is also one including this neighbor?

Proof. Every optimal solution needs to include either v or its neighbor; if it includes v, a valid cover of equal size can be constructed by removing v and adding the neighbor.

Bounded Search Trees: Improving Vertex Cover

Branch on vertices – many neighbors



remove v;
k-1 remain

remove v and all
its d neighbors;
k-d remain

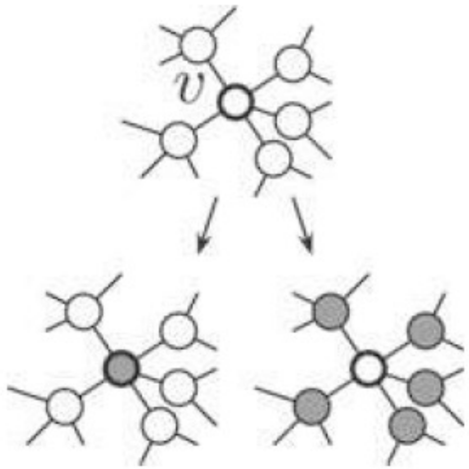
Q. Why if there is an optimal solution, there is also one like one of these?

Proof. Every optimal solution needs to cover all edges incident to v. If v is included it matches left. If v is not included, all neighbors need to be (right).

Bounded Search Trees: Improving Vertex Cover

Branch on vertices – many neighbors

remove v;
k-1 remain



remove v and all
its d neighbors;
k-d remain

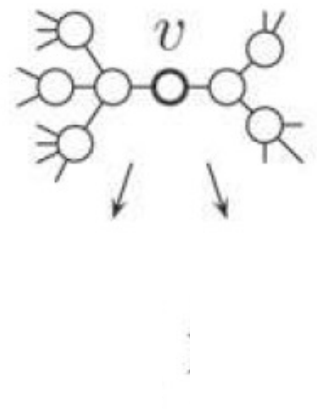
Q. When do we get the worst-case runtime?

A. For the smallest d, i.e. d=2.
T(k) = T(k-1) + T(k-2) + O(n)
Can we do better if d=2?

TU Delft

Bounded Search Trees: Improving Vertex Cover

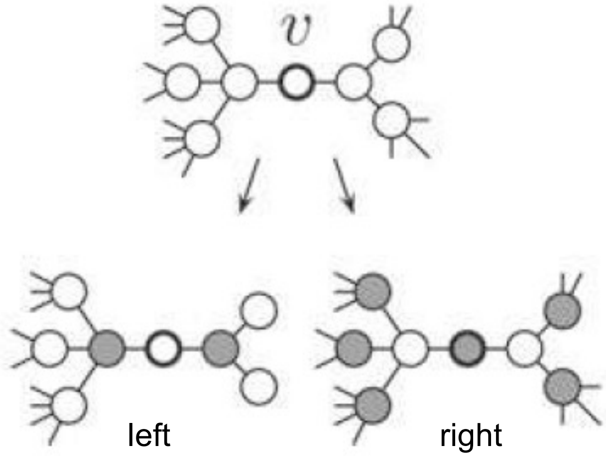
Branch on vertices – two neighbors



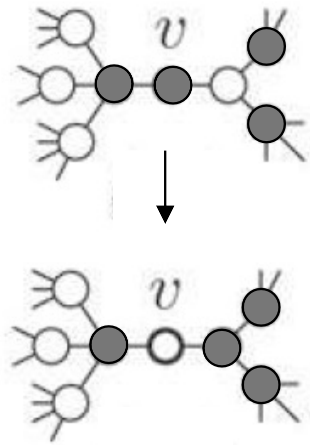
Q. Why if there is an optimal solution, there is also one like one of these?

Bounded Search Trees: Improving Vertex Cover

Branch on vertices – two neighbors



v and one neighbor?



Q. Why if there is an optimal solution, there is also one like one of these?

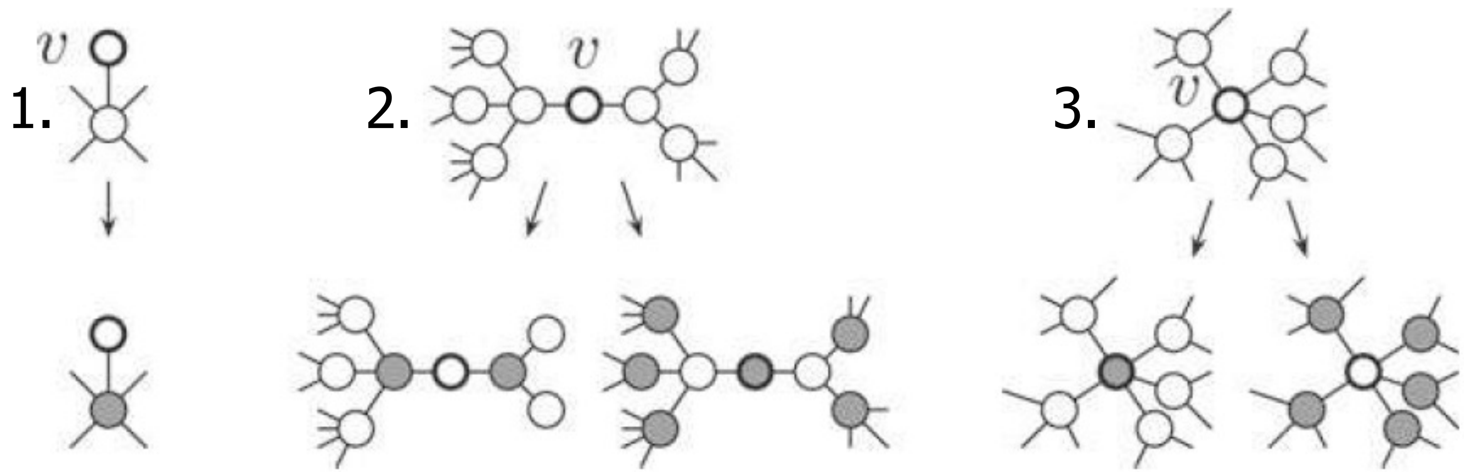
Proof. Every optimal solution needs to cover all edges incident to v.

- If v is not included its neighbors must be, and this matches *left*.
- Otherwise, if v is included
    - and one of its neighbors is, then it is possible to include the other neighbor instead of v, case *left* again.
    - or none of its neighbors, and thus neighbors of neighbors are: *right*.

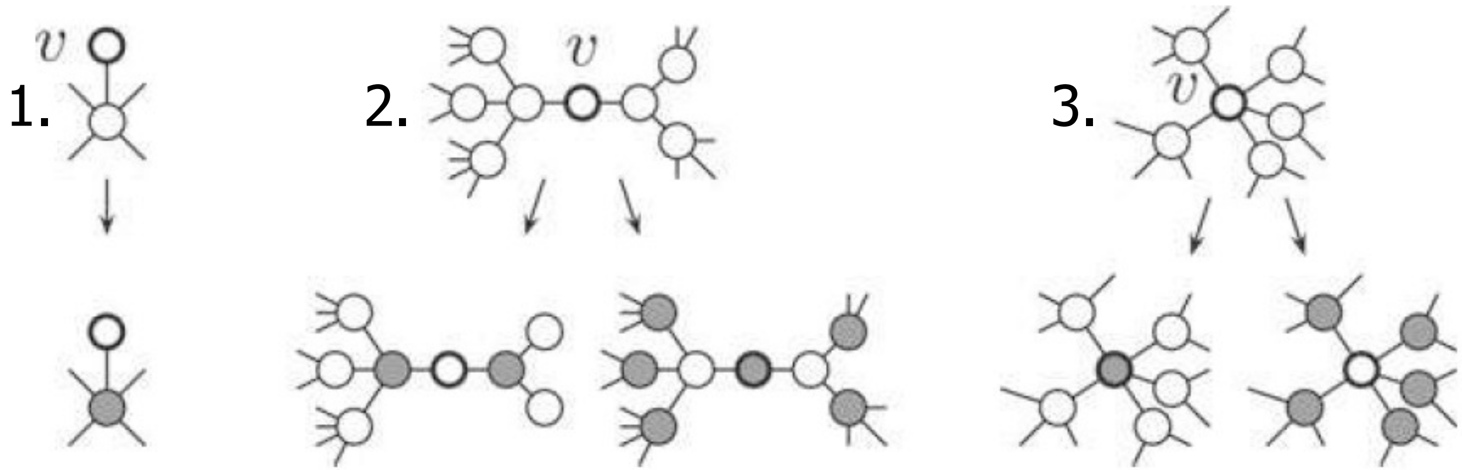
Bounded Search Trees: Improving Vertex Cover

More detailed analysis of sub-cases (try most efficient first, so:)

1. If there is a vertex of degree 0, remove it.
2. If there is a vertex of degree 1, put neighbor in cover.
3. Else, if there is a vertex v of degree 2
    1. put 2 neighbors of v in cover, or
    2. put v in cover together with all neighbors of 2 neighbors
4. Else, if there is a vertex v of degree 3 or more:
    1. put v in cover, or
    2. put all neighbors of v in cover



Bounded Search Trees: Improving Vertex Cover



Analyze the worst case

1. only one subproblem of size k-1 (thus linear in k)
2. two subproblems: one of size k-2 and one of size at most k-3
3. two subproblems: one of size k-1 and one of size at most k-3

So, case 3 is the worst case…

Recurrence relation describing the run time T(k)

$T(k) \leq T(k-1) + T(k-3) + O(n+m)$

Bounded Search Trees: Improving Vertex Cover

Recurrence relation describing the run time T(k):
$T(k) \leq T(k-1) + T(k-3) + O(n+m)$

O(n+m) for updating graph and selecting vertex with three or more neighbors

Evaluate this recurrence to compare to $O^*(2^k)$
Idea. Find α such that $T(k) \leq O^*(\alpha^k)$
1. ignore polynomial part of O(n+m)
2. rewrite T(k) to α^k
3. so solve α from $\alpha^k = \alpha^{k-1} + \alpha^{k-3}$, or equivalently (divide by α^{k-3})
4. $\alpha^3 = \alpha^2 + 1$
5. use Matlab to find out that $\alpha = 5^{1/4} \approx 1.47$, so T(k) is $O^*(1.47^k)$

Later improvements
By even more careful analysis of special (sub)cases:
- $O^*(1.32^k)$ in 1998 [1]
- $O^*(1.285^k)$ in 2001 [2]

[1] R. Balasubramanian, M. R. Fellows, and V. Raman, An improved fixed parameter algorithm for vertex cover, Inform. Process. Lett. 65 (1998), 163–168.
[2] J. Chen, I. Kanj, W. Jia, Vertex cover: further observations and further improvements, Journal of Algorithms 41 (2001) 280–301

Vertex cover: applications

- Reconfigurable arrays: where to place spare parts of a chip?
- Networks:
  - where to place packet filters?
  - where to place converters for wave-length-division multiplexing (to combine multiple signals on fiber-optic media)
- Wireless sensor: minimal set of sensor devices necessary to cover entire area
- Finding SNPs (Single Nucleotide Polymorphism, mutations in DNA) [1]

[1] G. Lancia, V. Afna, S. Istrail, L. Lippert, and R. Schwartz, *SNPs Problems, Complexity and Algorithms*, ESA 2002, LNCS 2161, pp. 182-193, 2001.

TU Delft

Fixed parameter tractable

Def. A problem of size n is *fixed parameter tractable (FPT) with respect to parameter k* if it can be solved in $f(k) \cdot p(n)$ time, where

- f is a (usually exponential) function depending only on the parameter k
- p is a polynomial function

To distinguish between behavior:

- $O(f(k) \cdot p(n))$
- $\Omega(n^{f(k)})$

Parameterized complexity was first described by Downey & Fellows (1999).

Q. Given the previous rules, what is f(k) for vertex cover FPT?

A. $f(k)=1.47^k$

p(n) is the time we need select an edge, and an upper bound on the time for preprocessing

1-Slide Summary on Search Trees

- root represents the complete problem
- children are smaller subproblems: alternatives for single decision (mutually exclusive, all need to be investigated)
- the smaller subcases the better (using worst case analysis)!
  - we may have different types of branches at various places in the tree (first do easier cases, e.g., start with vertices with a single neighbor)
- expressed as recursive algorithm
- prove correctness
- resolve recurrence of worst case by assuming exponential runtime

"Examples":
- **independent set**: special cases of 0, 1 and 2 neighbors
- **3 SAT**: 3 branches: $L_1=1$, $L_1=0$ and $L_2=1$, $L_1=L_2=0$ and $L_3=1$
- **vertex cover**: $O^*(2^k)$ and $O^*(1.47^k)$: special cases of 0 and 1; case with degree 2, but worst case is with degree 3

Fixed parameter tractable if runtime bounded by $O(f(k) \cdot p(n))$

TU Delft

Study Advice

Please read (about 15 pages)

1. Section 10.1 for BST from Jon Kleinberg and Eva Tardos, *Algorithm Design*, 2006.
2. Gerhard Woeginger, Exact algorithms for NP-hard problems: A survey, *Combinatorial Optimization*, LNCS 3570, pp 187-207, 2003: Section 1-2 background, Section 4 for BST
3. Falk Hueffner, Rold Niedermeier and Sebastian Wernicke, Techniques for Practical Fixed-Parameter Algorithms, *The Computer Journal*, 51(1):7–25, 2008: Section 1 background, and Section 3 for BST

Lab assignment 1 is about search trees

Homework assignments

- General idea of search trees
- Cluster editing

TU Delft