

Optimization In Graphs

Subset Sum

Input: $x_1, x_2, \dots, x_n \in \mathbb{N}$
 $T \in \mathbb{N}$ "target"

Goal: Find a subset of x_i 's that sum to T

Theorem

poly time algo for subset sum

poly-time $\Downarrow$ algo for SAT

cached-subset-sum(i, S)

```

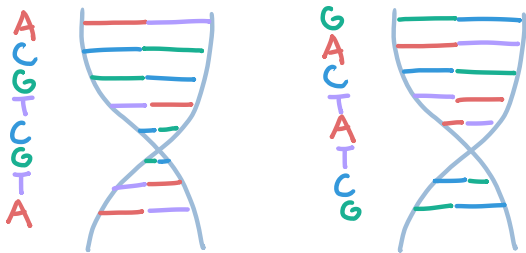
/* We assume there is an  $n \times T$  array denoted  $A[1..n, 1..T]$  (“A” for
“Answer”) allocated in global memory, with all entries initially set to
null.
*/

```

1. If $S = 0$ then return true.
2. If $S < 0$ then return false.
3. If $i > n$ then return false.
4. If $A[i, S]$ is null:
 - A. $A[i, S] \leftarrow \text{subset-sum}(i + 1, S) \vee \text{subset-sum}(i + 1, S - x_1)$.
5. Return $A[i, S]$.

$$A_j = \underbrace{0 \ 1 \ 0 \ 1 \ 0}_{\text{m slots } S_{or} \text{ each clause}} \underbrace{0 \ 1 \ 0}_{\text{n slots } S_{or} \text{ each variable}} \in \{0, 1\}^{mn}$$

Edit distance



insertions, deletions, substitutions
to convert x into y

$X =$ ^{del} ABCDEFG
 $Y =$ ABCEFGD_{ins} $\Rightarrow$ Edit. distance = 2

$$= m \cdot n \cdot O(1) = O(mn)$$

Step 1 Recursive spec

Six strings $x[1 \sim m], y[1 \sim n]$
 $\text{edit}(i, j) =$ minimum edit distance
 between $x[i, i+1 \sim m]$ and $y[j, j+1 \sim n]$

Step 2 implement

$$\text{edit}(i,j) = \begin{cases} 0 & i \leq m, j \leq n \\ 1 + \text{edit}(i+1, j) & i > m, j \leq n \\ 1 + \text{edit}(i, j+1) & i \leq m, j > n \\ 1 + \text{edit}(i+1, j+1) & i > m, j > n \\ \min \begin{cases} 1 + \text{edit}(i+1, j) \\ 1 + \text{edit}(i, j+1) \\ 1 + \text{edit}(i+1, j+1) \end{cases} & \text{when } x[i] \neq y[j] \\ \min \begin{cases} 1 + \text{edit}(i+1, j) \\ 1 + \text{edit}(i, j+1) \\ \text{edit}(i+1, j+1) \end{cases} & \text{when } x[i] = y[j] \end{cases}$$

Previously:

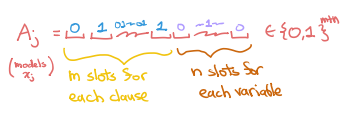
Subset Sum

Input: $x_1, x_2, \dots, x_n \in \mathbb{N}$
 $T \in \mathbb{N}$ "target"

Goal: Find a subset of x_i 's that sum to T

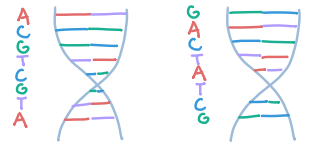
```
cached-subset-sum(i, S)
/* We assume there is an n x T array denoted A[1..n, 1..T] ("A" for "Answer") allocated in global memory, with all entries initially set to null. */
1. If S = 0 then return true.
2. If S < 0 then return false.
3. If i > n then return false.
4. If A[i, S] is null:
   A. A[i, S] ← subset-sum(i + 1, S) ∨ subset-sum(i + 1, S - x_i).
5. Return A[i, S].
```

Theorem
poly time algo for subset sum
↓
poly-time algo for SAT



no polynomial algo

Edit distance



insertions, deletions, substitutions to convert x into y

$x = ABCDEF G$
 $y = ABCEFGD$
 $\Rightarrow$ Edit. distance = 2

$(\# \text{ subproblems}) \times (\text{time per subproblem})$
 $= m \cdot n \cdot O(1) = O(mn)$

Step 1 Recursive spec

Two strings $x[1..m], y[1..n]$
 $\text{edit}(i, j) =$ minimum edit distance between $x[i..m]$ and $y[j..n]$

Step 2 implement

$\text{edit}(i, j) =$

0

$1 + \text{edit}(i+1, j)$

$1 + \text{edit}(i, j+1)$

$\min \begin{cases} 1 + \text{edit}(i+1, j) \\ 1 + \text{edit}(i, j+1) \\ 1 + \text{edit}(i+1, j+1) \end{cases}$

$\min \begin{cases} 1 + \text{edit}(i+1, j) \\ 1 + \text{edit}(i, j+1) \\ \text{edit}(i+1, j+1) \end{cases}$

$i > m, j > n$
 $j > n$
 $i > m$
when $x[i] \neq y[j]$
when $x[i] = y[j]$

yes polynomial algo

why?

(edit distance looks more complicated than subset sum IMO)

Today: similar themes for graph problems

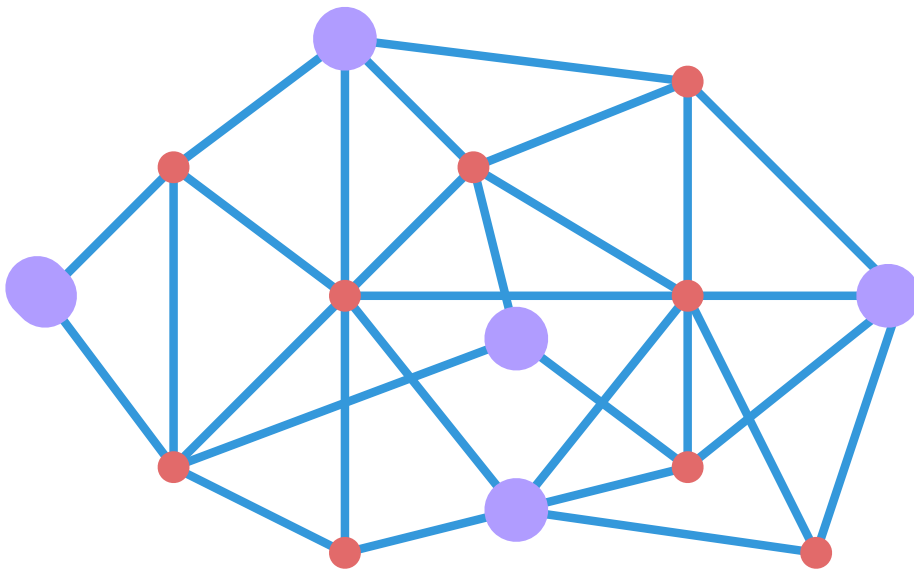
- ① max independent set
- ② graph coloring
- ③ dominating set (time permitting)

Independent set

Independent set

Input: undirected $G=(V,E)$, $k \in \mathbb{N}$

$S \subseteq V$ is independent if no $u,v \in S$ are connected by an edge.

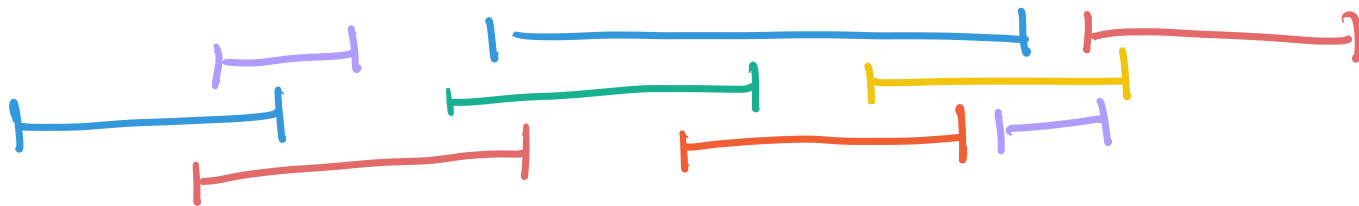


Goal: max cardinality independent set

(also have weighted version, decision version)

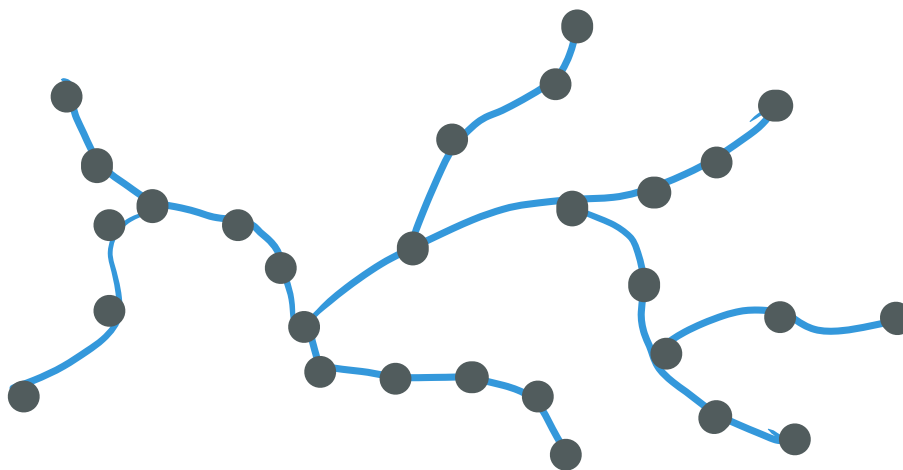
Special cases

① Intervals (w/ weights)



"independent" = non-overlapping

② Trees



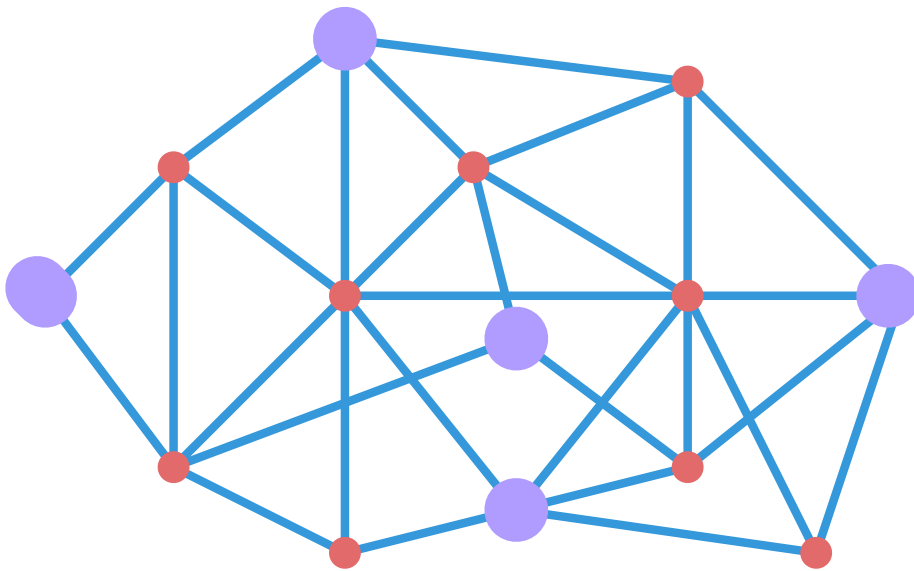
Independent set in { graphs, intervals, trees }

hard easy abstain

Independent set

Input: undirected $G=(V,E)$, $k \in \mathbb{N}$

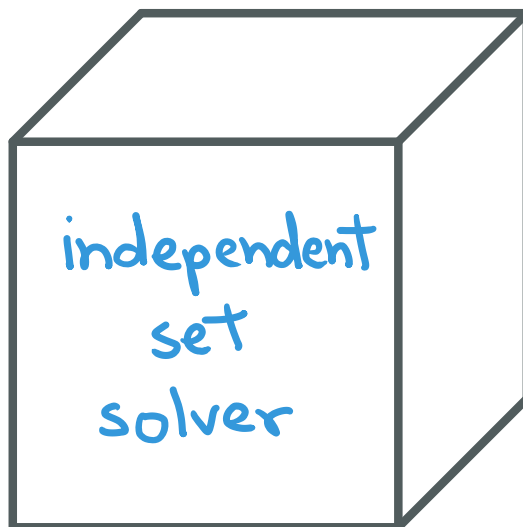
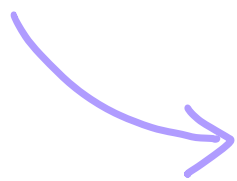
$S \subseteq V$ is independent if no $u,v \in S$ are connected by an edge.



Goal: max cardinality independent set

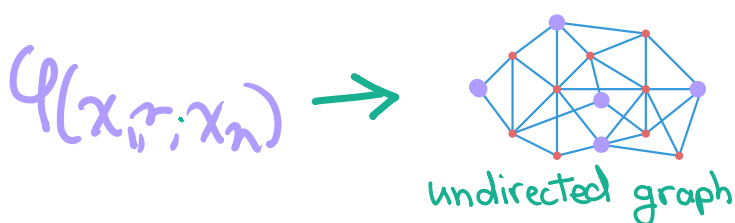
(also have weighted version, decision version)

$$\varphi(x_1, x_2, x_3) = (x_1 \vee x_2 \vee x_3) \wedge (\bar{x}_1 \vee \bar{x}_2 \vee x_3) \wedge \dots$$



YES, NO

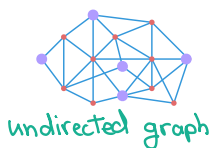
Goal: transform



s.t. size of max independent set

informs satisfiability of φ

we will build



s.t.

(φ is satisfiable)

$\Leftrightarrow$

(there is indep. set of size k)

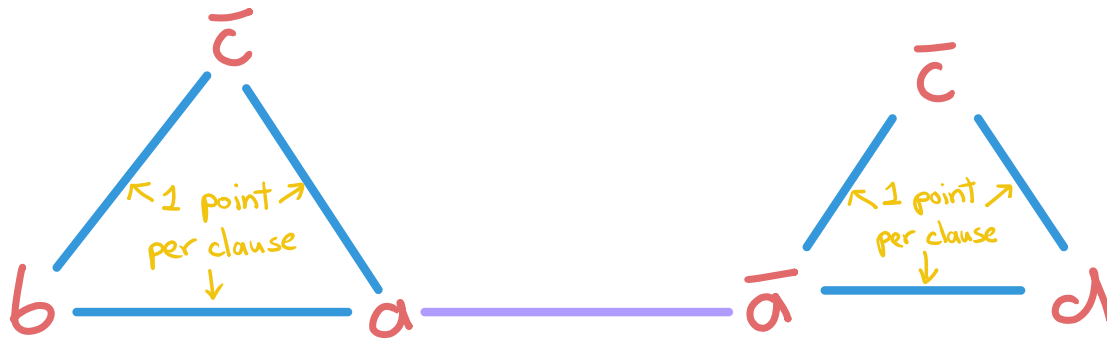
(for some k TBD)

SAT variables $\longleftrightarrow$ graph vertices

clauses $\longleftrightarrow$ Triangles

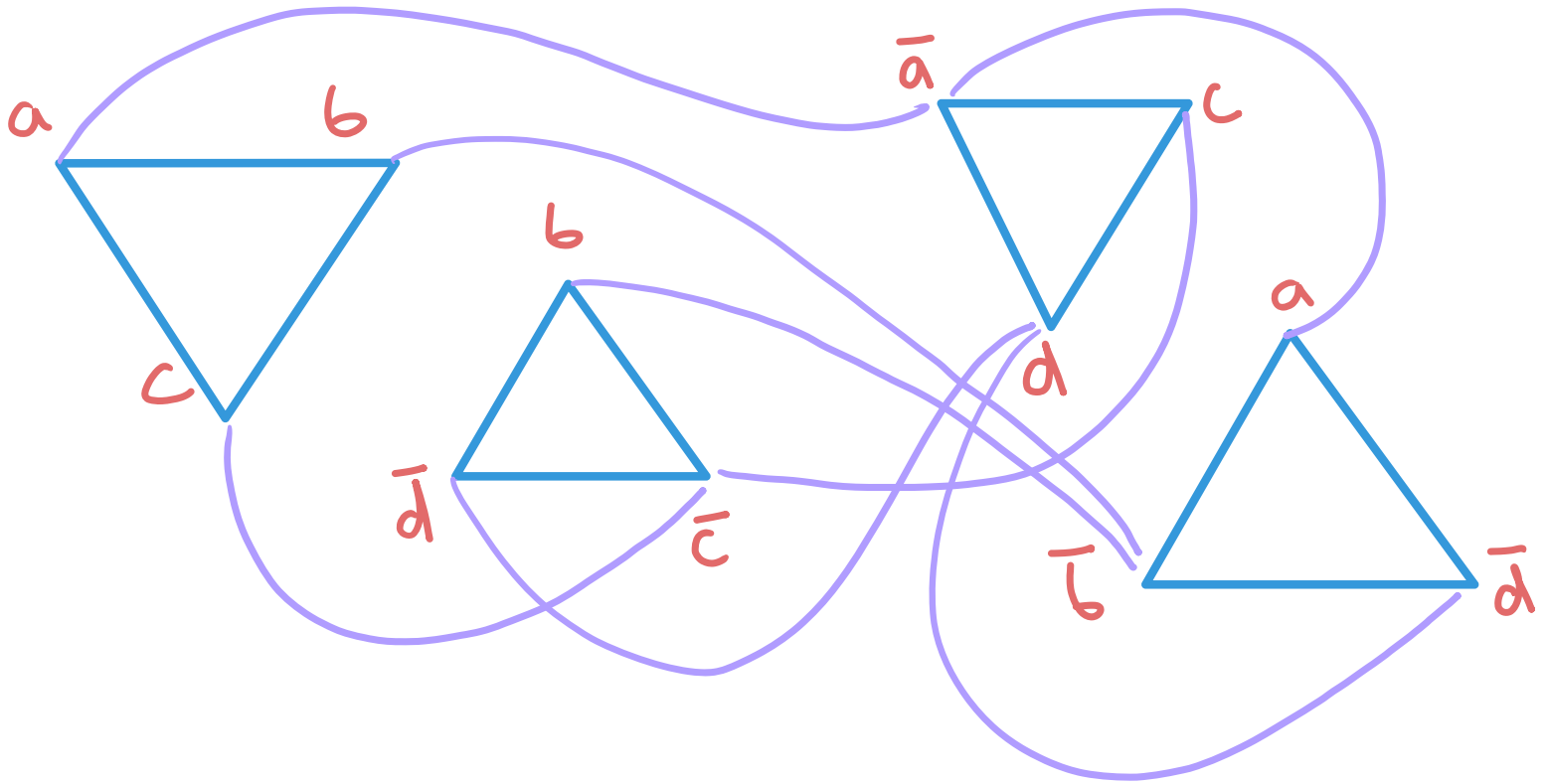
$$a \vee b \vee \bar{c}$$

$$\bar{a} \vee \bar{c} \vee d$$



cannot take
 $a=1$ and $a=0$

$$(a \vee b \vee c) \wedge (b \vee \bar{c} \vee \bar{d}) \wedge (\bar{a} \vee c \vee d) \wedge (a \vee \bar{b} \vee \bar{d})$$



m clauses $\rightarrow m$ cliques

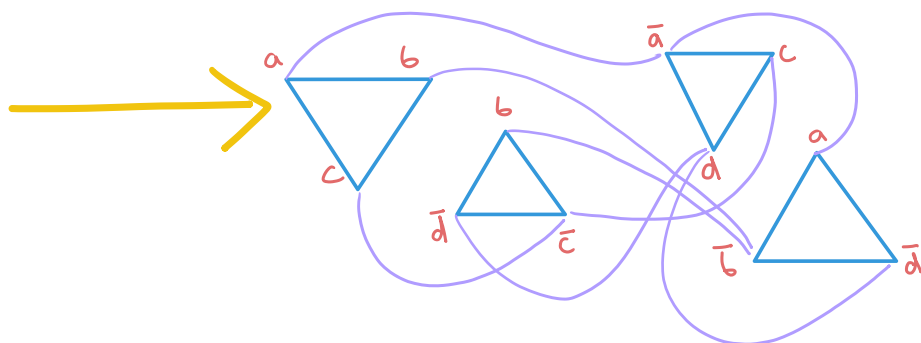
(in $\text{poly}(m, n)$
size graph)

$\varphi(x_1, \dots, x_n) =$

$(x_1 \vee \bar{x}_2)$

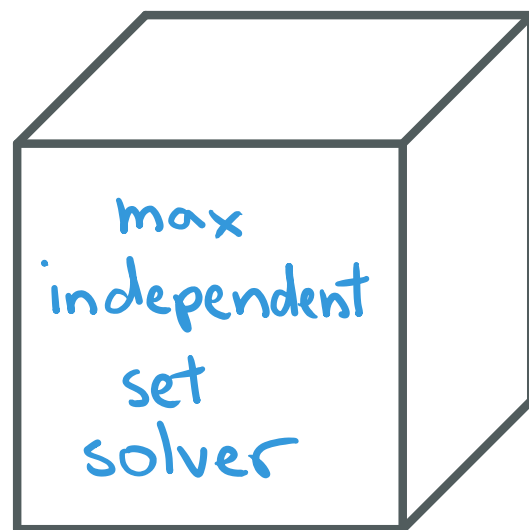
$\wedge (x_2 \vee x_3 \vee \bar{x}_4)$

$\wedge \dots$



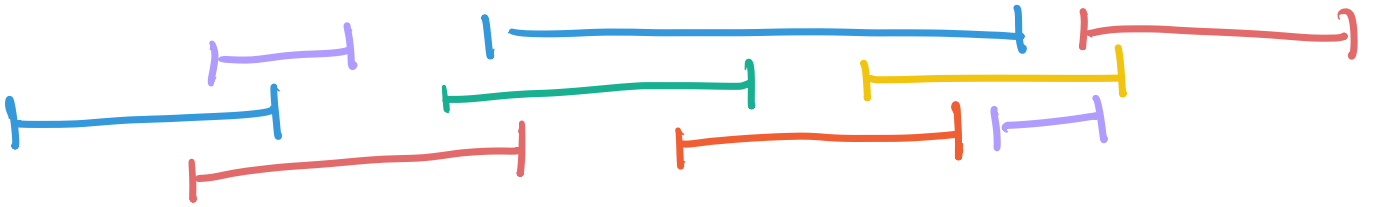
yes, no

if
is polytime, then
entire algorithm is
polytime.



Thus: max indep. set is at least
as hard as SAT

Independent set w/ weighted intervals



"independent" = non-overlapping

Step 1 Recursive spec

for a set of weighted intervals $\mathcal{I}$

(Max Weight
Independent Set)

MWIS-Intervals($\mathcal{I}$) =

max weight of any

subset of intervals of $\mathcal{I}$

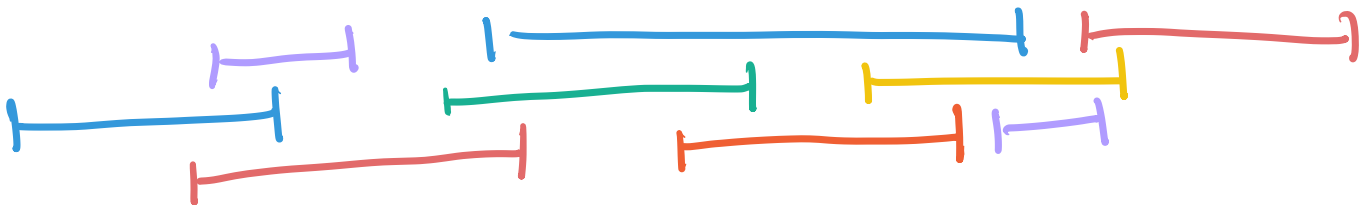
Step 2 Implement

$\text{MWIS-intervals}(\mathcal{I}, w)$ = the weight of the maximum weight independent subset of intervals of $\mathcal{I}$.

$\text{MWIS-intervals}(\mathcal{I}, w : \mathcal{I} \rightarrow \mathbb{R}_{>0})$

1. If $\mathcal{I} = \emptyset$ then return 0
2. Let $I \in \mathcal{I}$ be any interval
3. Let $\mathcal{I}' = \{J \in \mathcal{I} : J \cap I = \emptyset\}$ *// = set of intervals disjoint from I*
4. return the maximum of
 - /* The MWIS among those excluding I. */*
A. $\text{MWIS-intervals}(\mathcal{I} \setminus \{I\}, w)$
 - /* The MWIS among those including I. */*
B. $w(I) + \text{MWIS-intervals}(\mathcal{I}', w)$

Question: how to make $\uparrow$ faster?



Step 2 Implement

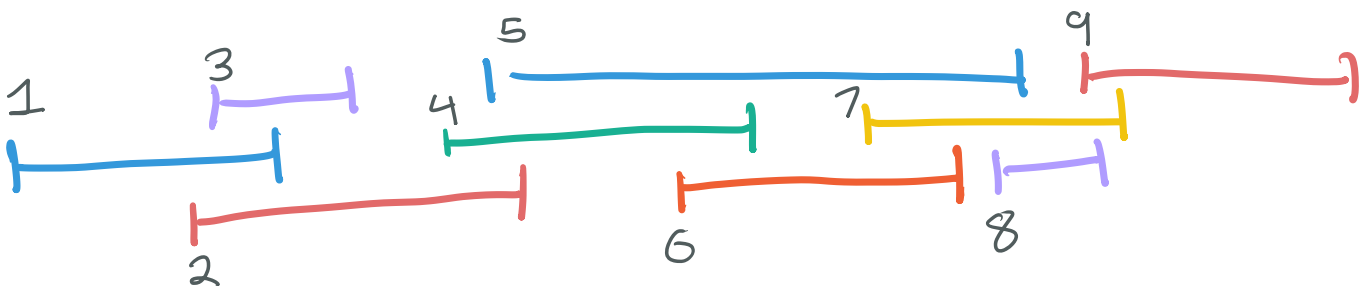
MWIS-intervals($\mathcal{I}$, $w : \mathcal{I} \rightarrow \mathbb{R}_{>0}$)

1. If $\mathcal{I} = \emptyset$ then return 0
2. Let $I \in \mathcal{I}$ be any interval
3. Let $\mathcal{I}' = \{J \in \mathcal{I} : J \cap I = \emptyset\}$ *// = set of intervals disjoint from I*
4. return the maximum of
 - /* The MWIS among those excluding I. */*
A. $\text{MWIS-intervals}(\mathcal{I} \setminus \{I\}, w)$
 - /* The MWIS among those including I. */*
B. $w(I) + \text{MWIS-intervals}(\mathcal{I}', w)$

Idea: make problem sequential
sort

$$\underline{\mathcal{I}} = \{I_1 = [a_1, b_1], I_2 = [a_2, b_2], \dots, I_n = [a_n, b_n]\}$$

increasing in left endpoint a_i



Same algorithm, but ordered:

MWIS-intervals($\mathcal{I} = \{I_1 = [a_1, b_1], \dots, I_n = [a_n, b_n]\}$, $w : \mathcal{I} \rightarrow \mathbb{R}_{>0}$)

/ We assume $I_1 = [a_1, b_1], \dots, I_n = [a_n, b_n]$ is sorted in increasing order of a_i . */*

1. If $n = 0$ then return 0

2. Let $i > 1$ be the first index such that $a_i > b_1$

// I_i is the first interval disjoint from I_1 , and $\{I_i, \dots, I_n\}$ = set of intervals disjoint from I_1

3. return the maximum of

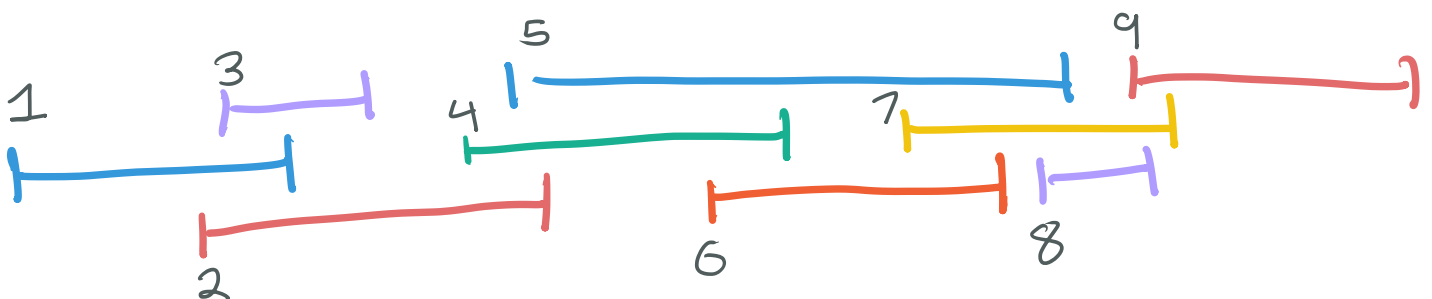
/ The MWIS among those excluding I_1 */*

A. MWIS-intervals($\{I_2, \dots, I_n\}, w$)

/ The MWIS among those including I_1 */*

B. $w(I_1) + \text{MWIS-intervals}(\{I_i, \dots, I_n\}, w)$

↑
suffix of ordered list!
n subproblems!



Dynamic Programming

① Spec

MWIS-intervals(i) = the maximum weight of any independent set of intervals among $\{I_i, I_{i+1}, \dots, I_n\}$.

② Code

$$\text{MWIS-intervals}(i) = \begin{cases} 0 & \text{if } i > n, \\ \max \left\{ \begin{array}{l} \text{MWIS-intervals}(i+1), \\ w(I_i) + \text{MWIS-intervals}(j(i)) \end{array} \right\} & \text{otherwise,} \end{cases}$$

where $j(i)$ is the first index j such that $a_j > b_i$.

Running time

(# subproblems) $\times$ (time per subproblem)

$$= n \times O(n) = O(n^2)$$

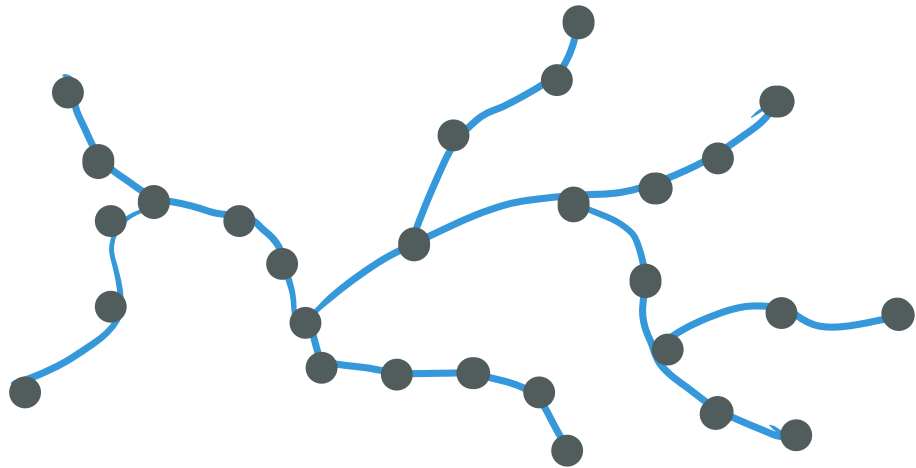
↑ scan for j ?

Better?

Binary search for $j \Rightarrow O(n \log n)$
time

also, $O(n)$ space

② Trees

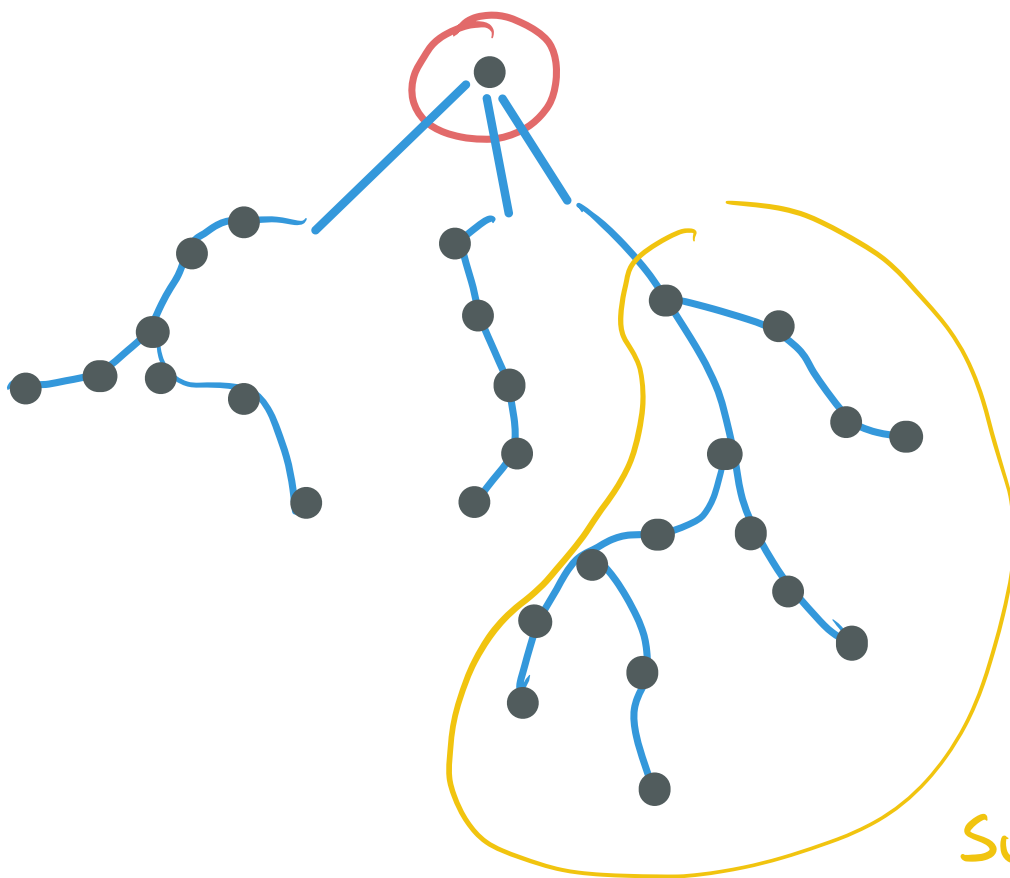
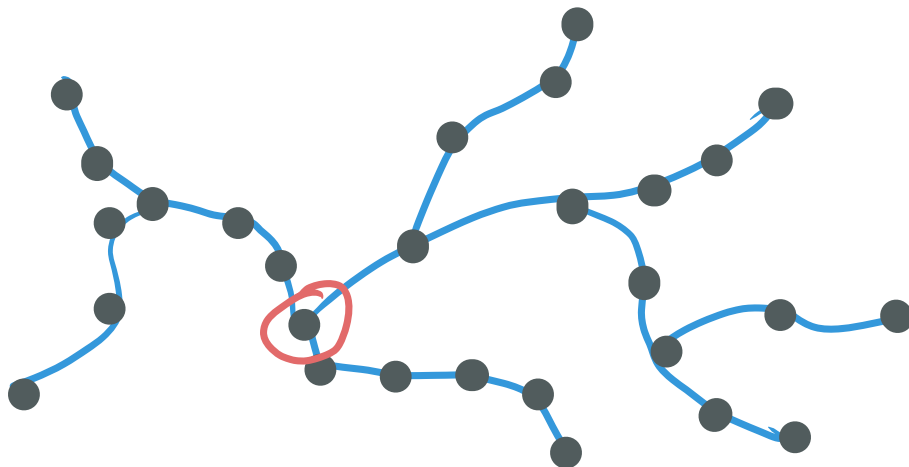


hard as graphs? easy as intervals?



how to order
subproblems?

Root it



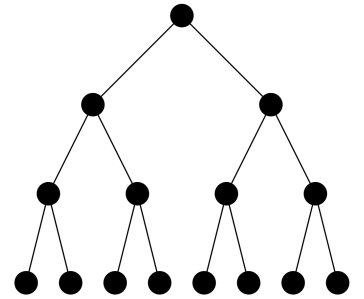
Subtree

① Recursive Spec

$MWIS(v)$ = the maximum weight of any independent set in the subtree rooted at v .

② Implement

$MWIS(v)$ = maximum of



① (exclude v)

sum of $MWIS(x)$ over children x of v

② (include v)

weight of v + sum of



over children x of v

We really want $MWIS(x)$ restricted to independent sets excluding x

Revisit ① Recursive Spec

↙ new boolean flag

MWIS(v , exclude-root): If exclude-root = true, then the maximum weight independent set in the subtree rooted at v among those excluding v . If exclude-root = false, then the maximum weight independent set in the subtree rooted at v (which may or may not include v).

② Implement

MWIS(v , exclude-root)

1. If exclude-root

/ Don't include v in the independent set */*

A. Return the sum of MWIS(x , false) over all children x of v

2. Otherwise return the maximum of:

/ Don't include v in the independent set */*

A. The sum of MWIS(x , false) over all children x of v .

/ Include v in the independent set */*

B. The weight of v plus the sum of MWIS(x , true) over all children x of v

Note: stronger recursion is
easier to implement

Running time

$$(\# \text{ subproblems}) \times (\text{time per subproblem})$$

$$= n \times O(n) = O(n^2)$$

↑
up to n children per node

Better?

total # children in tree is $< n$

Running time

$$\sum_{\text{node } v} (\text{time for MWIS}(v)) = \sum_{\text{node } v} (\# \text{ children of } v)$$

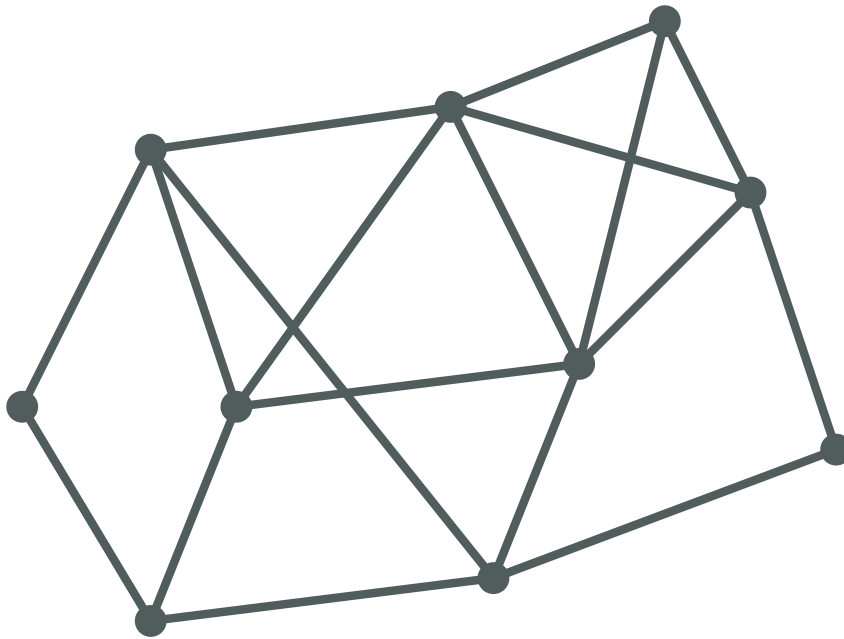
$$= O(\text{total \# children})$$

$$= O(n)$$

FASTER than intervals!

also, $O(n)$ space

Graph Coloring



Goal: color vertices so that no adjacent vertices have same color
(w/ minimum # colors)

decision: does k colors suffice?

* In "k-coloring",
 k is a fixed constant

Theorem

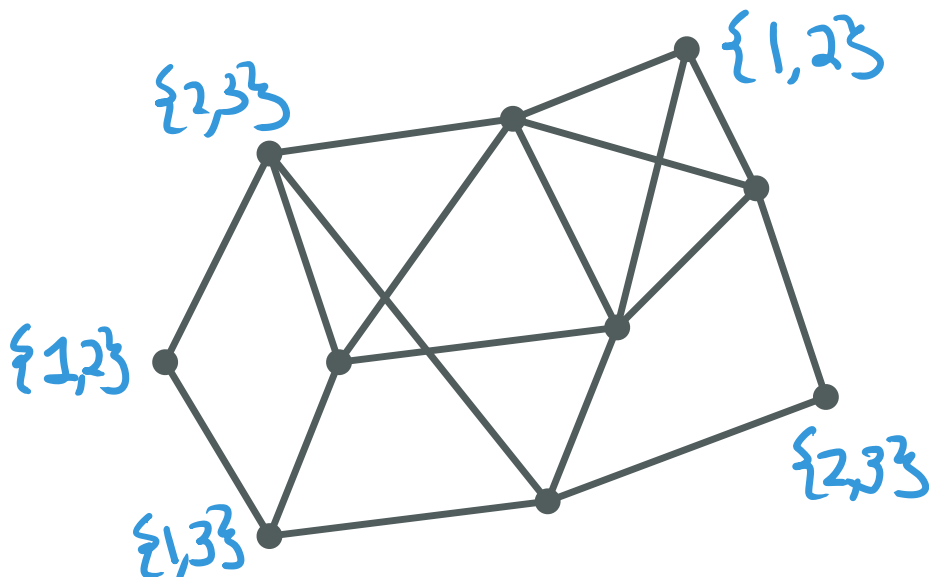
3-coloring is as hard as SAT

High level approach

3SAT $\rightarrow$ "subset" 3-coloring $\rightarrow$ 3-coloring

Subset 3-coloring

For each vertex v ,
subset $S_v \subseteq \{1, 2, 3\}$ of allowed colors

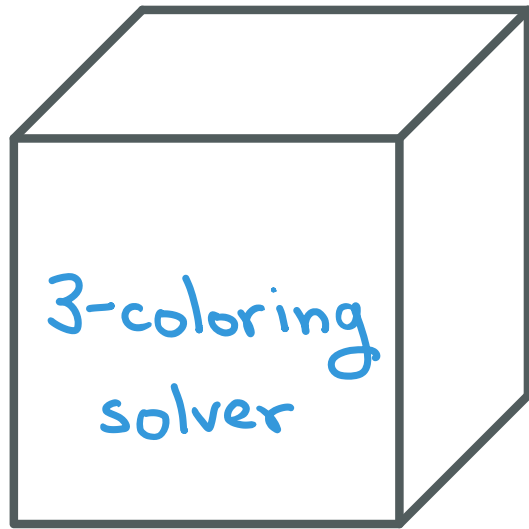
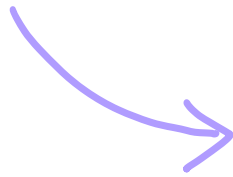


(unmarked)
= $\{1,2,3\}$

Theorem

3-coloring is as hard as SAT

$$\varphi(x_1, x_2, x_3) = (x_1 \vee x_2 \vee x_3) \wedge (\bar{x}_1 \vee \bar{x}_2 \vee x_4) \wedge \dots$$



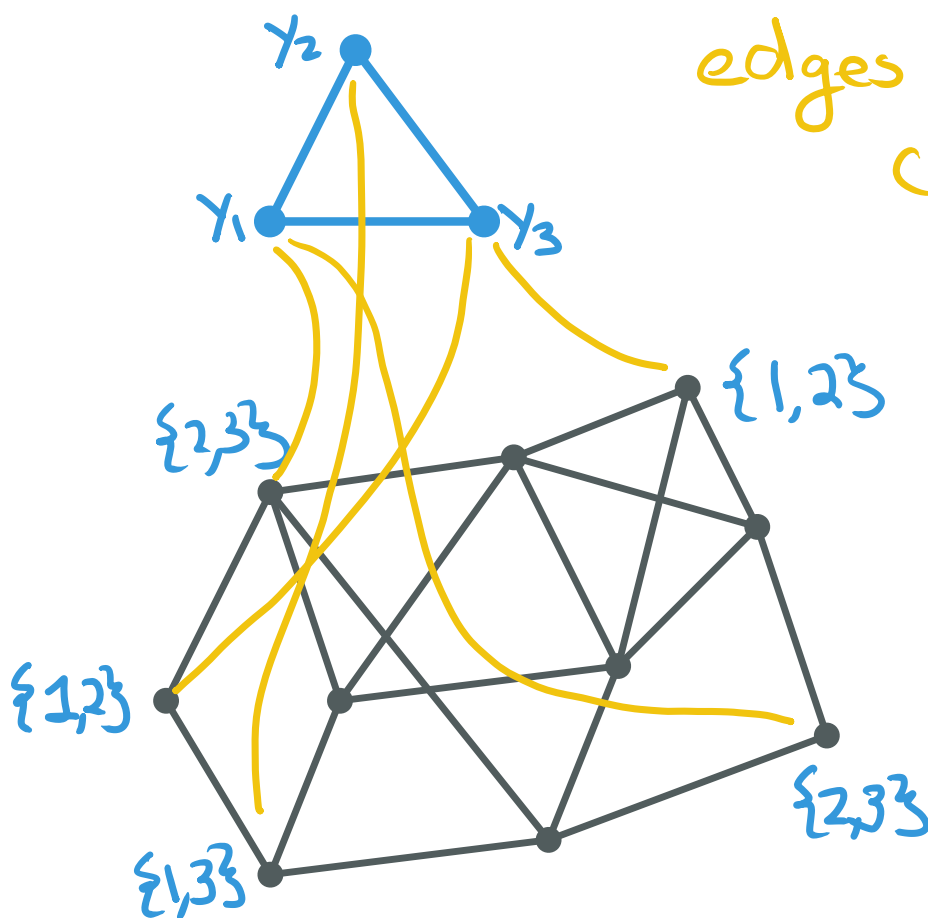
YES, NO



Subset
3-coloring $\rightarrow$ 3-coloring

(3-coloring solves subset 3-coloring)

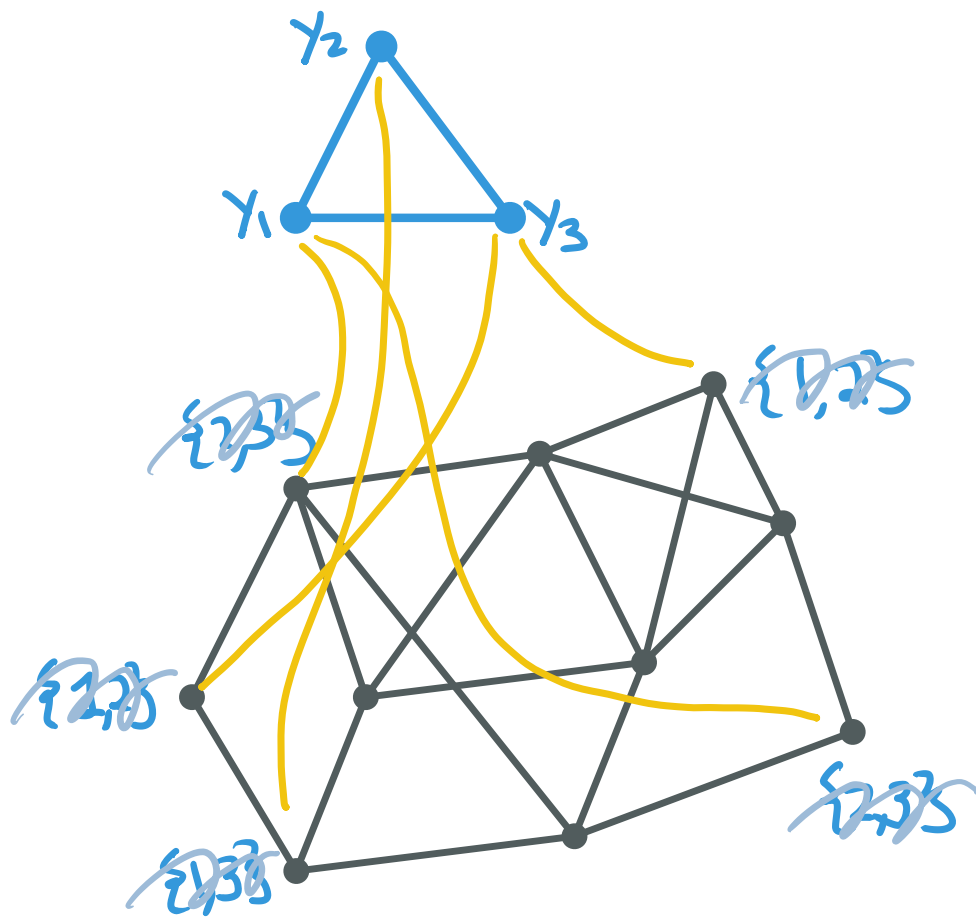
Idea "color pakette"



edges forbid
colors

(unmarked)
(= $\{1,2,3\}$)

Given 3-coloring of



wlog:

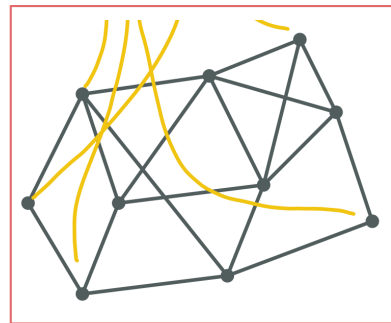
y_1 has color 1

y_2 has color 2

y_3 has color 3

(unmarked)
= $\{1,2,3\}$

the coloring on $\longrightarrow$
will satisfy the subset const.



Theorem

3-coloring is as hard as SAT

High level approach

3SAT $\rightarrow$ "subset" 3-coloring $\rightarrow$ 3-coloring

3SAT $\rightarrow$ Subset 3-coloring

Input: 3SAT $\mathcal{S}(x_1, \dots, x_n)$

Goal: instance of subset 3-coloring
that preserves feasibility

Convenient to name colors

"T" = "TRUE"

"F" = "FALSE"

"N" = "Neither"

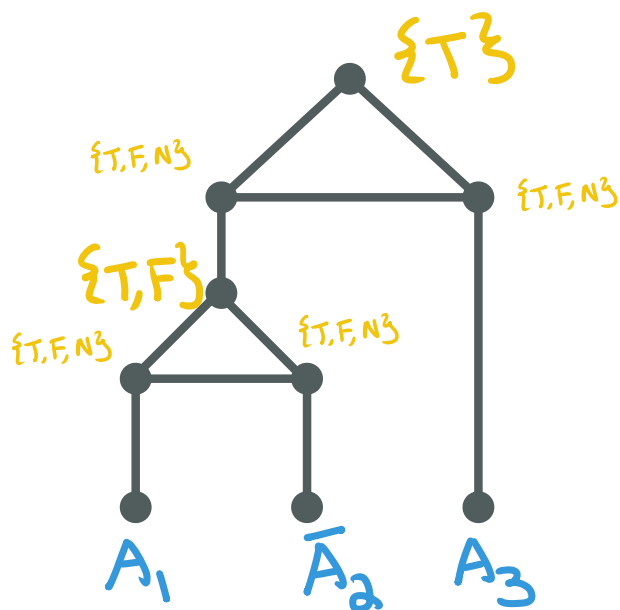
① variables



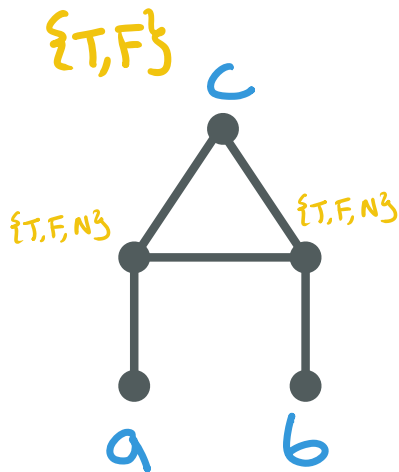
interpretation: $x_j = A_j.\text{color}$
 $\bar{x}_j = \bar{A}_j.\text{color}$

② clauses

$x_1 \vee \bar{x}_2 \vee x_3 \Rightarrow$



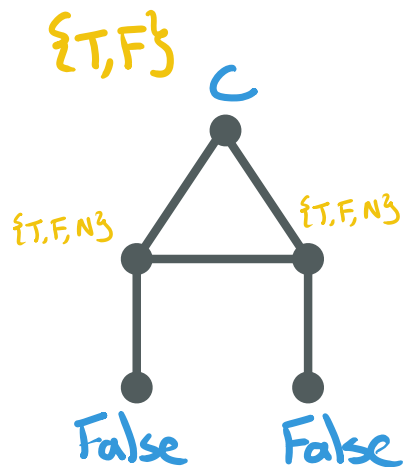
Idea: "or-gadget"



Lemma

$c.\text{color} = \text{true}$
possible iff
 $a \vee b = \text{False}$

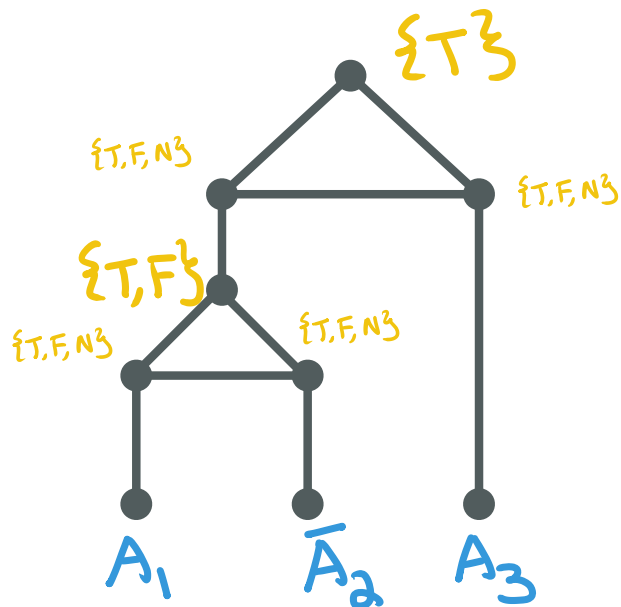
Proof



forces $c = \text{False}$

$x_1 \vee \bar{x}_2 \vee x_3 \Rightarrow$

combines
2 $\vee$ -gadgets



① variables

x_j

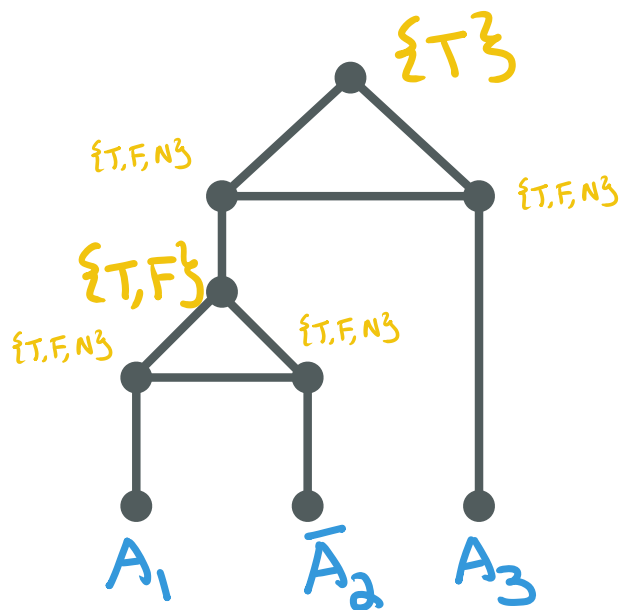
$\Rightarrow$



interpretation: $x_j = A_j \cdot \text{color}$
 $\bar{x}_j = \bar{A}_j \cdot \text{color}$

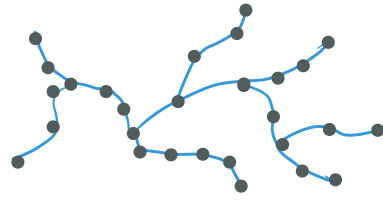
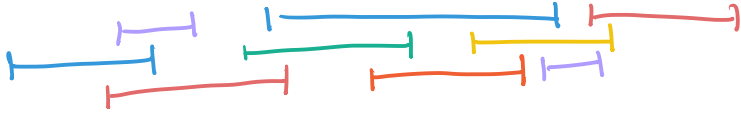
② clauses

$x_1 \vee \bar{x}_2 \vee x_3 \Rightarrow$



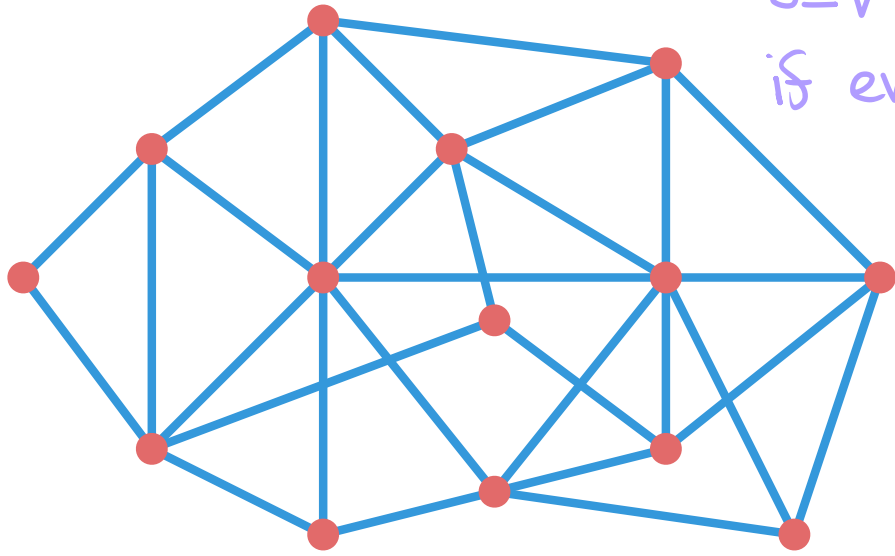
① + ② $\Rightarrow$ subset 3-coloring problem
that preserves satisfiability

what about intervals? trees?



these are PSO problems

Dominating set



$S \subseteq V$ is a dominating set
if every $v \in V$ is either

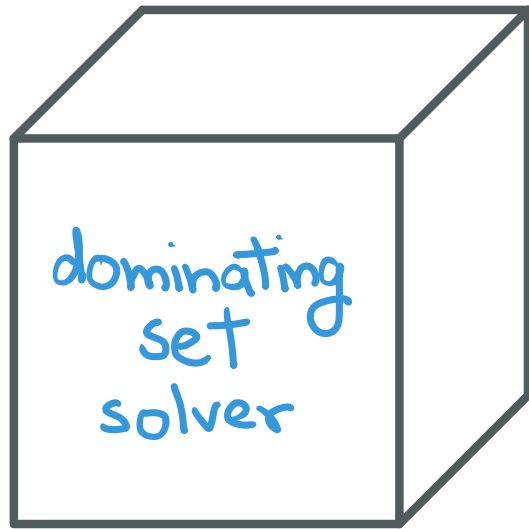
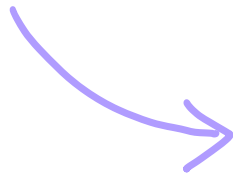
(a) in S

(b) a neighbor of
some $w \in S$

Theorem

dominating set is as hard as SAT

$$\varphi(x_1, \dots, x_n) = (x_1 \vee x_2 \vee x_3) \\ \wedge (\bar{x}_1 \vee \bar{x}_2 \vee x_4) \wedge \dots,$$



YES, NO



Chapter 5

Optimization in graphs

The two preceding discussions introduced some basic combinatorial problems, such as subset sum and edit distance, that lead to surprisingly different outcomes. Edit distance had a polynomial algorithm, while subset sum was revealed to be as general as SAT (up to polynomial time reductions), hence unlikely to yield a polynomial running time. What's the difference between subset sum and edit distance? Subset sum is not obviously harder than edit distance on first appearance. But for edit distance we were able to identify subproblems that decompose the initial problem into polynomially many subproblems; for subset sum, similar attempts came up short.

This lecture continues to investigate the subtle distinctions between problems that are polynomial time solvable, and those that resist our best algorithmic efforts. This time, all our problems are posed in undirected graphs. We will also consider special cases of undirected graphs to see if the additional structure changes the fundamental character of the problem. The problems introduced here are called the **independent set**, **(vertex) coloring**, and **dominating set** problems. They are all simple to define. Of these only the independent set problem will be presented completely. Vertex coloring and dominating set will be partly presented and partly delegated to exercises.

5.1 Independent set

The independent set problem is a graph problem defined as follows. Let $G = (V, E)$ be an undirected graph. Recall that graphs consist of vertices (V) and edges (E) that connect pairs of vertices. A set of vertices $S \subset V$ is **independent** if any two vertices $a, b \in S$ are not connected by an edge. The **maximum independent set problem** asks for the maximum cardinality independent set. In the decision version of the problem, the input consists of a graph $G = (V, E)$

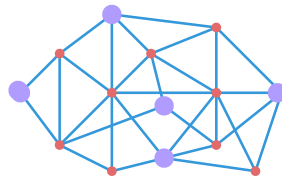
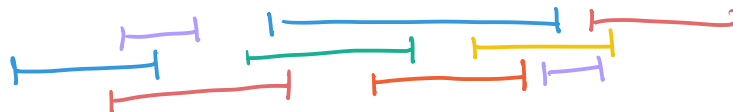


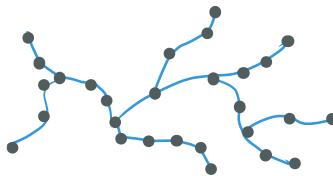
Figure 5.1: An independent set of size 5.

and an integer $k \in \mathbb{N}$, and asks if there exists an independent set S with size $|S| \geq k$. One can also consider the weighted version where we are also given positive vertex weights $w : V \rightarrow \mathbb{R}_{>0}$, and the goal is to choose the maximum weight independent set (or decide if the maximum weight is at least an input parameter T).

There are many special cases of independent set that are useful. We will discuss two of them. The first is for *intervals on a line*. The input consists of n intervals $\mathcal{I} = \{I_1 = [a_1, b_1], \dots, I_n = [a_n, b_n]\}$, with positive edge weights $w : \mathcal{I} \rightarrow \mathbb{R}_{>0}$. We say that a set of intervals $S \subset \mathcal{I}$ is independent if they are pairwise disjoint; i.e., $I \cap J = \emptyset$ for every two intervals $I, J \in S$. This is the same as the graph definition if we identify each interval as a vertex, and have an edge between every two intervals that overlap.



The second example is the special case where the graph G is a tree T . Since trees are a particularly simple class of graphs, it is interesting to investigate whether one can obtain a better algorithm for independent set in trees than in graphs in general.



5.1.1 Independent set in graphs

Let us first consider the most general formulation of maximum weight independent set in graphs. Let $G = (V, E)$ be a graph with m edges and n vertices. Let $w : V \rightarrow \mathbb{R}_{>0}$ assign positive weights to the vertices. The most general goal is to compute the maximum weight independent set $S \subset V$.

We first observe that such a set S can be found by brute force. Here is a recursive algorithm.

```

MWIS( $G = (V, E)$ ,  $w : V \rightarrow \mathbb{R}_{\geq 0}$ )
1. if  $V = \emptyset$  then return 0
2. let  $v$  be any vertex in  $V$ 
3. return the maximum of
    // all independent sets excluding  $v$ 
    A. MWIS( $G - v, w$ ) where  $G - v$  denotes the graph obtained by
       removing  $v$  and all its incident edges from  $G$ .
    // all independent sets including  $v$ 
    B.  $w(v) + \text{MWIS}(G', w)$  where  $G'$  obtained by removing  $v$  and its
       neighbors1 from  $G$ .

```

The recursive algorithm will take exponential time. This is slow, but it establishes a baseline. Can we do better? What about the simpler, unweighted setting?

Let us instead try to show that the problem should be hard - in particular, by proving the following.

Theorem 5.1. *A polynomial time algorithm for maximum cardinality independent set implies a polynomial time algorithm for SAT.*

We know that SAT can be reduced to 3-SAT in polynomial time, so it suffices to show that a polynomial time algorithm for maximum cardinality independent set implies a polynomial time algorithm for 3-SAT.

Let $f(x_1, \dots, x_n)$ be a 3-SAT formula with m clauses $C_1, \dots, C_m$ and n variables $x_1, \dots, x_n$. We interpret each C_i as a set of three elements out of the set of symbols $X = \{x_1, \bar{x}_1, \dots, x_n, \bar{x}_n\}$. We want to convert f into a graph, where somehow an independent set larger than some particular size would indicate that f is satisfiable.

¹Two vertices u and v are **neighbors** if they are connected by an edge.

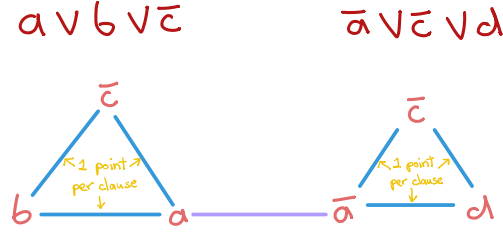


Figure 5.2: The gadget encoding disjunctions over three variables as triangles.

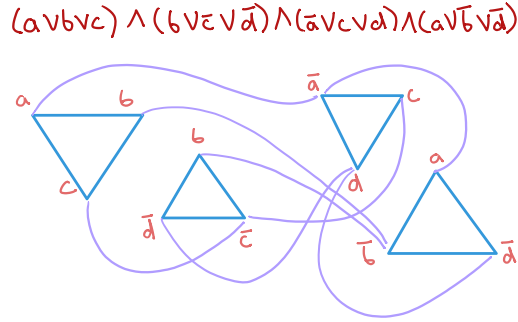


Figure 5.3: Encoding the 3-SAT formula $f(a, b, c, d) = (a \vee b \vee c) \wedge (b \vee \bar{c} \vee \bar{d}) \wedge (\bar{a} \vee c \vee d) \wedge (a \vee \bar{b} \vee \bar{d})$ as a graph, where the maximum independent set indicates whether or not f is satisfiable.

Clause gadgets. For each clause C , we make a triangle (i.e., three new vertices fully connected by edges). Each vertex in the triangle corresponds to a literal in the clause. For example, if $C = (x_1 \vee \bar{x}_2 \vee x_3)$, we create a triangle with three vertices, with one vertex corresponding to x_1 , $\bar{x}_2$, and x_3 . Thus a symbol such as x_1 or $\bar{x}_2$ may correspond to multiple vertices in the construction, with one vertex for each clause it appears in. So far, for m clauses, we have created a graph of m disjoint triangles, with $3m$ vertices and $3m$ edges in all. Note already that any independent set in G has at most one vertex per clause/triangle.

Our graph does not yet encode the fact that we can only set each variable x_i to be exactly one of true or false, and not both. Currently, we can take an independent set that (for example) contains vertices for both x_1 and $\bar{x}_1$. To encode this constraint, we add an edge between every pair of vertices of the form x_i and $\bar{x}_i$, for all i . With these additional edges, an independent set can no longer include variables corresponding to both x_i and $\bar{x}_i$.

This completes the constructions. In total the graph has $O(m)$ vertices and $O(m + n)$ edges. A minimal example is given in fig. 5.2. A more ornate example is given in fig. 5.3.

We now claim that $f(x_1, \dots, x_n)$ is satisfiable if and only if there is an independent set of size m in the graph G . Suppose first that there is an independent set S of size m . For each variable x_i , S may contain vertices corresponding to x_i , or vertices corresponding to $\bar{x}_i$, but not both, because of the edges we added in the second part of our construction. Consider the assignment where we set $x_i = 1$ if any of the x_i -vertices appear in S , or we set $x_i = 0$ if any of the $\bar{x}_i$ -vertices appear in S . (If neither type appears in S , then we set x_i to whatever we want.) We claim that this assignment satisfies all the clauses in f . We know that (a) there are m triangles (one per clause), (b) S has m vertices, and (c) S can have at most one vertex per triangle (to be independent). So S has exactly one vertex from every clause gadget. For each clause in f , our assignment satisfies the clause via the symbol for that vertex.

Conversely, suppose f is satisfiable, and fix a satisfying assignment. For each clause C , choose a literal that satisfies it, and add the corresponding vertex from C 's gadget in our graph to a set S . Since these choices are disjoint, S will have m vertices. We claim that S is independent. Indeed, picking one vertex per clause implies we satisfy all the triangle edges. Moreover we only choose a vertex corresponding to the symbol x_i if $x_i = 1$ in the satisfying assignment, and we only choose a vertex corresponding to the symbol $\bar{x}_i$ if $x_i = 0$. In particular, we can't choose both. So S will be independent over the second set of edges as well. Thus S is an independent set of size m .

To conclude, given a 3-SAT formula f with m clauses, we have built a graph G , of size polynomial in the size of f , that has an independent set of size m iff f is satisfiable. If we had a polynomial time algorithm for maximum independent set, we could run it on this graph and decide if f is satisfiable. This completes the proof.

5.1.2 Independent sets of intervals

While independent set on graphs turned out to be difficult, this does not preclude a more efficient algorithm for intervals, since this is a special case. Thus let us consider the problem for intervals with renewed optimism.

Let $\mathcal{I} = \{I_1 = [a_1, b_1], \dots, I_n = [a_n, b_n]\}$ be a set of intervals. Let $w : \mathcal{I} \rightarrow \mathbb{R}_{>0}$ assign positive weights to each interval. The goal is to compute the maximum weight independent set of intervals.

A lazy approach to this problem is to convert it to a graph by first comparing all pairs of intervals to identify all conflicting pairs. This would take quadratic time. (One can also do this faster - how?) Then one can compute brute force in the graphical setting as discussed above. But of course, this is far from efficient. More importantly, just because we can interpret the interval problem in terms of a graph, doesn't mean we should automatically treat it as a graph problem. This

is because the intervals have more structure than graphs (in general), and this structure in particular makes it more difficult to reduce from SAT.

For graphs we explored brute force algorithms that recursively test one element at a time. For the sake of concreteness, let us quickly rewrite that algorithm in terms of intervals. First, the recursive spec: for a set of intervals $\mathcal{I}$ and weights $w : \mathcal{I} \rightarrow \mathbb{R}_{>0}$, we define

MWIS-intervals($\mathcal{I}, w$) = the weight of the maximum weight independent subset of intervals of $\mathcal{I}$.

We now implement MWIS-intervals as specified above.

MWIS-intervals($\mathcal{I}, w : \mathcal{I} \rightarrow \mathbb{R}_{>0}$)

1. If $\mathcal{I} = \emptyset$ then return 0
2. Let $I \in \mathcal{I}$ be any interval
3. Let $\mathcal{I}' = \{J \in \mathcal{I} : J \cap I = \emptyset\}$ *// = set of intervals disjoint from I*
4. return the maximum of
 - /* The MWIS among those excluding I. */*
 - A. MWIS-intervals($\mathcal{I} \setminus \{I\}, w$)
 - /* The MWIS among those including I. */*
 - B. $w(I) + \text{MWIS-intervals}(\mathcal{I}', w)$

In our brute force algorithm, we are not selective about the order in which we process the elements. In a graph, there is no obvious order to the vertices anyway. In intervals, however, there are natural ways to order the intervals and this can make a big difference.

Let us sort $\mathcal{I}$ in increasing order of left endpoint (a_i) in $O(n \log n)$ time, and henceforth assume that

$$\mathcal{I} = \{I_1 = [a_1, b_1], \dots, I_n = [a_n, b_n]\}$$

lists the intervals in increasing order of a_i . Let us adjust the brute force algorithm again so that it processes the intervals in increasing order of left endpoint. The revised pseudocode is as follows.

```

MWIS-intervals( $\mathcal{I} = \{I_1 = [a_1, b_1], \dots, I_n = [a_n, b_n]\}$ ,  $w : \mathcal{I} \rightarrow \mathbb{R}_{>0}$ )

/* We assume  $I_1 = [a_1, b_1], \dots, I_n = [a_n, b_n]$  is sorted in increasing order
   of  $a_i$ . */
1. If  $n = 0$  then return 0
2. Let  $i > 1$  be the first index such that  $a_i > b_1$ 
   //  $I_i$  is the first interval disjoint from  $I_1$ , and  $\{I_i, \dots, I_n\}$  = set of
   intervals disjoint from  $I_1$ 
3. return the maximum of
   /* The MWIS among those excluding  $I_1$  */
   A. MWIS-intervals( $\{I_2, \dots, I_n\}, w$ )
   /* The MWIS among those including  $I_1$  */
   B.  $w(I_1) + \text{MWIS-intervals}(\{I_i, \dots, I_n\}, w)$ 

```

In this ordered version of brute force, every recursive call is for a subset of intervals of the form $\{I_i = [a_i, b_i], I_{i+1} = [a_{i+1}, b_{i+1}], \dots, I_n = [a_n, b_n]\}$ for some index i ; that is, a suffix of our *ordered* sequence of intervals. This limits the number of combinations of intervals in our subproblems: there are only $n + 1$ suffixes (including the empty set). We have an ideal situation for caching.

Let us now give a dynamic programming algorithm for the maximum weight independent set of intervals. Fix an input consisting of n intervals $\mathcal{I} = \{I_1 = [a_1, b_1], \dots, I_n = [a_n, b_n]\}$ and weights $w : \mathcal{I} \rightarrow \mathbb{R}_{>0}$. Sorting the intervals $\mathcal{I}$ in $O(n \log n)$, we assume the intervals are indexed in increasing order of left endpoint a_i . For $i \in \mathbb{N}$, we define

MWIS-intervals(i) = the maximum weight of any independent set of intervals among $\{I_i, I_{i+1}, \dots, I_n\}$.

We implement this with the following pseudocode.

$$\text{MWIS-intervals}(i) = \begin{cases} 0 & \text{if } i > n, \\ \max \left\{ \begin{array}{l} \text{MWIS-intervals}(i+1), \\ w(I_i) + \text{MWIS-intervals}(j(i)) \end{array} \right\} & \text{otherwise,} \end{cases}$$

where $j(i)$ is the first index j such that $a_j > b_i$. We can identify $j(i)$ by scanning the list of intervals. We apply dynamic programming to the above recursive code to make sure we never solve the same subproblem twice.

For the running time, we have n subproblems, each of which takes $O(n)$ time to compute, excluding recursive subcalls. (The $O(n)$ time comes from scanning for the next index $j(i)$.) The space usage is $O(n)$, since there are $O(n)$

many subproblems. The solution to the original problem is given by the call `MWIS-intervals(1)`. Correctness is inherited from brute force.

Can one do better? In fact, yes. For fixed i , we spend $O(n)$ time to find the first index $j(i)$ such that $a_j > b_i$. But instead one can do binary search to find $j(i)$, in $O(\log n)$ time. This improves the running time of each subproblem to $O(\log n)$, and the overall running time to $O(n \log n)$. This is our first examples where a dynamic programming algorithm is sped up by an adjustment to the implementation.

Taking a step back, we see that there is substantially more structure to intervals than a generic graph. A provably hard problem in graphs can be solved as fast as sorting in interval graphs!

5.1.3 Independent set in trees

We now consider the independent set problem in our second special case of graphs: trees. Here we are given a tree $T = (V, E)$ and (optionally) vertex weights $w : V \rightarrow \mathbb{R}$. The goal is to compute the maximum weight independent set $S \subset V$. The question really is whether trees are as general as graphs, or as structured as intervals on a line. Let us make two initial observations relating trees to our previous reduction.

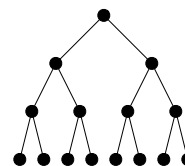
1. The reduction from 3SAT to independent set was full of triangles. But trees cannot have triangles, or any cycle, for that matter.
2. The graph corresponding to a set of intervals is also not a tree - any three overlapping intervals form a triangle.

There is clearly much more structure to trees than graphs. As mentioned above, trees have no cycles. If we remove an edge or a vertex, then we get two entirely disconnected parts. In fact one can always find a vertex whose removal breaks up the graph into subtrees with at most half the vertices (why?) - a remarkable property that is certainly not true for graphs.

Between this chapter and the last, we have seen multiple problems turn out to be easy when we take advantage of a clearly defined, *linear* ordering. But there is no clear notion of “left” and “right” in a tree. But while trees don’t have a linear structure, they do have a *hierarchical* structure that is very useful.

Fix any node $r \in V$ to be the root, and interpret T as a rooted tree. Now we can speak of “children”, “parents”, “subtrees” and so forth. There is a clear notion of “top” – the root r – and “bottom” – at the leaves. Let’s use this ordering to design a recursive algorithm for the maximum weight independent set.

From a hierarchical point of view, it is natural to design a recursive algorithm



using subtrees to define the subproblems. Consider the following recursive spec. For a vertex $v \in V$, we define

MWIS(v) = the maximum weight of any independent set in the subtree rooted at v .

Let us now sketch out an implementation MWIS(v). Fix v , and suppose we want to find the maximum weight independent set in the subtree rooted at v . A natural decision is whether or not to include v in the independent set. If we don't include v , then we can recurse on each of the children to compute the maximum weight independent set in each subtree. If we do include v , then... wait. Actually, it's not so obvious. When we recurse on a child x of v , this may return (the value of) an independent set that includes x . But this is incompatible with our decision to include v in the independent set. The following pseudocode summarizes our progress and poinpoints where we get stuck.

MWIS(v)

1. Return the maximum of

- /* Don't include v in the independent set */*
- A. The sum of MWIS(x) over all children x of v
- /* Include v in the independent set */*
- B. The weight of v plus the sum of ~~MWIS(x)~~ ??? over all children x of v

In step (1.B), where we got stuck, what did we really want? Not the MWIS in the subtree rooted at x , since this can include x . Rather we need the MWIS in the subtree rooted at x *among those that exclude x* . MWIS(x) simply doesn't offer this. The easiest way to address this deficiency is to strengthen the definition MWIS(x). We introduce a boolean flag, called `exclude-root` $\in \{0,1\}$, as an additional parameter, and define MWIS(v , `exclude-root`) as follows.

MWIS(v , `exclude-root`): If `exclude-root` = true, then the maximum weight independent set in the subtree rooted at v among those excluding v . If `exclude-root` = false, then the maximum weight independent set in the subtree rooted at v (which may or may not include v).

This stronger definition makes the implementation effortless.

MWIS(v , exclude-root)

1. If exclude-root
 - /* Don't include v in the independent set */*
 - A. Return the sum of MWIS(x , false) over all children x of v
2. Otherwise return the maximum of:
 - /* Don't include v in the independent set */*
 - A. The sum of MWIS(x , false) over all children x of v .
 - /* Include v in the independent set */*
 - B. The weight of v plus the sum of MWIS(x , true) over all children x of v

Good. It was a little trickier, but we have now established a recursive algorithm for computing the maximum weight independent set. The overall answer is given by MWIS(r , false). But our algorithm is not yet efficient. To this end – as usual – we apply caching. Let us save the answers to each MWIS(v , true) and MWIS(v , false) in a table² of size $2n$. Then we never compute the same answer twice.

Consider the running time. We have

$$(\# \text{ subproblems})(\text{time per subproblem}) = n \times O(n) = O(n^2).$$

Here we upperbound the “time per problem” by $O(n)$ because a vertex may have up to n children. But this analysis is too conservative. The total number of children over all nodes is n . A more refined analysis first observes that

$$(\text{time spent on MWIS}(v, \dots)) = O(\# \text{ children of } v).$$

Summing over all v , we have

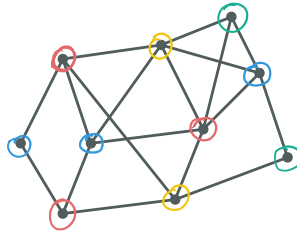
$$\sum_v (\text{time spent on MWIS}(v, \dots)) = O\left(\sum_v \# \text{ children of } v\right) = O(n).$$

So actually the running time is $O(n)$. We also mention that the space usage is $O(n)$ because there are $O(n)$ subproblems, each requiring constant space. Thus we have a $O(n)$ time algorithm for maximum weight independent set in trees. *Which is faster than for intervals! Faster than sorting!*

²We assume the vertices are number 1 through n . Otherwise one can number them in a preprocessing step.

5.2 Graph coloring

Graph coloring problems are an important topic in graph theory. Generally speaking, vertex coloring problems are problems where you want to color the vertices of an undirected graph so that any two adjacent vertices have different colors. More formally, we can define a **proper vertex coloring** as an assignment $\pi : V \rightarrow [k]$, for some integer k , such that for any edge $\{u, v\} \in E$, $\pi(u) \neq \pi(v)$. The goal is to find a vertex coloring $\pi : V \rightarrow [k]$ for k as small as possible.



Besides being a useful problem in practice (e.g., assigning non-overlapping frequency ranges to nearby cell-phone towers), it is an important problem in combinatorics, where graph theorists have discovered remarkable structural theorems such as the following famous theorem.

Theorem 5.2 (Brook's theorem, [Bro41]). *Let G be an undirected graph with maximum degree Δ . Then G can be colored in Δ colors unless G is either (a) a complete graph or (b) an odd cycle.*

We briefly point out there are many related graph problems of interest. For example, in edge coloring, the goal is to color the edges of an undirected graph so that no two adjacent edges³ have the same color. For edge coloring, *Vizing's theorem* asserts that every graph with maximum degree Δ can be edge-colored with at most Δ colors.

Let us return to vertex coloring. Here we are interested in the optimization problem rather than absolute bounds such as Brook's theorem. Given an undirected graph $G = (V, E)$, the goal is to color G with as few colors as possible. Or, as a decision problem: given an undirected graph $G = (V, E)$, and an integer $k \in \mathbb{N}$, can G be colored in k or fewer colors? There is also the k -coloring problem where k is fixed, but otherwise asks the same question: Given as input a graph $G = (V, E)$, can G be colored in k or fewer colors? Note that for k -coloring, an $n^{f(k)}$ running time, for any function f , counts as polynomial time.

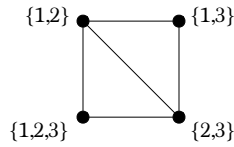
This problem is yet another search problem, since it is easy to check if an assignment is a proper coloring with at most k colors. So in particular we have

³Two edges are adjacent if they have a common endpoint.

an exponential time algorithm by brute force. Can one do better? What about the special case of $k = 3$ – 3-coloring?

Theorem 5.3. *A polynomial time algorithm for vertex 3-coloring implies a polynomial time algorithm for SAT.*

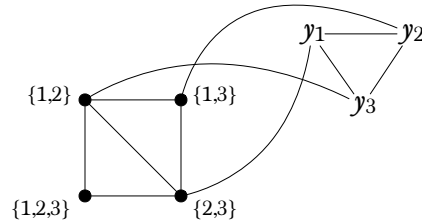
To prove the above it is convenient to introduce the following more general variant of k -coloring. In the **subset k -coloring problem**, each vertex v in the graph is annotated with a subset $S_v \subset [k]$ of colors available to that vertex. The goal is to properly color the graph $\pi : V \rightarrow [k]$ such that for each k , $\pi(v) \in S_v$.



Lemma 5.4. *Let $k \in \mathbb{N}$ be fixed. A polynomial time algorithm for k -coloring implies a polynomial time algorithm for subset- k coloring.*

Proof. Assume an instance of subset- k coloring defined by a graph $G = (V, E)$ with m edges and n vertices and, for each vertex v , a set $S_v \subset [k]$ of colors available to v .

To reduce to k -coloring, we will construct a new graph H that extends the original graph G . Starting from G , we first introduce a clique K_k of k colors. That is, we create k new vertices $y_1, \dots, y_k$ and add an edge between every pair of new vertices. Observe that any k -coloring of this clique requires every y_i to receive a distinct color, and this uses up all k colors. (Conceptually we interpret the color assigned to y_i as the i th color.) Now, for each vertex v , and each color i omitted from S_v , we add an edge between y_v and i . (Conceptually, this forbids v from taking i in a k -coloring.) Below we draw the construction for $k = 3$.



We claim that G is k -subset-colorable with respect to the sets $\{S_v : v \in V\}$ iff H is k -colorable. First suppose G is k -subset-colorable. Given any subset coloring π of G , we can extend π to a k -coloring on H by coloring $\pi(y_i) = i$ for each i . This does not create any monochromatic edges between the y_i 's, nor does it

decrease any monochromatic between a y_i and a vertex $v \in V_G$ because $i \notin S_v$ for any edge $\{y_i, v\}$.

Conversely, consider any proper coloring of $\pi : V_H \rightarrow [k]$ of H . Without loss of generality we may assume that $\pi(y_i) = i$ for each i .⁴ We need to show that $\pi(v) \in S_v$ for all v . Indeed, for any color $i \notin S_v$, the edge $\{y_i, v\}$ implies that $\pi(v) \neq \pi(y_i) = i$. ■

To prove theorem 5.3, it now suffices to show that subset 3-coloring can be used to solve SAT.

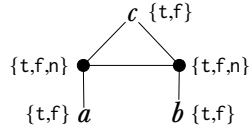
Theorem 5.5. *A polynomial time algorithm subset 3-coloring implies a polynomial time algorithm for 3-SAT.*

Let $f(x_1, \dots, x_n)$ be a 3-SAT formula with n variables $x_1, \dots, x_n$ and m clauses $C_1, \dots, C_m$. We want to create an instance of subset 3-coloring (of roughly the same size) that is feasible iff f is feasible.

We point out that the names of the colors in a coloring problem are arbitrary. For this reduction, it is convenient to name the colors true, false, and nonsense; or t, f, and n for short.

We first create vertices corresponding to the variables. For each variable x_j , we make two vertices A_j and $\bar{A}_j$ each constrained to have colors from the set $\{t, f\}$. We also add an edge between A_j and $\bar{A}_j$, which forces exactly one of A_j or $\bar{A}_j$ to be colored true, and the other to be colored false.

Lemma 5.6. *Consider the following instance of subset 3-coloring with five vertices.*



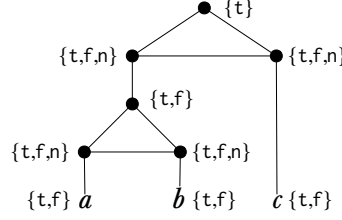
In any proper subset coloring, $c = \text{true}$ only if $a \vee b = \text{true}$.

Proof. By inspection. ■

Now we can construct a gadget for 3-CNF clauses. Recall that $(a \vee b \vee c) = ((a \vee b) \vee c)$. Likewise we can compose to OR gadgets as follows.

Lemma 5.7. *Consider the following instance of subset 3-coloring with 8 vertices.*

⁴This is because we only care if two colors have the same labels, which is invariant to permutation of the labels.



Then in any proper subset coloring of the above, $a \vee b \vee c = \text{true}$.

For each clause C_i , we identify the three vertices corresponding to the literals in that clause. (For example, $x_1 \vee \bar{x}_2 \vee x_3$ corresponds to the vertices A_1 , $\bar{A}_2$, and A_3 .) We build the 3CNF gadget, per lemma 5.7 over these three vertices. Each clause generates 6 new variables.

Between the two vertices A_j and $\bar{A}_j$ for each variable x_j , and the constant-size gadgets built over these vertices for each C_i , we obtain an instance of subset 3-coloring that is satisfiable iff f is satisfiable.

Indeed, given a satisfying assignment x , we color $A_j = \text{true}$ and $\bar{A}_j = \text{false}$ if $x_j = \text{true}$, and reverse the colors if $x_j = \text{false}$. One can then color each clause gadget appropriately. Conversely, given any proper coloring of the clauses, one constructs an assignment x by setting x_j to be the color of A_j . One can verify that, since the coloring was able to color the clause-gadgets properly, each clause in the formula is satisfied by x . This completes the proof.

Coloring with more than 3 colors. We have not shown that k -coloring is hard for $k > 3$. This is left to the reader in exercise 5.1.

Coloring in interval graphs and trees. As with the independent set problem, the reduction from SAT to 3-coloring does not forbid polynomial time algorithms for trees or intervals. This is left to the reader in exercise 5.3.

5.3 Dominating set

[picture of dominating set]

Let $G = (V, E)$ be an undirected graph. A set of vertices $S \subseteq V$ is a **dominating set** if every vertex $v \in V$ is either in S or the neighbor of a vertex in S . The minimum dominating set problem is to compute the minimum cardinality dominating set. The decision version additionally specifies an integer k and the question is whether there is a dominating set of size k (or greater). One can consider weighted versions of the problem where we have weights $w : V \rightarrow \mathbb{R}_{>0}$ and the goal is to compute the dominating set of minimum weight.

Perhaps it is not so surprising, after discussing independent set and coloring, that dominating set is also as hard as SAT.

Theorem 5.8. *A polynomial time algorithm for dominating set implies a polynomial time algorithm for SAT.*

Let $f(x_1, \dots, x_n)$ be a Boolean formula in CNF with n variables, m clauses, and total size p . We will construct an undirected graph G such that G has a dominating set of size n iff f is satisfiable. Here are the first steps in making G .

1. For every variable x_i , we create two vertices denoted v_i and $\bar{v}_i$. v_i is associated with the symbol x_i and the assignment $x_i = \text{true}$; $\bar{v}_i$ is associated with the symbol $\bar{x}_i$ and the assignment $x_i = \text{false}$.
2. For every clause C_j , we create a vertex w_j .
3. For every clause C_j and every variable x_i appearing as x_i (without negation) in C_j , we add the edge $\{v_i, w_j\}$. For every variable x_i appearing in C_j negated as $\bar{x}_i$, we add an edge from.
4. For every variable x_i , we add an edge between v_i and $\bar{v}_i$.

Let us try to explain the intuition of the construction so far. We have vertices for each assignment to each variable. We also have vertices for the clauses. We have edges between (vertices representing) clauses and the (vertices representing the) single-variable assignments that satisfy the clauses. The total number of variables is $O(m + n)$, and the number of edges is $O(p)$.

Now consider any satisfying assignment for f . We will construct a dominating set S with n vertices as follows. If the satisfying assignment has $x_i = \text{true}$, then we include the corresponding vertex v_i . If the satisfying assignment has $x_i = \text{false}$, then we include the corresponding vertex $\bar{v}_i$. So S has n vertices, with each vertex corresponding to a variable's assignment. To see that this is a dominating set, we observe that:

1. For every clause-vertex w_j corresponding to clause C_j , the vertex corresponding to a variable satisfying C_j will dominate w_j .
2. For each variable-vertex v_i or $\bar{v}_i$, the fact that S takes either v_i or $\bar{v}_i$ ensures that both v_i and $\bar{v}_i$ are dominated.

Conversely, suppose our graph has a dominating set S of size n . Ideally S would contain exactly one of the vertices, v_i or $\bar{v}_i$, for each variable x_i , and form a satisfying assignment accordingly. However S could contain neither v_i or $\bar{v}_i$, or both v_i and $\bar{v}_i$. So it is not clear that a dominating set of size n implies a satisfying assignment (and the reader might think of some simple counterexamples).

So to summarize the current state of affairs: our construction is such that if f has a satisfying set, then the graph has a dominating set with (at most) n vertices. However, a dominating set with n vertices does *not* imply a satisfying assignment for f . We ask the reader how to improve the construction to complete the proof.

We would like to force S to take at least one of v_i or $\bar{v}_i$ for each variable x_i . If so, and S also has size (at most) n , then it must take exactly one of these vertices for each x_i . How can we force every dominating set to take at least one of v_i or $\bar{v}_i$? We can introduce another auxiliary variable, which we denote u_i , that can only be dominated by v_i or $\bar{v}_i$ (or itself). So we add the following steps to our construction.

5. For every variable x_i , we create a (third) vertex u_i .
6. We add the edges $\{u_i, v_i\}$ and $\{u_i, \bar{v}_i\}$.

The introduction of u_i means that we have to have either u_i , v_i or $\bar{v}_i$ in any dominating set. Moreover, if a dominating set S has u_i , it can always be replaced with v_i and $\bar{v}_i$ while keeping S a dominating set.

Now, we can complete the proof. As before, a satisfying assignment to f maps to a dominating set of size n . Now, suppose S is a dominating set with at most n vertices. We know that S includes at least one of v_i , $\bar{v}_i$, or u_i for each i , in order to dominate u_i . Whenever S contains some u_i , we replace it with v_i or $\bar{v}_i$ (either one works); now we know that S includes at least one of v_i or $\bar{v}_i$. On the other hand, $|S| \leq n/2$ implies that S contains exactly one of v_i or $\bar{v}_i$. Consider the assignment $x_1, \dots, x_n$ corresponding to the vertices in S . For each clause C_j , the corresponding variable w_j is dominated by some v_i or $\bar{v}_i$ in S ; the corresponding assignment ($x_i = \text{true}$ or $x_i = \text{false}$) satisfies C_j . Thus we have a satisfying assignment.

To summarize, we have a dominating set of size n iff f is satisfiable. Our auxiliary graph is polynomial in the size of f , and the construction takes polynomial time. A polynomial time algorithm that computes the minimum dominating set can, in particular, decide if there is a dominating set of size n in the auxiliary graph of a given CNF $f(x_1, \dots, x_n)$. We would obtain a polynomial time algorithm for CNF SAT by first converting the algorithm to the graph described above and then computing a dominating set.

Dominating set for intervals and trees. What about dominating set in interval graphs, or in trees? We leave these questions to the reader in exercise 5.4.

5.4 Exercises

Exercise 5.1. Theorem 5.3 proved that a polynomial time algorithm for 3-coloring implies a polynomial time algorithm for SAT. Prove that for any $k > 3$, a polynomial time algorithm for k -coloring implies a polynomial algorithm for SAT.⁵

Exercise 5.2. Let k be fixed. Suppose you had access to a polynomial time algorithm for the decision version of k -coloring. Use this algorithm to obtain a polynomial time algorithm that obtains a k -coloring of a graph (if one exists).

Exercise 5.3. Recall that the k -coloring problem is to decide if there is a vertex coloring with at most k -colors. The subset k -coloring problem also specifies for each vertex v a subset $S_v \subseteq [k]$, and the k -coloring is restricted to have each vertex v colored by a color in S_v . For each of the following problems, either (a) design and analyze a polynomial time algorithm, or (b) prove that a polynomial time algorithm would imply a polynomial time algorithm for SAT.

1. The k -coloring problem in interval graphs.
2. The k -coloring problem in trees.
3. The subset k -coloring problem in interval graphs, for fixed k .
4. The subset k -coloring problem in trees, for fixed k .

Exercise 5.4. Recall the dominating set problem from section 5.3. Here we will consider the weighted version where the vertices are given positive weights, and the goal is to compute the minimum weight dominating set. For each of the following problems, either (a) design and analyze a polynomial time algorithm, or (b) prove that a polynomial time algorithm would imply a polynomial time algorithm for SAT.

1. The minimum weight dominating set problem in interval graphs.
2. The minimum weight dominating set problem in trees.

Exercise 5.5. Given a graph G , a **clique** is a set of vertices S such that every pair of vertices in S is connected by an edge. This is sort of like opposite of an independent set.

⁵It might be helpful to focus on $k = 4$ to develop your ideas.

1. Consider the problem of finding the largest weight clique from a set of weighted intervals. (Here a clique of intervals is a set of intervals where every two intervals are overlapping.) Either (a) show that this problem is as hard as solving SAT, or (b) design and analyze an algorithm the finds the maximum weight clique.
2. Consider the problem of finding the largest weight clique in a tree. Either (a) show that this problem is as hard as solving SAT, or (b) design and analyze an algorithm the finds the maximum weight clique in a tree.
3. Consider the problem of finding the largest weight clique in a graph G . Either (a) show that this problem is as hard as solving SAT, or (b) design and analyze an algorithm the finds the maximum weight clique.⁶

⁶Hint: Consider the “complement graph” $G' = (V, E')$ where there is an edge $\{u, v\}$ iff $\{u, v\}$ is *not* an edge in G .