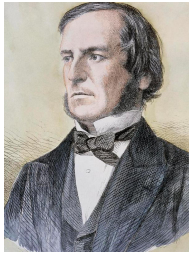


# The Longest Path

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"  $\bar{x}$  (or  $\neg x$ )



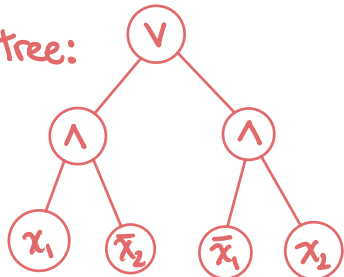
George Boole,  
1815-1864

XOR ("exclusive-or")

(TRUE iff  $x_1 \neq x_2$ )

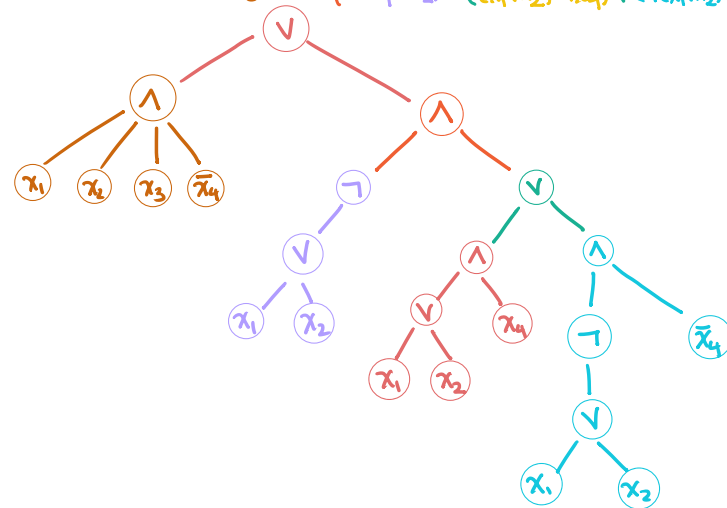
$$f(x_1, x_2) = (x_1 \wedge \bar{x}_2) \vee (\bar{x}_1 \wedge x_2)$$

as a tree:



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

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



"SAT Programming Language"

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

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

★ "if x then y else z"  $(x \wedge y) \vee (\bar{x} \wedge z)$

So Boolean algebra is very expressive

Boolean Satisfiability

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

• n variables • "size" m

e.g.  $\varphi(0, 1, 0, 0, \neg, 1, 1) = 0$

Goal: decide if  $\varphi$  is "satisfiable";

i.e., if there exists  $x_1, x_2, \dots, x_n \in \{0, 1\}$   
 s.t.  $\varphi(x_1, x_2, \dots, x_n) = 1$

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

- $n$  variables
- "size"  $m$

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

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

"x and y"  $x \wedge y$

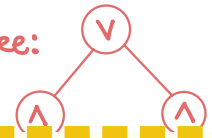
"x or y"  $x \vee y$

"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:



Goal: The Big Question

Is there a polynomial time algorithm for Boolean satisfiability?

$x_2$

$x_2 = x_3 x_4$

wise

$x_3 \wedge \bar{x}_4$

$x_2$

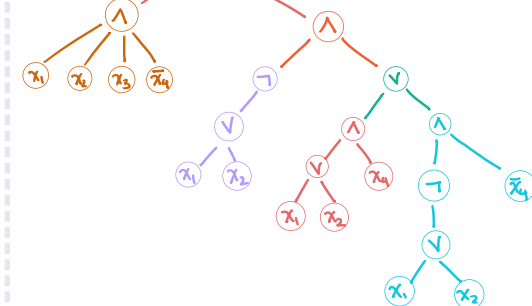
$x_2 \wedge x_4$

$(x_1 \vee x_2) \wedge \bar{x}_4$

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"



"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"

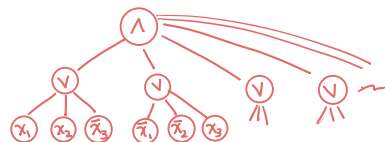
"(x=1, y=1) or (x=0, z=1)"  
 $(x \wedge y) \vee (\bar{x} \wedge z)$

## CNF (Conjunctive Normal Form)

"and-of-or's"

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

$$(\bar{x}_1 \vee x_2 \vee \bar{x}_3) \wedge (\bar{x}_1 \vee \bar{x}_2 \vee x_3) \wedge (x_1 \vee x_2) \wedge \dots$$



(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"

$$\bar{x} \quad (\text{or } \neg x)$$

★ "z = x"

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

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

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

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

★ "if x then y else z"

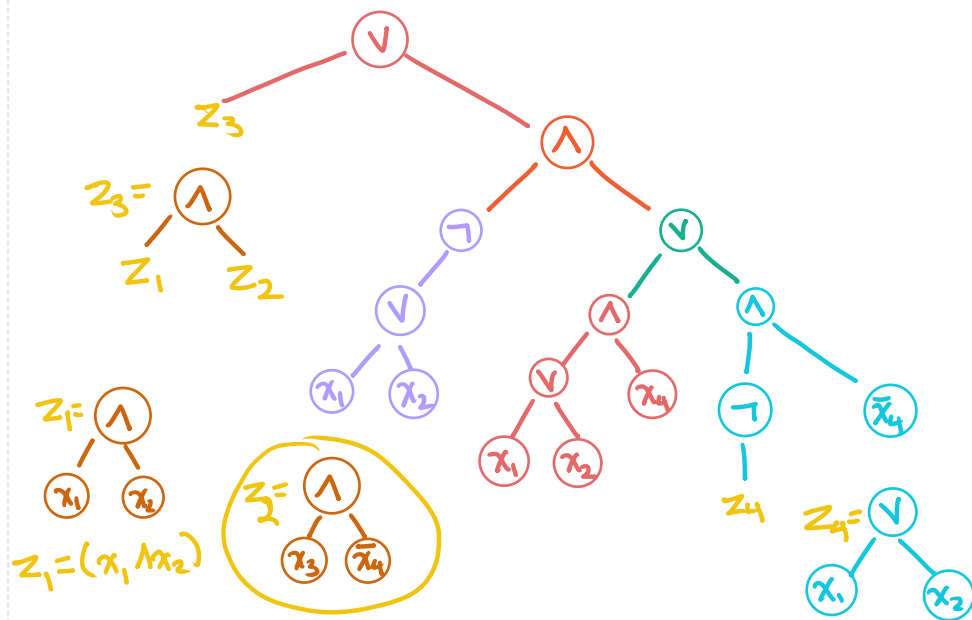
$$(x \wedge y) \vee (\bar{x} \wedge z)$$

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

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

So CNF is also very expressive

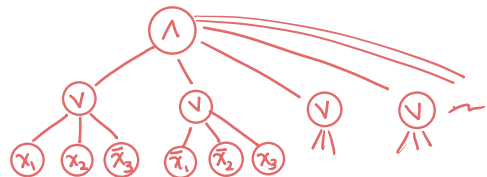
Theorem: there is a poly-time algo for Boolean SAT iff there is a poly-time algo for CNF-SAT.





more special case: **3-SAT**

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



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

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

"clause"

Is there a polytime algo for 3SAT?

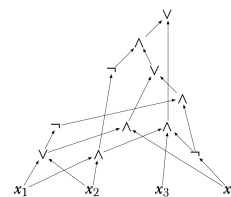
Theorem: there is a poly-time algo for Boolean SAT iff there is a poly-time algo for 3-SAT.

Conclusion

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

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

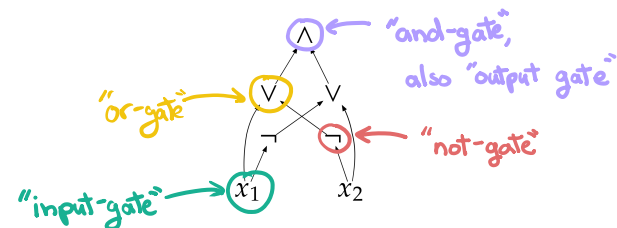
But we do know they are equivalent up to polynomial time!



(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

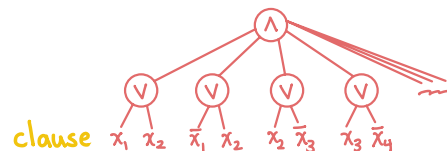
Theorem: there is a poly-time algo for Boolean SAT iff there is a poly-time algo for Circuit SAT

## 2-SAT

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

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

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



## 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)  $C = \text{true}$ ,  $\bar{C} = \text{false}$

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

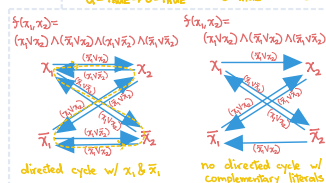
Key idea: assigning sinks to be true,  
sources to be false  
does not violate any edges

Implication graph

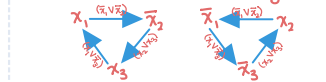
• vertex for each  $x_i, \bar{x}_i$

• for clause  $(a \vee b)$ ,

$\bar{a} \xrightarrow{(a \vee b)} b$   $\bar{b} \xrightarrow{(a \vee b)} a$   
"a = TRUE  $\Rightarrow$  b = TRUE" "b = TRUE  $\Rightarrow$  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 SAT<sup>(CNF)</sup>

Abstain

2SAT

66%

44%

2

Max 2-SAT

60%

35%

5

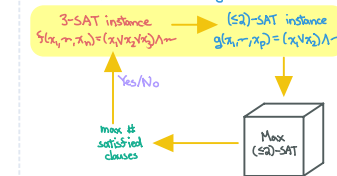
Fix a clause  $(a \vee b \vee c)$   
introduce new variable  $y$

consider the 10 clauses

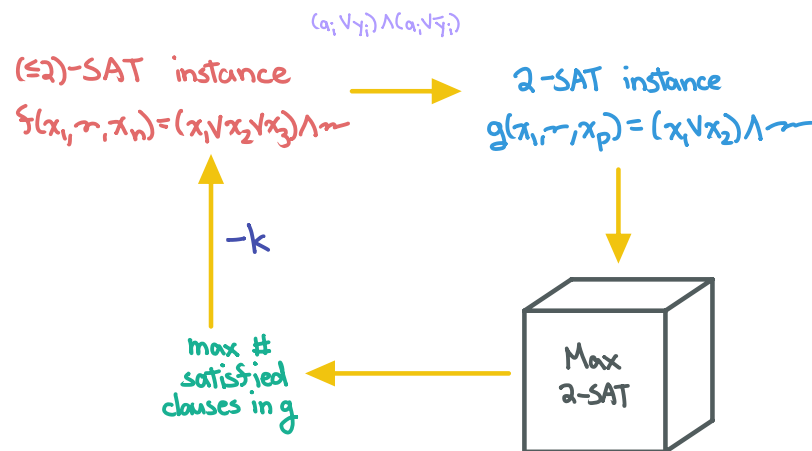
$(a), (b), (c), (\bar{a} \vee \bar{b}), (\bar{b} \vee \bar{c}), (\bar{c} \vee \bar{a}),$   
 $(y), (a \vee \bar{y}), (b \vee \bar{y}), (c \vee \bar{y})$

| $(a, b, c) =$                                                                     | $(f, f, f)$ | $(t, f, f)$ | $(t, t, f)$ | $(t, t, t)$ |
|-----------------------------------------------------------------------------------|-------------|-------------|-------------|-------------|
| $(a), (b), (c)$                                                                   | 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 (2,2)-SAT implies a polynomial time algorithm for SAT.



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



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 SAT (CNF-) | Abstain |
|------------|---------------|--------------------|---------|
| Subset Sum | 10%           | 30%                | ○       |

★Dynamic Programming★

Recursive Spec  
(Subset Sum)  
SS(i, S) = TRUE if there is a subset of  $x_1, \dots, x_n$  summing to S, and FALSE otherwise

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

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<sub>i</sub>)

Running time:  
O(nT) subproblems  
O(1) time per subproblem  
O(nT) time overall

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

Literals  $x = \{x_1, \bar{x}_1, \dots, x_n, \bar{x}_n\}$  ↔ Long #'s  $\{A_1, \bar{A}_1, \dots, A_m, \bar{A}_m\}$  and  $\{B_1, \bar{B}_1, \dots, B_n, \bar{B}_n\}$   
T = 33 ~~~ 3 1 1 ~~~ 1  
m clauses n variables  
3-SAT instance  $\phi(x_1, \dots, x_n) = (x_1 \vee x_2 \vee x_3) \wedge \dots$   
Yes/No for  $x_1, x_2, \dots, x_p \in \mathbb{N}$  T ∈ N  
Subset Sum  
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  $x_j$ :

$A_j = \underbrace{0 \ 1 \ 01 \dots 1}_{m \text{ slots for each clause}} \underbrace{0 \ 0 \dots 01 \dots 0}_{n \text{ slots for each variable}}$   
(models)  $x_j$

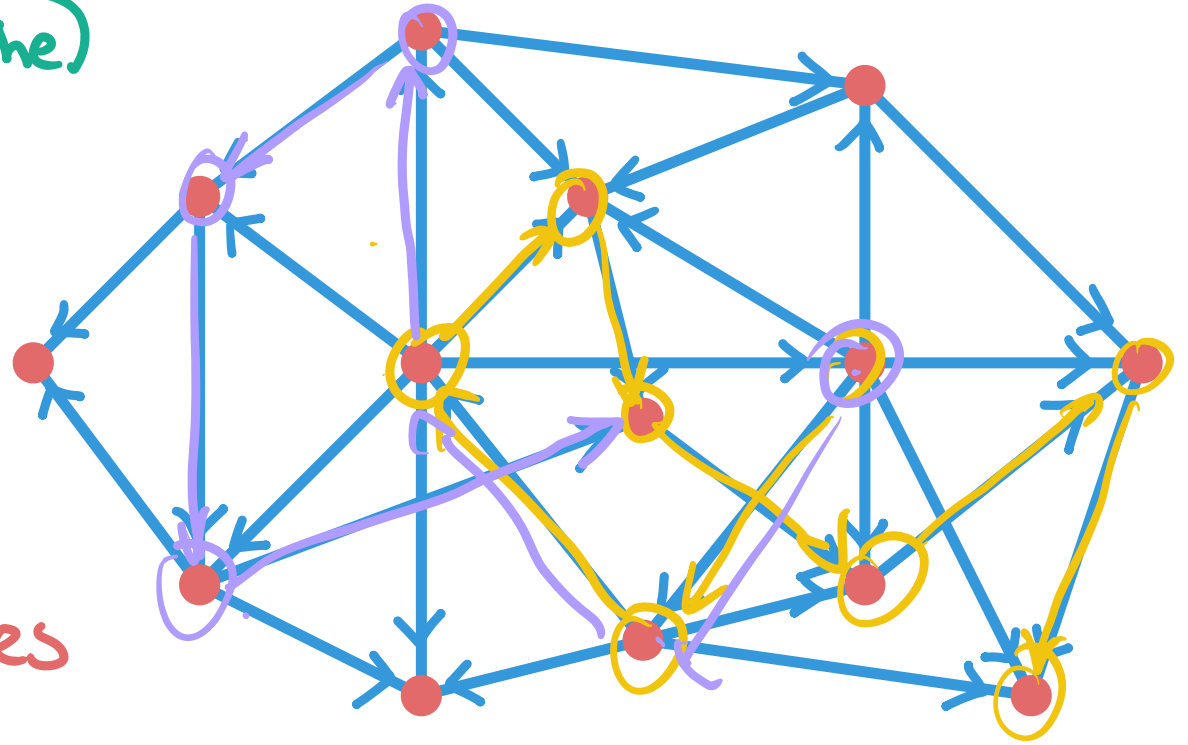
1. Recast 3SAT as "subset" problem  
 $X = \text{literals } \{x_1, \bar{x}_1, x_2, \bar{x}_2, \dots, x_n, \bar{x}_n\}$   
clause  $(x_1 \vee \bar{x}_2 \vee x_3) = \text{set } \{x_1, \bar{x}_2, x_3\} \subset X$   
Goal: Find SCX sit.  
ⓐ for each clause  $C_i, C_i \cap S \neq \emptyset$   
ⓑ for each  $x_j, S$  contains 1 of  $\{x_j, \bar{x}_j\}$  (not  $\bar{x}_j$ )

• for each clause  $C_i$ , set  $C_i$ th digit to 1 if  $x_j \in C_i$ , else 0  
• set " $x_j$ th digit" to 1, else 0

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

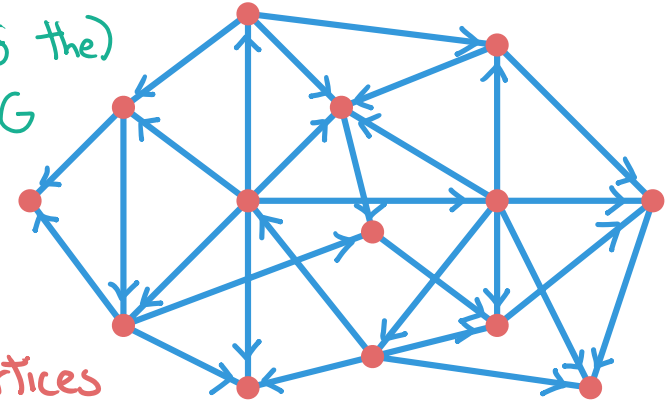
3. Undirected

today: longest path problem (vertices cannot repeat)

Find the (length of the)  
longest path in  $G$

Hamiltonian path:

path visiting all vertices



does  $G$  have a Hamiltonian path?

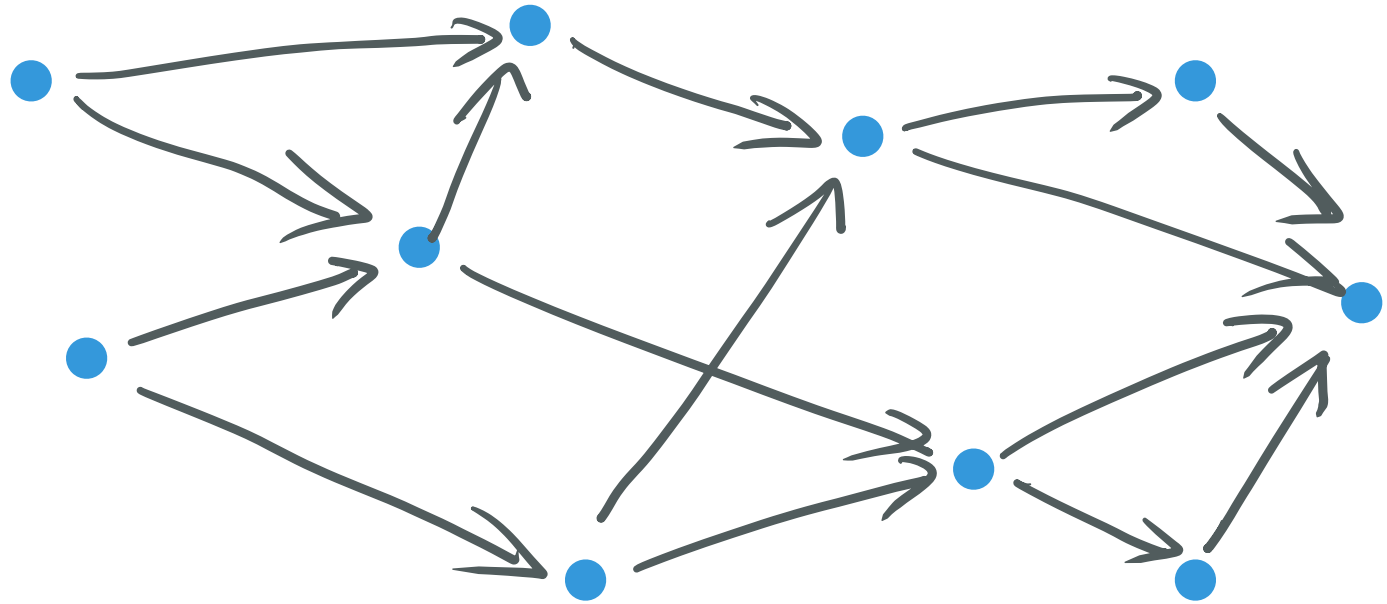
|            | Polytime Algo | Hard as SAT <small>(CNF-)</small> | Abstain |
|------------|---------------|-----------------------------------|---------|
| DAG        | 10            | 2                                 |         |
| Directed   | 5             | 1                                 |         |
| Undirected | 3             | <del>1</del> 9                    |         |

walk

Longest ~~path~~ in a DAG

no cycles

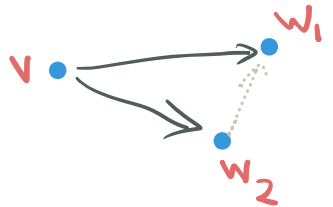
$\Rightarrow$  walk = path



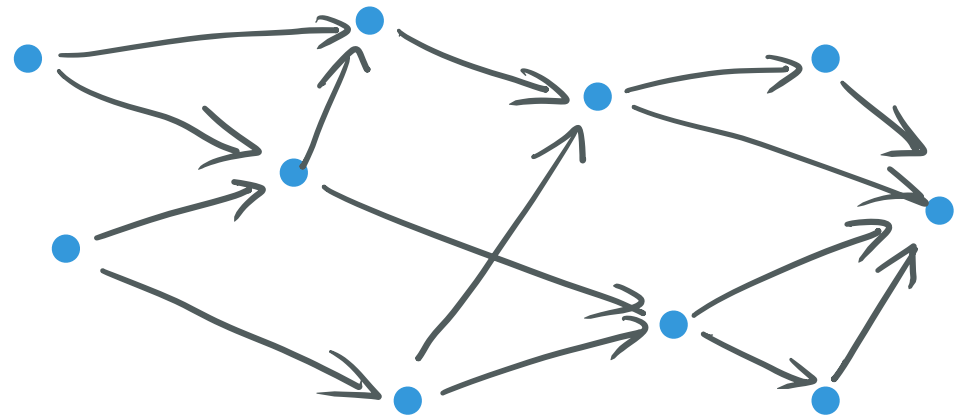
Recursion

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

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



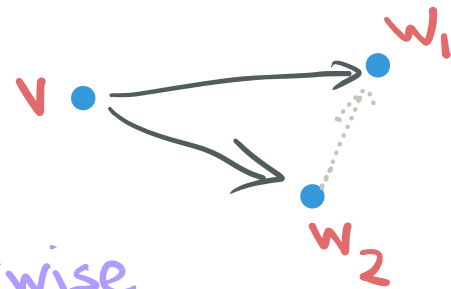
# Longest **walk** in DAGs



## Recursion

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

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



## Running time w/ caching

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

# Dynamic programming in DAGs

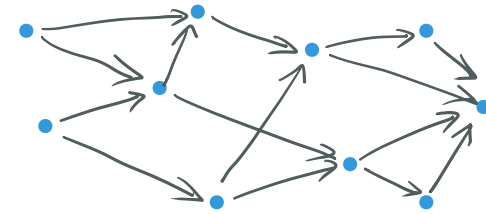
topological order

$\Rightarrow$  induction, recursion

dynamic programming, caching  $\Rightarrow$   
efficient (even linear time) algos

Longest <sup>walk</sup>~~path~~ in a DAG

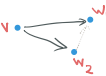
no cycles  
 $\Rightarrow$  walk = path



Recursion

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

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



Running time w/ caching

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

Many hard problems become much easier in DAGs



Three settings:

✓ 1. DAG

2. Directed

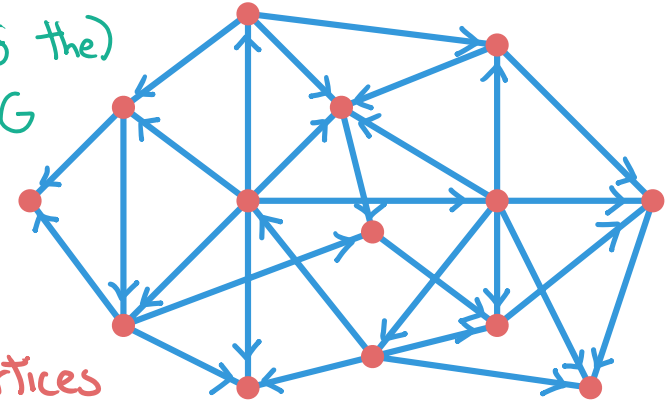
3. Undirected

today: longest path problem (vertices cannot repeat)

Find the (length of the)  
longest path in  $G$

Hamiltonian path:

path visiting all vertices



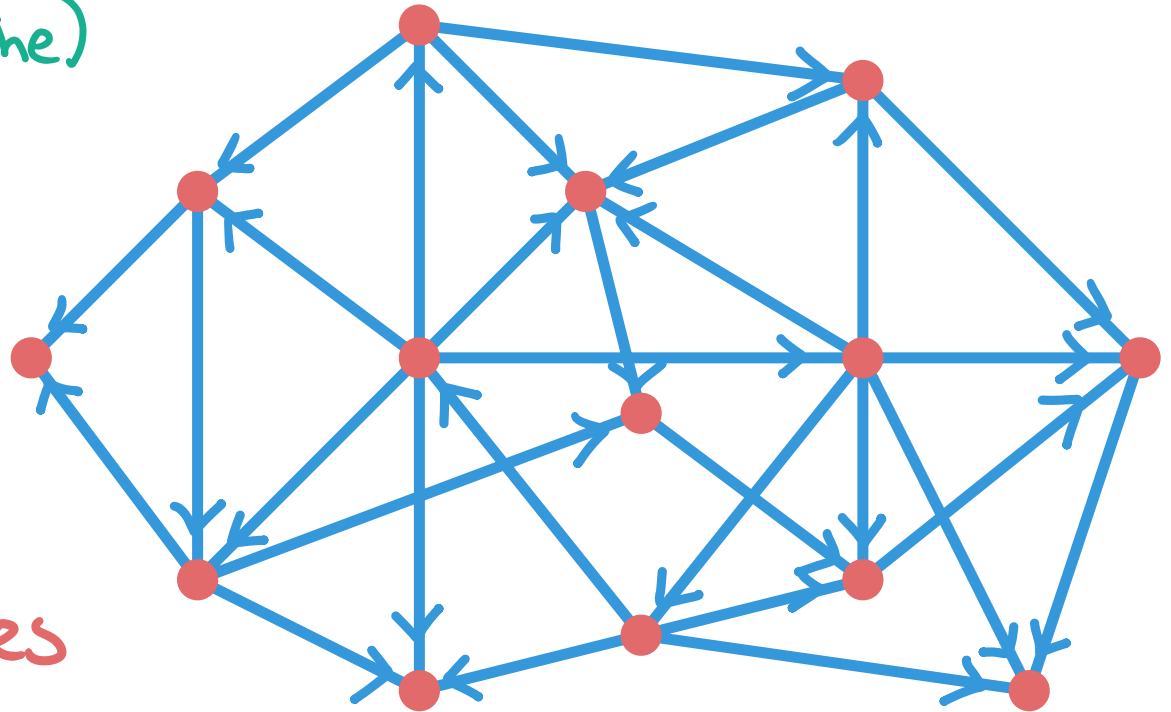
does  $G$  have a Hamiltonian path?

|            | Polytime Algo | Hard as SAT <small>(CNF-)</small> | Abstain |
|------------|---------------|-----------------------------------|---------|
| DAG        | ✓             |                                   |         |
| Directed   |               |                                   |         |
| Undirected |               |                                   |         |

today: longest path problem  
(vertices cannot repeat)

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

Hamiltonian path:  
path visiting all vertices



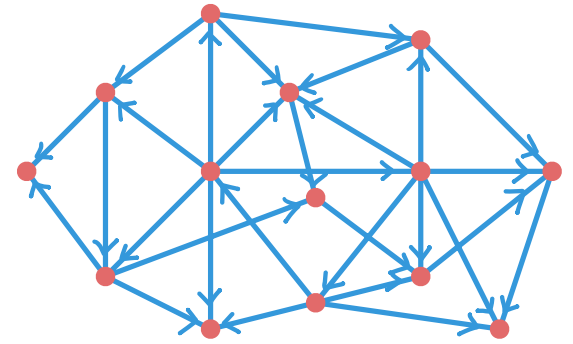
does  $G$  have a Hamiltonian path?

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.*

$$\begin{aligned}\varphi(x_1, \dots, x_n) = & (x_1 \vee \bar{x}_2) \\ & \wedge (x_2 \vee x_3 \vee \bar{x}_4) \\ & \wedge \dots\end{aligned}$$

yes, no



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

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


binary decision

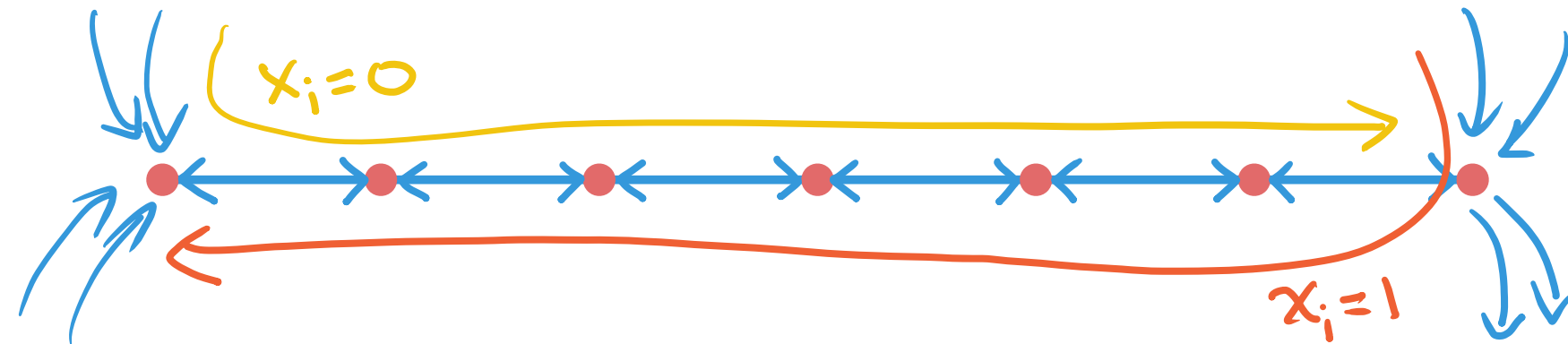
how to model  $x_i \in \{0, 1\}$  as a graph?

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

binary decision

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

Long bidirected path



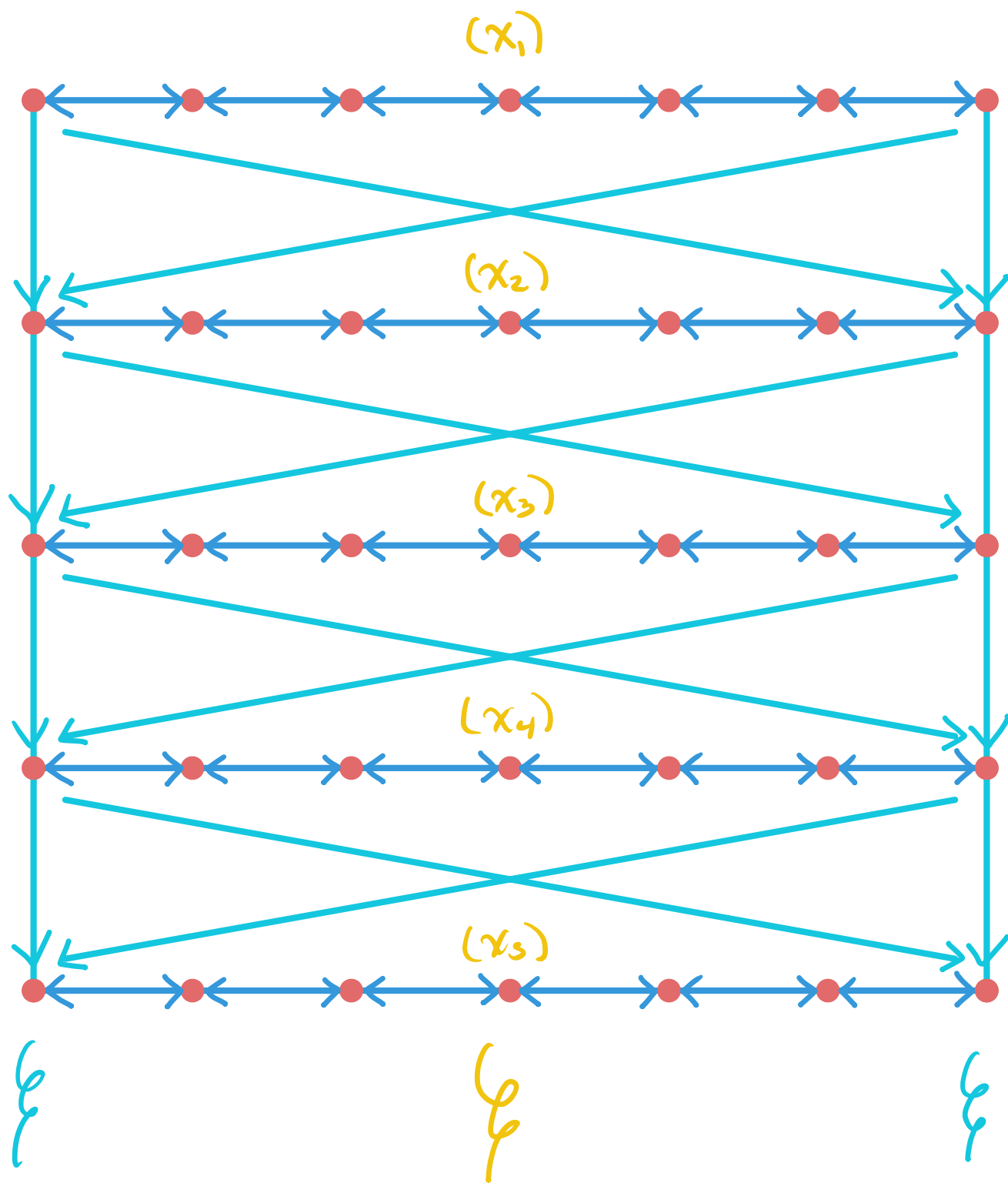
Only way to visit all the vertices

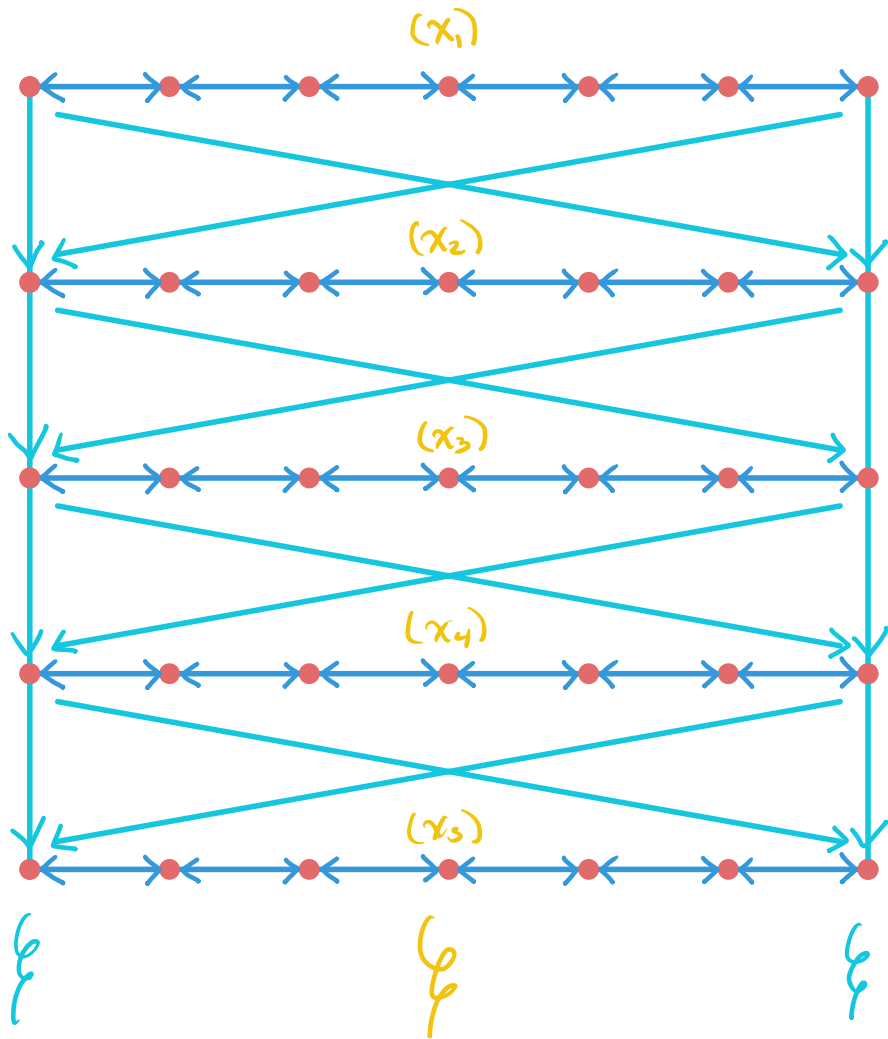
is from left to right



or from right to left







Now we can model choosing

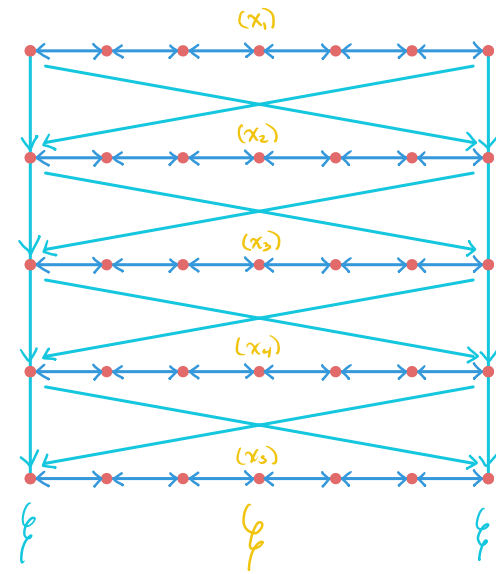
$x_i = 0$  (left  $\rightarrow$  right on  $x_i$ 's path)

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

within overall path

clauses?

e.g.  $x_1 \vee \bar{x}_2$

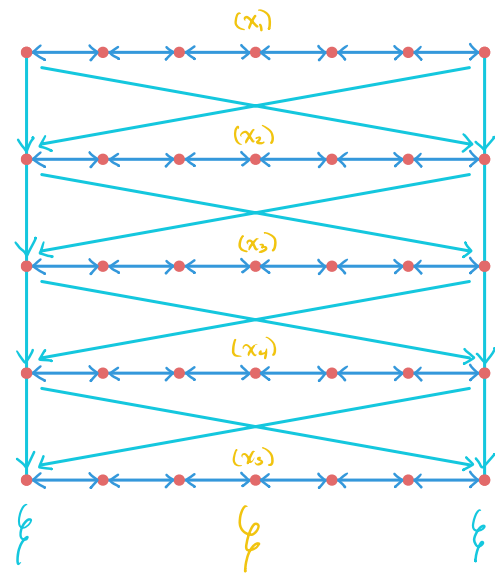


recall we want:

satisfy all clauses  $\Leftrightarrow$  path through all vertices



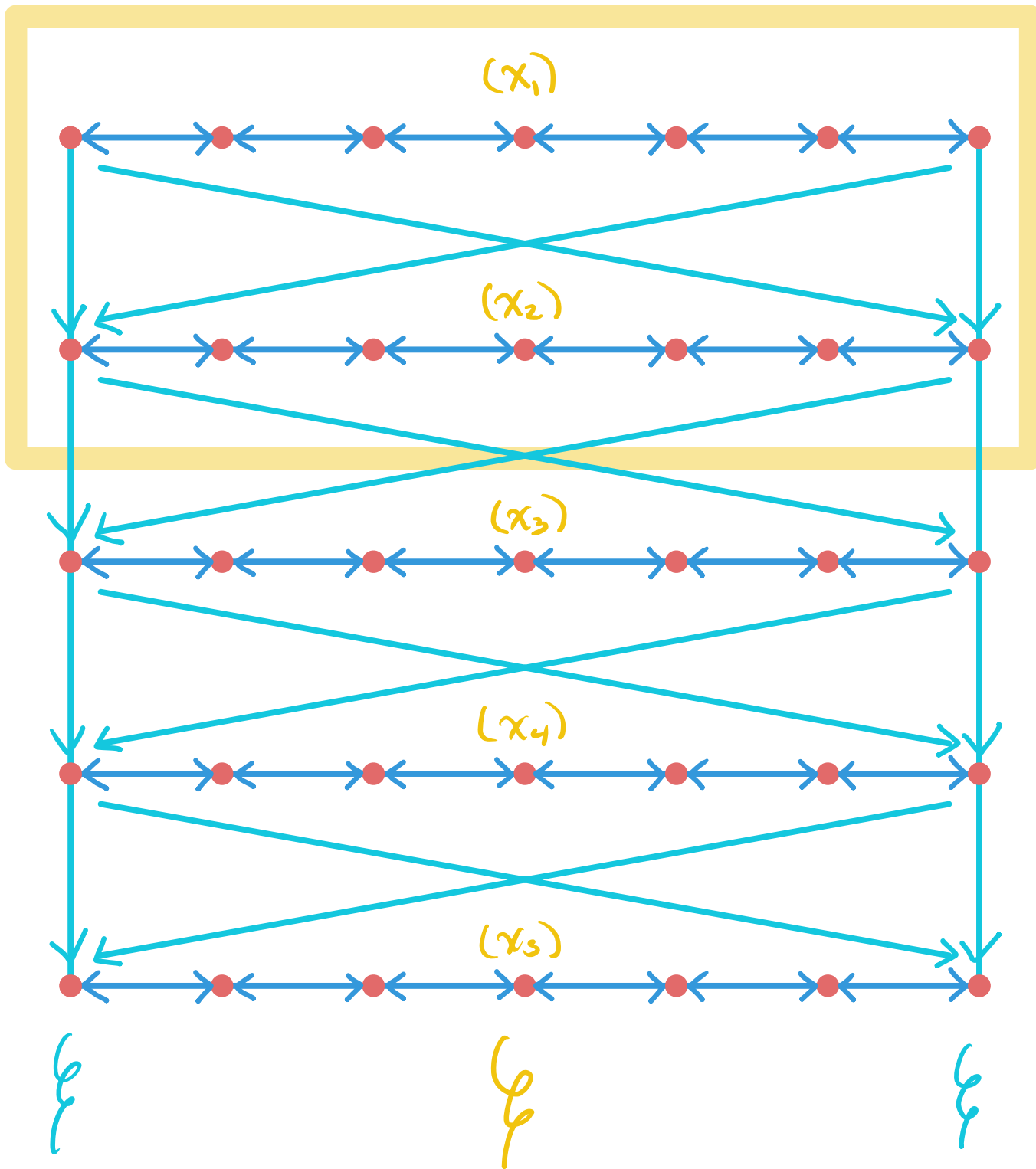
e.g.  $x_1 \vee \overline{x_2}$

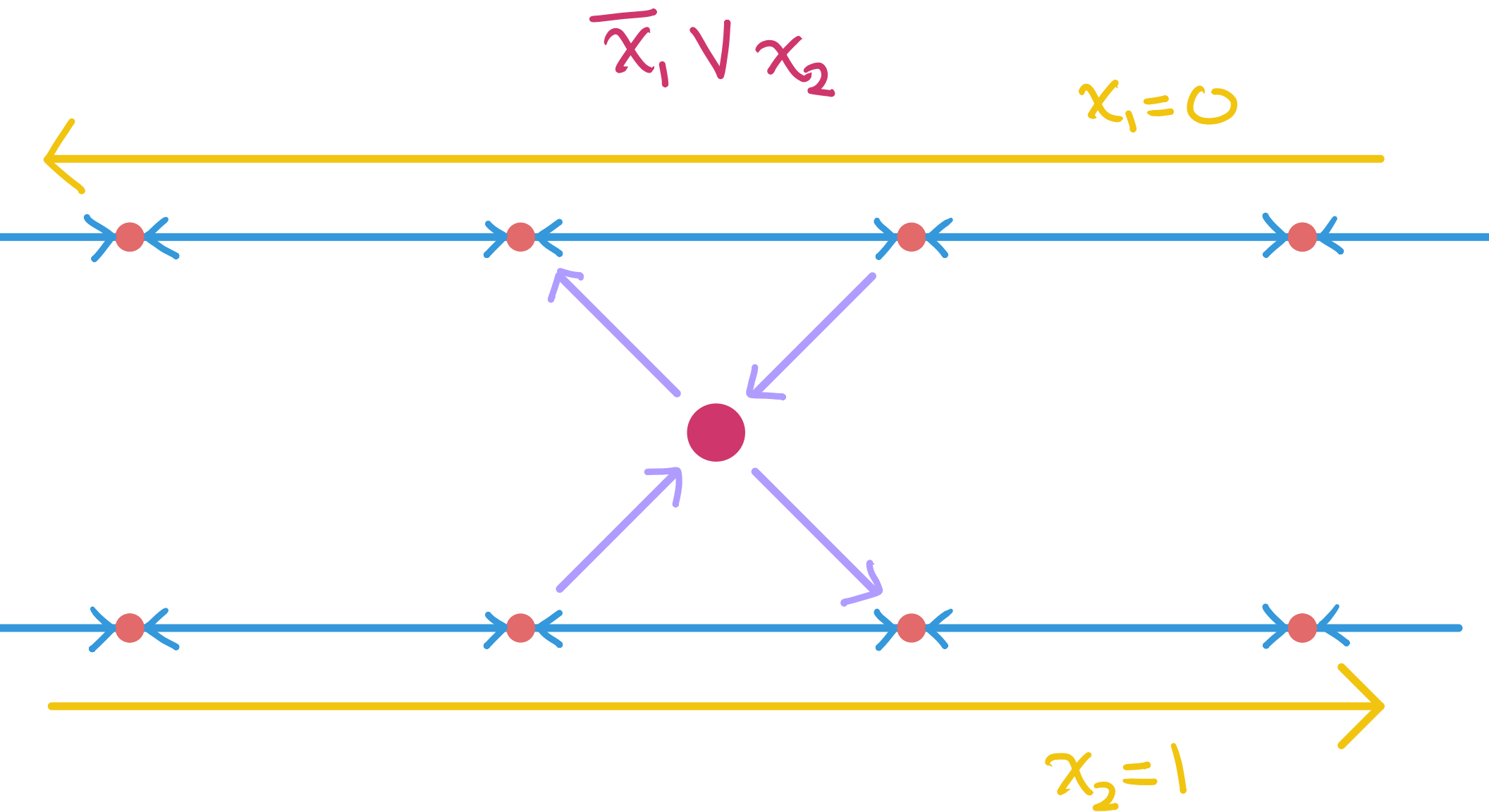


recall we want:

satisfy all clauses  $\Leftrightarrow$  path through all vertices

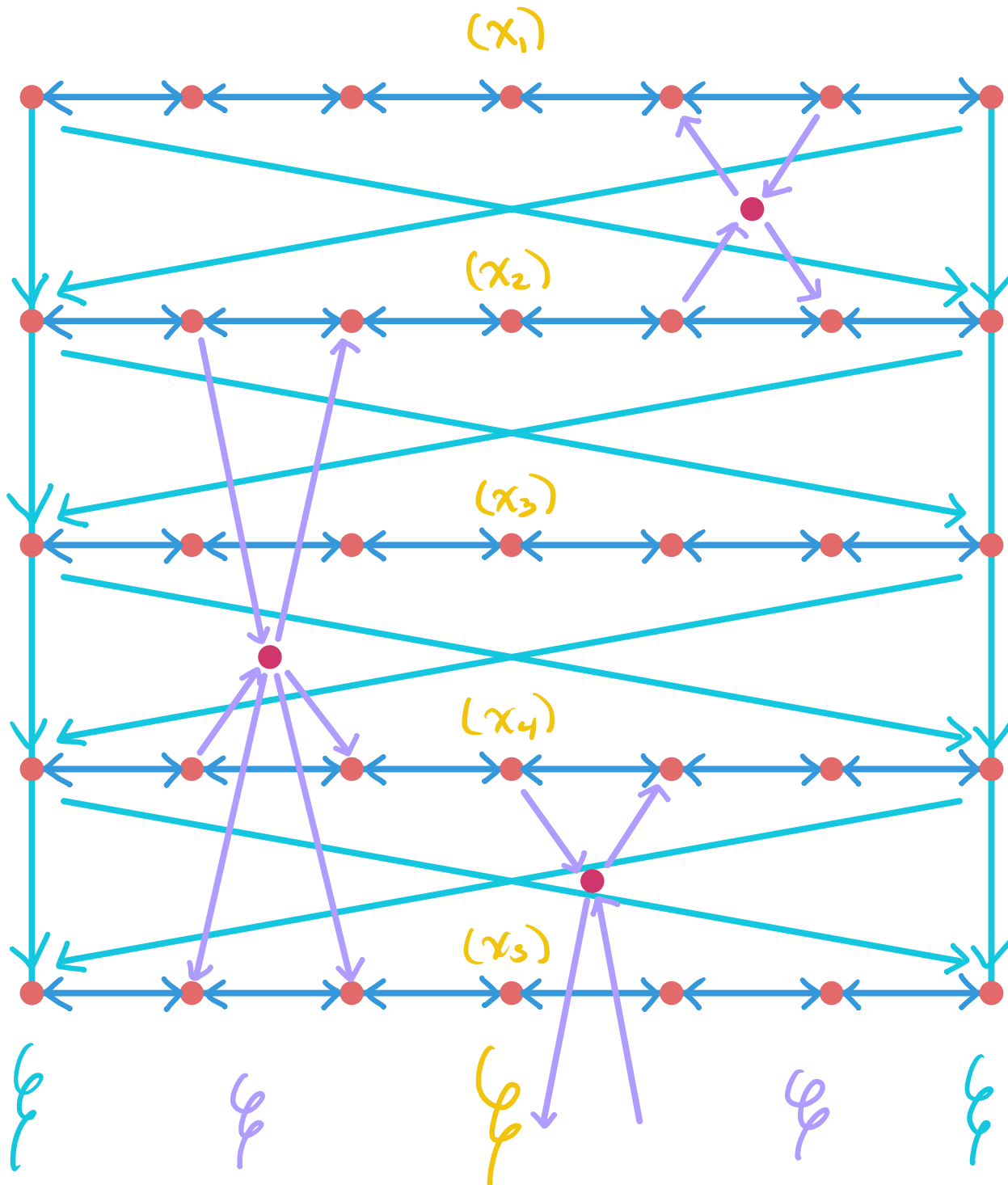
Idea: make a new vertex that we can visit iff we set  $x_1=1$  or  $x_2=0$





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  $\Leftrightarrow$  path through all vertices

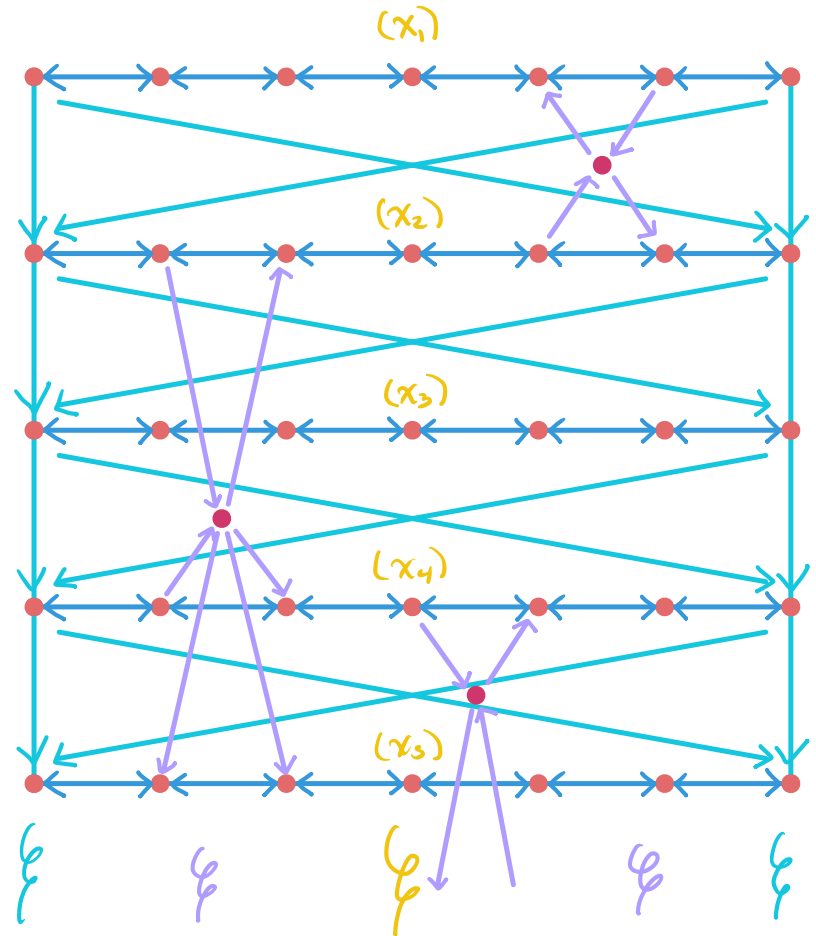
$$\varphi(x_1, \dots, x_n) =$$

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

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

$$\wedge \dots$$

$\Rightarrow$



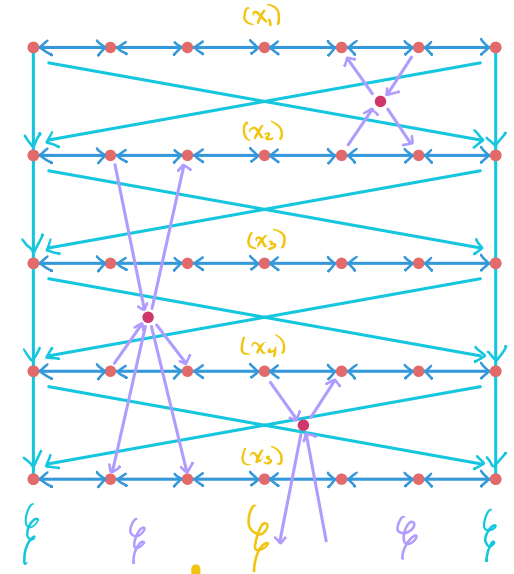
$m$  clauses,  $n$  vertices

$O(mn)$  vertices

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

$$\begin{aligned} \varphi(x_1, \dots, x_n) = & (x_1 \vee \bar{x}_2) \\ & \wedge (x_2 \vee x_3 \vee \bar{x}_4) \\ & \wedge \dots \end{aligned}$$

yes, no



Three settings:

✓ 1. DAG

✓ 2. Directed

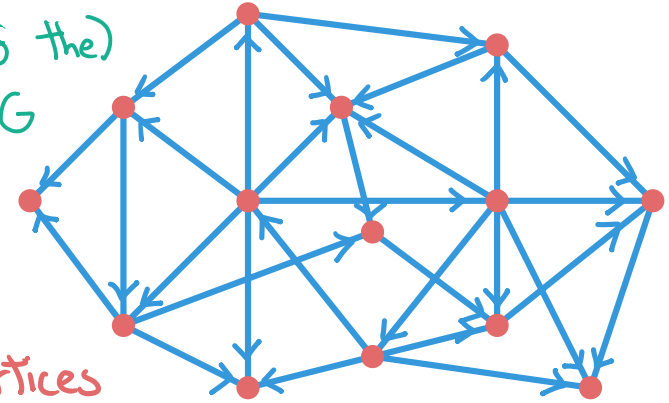
3. Undirected

today: longest path problem (vertices cannot repeat)

Find the (length of the)  
longest path in  $G$

Hamiltonian path:

path visiting all vertices



does  $G$  have a Hamiltonian path?

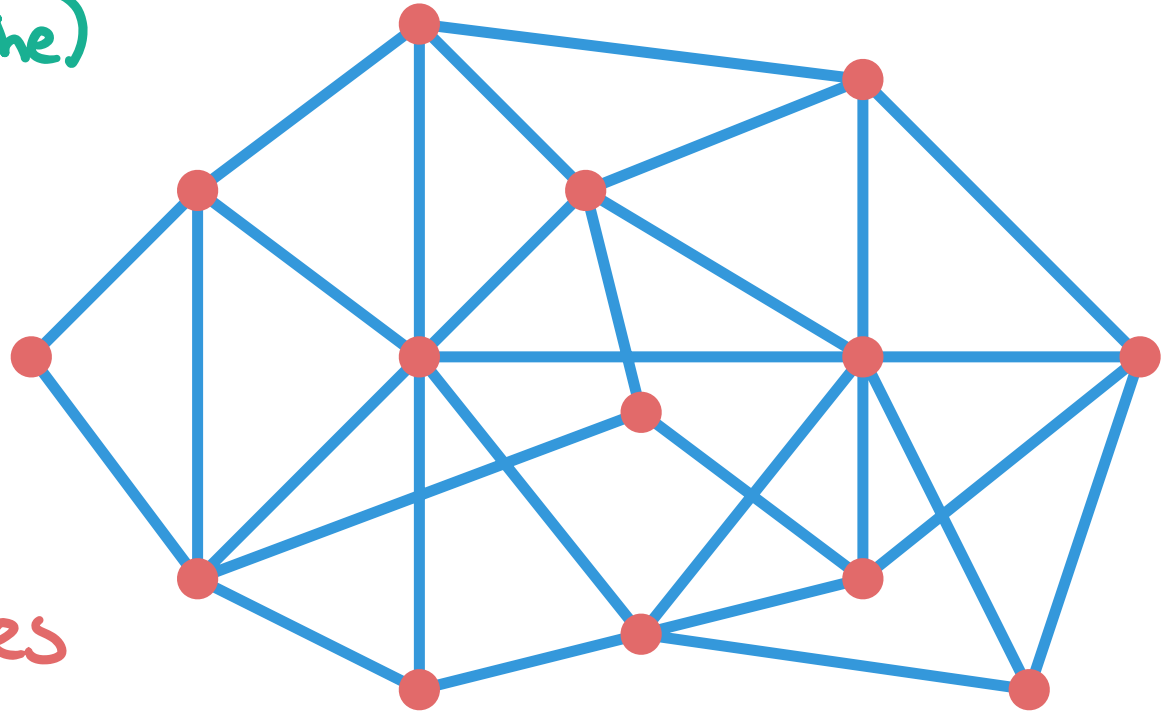
|            | Polytime Algo | Hard as SAT <small>(CNF-)</small> | Abstain |
|------------|---------------|-----------------------------------|---------|
| DAG        | ✓             |                                   |         |
| Directed   |               | ✓                                 |         |
| Undirected |               |                                   |         |

today: longest path <sup>(vertices cannot repeat)</sup> in undirected graphs

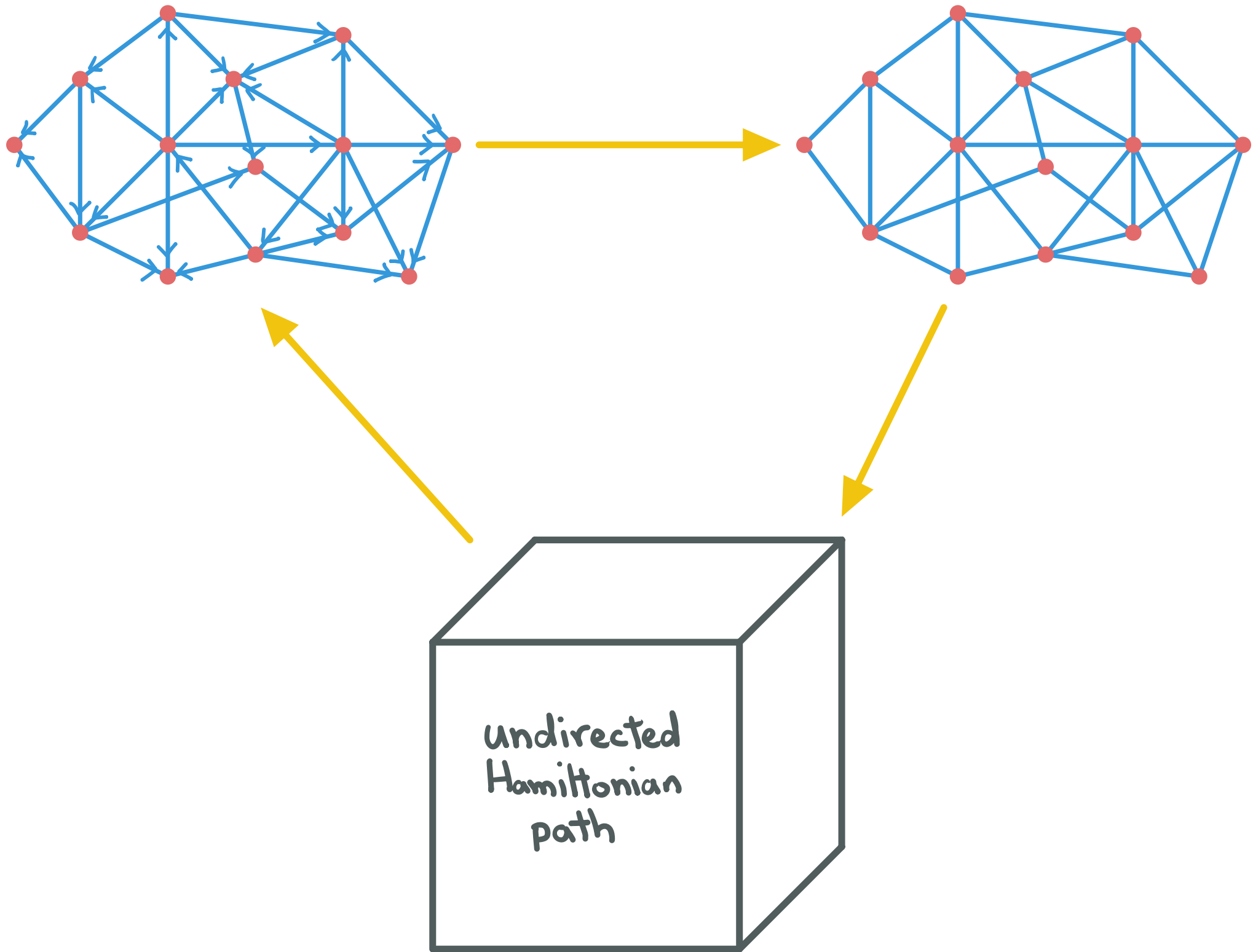
Find the (length of the) <sup>(eg. # edges)</sup> longest path in  $G$

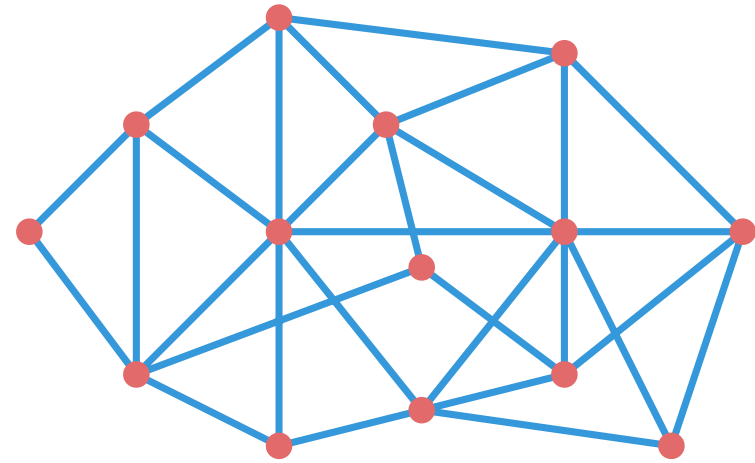
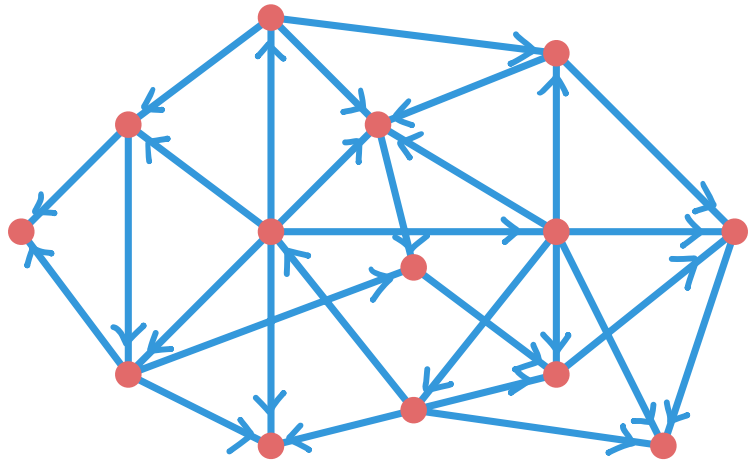
Hamiltonian path:  
path visiting all vertices

does  $G$  have a Hamiltonian path?

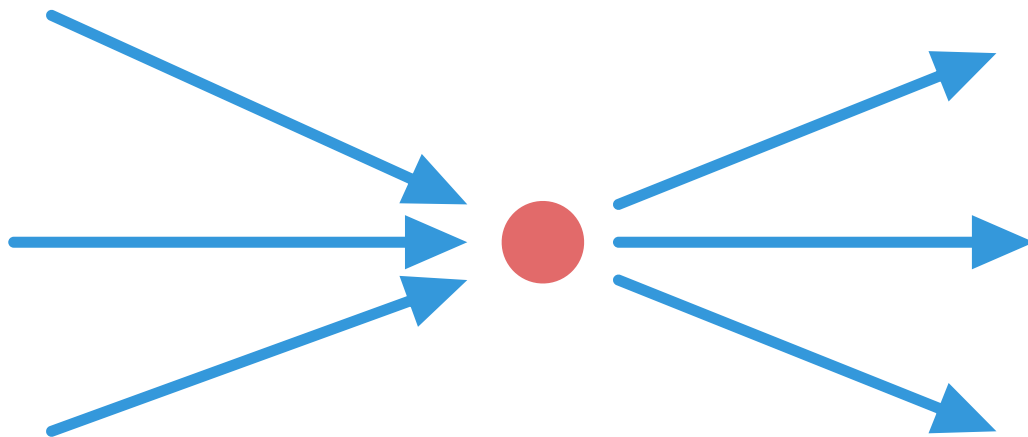








ideas?



**Exercise 12.3.** A *Hamiltonian cycle* is a cycle that visits every vertex exactly once.

1. 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 (the faster the better), or (b) prove that a polynomial time algorithm would imply a polynomial time algorithm for SAT.

2. Give a polynomial time reduction from the Hamiltonian cycle problem to the Hamiltonian path problem (showing directly that a polynomial time algorithm for Hamiltonian path implies a polynomial algorithm for Hamiltonian cycle).

## Chapter 12

### The longest path

Chapter 7 developed algorithms for computing shortest  $(s, t)$ -paths and walks in directed graphs. Here we 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 12.1.) 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.

#### 12.1 The longest path in a DAG

Let  $G = (V, E)$  be a directed acyclic graph (DAG). 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  $G$ .

Let us develop a simple recursive algorithm. We want the length of the longest path/walk in  $G$ . (Getting the actual longest path itself follows easily.) Where do

we start the path? Who knows – so let’s throw the starting vertex in the recursive specification as a parameter. So we 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  $v \in V$ .

We implement **longest-walk**, as specified above, as follows.

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

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 12.2). 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  $d^+(v)$ ). Summing  $d^+(v)$  over all  $v$  counts every edge exactly once, so the total running time is

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

So the longest path problem can be solved in linear time in a DAG, by dynamic programming.

Of course this only gives the maximum length of any path. We can extract the actual path as follows. We can assume we’ve already computed **longest-walk**( $v$ ) for all  $v$ . Suppose we want to extract the longest path from a path  $v$ . Assuming  $v$  is not a sink, the longest path from  $v$  consists of  $v$  plus the longest path from the vertex  $w$  such that  $(v, w) \in \delta^+(v)$  and **longest-walk**( $v$ ) = **longest-walk**( $w$ ) + 1. We can use the stored **longest-walk** values to identify  $w$ , and recursively obtain the rest

of the path from  $v$ . (This technique is sometimes called “backtracking”, and can be used in other dynamic programming algorithms to construct an optimal solution.)

Stepping back, this problem highlights an important algorithmic feature of DAGs: dynamic programming can be applied to DAG’s using the topological order to guide the induction argument.

## 12.2 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?

---

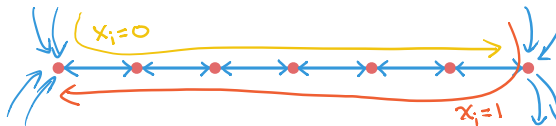
The following theorem indicates that the longest path problem – and even the special case of Hamiltonian path – is much harder than in DAG’s.

**Theorem 12.1.** *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 12.1.

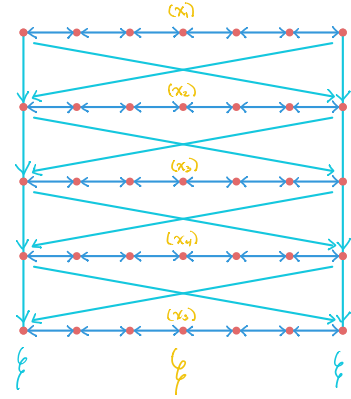
We take as input a SAT formula  $f(x_1, \dots, x_n)$  in CNF with  $m$  clauses. Based on  $f$ , we construct a directed and unweighted graph  $G$  with  $M$  edges and  $N$  vertices, where  $M$  and  $N$  are TBD. It will have the feature that  $f$  is satisfiable iff  $G$  has a Hamiltonian path. 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  $x_j = \text{true}$  or  $x_j = \text{false}$ ) 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  $x_j = \text{true}$  and the other as setting  $x_j = \text{false}$ .



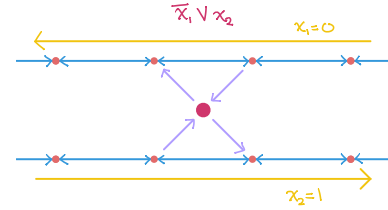


We arrange these paths, one for each  $x_i$ , in a “ladder”-like structure, strung together by edges connecting the endpoints. In particular we will add all four directed combinations from the endpoints of the path of one variable  $x_k$  to the endpoints of the path of the next variable  $x_{k+1}$ . 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.



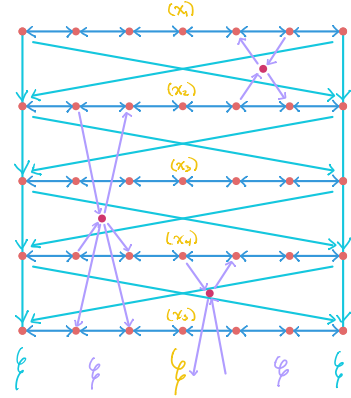
We have not yet encoded any clauses. To this end we have the following gadget. Let us (arbitrarily) fix, for each vertex  $x_j$ , the left to right direction as denoting  $x_j = \text{true}$ , and the right to left direction as denoting  $x_j = \text{false}$ . We want to design a gadget for each clause that is “free” when we travel along the direction of a vertex that corresponds to satisfying the clause.

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  $x_j = \text{true}$ ), 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  $x_j = \text{false}$ ), 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

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

where the  $\dots$  iterates through 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  $j = 1, \dots, n - 1$ , we add all 4 combinations of directed edges from  $s_j$  and  $t_j$  to  $s_{j+1}$  and  $t_{j+1}$ . 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 12.1.

### 12.3 The traveling salesman problem

Let  $G = (V, E)$  be a directed graph with positive edge weights<sup>1</sup>. A *tour* of  $G$  is a walk, starting and ending at the same vertex<sup>2</sup>, that visits every vertex at least once. The *traveling salesman problem* is to compute the minimum length tour.

The traveling salesman problem is extremely useful as a prototypical planning and routing problem (broadly construed). It has inspired a number of techniques both in theory and in practice, and also captured the popular imagination. See <https://www.math.uwaterloo.ca/tsp/history/index.html> for more on the history of this problem.

The problem turns out to be difficult, as it is a thinly disguised Hamiltonian path problem. Indeed, let  $G = (V, E)$  be a directed graph in which we want to find a Hamiltonian path. Let us add a new vertex  $s^*$  with edges  $(s^*, v)$  and  $(v, s^*)$  to and from  $s^*$ , for every  $v \in V$ . Call this new graph  $G^*$ .

Each Hamiltonian path map to a Hamiltonian cycle in  $G^*$  by connecting the beginning and end through  $s^*$ . The inverse mapping takes a Hamiltonian cycle in  $G^*$  and produces a Hamiltonian path by splitting the cycle at  $s^*$ .

Finally, we point out that a Hamiltonian cycle is a tour of  $G^*$  with  $n^* = n + 1$  edges. So a TSP solver can find a Hamiltonian cycle in  $G^*$ , hence a Hamiltonian path in  $G$ .

**Theorem 12.2.** *A polynomial time algorithm for the (directed) traveling salesman problem implies a polynomial time algorithm for SAT.*

### 12.4 Additional notes and materials

**Spring 2024 lecture notes.** Click on the links below for the following files:

<sup>1</sup>One could also consider negative edge weights; we restrict to positive edge weights for simplicity.

<sup>2</sup>Such a walk is sometimes called a *circuit*, but should not be confused with logical circuits that have been discussed elsewhere.

- [Handwritten notes prepared before the lecture.](#)
- [Handwritten notes annotated during the presentation.](#)
- [Recorded video lecture.](#)

**Spring 2023 lecture materials.** Click on the links below for the following files:

- [Handwritten notes prepared before the lecture.](#)
- [Handwritten notes annotated during the presentation.](#)
- [Recorded video lecture.](#) (There were some issues with the audio.)

**Spring 2022 lecture materials.** Click on the links below for the following files:

- [Handwritten notes prepared before the lecture.](#)
- [Handwritten notes annotated during the presentation.](#)
- [Recorded video lecture.](#)

## 12.5 Exercises

**Exercise 12.1.** Suppose you had a polynomial time algorithm for the decision version of the Hamiltonian path problem. That is, given a directed graph  $G$ , the algorithm returns **true** if  $G$  has a Hamiltonian path, and **false** if not. Consider the problem of constructing a Hamiltonian path: that is, if  $G$  has a Hamiltonian path, then return a Hamiltonian path. Show how to use the polynomial time algorithm for the decision problem to obtain a polynomial time algorithm for Hamiltonian path.

**Exercise 12.2.** 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 12.3.** A *Hamiltonian cycle* is a cycle that visits every vertex exactly once.

1. 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 (the faster the better), or (b) prove that a polynomial time algorithm would imply a polynomial time algorithm for SAT.

2. Give a polynomial time reduction from the Hamiltonian cycle problem to the Hamiltonian path problem (showing directly that a polynomial time algorithm for Hamiltonian path implies a polynomial algorithm for Hamiltonian cycle).

**Exercise 12.4.** Consider the Hamiltonian path problem in *undirected* graphs. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that a polynomial time algorithm would imply a polynomial time algorithm for SAT.

**Exercise 12.5.** 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  $O(n! \text{poly}(n))$  time. Can one do better? Design and analyze an algorithm that computes a Hamiltonian path in  $O(2^n \text{poly}(n))$  time. (The smaller the  $\text{poly}(n)$  term the better, but the key point is to reduce  $n!$  to  $2^n$ .)

**Exercise 12.6.** Let  $G = (V, E)$  be a directed graph with real-valued edge lengths  $\ell : E \rightarrow \mathbb{R}$ , and let  $s, t \in V$ . Consider the problem of computing the length of the shortest path from  $s$  to  $t$ . For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that a polynomial time algorithm would imply a polynomial time algorithm for SAT.

**Exercise 12.7.** Here we consider an extension of shortest  $(s, t)$ -walks where one has to visit a family of vertices specified by the input. The input consists of a directed graph  $G = (V, E)$  with positive edge lengths  $\ell : E \rightarrow \mathbb{R}_{>0}$ , as well as a list of vertices  $x_1, \dots, x_k \in V$ . The high-level goal in both of the following problems is to compute the length of the shortest walk that visits all  $k$  vertices.

1. Suppose you are allowed to visit  $x_1, \dots, x_k$  in any order. Consider the problem of computing the length of the shortest walk visiting all of  $x_1, \dots, x_k$  in any order. (You may assume no vertices repeat in  $x_1, \dots, x_k$ .) For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that a polynomial time algorithm would imply a polynomial time algorithm for SAT.
2. Suppose you have to visit  $x_1, \dots, x_k$  in the listed order. (Here vertices may repeat, but for simplicity you may assume that any two consecutive vertices  $v_i$

and  $v_{i+1}$  are distinct.) Consider the problem of computing the length of the shortest walk visiting  $x_1, \dots, x_k$  in order. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that a polynomial time algorithm would imply a polynomial time algorithm for SAT.

**Exercise 12.8.** The following 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. Let  $G = (V, E)$  be a directed graph with  $m$  edges and  $n$  vertices. For each of the following problems, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that a polynomial time algorithm would imply a polynomial time algorithm for SAT.

1. Compute the length of the longest walk in  $G$ .
2. Compute the maximum number of distinct vertices of any walk in  $G$ .
3. Compute the maximum number of distinct edges of any walk in  $G$ .

**Exercise 12.9.** Let  $G = (V, E)$  be a directed graph with vertex weights  $w : V \rightarrow \mathbb{R}$ . We say that the *vertex weight of a walk* going through vertices  $v_1, v_2, \dots, v_k$  (in order, with vertices possible repeating) is the sum weight  $w(v_1) + w(v_2) + \dots + w(v_k)$ .

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 (the faster the better), or (b) prove that a polynomial time algorithm would imply a polynomial time algorithm for SAT.

**Exercise 12.10.** 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<sup>3</sup> in  $G$  where the labels of the vertices are (strictly) increasing.

1. Suppose  $G$  is a DAG. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that a polynomial time algorithm would imply a polynomial time algorithm for SAT.

---

<sup>3</sup>Recall that a path is a walk that does not repeat vertices.

2. Consider now the problem for general graphs. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that a polynomial time algorithm would imply a polynomial time algorithm for SAT.
3. Suppose instead we ask for the length of the longest path in  $G$  where  $G$  is a general graph and the labels of the vertices are weakly increasing.<sup>4</sup> For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that a polynomial time algorithm would imply a polynomial time algorithm for SAT.

**Exercise 12.11.** Let  $G = (V, E)$  be a simple<sup>5</sup> 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, red, white, blue... (The first edge may be any color.) Consider the problem of computing the length of the longest American path in the graph. For this problem in each of the following settings, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that a polynomial time algorithm would imply a polynomial time algorithm for SAT. You may assume that one can look up the color of a path in constant time.

1.  $G$  is a DAG.
2.  $G$  is a simple directed graph.

**Exercise 12.12.** 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. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that a polynomial time algorithm would imply a polynomial time algorithm for SAT.

**Exercise 12.13.** Let  $G = (V, E)$  be a DAG, and  $s, t \in V$ . Consider the problem of computing the *number* of paths from  $s$  to  $t$  in  $G$ .<sup>6</sup> For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that a polynomial time algorithm would imply a polynomial time algorithm for SAT.

---

<sup>4</sup>A sequence  $x_1, \dots, x_k$  is weakly increasing if  $x_1 \leq x_2 \leq \dots \leq x_k$ .

<sup>5</sup>i.e.,  $G$  has no parallel edges.

<sup>6</sup>Technically, there could be as many as  $n!$  paths in which case each arithmetic operation would take  $O(\log(n!)) = O(n \log n)$  time. For this exercise, let us assume for simplicity that arithmetic operations take  $O(1)$  time anyway.

**Exercise 12.14.** A polygonal path is a sequence of line segments joined end-to-end; the endpoints of these line segments are called the vertices of the path. The length of a polygonal path is defined as the sum of the lengths of its segments. A polygonal path with vertices  $(x_1, y_1), (x_2, y_2), \dots, (x_k, y_k)$  is *monotonically increasing* if  $x_i < x_{i+1}$  and  $y_i < y_{i+1}$  for every index  $i$  – informally, each vertex of the path is above and to the right of its predecessor.

Consider the problem of computing the length of the longest monotonically increasing path among a set  $S$  of  $n$  points. Here the input consists of a set  $S$  of  $n$  points in the plane, represented as two arrays  $X[1..n]$  and  $Y[1..n]$  (for the  $x$ -coordinates and  $y$ -coordinates, respectively). You may assume access to a subroutine  $\text{Length}(x, y, x', y')$  that returns the length of the segment from  $(x, y)$  to  $(x', y')$  in  $O(1)$  time. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that a polynomial time algorithm would imply a polynomial time algorithm for SAT.

**Exercise 12.15. It's-a me, Mario!** This problem is inspired by Super Mario World, where Mario must make it from some starting point to the castle, collecting as many coins as possible along the way.

Let  $G = (V, E)$  be a directed graph where each vertex  $v \in V$  has  $C_v \geq 0$  coins. For a walk in  $G$ , we say that the *number of coins collected by the walk* is the total sum of  $C_v$  over all *distinct* vertices  $v$  in the walk. (If we visit a vertex  $v$  more than once, we still only get  $C_v$  coins total.) Given  $s, t \in V$ , the goal is to compute the maximum number of coins collected by any  $(s, t)$ -walk.

1. (5 points) Let  $G$  be a DAG. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that a polynomial time algorithm would imply a polynomial time algorithm for SAT.
2. (5 points) Let  $G$  be a general directed graph. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that a polynomial time algorithm would imply a polynomial time algorithm for SAT.

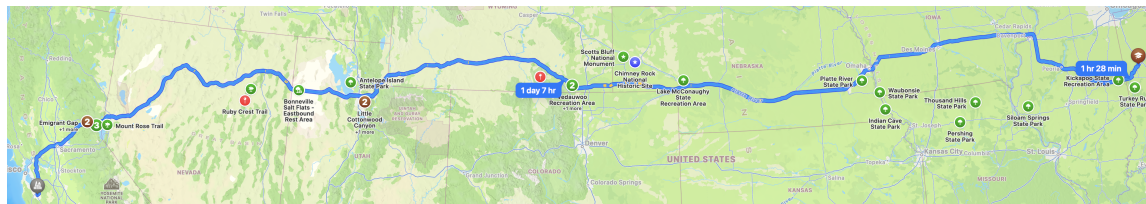
**Exercise 12.16.** Let  $G = (V, E)$  be a directed graph with no parallel edges. We say that a walk  $w$  in  $G$  has *distinct edges* if no edge appears twice in  $w$ . Consider the problem of computing the length of the longest walk in  $G$  with distinct edges. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that a polynomial time algorithm would imply a polynomial time algorithm for SAT.



**Exercise 12.17.** Let  $G = (V, E)$  be a directed graph with  $m$  edges,  $n$  vertices, and nonnegative integer edge lengths  $\ell : E \rightarrow \mathbb{Z}_{\geq 0}$ . (Edges may have length 0.) Given two vertices  $s$  and  $t$ , there may be many shortest paths (with respect to the edge lengths  $\ell$ ) from  $s$  to  $t$ . For each of the following problems, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that a polynomial time algorithm would imply a polynomial time algorithm for SAT.

1. (5 points) Compute the minimum number of edges of any shortest  $(s, t)$ -path (with respect to  $\ell$ ) from  $s$  to  $t$ .
2. (5 points) Compute the maximum number of edges of any shortest  $(s, t)$ -path (with respect to  $\ell$ ) from  $s$  to  $t$ .

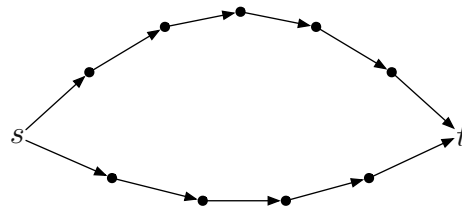
**Exercise 12.18.** Imagine you are planning a long road trip to San Jose, CA.



This time you haven't fixed your route and you're not so obsessed with gas stations. You decided you want to do some sightseeing along the way and you've marked out some points of interest across America. Your goal is to visit as many of these points as possible, but you are very strict about the phrase "along the way". You should never go backwards – you should always make progress towards your destination, even if its not the fastest way to San Jose.

We model this problem with a directed graph  $G = (V, E)$  with positive real edge lengths  $\ell : E \rightarrow \mathbb{R}_{>0}$ . You have a vertex  $s$  marking the start and a vertex  $t$  marking the end. You are given a set  $A \subseteq V$  of size  $p = |A|$ , which define the tourist attractions you would like to see. The goal is to compute the maximum number of attractions that you can visit along any walk  $(v_0 = s), v_1, \dots, (v_k = t)$  from  $s$  to  $t$  such that for  $i = 1, \dots, k$ , the distance from  $v_i$  to  $t$  is (strictly) less than the distance from  $v_{i-1}$  to  $t$  (with respect to  $\ell$ ). For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that a polynomial time algorithm would imply a polynomial time algorithm for SAT.

**Exercise 12.19.** Let  $G = (V, E)$  be a directed graph and  $s, t \in V$ . We say that two  $(s, t)$ -paths  $p, q : s \rightsquigarrow t$  are *internally disjoint* if they are vertex disjoint except at their endpoints  $s$  and  $t$ .



Consider the problem of computing the maximum number of edges of any two internally disjoint  $(s, t)$ -paths (for fixed  $s$  and  $t$ ). For example, the picture above gives two internally disjoint  $(s, t)$ -paths with a total of 11 edges. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that a polynomial time algorithm would imply a polynomial time algorithm for SAT.