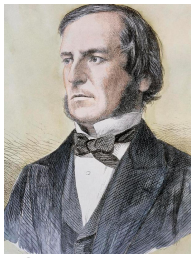
The Longest Path

Boolean logic $x, y \in \{\text{TRUE}, \text{FALSE}\} = \{1, 0\}$

"x and y" $x \wedge y$

"x or y" $x \vee y$

"not x" $\overline{x}$ (or $\neg x$)

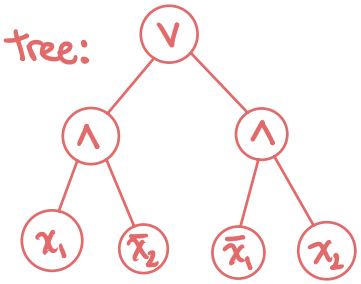


George Boole, 1815-1864

XOR ("exclusive-or") (TRUE iff $x_1 \neq x_2$)

$f(x_1, x_2) = (x_1 \wedge \overline{x}_2) \vee (\overline{x}_1 \wedge x_2)$

as a tree:



"SAT Programming Language"

"x and y" $x \wedge y$

"x or y" $x \vee y$

"not x" $\overline{x}$ (or $\neg x$)

"z = x" "both true or both false" $(z \wedge x) \vee (\overline{z} \wedge \overline{x})$

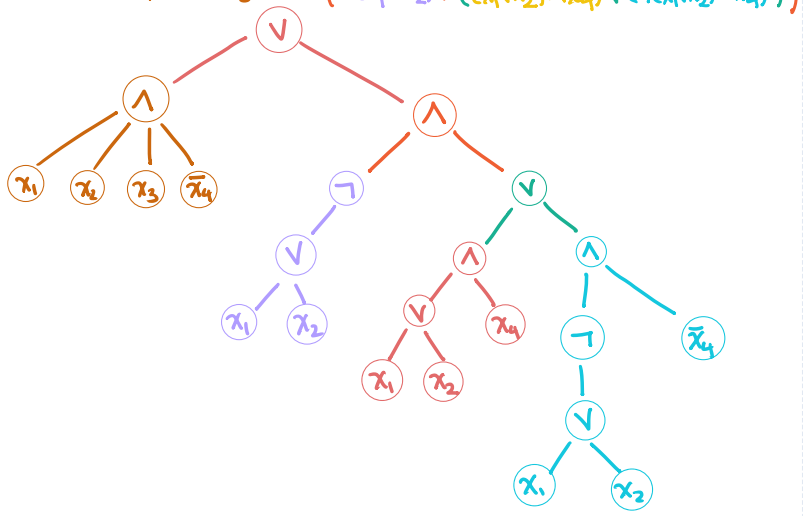
"if x then y else z" "(x=1, y=1) or (x=0, z=1)" $(x \wedge y) \vee (\overline{x} \wedge z)$

So Boolean algebra is very expressive

Adding two bits

$f(x_1, x_2, x_3, x_4) = \begin{cases} 1 & \text{if } x_1 + x_2 = x_3 x_4 \\ 0 & \text{otherwise} \end{cases}$

$f(x_1, x_2, x_3, x_4) = (x_1 \wedge x_2 \wedge x_3 \wedge \overline{x}_4) \vee (\neg(x_1 \wedge x_2) \wedge ((x_1 \vee x_2) \wedge x_4) \vee (\neg(x_1 \vee x_2) \wedge \overline{x}_4))$



Boolean Satisfiability

Input: Boolean formula $\varphi(x_1, x_2, \sim, x_n)$

- n variables
- "size" m

e.g. $\varphi(\underset{\text{(false)}}{0}, \underset{\text{(true)}}{1}, 0, 0, \sim, 1, 1) = \underset{\text{(false)}}{0}$

Goal: decide if φ is "satisfiable";
i.e., if there exists $x_1, x_2, \sim, x_n \in \{0, 1\}$
s.t. $\varphi(x_1, x_2, \sim, x_n) = 1$

"search problem": we can recognize an answer (e.g. satisfying assignment x) when we see it (by evaluating $f(x)$).

- at the very least: we can try all x.
- harder "search problem" than "over/under"

Boolean Satisfiability

Input: Boolean formula $\varphi(x_1, x_2, \sim, x_n)$

- n variables
- "size" m

e.g. $\varphi(\overset{\text{(false)}}{0}, \overset{\text{(true)}}{1}, 0, 0, \sim, 1, 1) = \overset{\text{(false)}}{0}$

Goal: "search" problem

The Big Question

Is there a polynomial time algorithm for Boolean satisfiability?

answer (e.g. satisfying assignment x) when we see it (by evaluating $f(x)$).

- at the very least: we can try all x.
- harder "search problem" than "over/under"

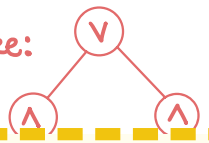
Boolean logic $x, y \in \{\text{TRUE}, \text{FALSE}\} = \{1, 0\}$

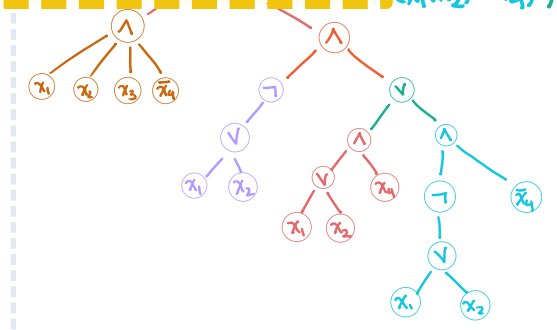
"x and y"	$x \wedge y$
"x or y"	$x \vee y$
"not x"	$\bar{x}$ (or $\neg x$)

XOR ("exclusive-or") (TRUE iff $x_1 \neq x_2$)

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

as a tree:



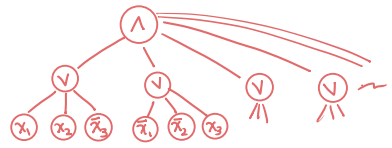


"SAT Programming Language"

- "z = x" — "both true or both false" $(z \wedge x) \vee (\bar{z} \wedge \bar{x})$
- "if x then y else z" — "(x=1, y=1) or (x=0, z=1)" $(x \wedge y) \vee (\bar{x} \wedge z)$

CNF (Conjunctive Normal Form)

"and-of-or's"



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

"clause"

$$(\bar{x}_1 \vee x_2 \vee \bar{x}_3) \wedge (\bar{x}_1 \vee \bar{x}_2 \vee x_3) \wedge (\dots) \wedge \dots$$

(CNF-) SAT problem

Given boolean formula f in CNF, decide if f is satisfiable

More modest Question

Is there a polynomial time algorithm for CNF-satisfiability?

"CNF-SAT Programming Language"

"x and y" $x \wedge y$

"x or y" $x \vee y$

"not x" $\bar{x}$ (or $\neg x$)

* "z = x"

"both true or both false"

$$(z \wedge x) \vee (\bar{z} \wedge \bar{x})$$
$$= (z \vee (\bar{z} \wedge \bar{x})) \wedge (x \vee (\bar{z} \wedge \bar{x}))$$
$$= (z \vee \bar{z}) \wedge (z \vee \bar{x}) \wedge (x \vee \bar{z}) \wedge (x \vee \bar{x})$$
$$= (z \vee \bar{x}) \wedge (x \vee \bar{z})$$

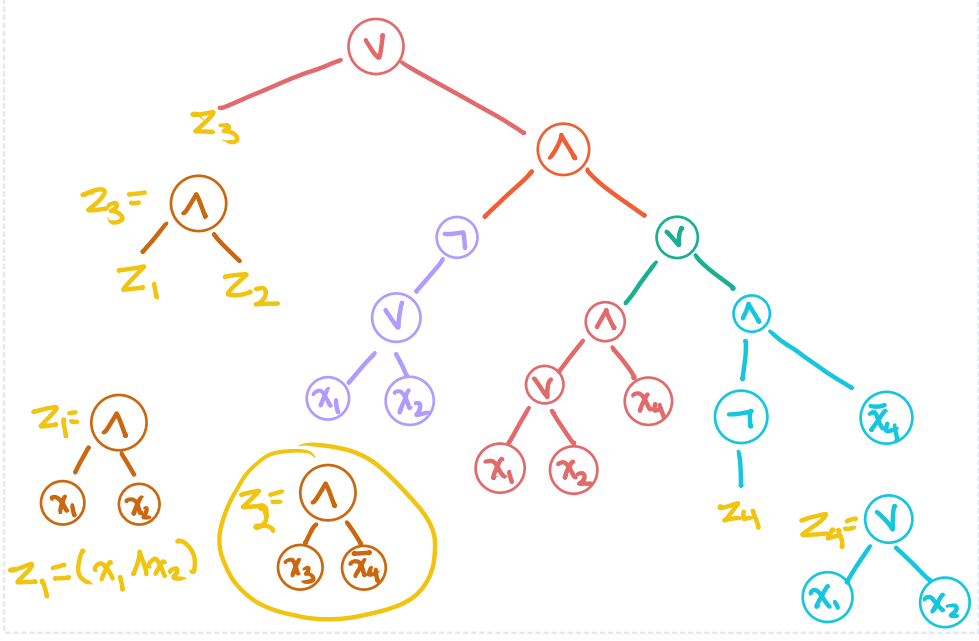
* "if x then y else z"

"(x=1, y=1) or (x=0, z=1)"

$$(x \wedge y) \vee (\bar{x} \wedge z)$$
$$= (x \vee \bar{x}) \wedge (y \vee \bar{x}) \wedge (x \vee z) \wedge (x \vee \bar{z})$$
$$= (y \vee \bar{x}) \wedge (x \vee z) \wedge (x \vee \bar{z})$$

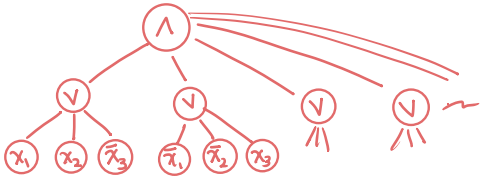
So CNF is also very expressive

Theorem: there is a poly-time algo for Boolean SAT iff there is a poly-time algo for CNF-SAT.



more special case: **3-SAT**

like CNF-SAT, except w/ exactly 3 variables per clause.



$f(x_1, \sim, x_n) =$

"clause"

$$(\bar{x}_1 \vee x_2 \vee \bar{x}_3) \wedge (\bar{x}_1 \vee \bar{x}_2 \vee x_3) \wedge (\sim \vee \sim \vee \sim) \wedge \sim$$

Is there a polytime algo for 3SAT?

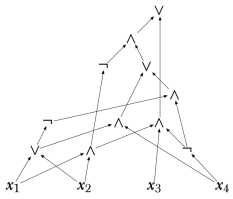
Theorem: there is a poly-time algo for Boolean SAT iff there is a poly-time algo for 3-SAT.

Conclusion

We want, but don't have, polynomial time algos for

Boolean SAT, CNF SAT, 3-SAT, or Circuit SAT

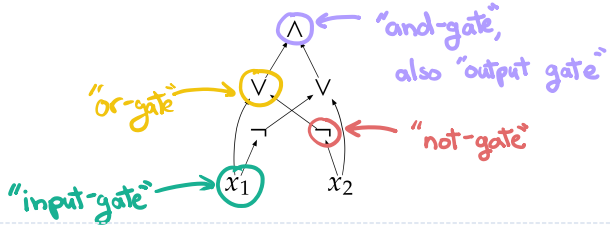
But we do know they are equivalent up to polynomial time!



(definition)

logical gates arranged as vertices in a directed acyclic graph

"gates": take some inputs bits, output 1 bit



Circuit SAT

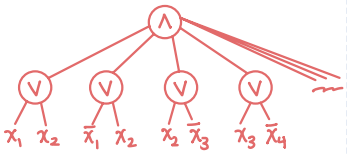
given a circuit, decide if it is satisfiable

Theorem: there is a poly-time algo for Boolean SAT iff there is a poly-time algo for Circuit SAT

2-SAT

like CNF-SAT, except w/ exactly 2 variables per clause.

$f(x_1, \sim, x_n) =$



$(x_1 \vee x_2) \wedge (\bar{x}_1 \vee x_2) \wedge (x_2 \vee \bar{x}_3) \wedge (x_3 \vee \bar{x}_4) \wedge \sim$

Max 2-SAT

like 2-SAT, except goal is to satisfy *maximum number of clauses.*
(not necessarily all)

Is there a polytime algo for Max 2-SAT?

let C be a sink SCC.

Complement $\bar{C}$ is a source SCC (why?)

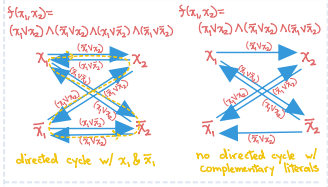
Assign (all literals in) C = true, $\bar{C}$ = false

Remove C and $\bar{C}$ and recurse on remaining SCC's

Key idea: assigning sinks to be true, sources to be false does not violate any edges

Implication graph
- vertex for each $x_i, \bar{x}_i$
- for clause $(a \vee b)$,

$\bar{a} \xrightarrow{(a \vee b)} b$ "$\bar{a}$=TRUE ⇒ b=TRUE" $\bar{b} \xrightarrow{(a \vee b)} a$ "$\bar{b}$=TRUE ⇒ a=TRUE"

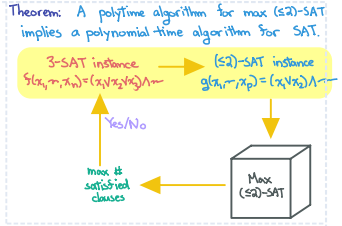


all literals in SCC have same assignment

each SCC C has complement SCC $\bar{C}$ w/ negated literals & reversed edges

	Polytime Algo	Hard as (CNF-) SAT	Abstain
2SAT	66%	44%	2
Max 2-SAT	60%	35%	5

Fix a clause $(a \vee b \vee c)$
introduce new variable y
consider the 10 clauses
$(a), (b), (c), (\bar{a} \vee \bar{b}), (\bar{b} \vee \bar{c}), (\bar{c} \vee \bar{a}),$
$(y), (a \vee \bar{y}), (b \vee \bar{y}), (c \vee \bar{y})$



$(a, b, c) =$	(f, f, f)	(t, f, f)	(t, t, f)	(t, t, t)
$(a), (b), (c)$	0	1	2	3
$(\bar{a} \vee \bar{b}), (\bar{b} \vee \bar{c}), (\bar{c} \vee \bar{a})$	3	3	2	0
$\max_{y \in \{t,f\}} (y), (a \vee \bar{y}), (b \vee \bar{y}), (c \vee \bar{y})$	3	3	3	4
total	6	7	7	7

Theorem: A polytime algorithm for max 2-SAT implies a polynomial time algorithm for SAT.

$(a_i \vee y_i) \wedge (a_i \vee \bar{y}_i)$

(≤2)-SAT instance
$f(x_1, \sim, x_n) = (x_1 \vee x_2 \vee x_3) \wedge \sim$

→ 2-SAT instance
$g(x_1, \sim, x_p) = (x_1 \vee x_2) \wedge \sim$

→ Max 2-SAT

→ max # satisfied clauses in g

−k → (≤2)-SAT instance

Subset Sum

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

Goal: find a subset of x_i's that sum to T
(decision version: decide if such a subset sum exists)

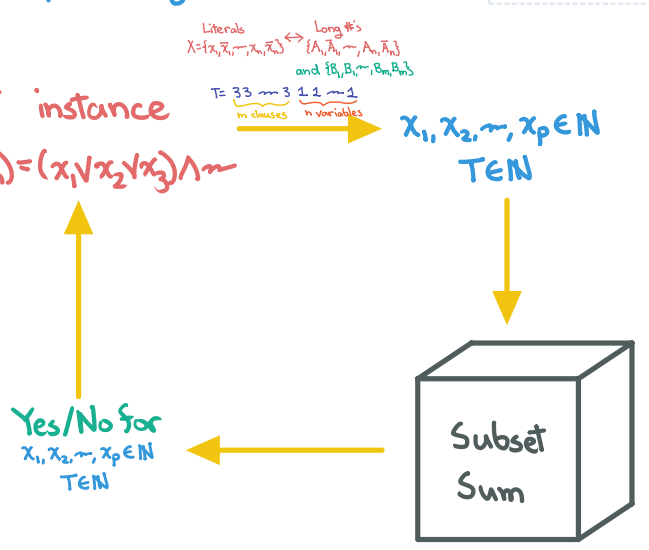
e.g. 14, 13, 17, 7, 18, 18, 10, 4, 11, 10
$T = 52$

Another "search" problem

	Polytime Algo	Hard as (CNF-) SAT	Abstain
Subset Sum	10%	30%	0

Theorem: A polytime algo for subset-sum implies a polytime algo for SAT

Subset Sum
Input: $x_1, x_2, \dots, x_n \in \mathbb{N}$
Goal: find a subset of x_i's that sum to T
(decision version: decide if such a subset sum exists)
e.g. 14, 13, 17, 7, 18, 18, 10, 4, 11, 10
$T = 52$



Key idea: numbers are bit-strings too! ← use many digits; ← each digit encodes 1 constraint

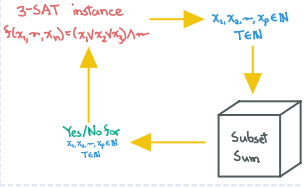
Key idea: numbers are bit-strings too!
- use many digits
- each digit encodes 1 constraint

2. Start making #'s:
- we use mtn digits base 4
- for the literal x_j:

A_j (models x_j) = 0 1 01…1 1 | 0 0 …1… 0
(m slots for each clause | n slots for each variable)

- for each clause C_i, set C_ith digit to 1 if $x_j \in C_i$ (not $\bar{x}_j$), else 0
- set "x_jth digit" to 1, else 0

Subset Sum
Input: $x_1, x_2, \dots, x_n \in \mathbb{N}$
Goal: find a subset of x_i's that sum to T
(decision version: decide if such a subset sum exists)
e.g. 14, 13, 17, 7, 18, 18, 10, 4, 11, 10
$T = 52$



1. Recast 3SAT as "subset" problem
X = literals $\{x_1, \bar{x}_1, x_2, \bar{x}_2, \dots, x_n, \bar{x}_n\}$
clause $(x_1 \vee \bar{x}_2 \vee x_3)$ = set $\{x_1, \bar{x}_2, x_3\} \subset X$
Goal: find $S \subset X$ s.t.
(a) for each clause C_i, $C_i \cap S = \emptyset$
(b) for each x_j, S contains 1 (exactly) of $\{x_j, \bar{x}_j\}$

Dynamic Programming

Subset Sum
Input: $x_1, x_2, \dots, x_n \in \mathbb{N}$
Goal: find a subset of x_i's that sum to T
(decision version: decide if such a subset sum exists)
e.g. 14, 13, 17, 7, 18, 18, 10, 4, 11, 10
$T = 52$

Recursive Spec

(Subset-Sum) $SS(i, S)$ = TRUE if there is a subset of $x_i, \dots, x_n$ summing to S, and FALSE otherwise

Implementation

```
SS(i, S)
1. If S<0 then return FALSE
2. If S=0 then return TRUE
3. If n=0 then return FALSE
4. Return SS(i+1, S) V SS(i+1, S-x_i)
```

Running time:

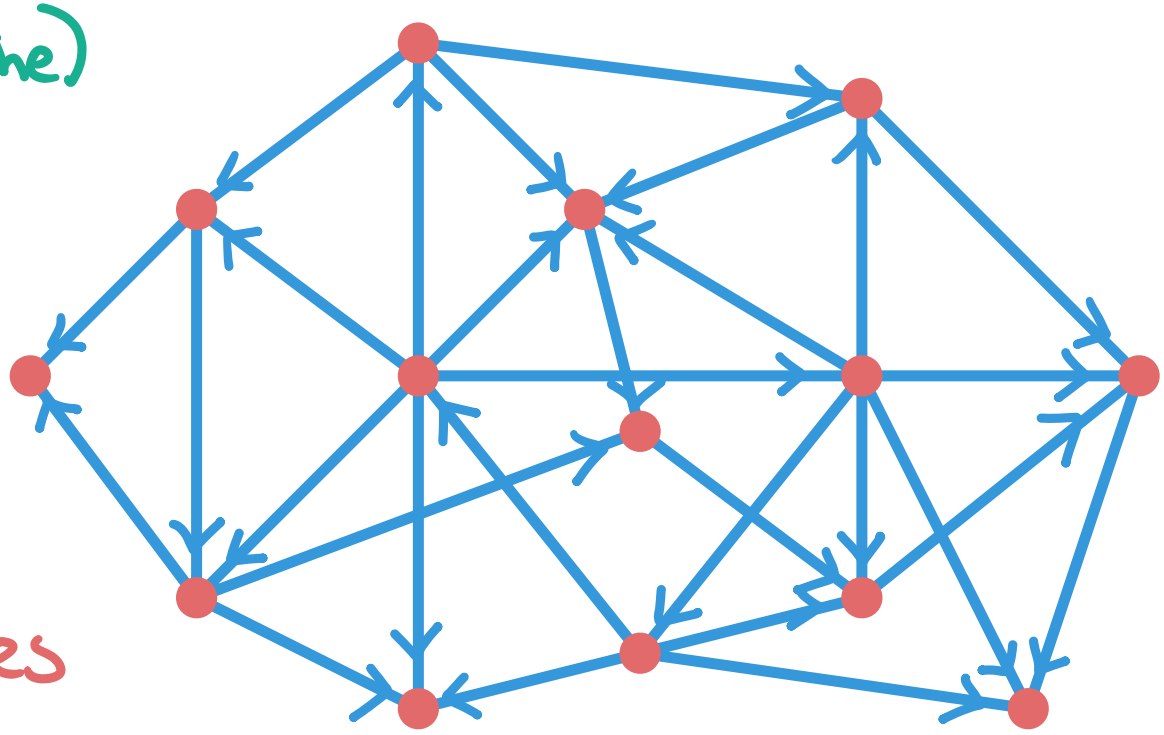
$O(nT)$ subproblems

$O(1)$ time per subproblem

$O(nT)$ time overall

today: longest path problem (vertices cannot repeat)

Find the (length of the) longest path in G (eg. # edges)



Hamiltonian path:

path visiting all vertices

does G have a Hamiltonian path?

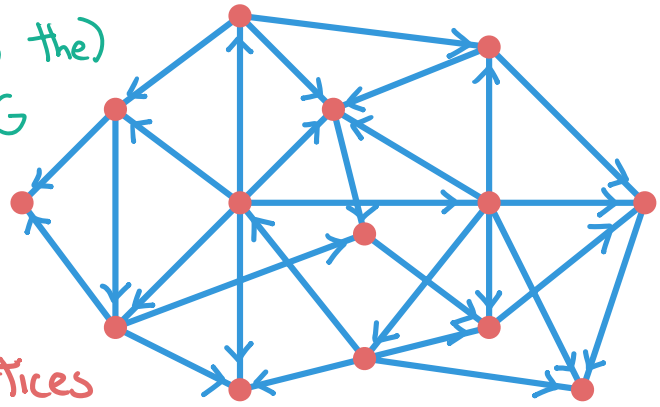
related problems: longest walk, shortest paths

Three settings:

1. DAG
2. Directed
3. Undirected

today: longest path (vertices cannot repeat) problem

Find the (length of the) longest path in G



Hamiltonian path:

path visiting all vertices

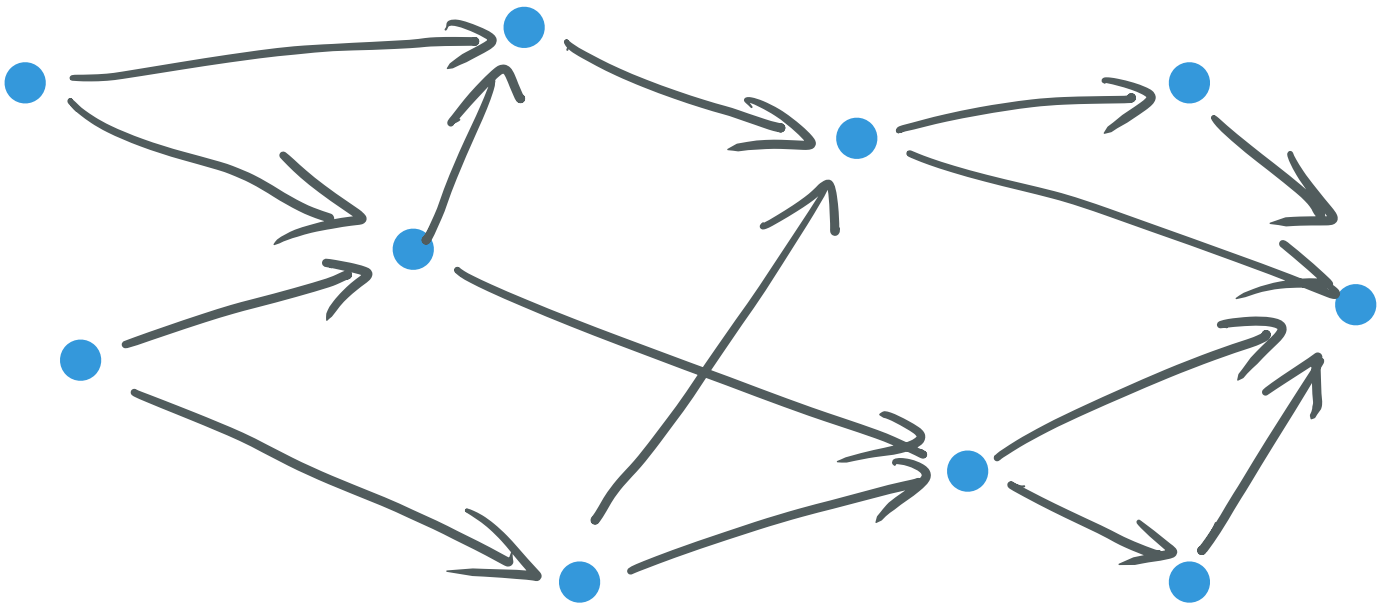
does G have a Hamiltonian path?

	Polytime Algo	Hard as (CNF-) SAT	Abstain
DAG			
Directed			
Undirected			

Longest ~~path~~ walk in a DAG

no cycles

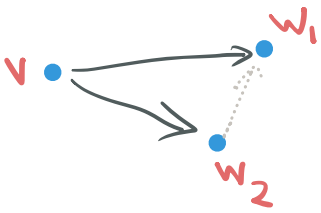
⇒ walk = path



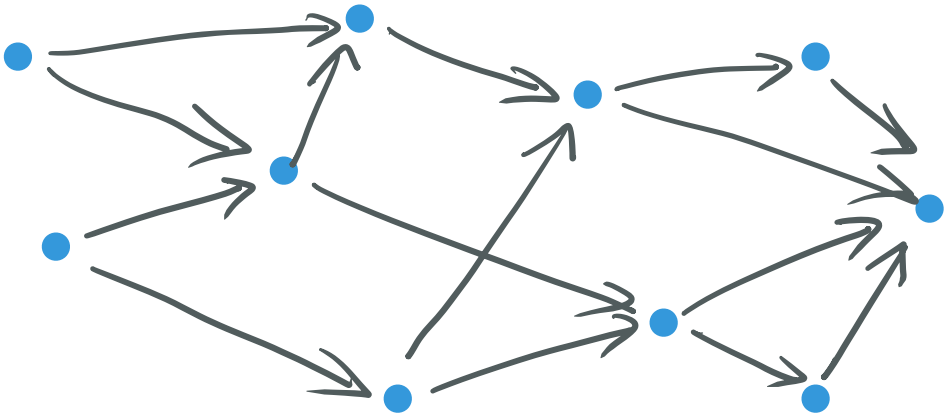
Recursion

$LW(v)$ = length of Longest Walk from v

$$LW(v) = \begin{cases} 0 & \text{if } v \text{ is a sink} \\ 1 + \max_{(v,w) \in \partial^+(w)} LW(w) & \text{otherwise} \end{cases}$$



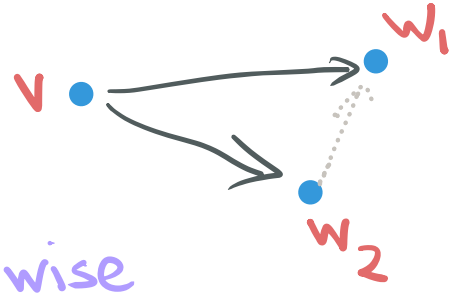
Longest walk in DAGs



Recursion

$LW(v)$ = length of Longest Walk from v

$$LW(v) = \begin{cases} 0 & \text{if } v \text{ is a sink} \\ 1 + \max_{(v,w) \in \vec{\partial}^+(w)} LW(w) & \text{otherwise} \end{cases}$$



Running time w/ caching

$$\sum_{v \in V} (\text{time for } LW(v)) = \sum_{v \in V} 1 + \deg^+(v) = O(m+n)$$

(out-degree)

Dynamic programming in DAGs

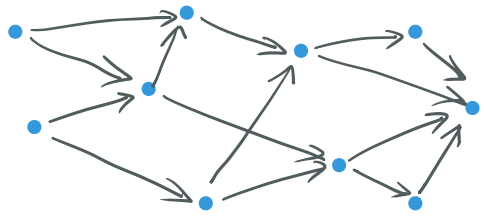
topological order

⇒ induction, recursion

dynamic programming, caching ⇒ efficient (even linear time) algos

Many hard problems become much easier in DAGs

Longest ~~path~~ walk in a DAG



no cycles
⇒ walk = path

Recursion

$LW(v)$ = length of Longest Walk from v

$$LW(v) = \begin{cases} 0 & \text{if } v \text{ is a sink} \\ 1 + \max_{(v,w) \in E^+(w)} LW(w) & \text{otherwise} \end{cases}$$

Running time w/ caching

$$\sum_{v \in V} (\text{time for } LW(v)) = \sum_{v \in V} 1 + \deg^+(v) = O(m+n)$$

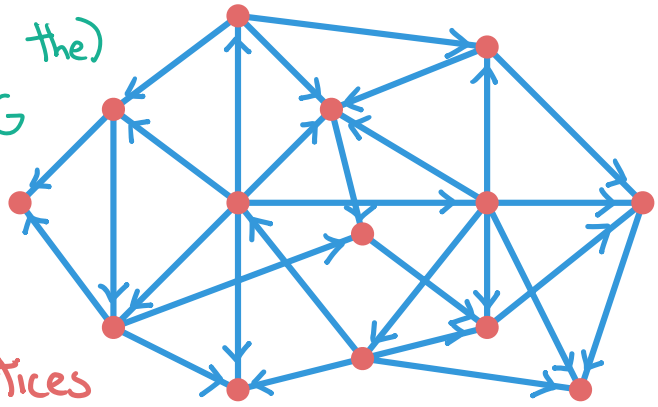
(out-degree)

Three settings:

1. DAG ✓
2. Directed
3. Undirected

today: longest path (vertices cannot repeat) problem

Find the (length of the) longest path in G



Hamiltonian path:

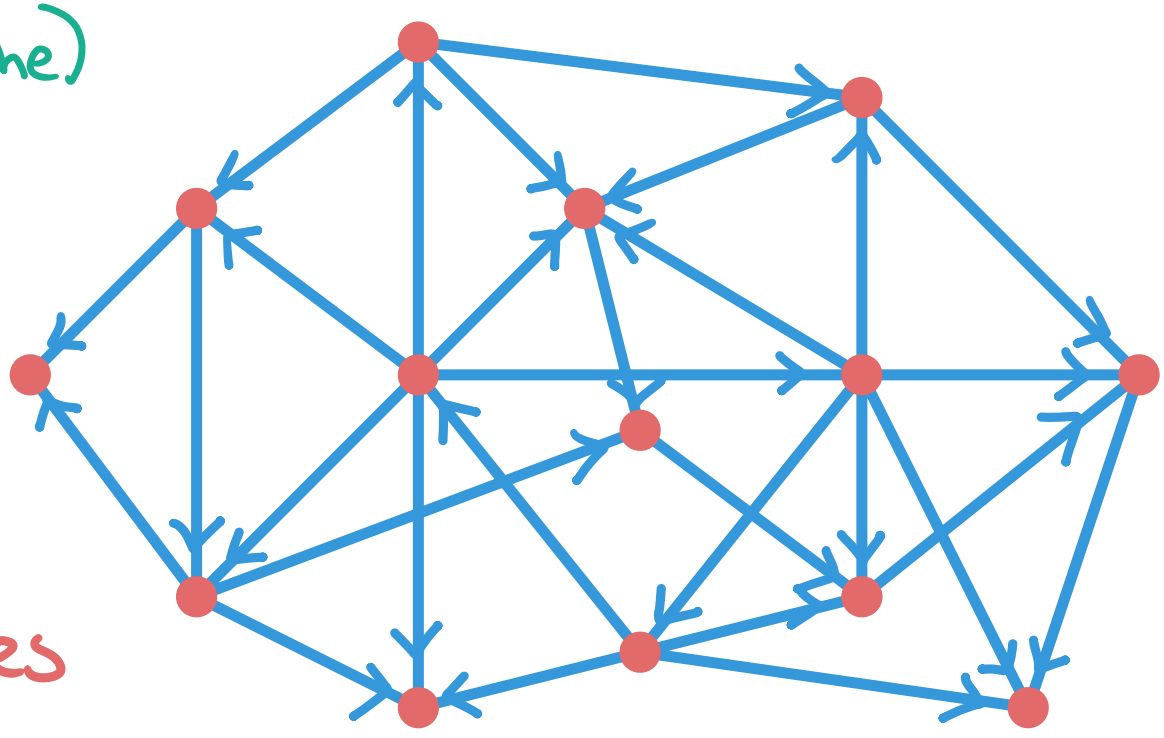
path visiting all vertices

does G have a Hamiltonian path?

	Polytime Algo	Hard as (CNF-) SAT	Abstain
DAG	✓		
Directed			
Undirected			

today: longest path (vertices cannot repeat) problem

Find the (length (eg. # edges) of the) longest path in G



Hamiltonian path:

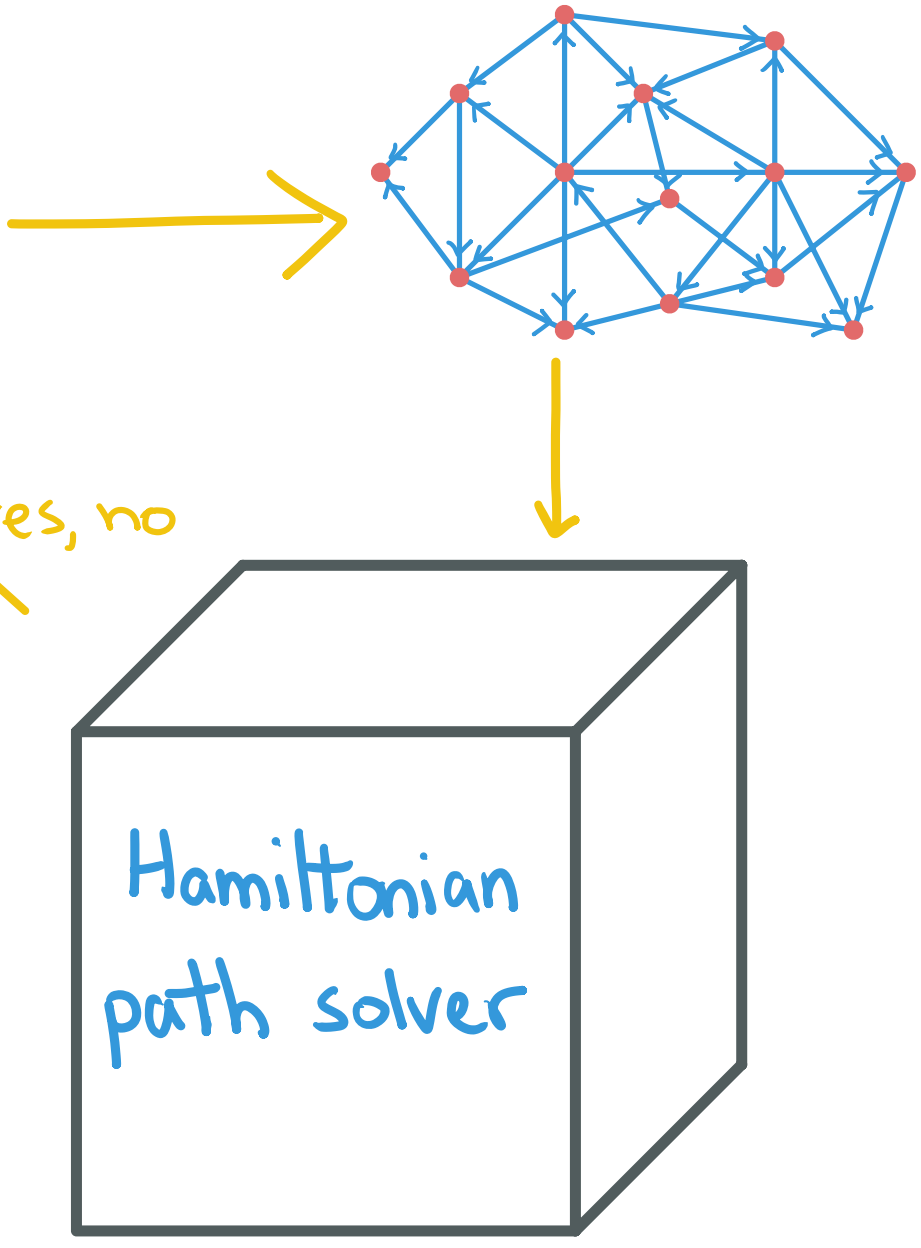
path visiting all vertices

does G have a Hamiltonian path?

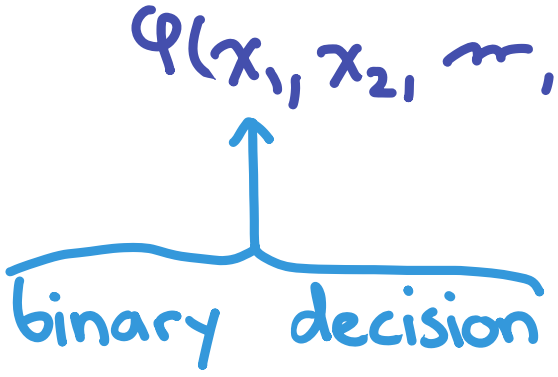
does G have a Hamiltonian path?

Theorem 7.1. *A polynomial time algorithm for the Hamiltonian path problem implies a polynomial time algorithm for SAT.*

$\varphi(x_1, \ldots, x_n) = (x_1 \vee \bar{x}_2) \wedge (x_2 \vee x_3 \vee \bar{x}_4) \wedge \cdots$



$$\varphi(x_1, x_2, \cdots, x_n)$$



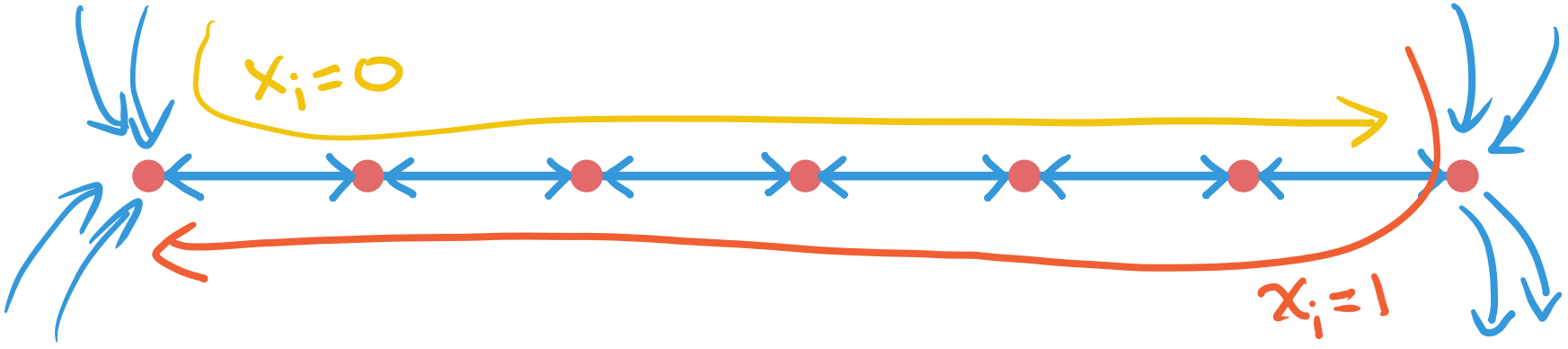
how to model $x_i \in \{0,1\}$ as a graph?

$\varphi(x_1, x_2, \dots, x_n)$

binary decision

model $x_i \in \{0,1\}$ as a graph?

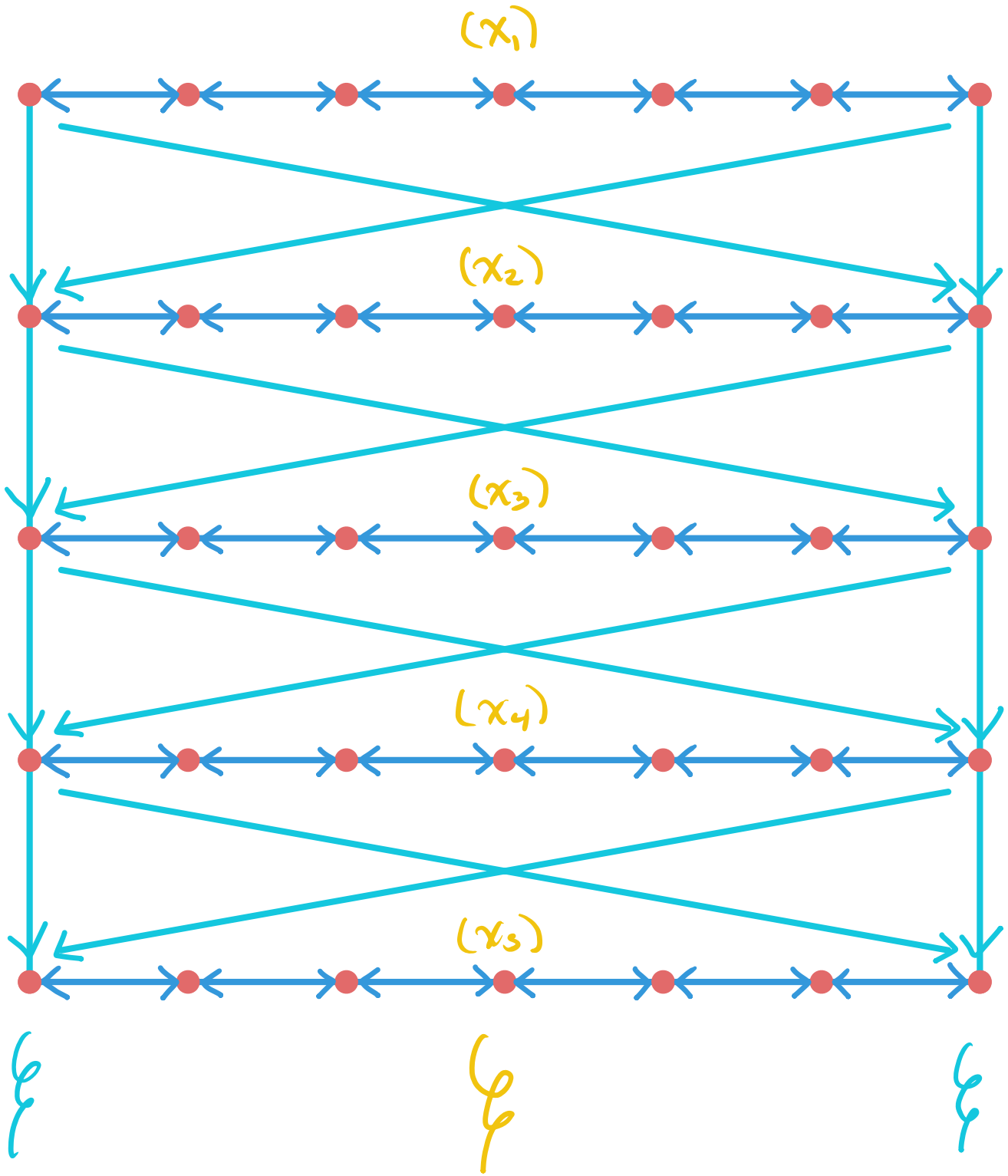
Long bidirected path

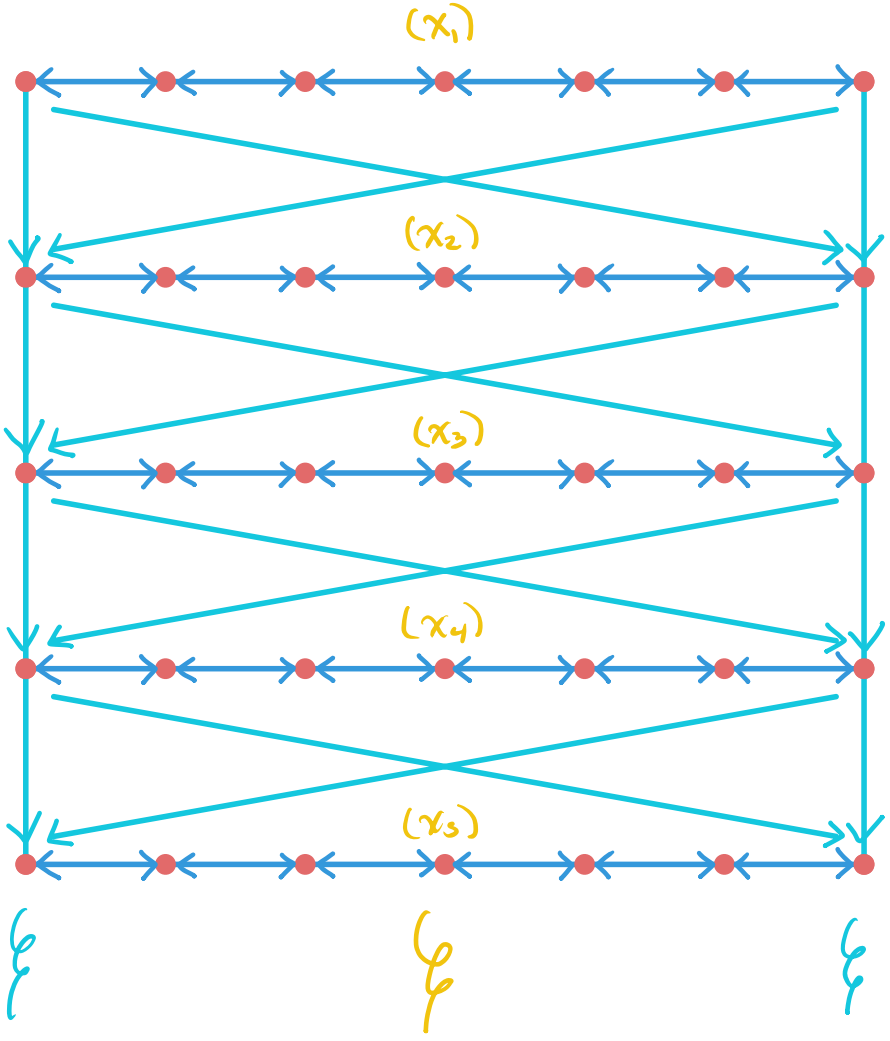


only way to visit all the vertices

is from left to right →

or from right to left ←


(x_1)
(x_2)
(x_3)
(x_4)
(x_5)



Now we can model choosing

$x_i = 0$ (left → right on x_i's path)

$x_i = 1$ (right → left on x_i's path)

within overall path

clauses?

e.g. $x_1 \vee \overline{x_2}$

recall we want:

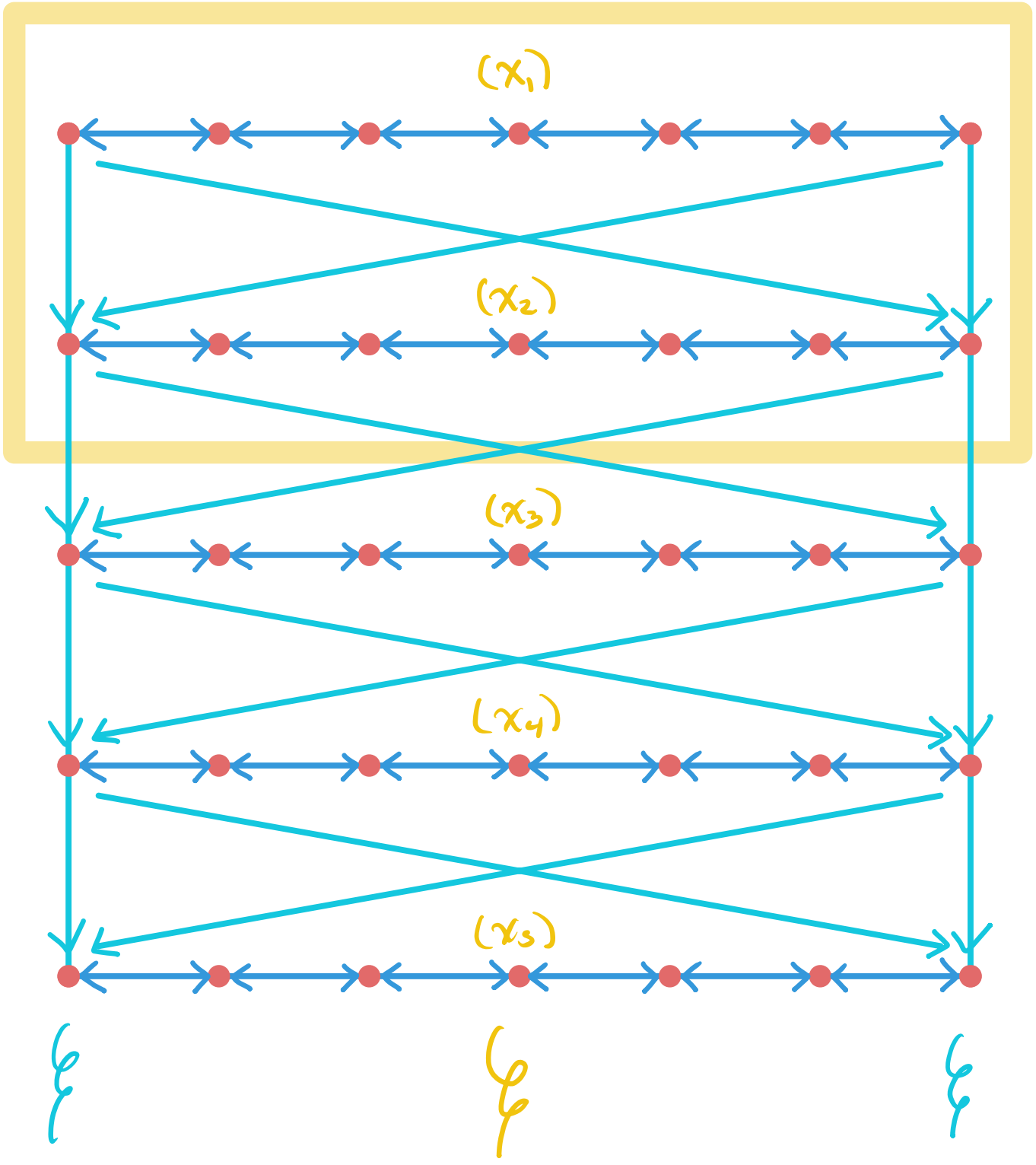
satisfy all clauses <=> path through all vertices

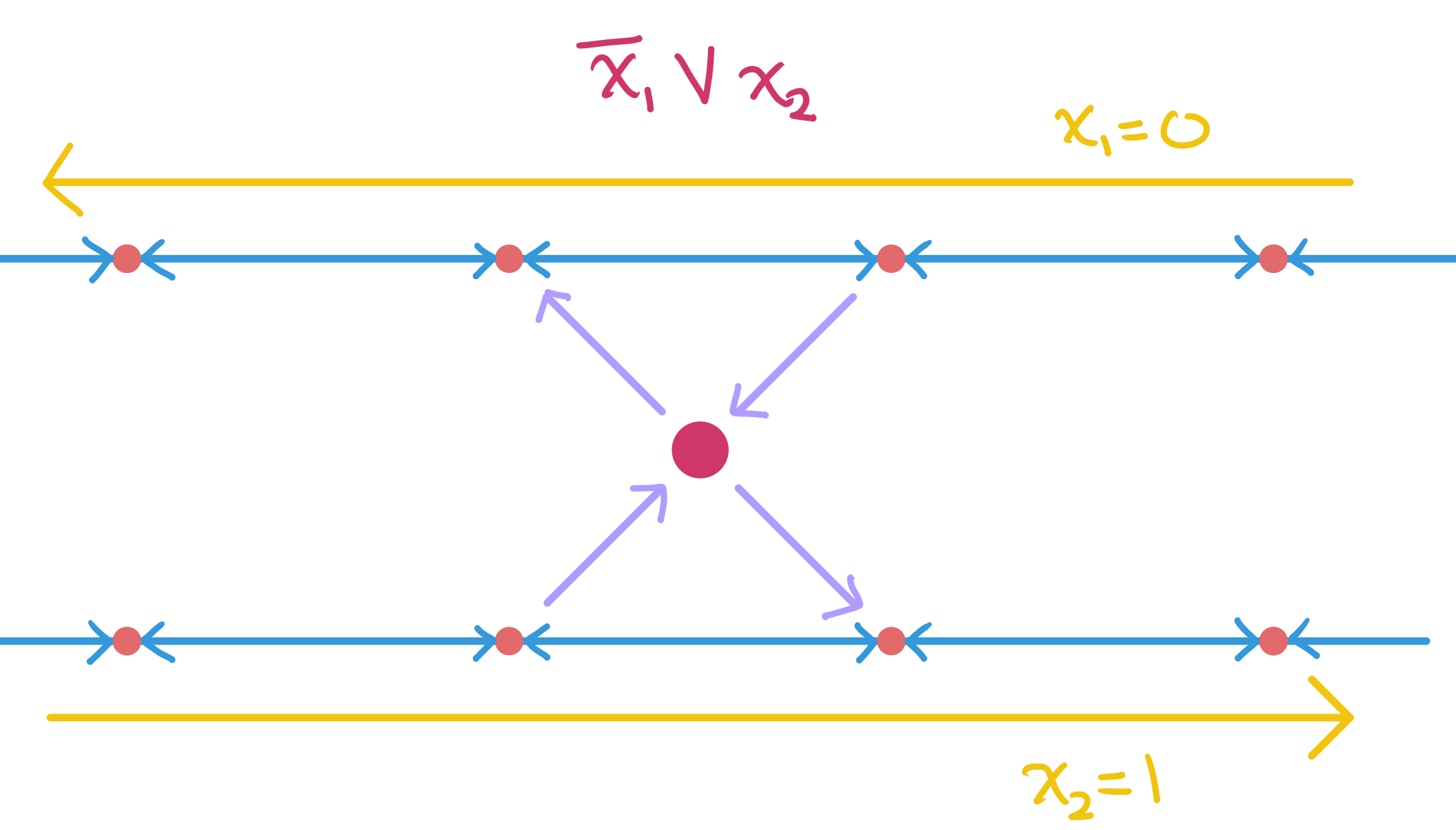
e.g. $x_1 \vee \overline{x_2}$

recall we want:

satisfy all clauses $\Leftrightarrow$ path through all vertices

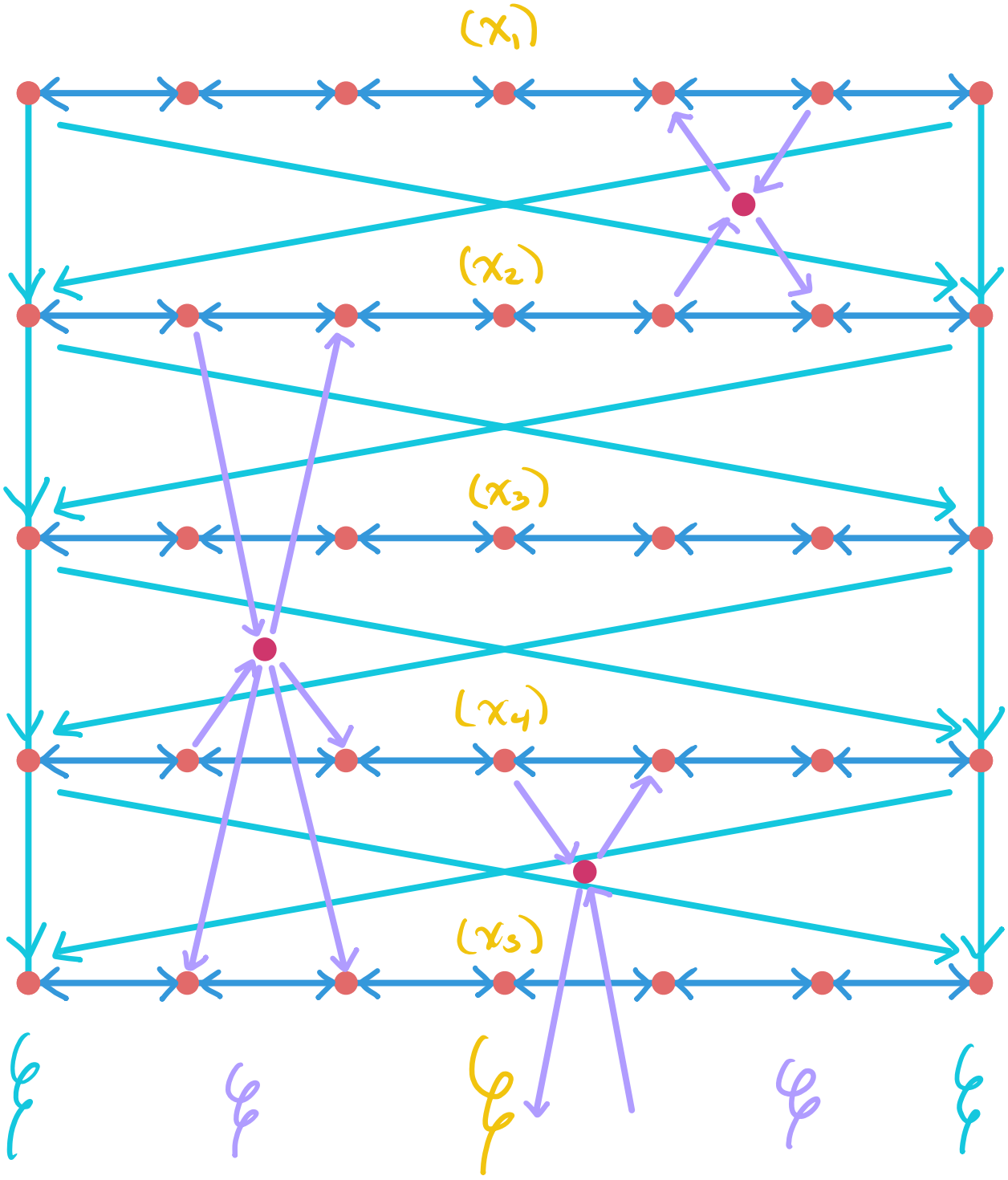
Idea: make a new vertex that we can visit iff we set $x_1 = 1$ or $x_2 = 0$


(x_1)
(x_2)
(x_3)
(x_4)
(x_5)



new vertex ● can be visited

iff we go in the $x_1 = 0$ direction or $x_2 = 1$ direction



Note

paths need to be long enough to accomodate all clauses (e.g., length m)

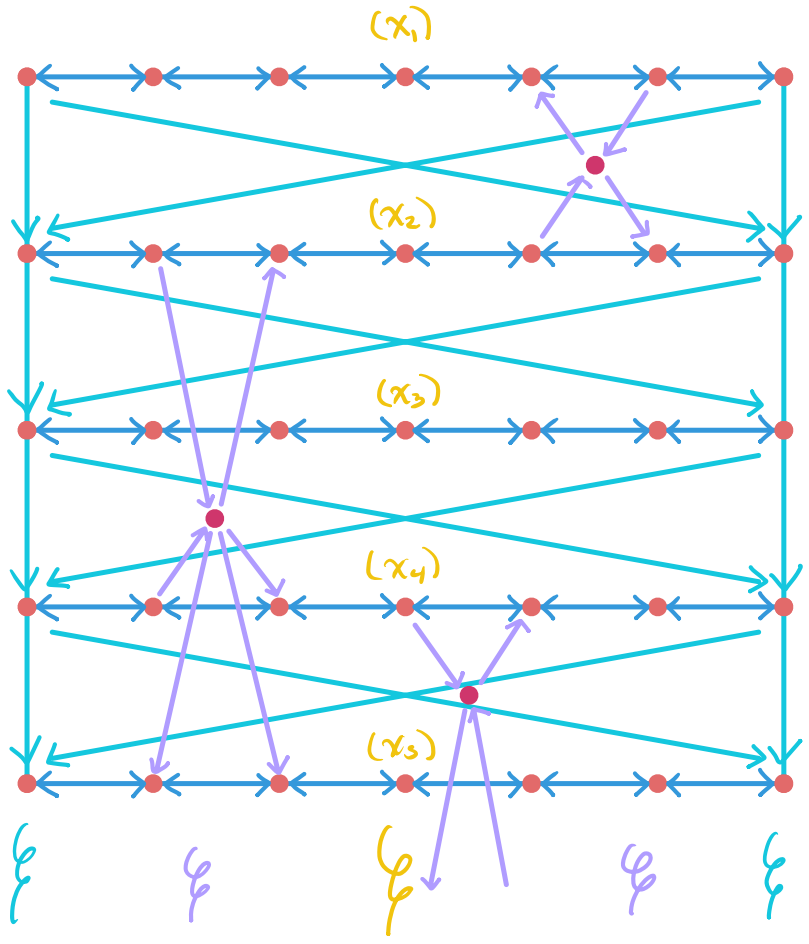
satisfy all clauses <=> path through all vertices

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

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

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

$\wedge \dots$

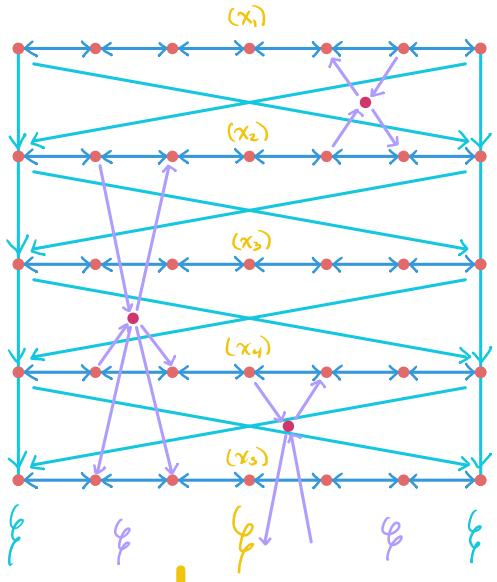


m clauses, n vertices

O(mn) vertices

Theorem 7.1. *A polynomial time algorithm for the Hamiltonian path problem implies a polynomial time algorithm for SAT.*

$$\varphi(x_1, \sim, x_n) = (x_1 \vee \bar{x}_2) \wedge (x_2 \vee x_3 \vee \bar{x}_4) \wedge \sim$$



yes, no

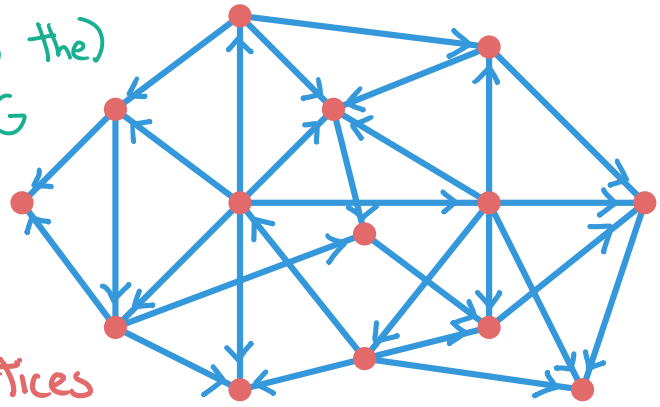
Hamiltonian path solver

Three settings:

1. DAG
2. Directed
3. Undirected

today: longest path (vertices cannot repeat) problem

Find the (length of the) longest path in G



Hamiltonian path:

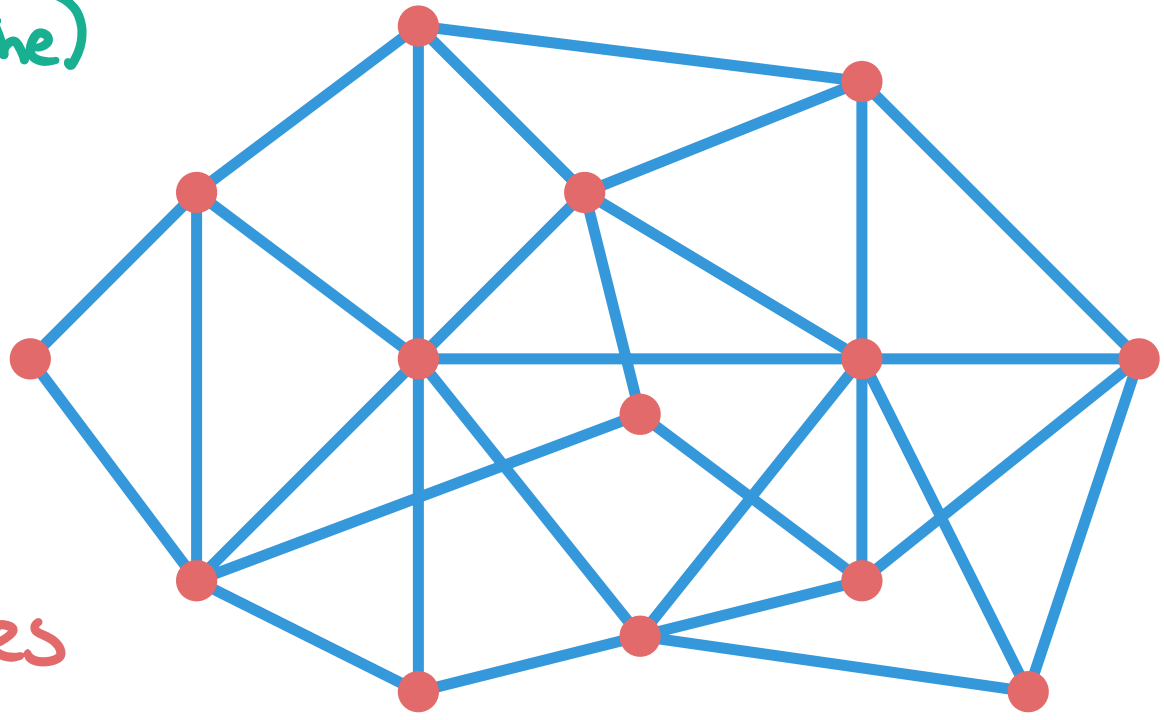
path visiting all vertices

does G have a Hamiltonian path?

	Polytime Algo	Hard as (CNF-) SAT	Abstain
DAG	✓		
Directed		✓	
Undirected			

today: longest path (vertices cannot repeat) in undirected graphs

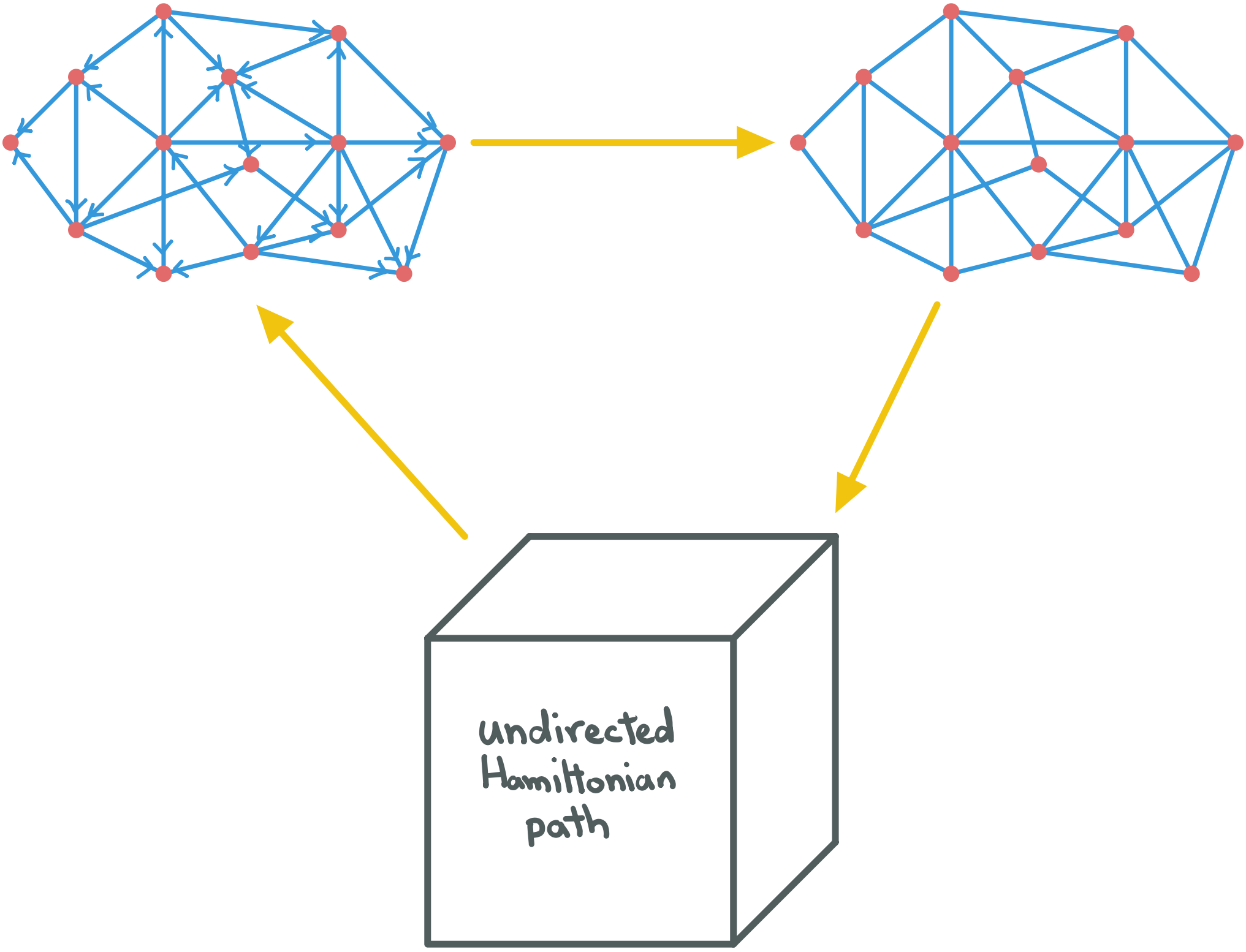
Find the (length of the) (eg. # edges) longest path in G

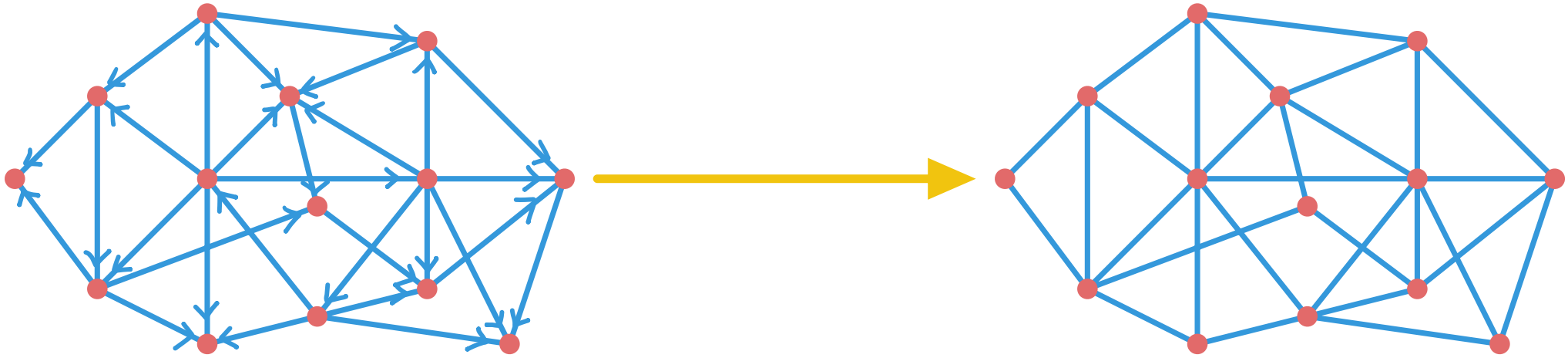


Hamiltonian path:

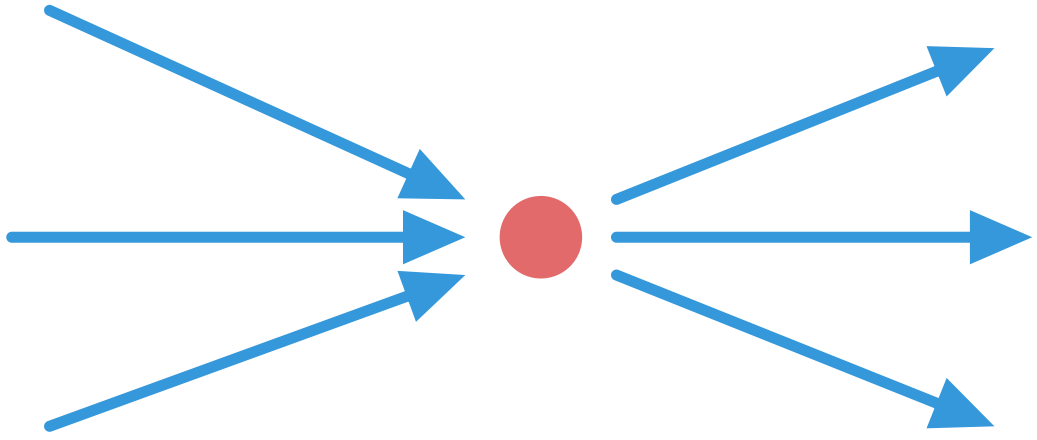
path visiting all vertices

does G have a Hamiltonian path?


Undirected
Hamiltonian
path



ideas?



Chapter 7

The longest path

Last chapter introduced directed graphs, and focused on questions concerning connectivity between vertices. Among the structural ideas that emerged were topological orderings in acyclic graphs, strongly connected components, and the relations between them. Algorithmically we investigated depth-first search and its applications, highlighted by a linear time algorithm to decompose the entire graph into its strongly connected components.

We continue to focus on directed graphs, and consider the problem of finding the longest path in G. (Recall that vertices cannot repeat in a path, by definition.) One can extend the problem to edge and vertex weights though we focus on the unweighted setting for simplicity. The decision version of the problem includes in the input an integer k, and asks if there is a path with at least k vertices. (The search version reduces to the decision version: see exercise 7.1.) An important special case of the decision question is where $k = n$. That is, does the graph contain a path traversing all the vertices? Such a path is called a **Hamiltonian path**; the "Hamiltonian path problem" is to decide if G has a Hamiltonian path.

We will consider the longest path problem in two settings. First we will consider the special case of DAG's, which are easier to work with. After that we will consider general graphs.

7.1 The longest path in a DAG

Let $G = (V, E)$ be a directed acyclic graph (DAG). Recall that DAG's were first introduced when discussing circuit-SAT (section 2.3), and played an important role in our discussion on DFS and strongly connected components (chapter 6). Our goal is to compute the longest path in G.

We encourage the reader to attempt the problem before proceeding.

The assumption that G has no cycles is very convenient because it implies that all walks in G are automatically paths. Indeed, vertices *cannot* repeat, since the repetition implies a cycle. Our problem, then, is the same as finding the longest walk in G; the constraint of avoiding repetition is irrelevant.

Let us develop a simple recursive algorithm. We want the length of the longest path/walk in G. (The actual longest path itself will follow easily) Where to start the path? Who knows – so let's throw the starting vertex in the recursive specification as a parameter. Thus let us define:

> `longest-walk(`v`)` = the length of the longest walk starting from v (as measured by the number of edges).

Given `longest-walk(`v`)` as defined above, we will return the maximum `longest-walk(`v`)` over all $v \in V$.

Now, to implement `longest-walk` as specified, we have the following. (Recall that a sink vertex is a vertex with no outgoing edges.)

$$\texttt{longest-walk}(v) = \begin{cases} 0 & \text{if } v \text{ is a sink} \\ 1 + \max_{(v,w) \in \partial^+(v)} \texttt{longest-walk}(v) & \text{otherwise.} \end{cases}$$

As with any recursive algorithm, we should first justify that this algorithm terminates in finite time – more precisely, that there are no potential cyclic dependencies in the recursive calls. Here we invoke the assumption that G is a DAG. Suppose we start at a vertex v. The vertices associated with recursive sequence calls will form a walk from v, and since G is a DAG, the walk cannot circle back to v. Another way to see this is to sort the vertices in topological order, with sources towards the front and sinks towards the bottom of the list. Then all arcs are directed down the list. Consequently the recursive calls from `longest-walk(`v`)` only go down the topological order – and never back up create an infinite loop.

Now, directly running the recursive algorithm above has a worst case running time of $O(n!)$ (see exercise 7.2). But suppose we cache our answers – that is, we apply dynamic programming – and visit each vertex only once. Excluding recursive subcalls, the time to compute `longest-walk(`v`)` is proportional to the out-degree of v (denoted $\deg^+(v)$). Summing $\deg^+(v)$ over all v counts every edge exactly once, so the total running time is

$$O\left(\sum_{v \in V} 1 + \deg^+(v)\right) = O(m + n).$$

So the longest path problem can be solved in linear time in a DAG.

Of course this only gives the maximum length of any path. We can extract the actual path as follows. For each v, we record the vertex w following v on the

longest walk from v. We can then recover the longest walk from v by following these starting from v. (This is essentially the same idea, used generically in other dynamic programming situations, where the optimal choice for each subproblem is stored in a secondary table, in order to reconstruct the object attaining the optimum value afterwards.)

Stepping back, this problem highlights a vital algorithmic feature of DAG's, which is that we can apply the ideas of dynamic programming to DAG's, using the topological order to guide the induction.

7.2 The longest path in general graphs

Having obtained a (rather simple) linear time algorithm for the longest path problem in DAG's, we consider the problem in a general directed graph G. In particular, G may have cycles. As remarked above, the recursive algorithm may not terminate when there are cycles. Can we somehow salvage the approach?

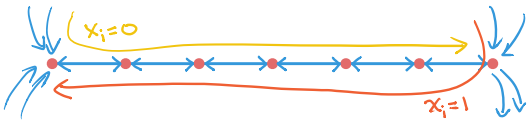
The following theorem indicates that the longest path problem – and even the special case of Hamiltonian path – is much harder than in DAG's.

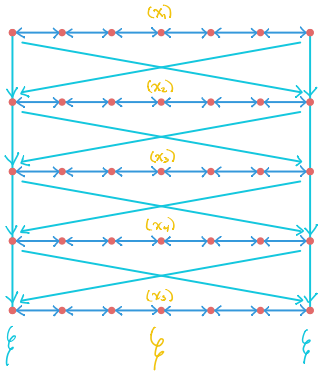
Theorem 7.1. *A polynomial time algorithm for the Hamiltonian path problem implies a polynomial time algorithm for SAT.*

The rest of this section is devoted to proving theorem 7.1.

We take as input a SAT formula $f(x_1, \dots, x_n)$ in CNF with m clauses. Based on f, we construct a directed and unweighted graph G with M edges and N vertices, where M and N are TBD. It will have the feature that f is satisfiable iff G has a Hamiltonian path. We first sketch out the main ideas at a high level. More specific details (like exactly how many vertices of some type) will be supplied at the end once the high level ideas are in place.

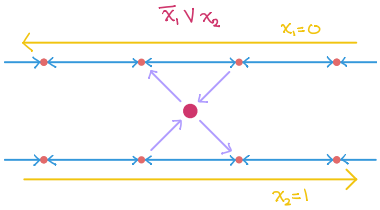
Speaking informally, for each variable x_j, we want to encode the choice of x_j (whether $x_j = \texttt{true}$ or $x_j = \texttt{false}$) as a graph. One way to do this is via a long, bidirected path for each x_j. A Hamiltonian path can only traverse the path in one of the two directions. So we can interpret one direction as setting $x_j = \texttt{true}$ and the other as setting $x_j = \texttt{false}$.





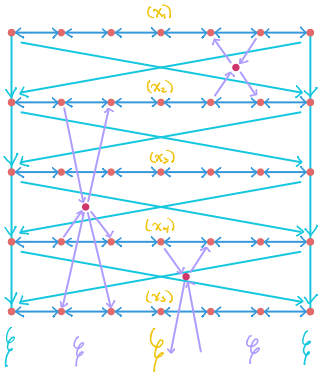
We arrange these paths, one for each x_i, in a "ladder"-like structure, strung together by edges connecting the endpoints. In particular we will add all four directed combinations from the endpoints of the path of one variable x_k to the endpoints of the path of the next variable x_{k+1}. Then any Hamiltonian path must commit to going left or right along each rung of the ladder as it works its way from the top to the bottom.

We have not yet encoded any clauses. To this end we have the following gadget. Let us (arbitrarily) fix, for each vertex x_j, the left to right direction as denoting $x_j = \texttt{true}$, and the right to left direction as denoting $x_j = \texttt{false}$. We want to design a gadget for each clause that is "free" when we travel along the direction of a vertex that corresponds to satisfying the clause. To be specific, fix a clause C_i (where $i \in [m]$). We introduce a new vertex c_i for that clause. For each variable x_j that appears in C_i as x_j (as opposed to $\bar{x}_j$), pick two consecutive (unused) nodes along x_j's path. In the left to right direction (corresponding to $x_j = \texttt{true}$), add an edge from the first of these vertices to the clause-vertex c_i, and an edge from the clause-vertex c_i to the second of these vertices. For each variable x_j that appears in C_i in negated form $\bar{x}_j$, we do something similar except in the opposite direction. We pick two consecutive unused nodes along x_j's path. Moving in the right to left direction (corresponding to $x_j = \texttt{false}$), we add an edge from the first of these vertices to the clause-vertex c_i, and an edge from the clause-vertex c_i to the second of these vertices. See the picture below.



Any Hamiltonian path has to visit each clause vertex c_i. The construction is such that if a solution is already traversing the path of some x_j in the direction that satisfies C_i, then it can detour and visit that clause along the way.

The picture on the right sketches the construction which now decorates the "ladder" of paths for each x_j with clause gadgets connecting different paths. A Hamiltonian path will start from one of the two



corners and weave through the ladder down to the bottom, choosing one of the two directions for each step.

This finishes the high-level description of the algorithm. We now describe the construction formally for the sake of completeness.

1. For each variable x_j, we create two variables s_j and t_j, which will be opposite ends of the path for x_j.
2. For each variable x_j (where $j \in [n]$), and each clause C_i (where $i \in [m]$), we create two variables $a_{j,i}$ and $b_{j,i}$. For each i, $a_{j,i}$ and $b_{j,i}$ are reserved for a possible gadget for clause C_i. We also have $m+1$ addition vertices $d_{j,1}$ that will act as dividers.
3. For each variable x_j, consider the sequence

   $$(s_j, d_{j,0}, a_{j,1}, b_{j,1}, d_{j,1}, a_{j,2}, b_{j,2}, d_{j,2}, \ldots, d_{j,m-1}, a_{j,m}, b_{j,m}, d_{j,m}, t_j),$$

   where the $\ldots$ iterates though the vertices $a_{j,i}, b_{j,i}, d_{k,i}$ and in increasing order of i. We add directions in both directions between every consecutive pair of edges. Thus creates a "rung".
4. For $j = 1, \ldots, n$, we add all 4 combinations of directed edges from s_j and t_j to s_{j+1} and t_{j+1}. For $j = n$, we add 4 directed edges from $v_{n,0}$ and $v_{n,m}$ to $v_{1,0}$ and $v_{1,m}$. This strings the rungs together to form a "ladder".
5. For each clause C_i, we create a vertex c_i.
6. For each clause C_i, and each literal of the form x_j in C_i, we add directed edges from $x_{j,i-1}$ to c_i and from c_i to $x_{j,i}$. For each literal of the form $\bar{x}_j$ in C_i, we add directed edges from $x_{j,i}$ to c_i and from c_i to $x_{j,i-1}$.

A helpful fact about the construction is that to avoid revisiting any vertex, one has to traverse all the vertices corresponding to a variable x_j in one direction or the other. The dividers make it easier to prove this and we leave the proof to the reader.

We claim that f is satisfiable iff and only G has a Hamiltonian. First of all, given a satisfying assignment $x \in \{0,1\}^n$, we obtain a Hamiltonian path by first using each x to decide in which direction to traverse the path for each x_j. Each clause vertex can be visited as we traverse the path of the satisfying vertex. Conversely, given a Hamiltonian path - which, in particular, traverses each path gadget in a single direction - we obtain a satisfying assignment by mapping the directions to `true`/`false` assignments.

This completes the proof of theorem 7.1.

7.3 Exercises

Exercise 7.1. Show how to use a polynomial time algorithm for the decision version of the longest path problem ("Is there a path of length k?") gives a polynomial time algorithm for the search version ("Find the longest path in the graph.").

Exercise 7.2. Recall the recursive algorithm for longest path in DAG's.

1. Describe a DAG with n vertices where the recursive algorithm (without dynamic programming) would take $O(n!)$ time.
2. More generally, show that for any integers $k \leq n$, there is a DAG with n vertices such that
   (a) Every vertex v has out-degree $\leq k$.
   (b) The recursive algorithm would take $(n-k)^k(k-1)!$ on this input.

Exercise 7.3. A *Hamiltonian cycle* is a cycle that visits every vertex exactly once. Consider the problem of deciding whether a given directed graph has a Hamiltonian cycle. For this problem, 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.

Exercise 7.4. While we showed that Hamiltonian path is unlikely to permit polynomial time algorithms, perhaps it is still interesting to try to develop faster exponential time algorithms. The simplest approach enumerates all permutations of the vertices and checks to see if they describe a Hamiltonian cycle; this takes $O\big(n!\,\mathrm{poly}(n)\big)$ time. Can one do better? Design and analyze an algorithm that computes a Hamiltonian path in $O\big(2^n\,\mathrm{poly}(n)\big)$ time. (The smaller the $\mathrm{poly}(n)$ term the better, but the key point is to reduce $n!$ to 2^n.)

Exercise 7.5. The following two problems consider variations for the longest path problem that instead ask for longest walks (with varying definitions of "longest"). Here we recall that a walk may repeat vertices and edges, whereas a path cannot. For both problems, either design and analyze a polynomial time algorithm, or prove that a polynomial time algorithm for the problem implies a polynomial time algorithm for SAT.

1. Given a directed graph G, compute a walk that visits the greatest number of distinct vertices.
2. Given a directed graph G, compute a walk that traverses the greatest number of distinct edges.

Exercise 7.6. Let $G = (V, E)$ be a directed graph with vertex weights $w : V \to \mathbb{R}$. We say that the *vertex weight of a walk* going through vertices $v_1, v_2, \ldots, v_k$ (in order, with vertices possible repeating) is the sum weight $w(v_1) + w(v_2) + \cdots + w(v_k)$.

Let $k \in \mathbb{N}$ be a given parameter with $k \leq n$. Consider the problem of computing the minimum vertex weight of any walk with exactly k vertices. For this problem, 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.

Exercise 7.7. Let $G = (V, E)$ be a directed graph with m edges and n vertices, where each vertex $v \in V$ is given an integer label $\ell(v) \in \mathbb{N}$. The goal is to find the length of the longest path[1] in G for which the labels of the vertices are strictly increasing.

1. (7 points) Suppose G is a DAG. For this problem, either (a) design and analyze a polynomial time algorithm, or (b) prove that a polynomial time algorithm implies a polynomial time algorithm for SAT.

2. (3 points) Consider now the problem for general graphs. Either (a) design and analyze an polynomial time algorithm, or (b) prove that a polynomial time algorithm implies a polynomial time algorithm for SAT.

Exercise 7.8. Let $G = (V, E)$ be a directed graph where each edge is colored `red`, `white`, or `blue`. An American path is a path where the edge colors alternate `red`, `white`, `blue`. The American Hamiltonian path problem is to decide if there is an American path that visits all the vertices. For this problem, 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.

Exercise 7.9. Consider a directed graph $G = (V, E)$ that is **transitively closed**. That is, a vertex u can reach a vertex v iff there is an edge from u to v.

Consider the problem of finding a Hamiltonian path in a transitively closed graph. That is, the input consists of a transitively closed graph, and the goal is to find a Hamiltonian path in the graph.

Either (a) show that this problem is NP-Hard, or (b) design and analyze an efficient algorithm for this problem.

Exercise 7.10. The following problem (and more elaborate extensions) appear in reinforcement learning. Let $G = (V, E)$ be a directed graph and let the edges be annotated by positive edge weights $r : E \to \mathbb{R}_{>0}$. We think of the vertices as

[1] Recall that a path is a walk that does not repeat vertices.

states of some device under our control, and the edge weights $r(e)$ as rewards obtained by traversing the edge e as follows. Let $s \in V$ be a fixed starting vertex / state.

1. Given an integer $k \leq n$, the goal is to compute a walk of length k maximizing the sum of rewards along that walk. (If you repeat an edge, you get the same reward each time you repeat the edge.) For this problem, 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.

2. Here we also incorporate a *discount rate*. Let $\alpha \in (0,1)$ be given. Given a walk with edges $e_1, \ldots, e_k$, the *discounted total reward* of the walk is given by

$$r(e_1) + \alpha r(e_2) + \cdots + \alpha^{k-1} r(e_k).$$

   The idea is that if we think of each edge traversal as also taking a unit of time, then the rewards attained far off in the future are worth less than the rewards attained now.

   Given an integer $k \leq n$, the goal is to compute a walk of length k maximizing the discounted total reward of the walk. For this problem, 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.