Algorithmics

Lecture 9

Dr. Oana Andrei

School of Computing Science
University of Glasgow

oana.andrei@glasgow.ac.uk

Section 4 – NP–completeness

Introduction (examples and discussion)

NP–complete problems

The classes P and NP

Polynomial–time reductions

Formal definition of NP–completeness

How to prove a problem is NP–complete

The class P

P is the class of all decision problems that can be solved in polynomial time

Fortunately, many problems are in P

- is there a path of length ≤K from vertex u to vertex v in a graph G?
- is there a spanning tree of weight ≤K in a graph G?
- is a graph G bipartite?
- is a graph G connected?
- deadlock detection: does a directed graph D contain a cycle?
- text searching: does a text t contain an occurrence of a string s?
- string distance: is d(s,t)≤K for strings s and t?
- …

P often extended to include search and optimisation problems

- what is the minimum length path between vertex u and vertex v

The class NP

The decision problems solvable in non-deterministic polynomial time

- a non-deterministic algorithm can make non-deterministic choices
  - the algorithm is allowed to guess (so when run can give different answers)
- hence is apparently more powerful than a normal deterministic algorithm

P is certainly contained within NP

- a deterministic algorithm is just a special case of a non-deterministic one

But is that containment strict?

- there is no problem known to be in NP and known not to be in P

The relationship between P and NP is the most notorious unsolved question in computing science

- there is a million dollar prize if you can solve this question

Non-deterministic algorithms (NDAs)

Such an algorithm has an extra operation: non-deterministic choice

```
int nonDeterministicChoice(int n)
// returns a positive integer chosen from the range 1,…,n
```

– an NDA has many possible executions depending on values returned

An NDA "solves" a decision problem π if

– for a 'yes'-instance I of π there is some execution that returns 'yes'

– for a 'no'-instance I of π there is no execution that returns 'yes'

and "solves" a decision problem π in polynomial time if

– for every 'yes'-instance I of π there is some execution that returns 'yes' which uses a number of steps bounded by a polynomial in the input

– for a 'no'-instance I of π there is no execution that returns 'yes'

Non-deterministic algorithms (NDAs)

An NDA "solves" a decision problem π if

- for a 'yes'-instance I of π there is some execution that returns 'yes'
- for a 'no'-instance I of π there is no execution that returns 'yes'

Clearly such algorithms are not useful in practice

- who would use an algorithm that sometimes gives the right answer

However they are a useful mathematical concept for defining the classes of NP and NP-complete problems

Non-deterministic algorithms – Example

Graph colouring

```
 // return true if graph g is k-colourable and false otherwise
boolean nDGC(Graph g, int k){

 for (each vertex v in g) v.setColour(nonDeterministicChoice(k));

 for (each edge {u,v} in g)
  if (u.getColour() == v.getColour()) return false;
 return true;
}
```

"verify" the colouring

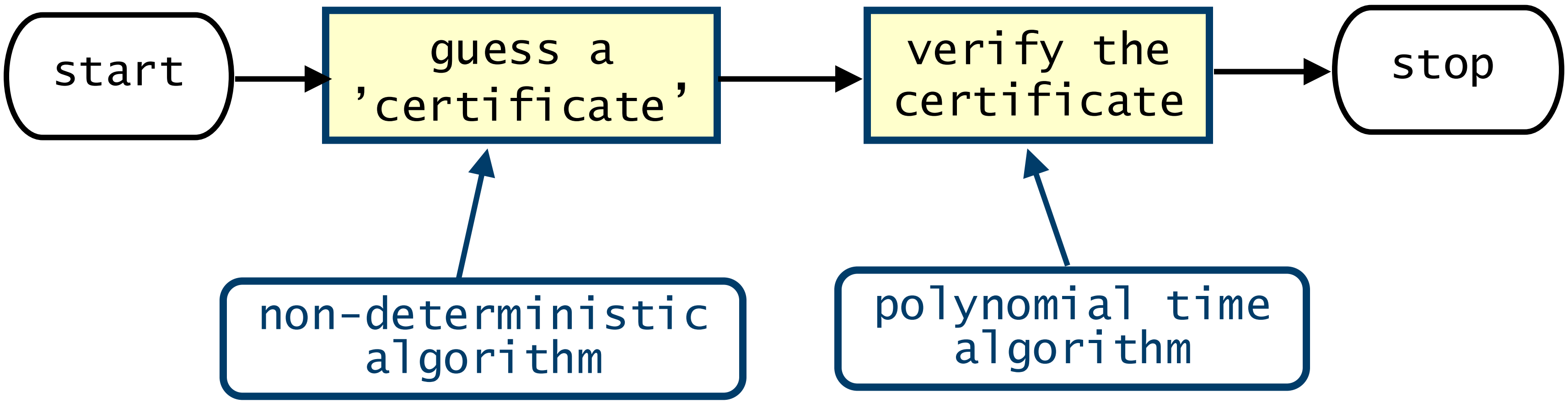
"guess" a colour for each vertex

Non–deterministic algorithms

An non–deterministic algorithm can be viewed as

– a guessing stage (non–deterministic)

– a checking stage (deterministic and polynomial time)



Section 4 – NP-completeness

Introduction (examples and discussion)

NP-complete problems

The classes P and NP

Polynomial-time reductions

Formal definition of NP-completeness

How to prove a problem is NP-complete

Polynomial time reductions

A **polynomial-time reduction (PTR)** is a mapping f from a decision problem Π_1 to a decision problem Π_2 such that:

for every instance I_1 of Π_1 we have

- the instance $f(I_1)$ of Π_2 can be constructed in polynomial time
- $f(I_1)$ is a 'yes'-instance of Π_2 if and only if I_1 is a 'yes'-instance of Π_1

We write $\Pi_1 \propto \Pi_2$ as an abbreviation for:

there is a polynomial-time reduction from Π_1 to Π_2

Polynomial time reductions – Properties

Transitivity: $\Pi_1 \propto \Pi_2$ and $\Pi_2 \propto \Pi_3$ implies that $\Pi_1 \propto \Pi_3$

Since $\Pi_1 \propto \Pi_2$ and $\Pi_2 \propto \Pi_3$ we have

- a PTR f from Π_1 to Π_2
- a PTR g from Π_2 to Π_3

Now for any instance I_1 of Π_1 since f is PTR we have

- $I_2=f(I_1)$ is an instance of Π_2 that can be constructed in polynomial time
- I_2 has the same answer as I_1

and since g is a PTR we have

- $I_3=g(I_2)$ is an instance of Π_3 that can be constructed in polynomial time
- I_3 has the same answer as I_2

Polynomial time reductions – Properties

Transitivity: $\Pi_1 \propto \Pi_2$ and $\Pi_2 \propto \Pi_3$ implies that $\Pi_1 \propto \Pi_3$

Since $\Pi_1 \propto \Pi_2$ and $\Pi_2 \propto \Pi_3$ we have

- a PTR f from Π_1 to Π_2
- a PTR g from Π_2 to Π_3

Putting the results together: for any instance I_1 of Π_1

- $I_3 = g(f(I_1))$ is an instance of Π_3 constructed in polynomial time
- I_3 has the same answer as I_1
- i.e. the composition of f and g is a PTR from from Π_1 to Π_3

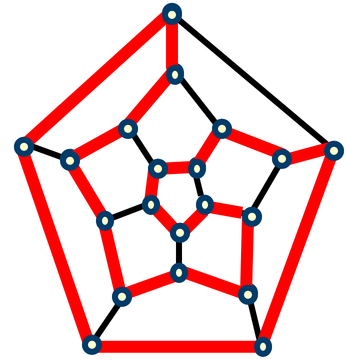
Polynomial time reductions – Properties

Relevance to P: $\Pi_1 \propto \Pi_2$ and $\Pi_2 \in P$ implies that $\Pi_1 \in P$

- to solve an instance of Π_1, reduce it to an instance of Π_2
- roughly speaking, $\Pi_1 \propto \Pi_2$ means that Π_1 is 'no harder' than Π_2
  i.e. if we can solve Π_2, then we can solve Π_1 without much more effort
  - just need to additional perform a polynomial time reduction
- but maybe that Π_2 is harder to solve than Π_1
  - we only map to easy to solve instances of Π_2

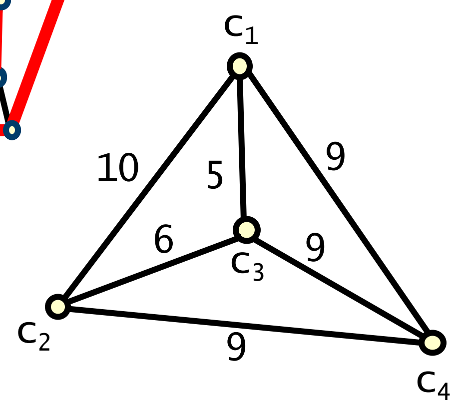
Polynomial time reductions – Example

Reducing Hamiltonian cycle problem to travelling salesperson problem



Hamiltonian Cycle Problem (HC)

- instance: a graph G
- question: does G contain a cycle that visits each vertex exactly once?



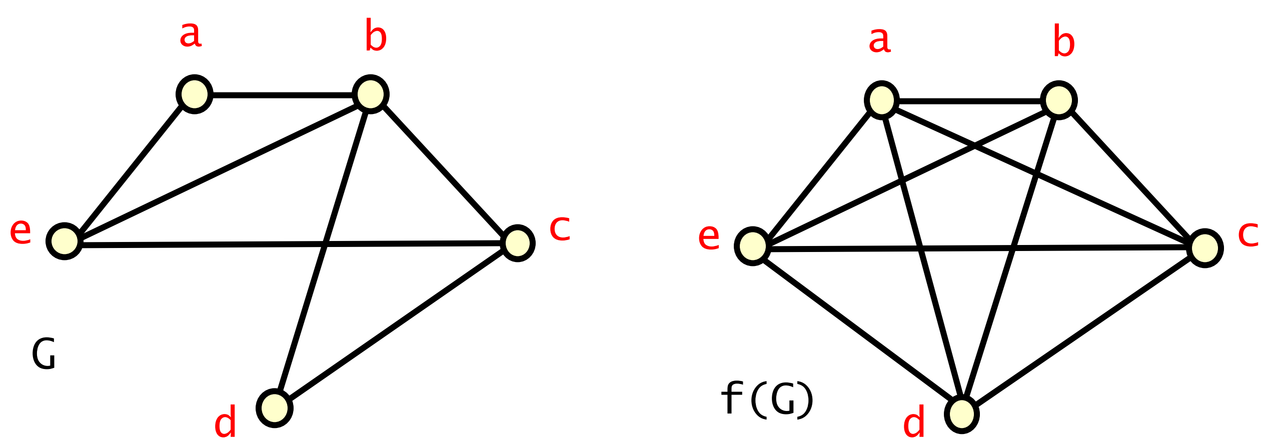
Travelling Salesperson Decision Problem (TSDP)

- instance: a set of n cities and integer distance d(i,j) between each pair of cities i,j, and a target integer K
- question: is there a permutation p of {1,2,…,n} such that $d(p_1,p_2) + d(p_2,p_3) + \ldots + d(p_{n-1},p_n) + d(p_n,p_1) \leq K$?
  - i.e. is there a ’travelling salesperson tour’ of length ≤K

Polynomial time reductions – Example

Reducing Hamiltonian cycle problem to travelling salesperson problem

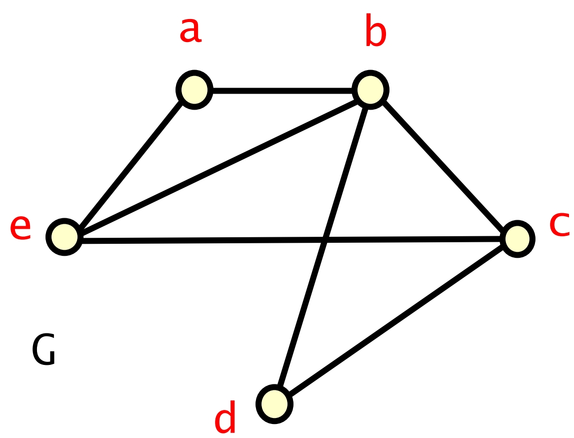
- G = (V,E) is an instance of HC
- construct TSDP instance f(G) where
  - cities = V

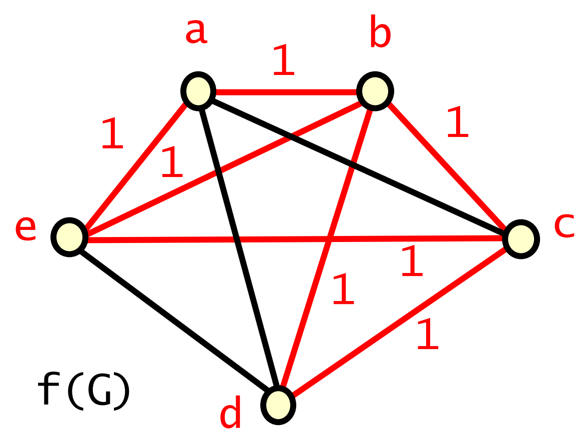


Polynomial time reductions – Example

Reducing Hamiltonian cycle problem to travelling salesperson problem

- G = (V,E) is an instance of HC
- construct TSDP instance f(G) where
  - cities = V
  - d(u,v)=1 if {u,v}∈E (is an edge of G)

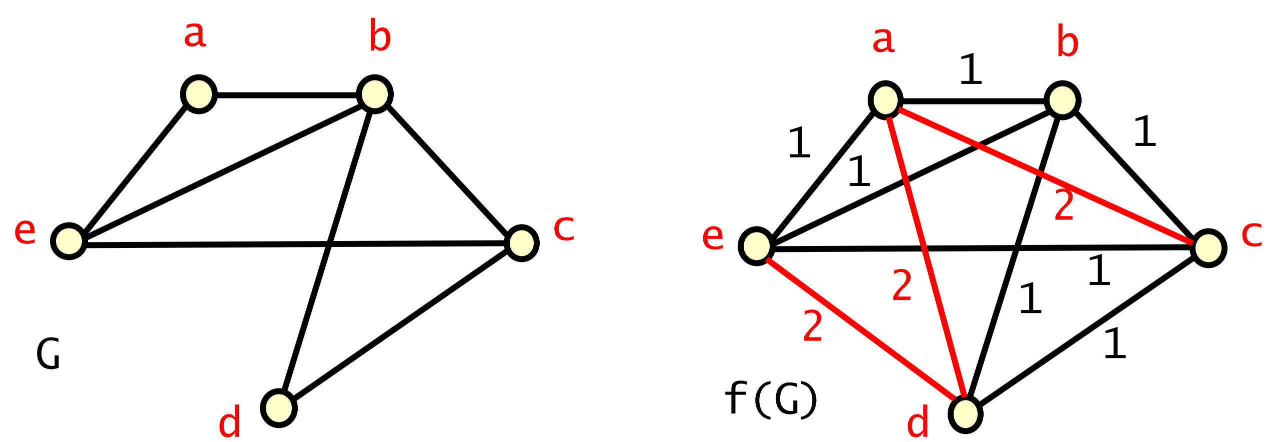




Polynomial time reductions – Example

Reducing Hamiltonian cycle problem to travelling salesperson problem

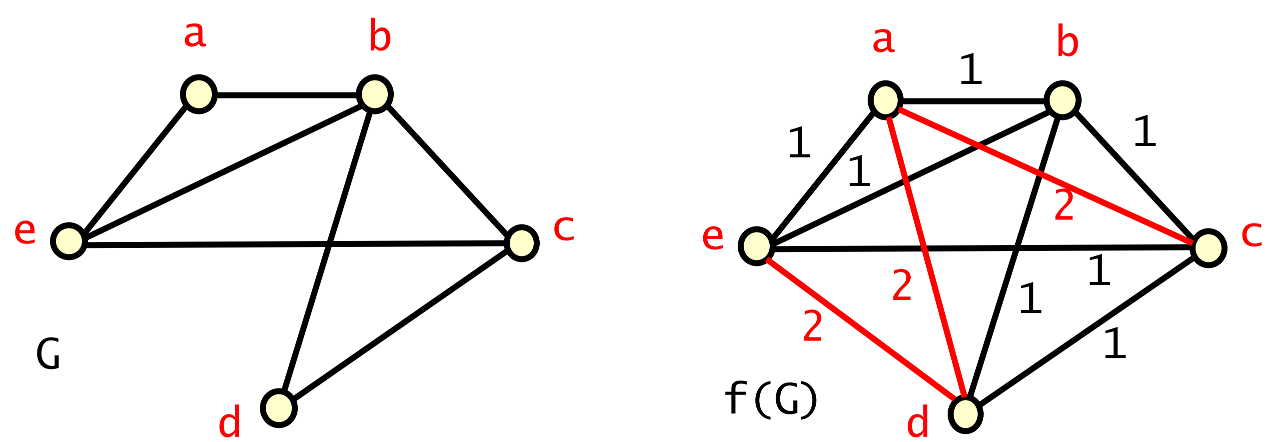
- `G = (V,E)` is an instance of HC
- construct TSDP instance `f(G)` where
  - `cities = V`
  - `d(u,v)=1` if `{u,v}∈E` and 2 otherwise (is not an edge of G)



Polynomial time reductions – Example

Reducing Hamiltonian cycle problem to travelling salesperson problem

- G = (V,E) is an instance of HC
- construct TSDP instance f(G) where
  - cities = V
  - d(u,v)=1 if {u,v}∈E and 2 otherwise (is not an edge of G)
  - K = |V|

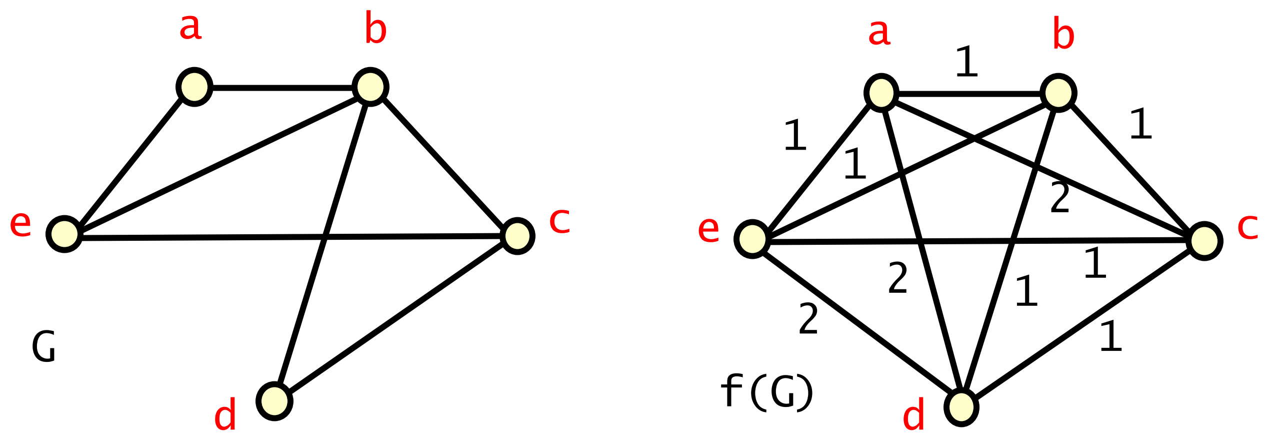


Polynomial time reductions – Example

Reducing Hamiltonian cycle problem to travelling salesperson problem

– G = (V,E) is an instance of HC

– construct TSDP instance f(G)



– f(G) can be constructed in polynomial time

– f(G) has a tour of length ≤|V| if and only if G has a Hamiltonian cycle (tour includes |V| edges so cannot take any of the edges with weight 2)

– therefore TSDP∈P implies that HC∈P

– equivalently HC∉P implies that TSDP∉P (contrapositive)

Section 4 – NP-completeness

Introduction (examples and discussion)

NP-complete problems

The classes P and NP

Polynomial-time reductions

Formal definition of NP-completeness

How to prove a problem is NP-complete

NP-completeness

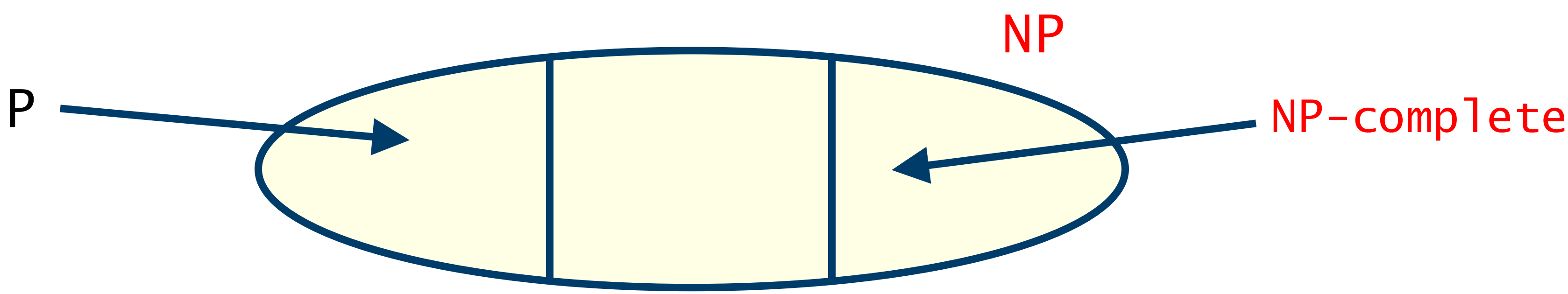
A decision problem π is NP-complete if

1. π∈NP
2. for every problem π' in NP: π' is polynomial-time reducible to π

Consequences of definition

- if π is NP-complete and can show that π∈P, then P = NP
- every problem in NP can be solved in polynomial time by reduction to Π
- supposing P ≠ NP, if π is NP-complete, then π∉P

The structure of NP if P ≠ NP



NP hard problems

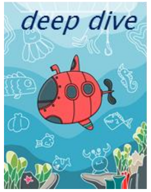
An NP-complete problem is as hard as the hardest problems in NP

A problem is NP-hard if every problem in NP can be reduced to it in polynomial time.

- no requirement that the problem itself must be in NP
- NP-hard problem is at least as hard as the hardest problems in NP, but it might not necessarily be in NP
- it may not be possible to verify a solution in polynomial time for an NP-hard problem

All NP-complete problems are NP-hard, but not all NP-hard problems are NP-complete

NP hard problems



All NP-complete problems are NP-hard, but not all NP-hard problems are NP-complete

- NP-complete problems are solvable in polynomial time by a nondeterministic Turing machine and have polynomial-time verifiable solutions
- NP-hard problems encompass a broader category that includes problems for which verifying a solution might not be feasible in polynomial time

NP-completeness mainly applies to decision problems

- problems with a yes/no answer

NP-hardness applies more broadly to decision problems, optimization problems, and search problems

Section 4 – NP-completeness

Introduction (examples and discussion)

NP-complete problems

The classes P and NP

Polynomial-time reductions

Formal definition of NP-completeness

How to prove a problem is NP-complete

Proving NP-completeness

A decision problem π is NP-complete if

1. $\pi \in NP$
2. for every problem π' in NP: π' is polynomial-time reducible to π

How can we possibly prove any problem to be NP-complete?

- it is not feasible to describe a reduction from every problem in NP
- however, suppose we knew just one NP-complete problem π_1

To prove π_2 is NP-complete enough to show

- π_2 is in NP
- there exists a polynomial-time reduction from π_1 to π_2

Proving NP-completeness

A decision problem Π is NP-complete if

1. $\Pi \in$ NP
2. for every problem Π' in NP: Π' is polynomial-time reducible to Π

Suppose we knew just one NP-complete problem Π_1, then to prove Π_2 is NP-complete it is enough to show

- Π_2 is in NP
- there exists a polynomial-time reduction from Π_1 to Π_2

Correctness of the approach

- for any $\Pi \in$ NP, since Π_1 is NP-complete we have $\Pi \propto \Pi_1$
- since $\Pi \propto \Pi_1$, $\Pi_1 \propto \Pi_2$ and $\propto$ is transitive, it follows that $\Pi \propto \Pi_2$
- since $\Pi \in$ NP was arbitrary, $\Pi \propto \Pi_2$ for all $\Pi \in$ NP
- and hence Π_2 is NP-complete

Proving NP-completeness

The first NP-complete problem?

Name: Satisfiability (SAT)

Instance: Boolean expression **B** in **conjunctive normal form (CNF)**

- CNF: $C_1 \wedge C_2 \wedge \ldots \wedge C_n$ where each C_i is a clause
- Clause C: $(l_1 \vee l_2 \vee \ldots \vee l_m)$ where each l_j is a literal
- Literal l: a variable x or its negation $\neg x$

Question: is B satisfiable?

- i.e. can values be assigned to the variables that make B true?

Example:

- $B = (x_1 \vee x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_3 \vee \neg x_4) \wedge (\neg x_2 \vee x_4) \wedge$
  $(x_2 \vee \neg x_3 \vee x_4)$
- B is satisfiable: x_1=**true**, x_2=**false**, x_3=**true**, x_4=**true**

Proving NP-completeness

The first NP-complete problem?

Cook's Theorem (1971): Satisfiability (SAT) is NP-complete

- the proof consists of a generic polynomial-time reduction to SAT from an abstract definition of a general problem in the class NP
- the generic reduction could be instantiated to give an actual reduction for each individual NP problem

Given Cook's theorem, to prove a decision problem π is NP-complete it is sufficient to show that:

- π is in NP
- there exists a polynomial-time reduction from SAT to π

Clique is NP-complete

Name: Clique Problem (CP)

Instance: a graph G and a target integer K

Question: does G contain a clique of size K?

- i.e. a set of K vertices for which there is an edge between all pairs

To prove Clique is NP -complete

- show CP is in NP (straightforward)
  - guess the set of K vertices
  - check if it's a clique (in polynomial time)
  - get "yes"-instances and "no"-instances
- there exists a polynomial-time reduction from SAT to CP
  - proof at the end of the slide notes, not examinable
  - video of the proof and example available on Moodle

Problem restrictions

A restriction of a problem consists of a subset of the instances of the original problem

- if a restriction of a given decision problem Π is NP-complete, then so is Π
- given NP-complete problem Π, a restriction of Π might be NP-complete or it might be easier to solve

For example a clique restricted to cubic graphs is in P

- (a cubic graph is a graph in which every vertex belongs to 3 edges)
- a largest clique has size at most 4 so exhaustive search is $O(n^4)$
- for any target $K>4$ we directly return the answer "no"

While graph colouring restricted to cubic graphs is NP-complete

- not proved here

Problem restrictions

K-colouring

- restriction of Graph Colouring for a fixed number K of colours
- 2-colouring is in P (it reduces to checking the graph is bipartite)
- 3-colouring is NP-complete

K-SAT

- restriction of SAT in which every clause contains exactly K literals
- 2-SAT is in P (proof is a tutorial exercise)
- 3-SAT is NP-complete
- showing 3-SAT $\in$ NP is easy we will just find the polynomial-time reduction SAT $\propto$ 3-SAT

SAT $\propto$ 3-SAT

Given instance B of SAT will construct an instance B' of 3-SAT

For each clause C of B we construct a number of clauses of B'

- if $C = l_1$, we introduce 2 addition variables x_1 and x_2 and add the clauses $(l_1 \lor x_1 \lor x_2)$, $(l_1 \lor x_1 \lor \neg x_2)$, $(l_1 \lor \neg x_1 \lor x_2)$, $(l_1 \lor \neg x_1 \lor \neg x_2)$ to B'

- B' holds if and only if all the clauses $(l_1 \lor x_1 \lor x_2)$, $(l_1 \lor x_1 \lor \neg x_2)$, $(l_1 \lor \neg x_1 \lor x_2)$, $(l_1 \lor \neg x_1 \lor \neg x_2)$ hold (B' is a conjunction of clauses)

- for any assignment to x_1 and x_2 for all the clauses to hold requires l_1 to holds (be `true`)
- i.e. all clauses hold if and only if the clause C holds

SAT $\propto$ 3-SAT

Given instance B of SAT will construct an instance B' of 3-SAT

For each clause C of B we construct a number of clauses of B'

- if $C = l_1$, we introduce 2 addition variables x_1 and x_2 and add the clauses $(l_1 \lor x_1 \lor x_2)$, $(l_1 \lor x_1 \lor \neg x_2)$, $(l_1 \lor \neg x_1 \lor x_2)$, $(l_1 \lor \neg x_1 \lor \neg x_2)$ to B'

- if $C = (l_1 \lor l_2)$, we introduce 1 additional variable y and add the clauses $(l_1 \lor l_2 \lor y)$ and $(l_1 \lor l_2 \lor \neg y)$ to B'

- B' holds if and only if both the clauses $(l_1 \lor l_2 \lor y)$ and $(l_1 \lor l_2 \lor \neg y)$ hold

- for any assignment to y this requires $(l_1 \lor l_2)$ holds
  i.e. both clauses hold if and only if the clause C holds

SAT $\propto$ 3-SAT

Given instance B of SAT will construct an instance B' of 3-SAT

For each clause C of B we construct a number of clauses of B'

- if $C = l_1$, we introduce 2 addition variables x_1 and x_2 and add the clauses $(l_1 \vee x_1 \vee x_2)$, $(l_1 \vee x_1 \vee \neg x_2)$, $(l_1 \vee \neg x_1 \vee x_2)$, $(l_1 \vee \neg x_1 \vee \neg x_2)$ to B'
- if $C = (l_1 \vee l_2)$, we introduce 1 addition variable y and add the clauses $(l_1 \vee l_2 \vee y)$ and $(l_1 \vee l_2 \vee \neg y)$ to B'
- if $C = (l_1 \vee l_2 \vee l_3)$, we add the clause C to B'

SAT $\propto$ 3-SAT

Given instance B of SAT will construct an instance B' of 3-SAT
For each clause C of B we construct a number of clauses of B'

- if $C = l_1$, we introduce 2 addition variables x_1 and x_2 and add the clauses $(l_1 \vee x_1 \vee x_2), (l_1 \vee x_1 \vee \neg x_2), (l_1 \vee \neg x_1 \vee x_2)$, $(l_1 \vee \neg x_1 \vee \neg x_2)$ to B'

- if $C = (l_1 \vee l_2)$, we introduce 1 addition variable y and add the clauses $(l_1 \vee l_2 \vee y)$ and $(l_1 \vee l_2 \vee \neg y)$ to B'

- if $C = (l_1 \vee l_2 \vee l_3)$, we add the clause C to B'

- if $C = (l_1 \vee \ldots \vee l_k)$ and $k>3$, we introduce $k-3$ additional variables $z_1,\ldots,z_{k-3}$ and add the clauses $(l_1 \vee l_2 \vee z_1)$, $(\neg z_1 \vee l_3 \vee z_2)$, $(\neg z_2 \vee l_4 \vee z_3),\ldots,$ $(\neg z_{k-4} \vee l_{k-2} \vee z_{k-3})$, $(\neg z_{k-3} \vee l_{k-1} \vee l_k)$ to B'

SAT $\propto$ 3-SAT

Given instance B of SAT will construct an instance B' of 3-SAT

For each clause C of B we construct a number of clauses of B'

- if $C = l_1$, we introduce 2 addition variables x_1 and x_2 and add the clauses $(l_1 \vee x_1 \vee x_2)$, $(l_1 \vee x_1 \vee \neg x_2)$, $(l_1 \vee \neg x_1 \vee x_2)$, $(l_1 \vee \neg x_1 \vee \neg x_2)$ to B'
- if $C = (l_1 \vee l_2)$, we introduce 1 addition variable y and add the clauses $(l_1 \vee l_2 \vee y)$ and $(l_1 \vee l_2 \vee \neg y)$ to B'
- if $C = (l_1 \vee l_2 \vee l_3)$, we add the clause C to B'
- if $C = (l_1 \vee \ldots \vee l_k)$ and $k>3$, we introduce $k-3$ additional variables $z_1,\ldots,z_{k-3}$ and add the clauses $(l_1 \vee l_2 \vee z_1)$, $(\neg z_1 \vee l_3 \vee z_2)$, $(\neg z_2 \vee l_4 \vee z_3)$,…, $(\neg z_{k-4} \vee l_{k-2} \vee z_{k-3})$, $(\neg z_{k-3} \vee l_{k-1} \vee l_k)$ to B'
- again all clauses hold if and only if C holds

Coping with NP-completeness

What to do if faced with an NP-complete problem?

Maybe only a restricted version is of interest (which maybe in P)

- e.g. `2-SAT`, `2-colouring` are in `P`

Seek an exponential-time algorithm improving on exhaustive search

- e.g. backtracking, branch-and-bound
- should extend the set of solvable instances in a reasonable time

For an optimisation problem (e.g. calculating min/max value)

- settle for an approximation algorithm that runs in polynomial time
- especially if it gives a provably good result (within some factor of optimal)
- use a heuristic
  - e.g. genetic algorithms, simulated annealing, neural networks

For a decision problem

- settle for a probabilistic algorithm correct answer with high probability

Next – Section 5 – Computability

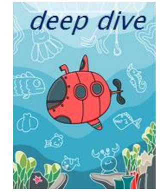
Introduction

The halting problem

Models of computation

- finite-state automata
- pushdown automata
- Turing machines
- Counter machines
- Church–Turing thesis

Clique is NP-complete: proof



Name: Clique Problem (CP)

Instance: a graph G and a target integer K

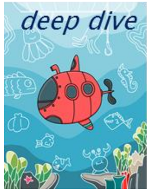
Question: does G contain a clique of size K?

- i.e. a set of K vertices for which there is an edge between all pairs

To prove Clique is NP -complete

- show CP is in NP (straightforward)
  - guess the set of K vertices
  - check if it’s a clique (in polynomial time)
  - get “yes”-instances and “no”-instances
- there exists a polynomial-time reduction from SAT to CP

Clique is NP-complete



To complete the proof we need to show **SAT $\propto$ CP**

- i.e. a polynomial time reduction from SAT to CP

This is not examinable – this is just to show you that it is possible to build PTRs between very different problems

Clique is NP-complete

To complete the proof we need to show SAT $\propto$ CP

- i.e. a polynomial time reduction from SAT to CP

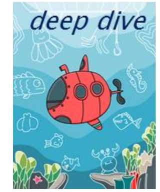
Given an instance B of SAT we construct (G,K) an instance of CP

- K number of clauses of B
- vertices of G are pairs (l,C) where l is a literal in clause C
- {(l,C),(m,D)} is an edge of G if and only if $l \neq \neg m$ and $C \neq D$
  - recall that $\neg(\neg x)=x$ so $l \neq \neg m$ is equivalent to $\neg l \neq m$
  - edge if distinct literals from different clauses can be satisfied simultaneously
- polynomial time construction ($O(n^2)$ where n is the number of literals)
  - worst case: to construct edges we need to compare every literal with every other literal

This is a polynomial time reduction since:

- B has a satisfying assignment if and only if G has a clique of size K

Clique is NP-complete



To prove it is a polynomial time reduction we can show:

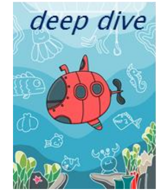
If B has a satisfying assignment, then

- if we choose a `true` literal in each clause the corresponding vertices form a clique of size K in G

If G has a clique of size K, then

- assigning each literal associated with a vertex in the clique to be `true` yields a satisfying assignment for B

Clique is NP-complete



Why does the construction work?

`{(l,C),(m,D)}` is an edge if and only if `l≠¬m` and `C≠D`

- only edges between literals in distinct clauses
- only edges between literals that can be satisfied simultaneously

Therefore in a clique of size K (recall K is the number of clauses)

- must include one literal from each clause (i.e. from K clauses)
- we can satisfy all the literals in the clique simultaneously
- this means we can satisfy all clauses
  - a clause is a disjunction of literals and we can satisfy one of the literals
- and therefore satisfy B
  - B is the conjunction of the clauses and we satisfy all the clauses

Clique is NP-complete – Example

$B = (x_1 \vee x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_3 \vee \neg x_4) \wedge (\neg x_2 \vee x_4) \wedge (x_2 \vee \neg x_3 \vee x_4)$

- there are $K = 4$ clauses

The graph G

- vertices of G are pairs
  (l, C) where l is a literal
  in clause C

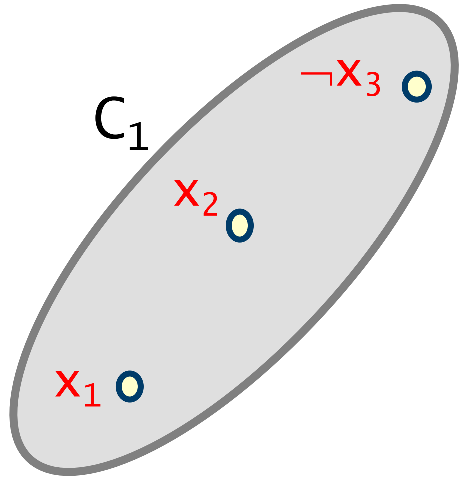
Clique is NP-complete – Example

$B = (x_1 \vee x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_3 \vee \neg x_4) \wedge (\neg x_2 \vee x_4) \wedge (x_2 \vee \neg x_3 \vee x_4)$

- there are $K = 4$ clauses

The graph G

- vertices of G are pairs (l, C) where l is a literal in clause C



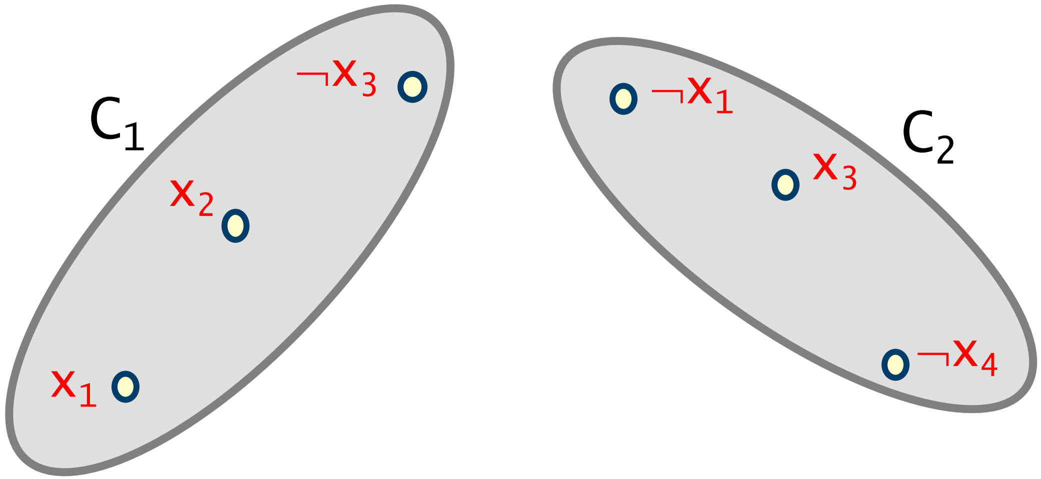
Clique is NP-complete – Example

$B = (x_1 \lor x_2 \lor \neg x_3) \land (\neg x_1 \lor x_3 \lor \neg x_4) \land (\neg x_2 \lor x_4) \land (x_2 \lor \neg x_3 \lor x_4)$

– there are $K = 4$ clauses

The graph G

– vertices of G are pairs (l,C) where l is a literal in clause C



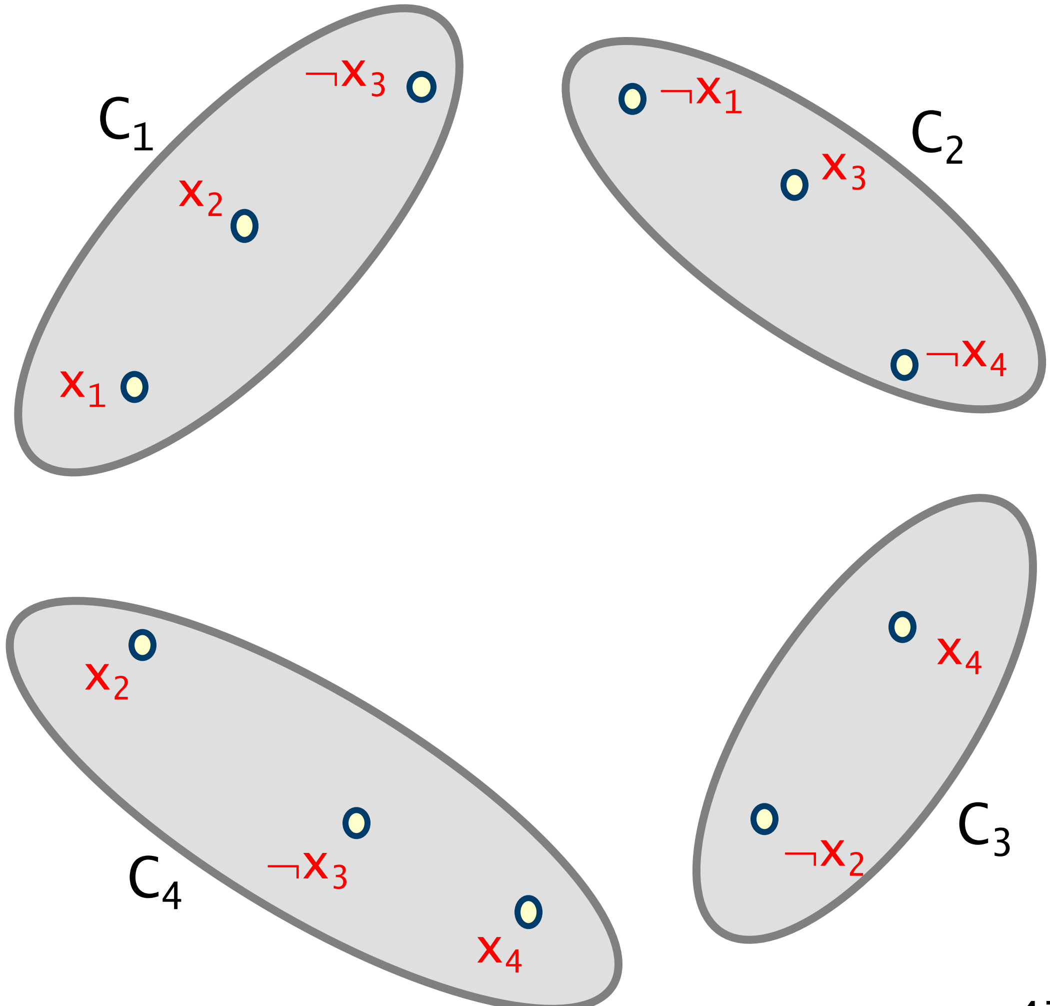
Clique is NP-complete – Example

$B = (x_1 \vee x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_3 \vee \neg x_4) \wedge (\neg x_2 \vee x_4) \wedge (x_2 \vee \neg x_3 \vee x_4)$

– there are $K = 4$ clauses

The graph G

– vertices of G are pairs (l, C) where l is a literal in clause C



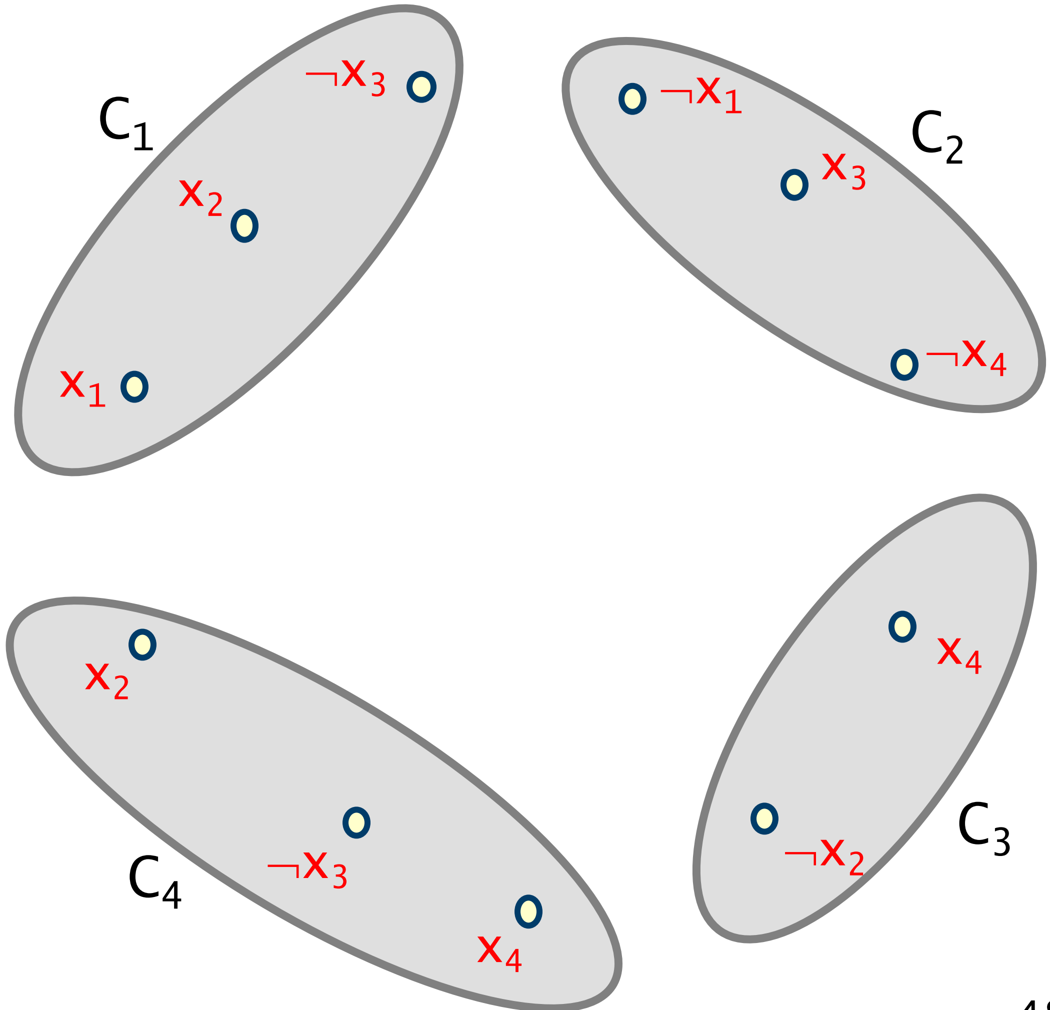
Clique is NP-complete – Example

$B = (x_1 \vee x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_3 \vee \neg x_4) \wedge (\neg x_2 \vee x_4) \wedge (x_2 \vee \neg x_3 \vee x_4)$

- there are $K = 4$ clauses

The graph G

- vertices of G are pairs (l, C) where l is a literal in clause C
- $\{(l, C), (m, D)\}$ is an edge if and only if $l \neq \neg m$ and $C \neq D$



Clique is NP-complete – Example

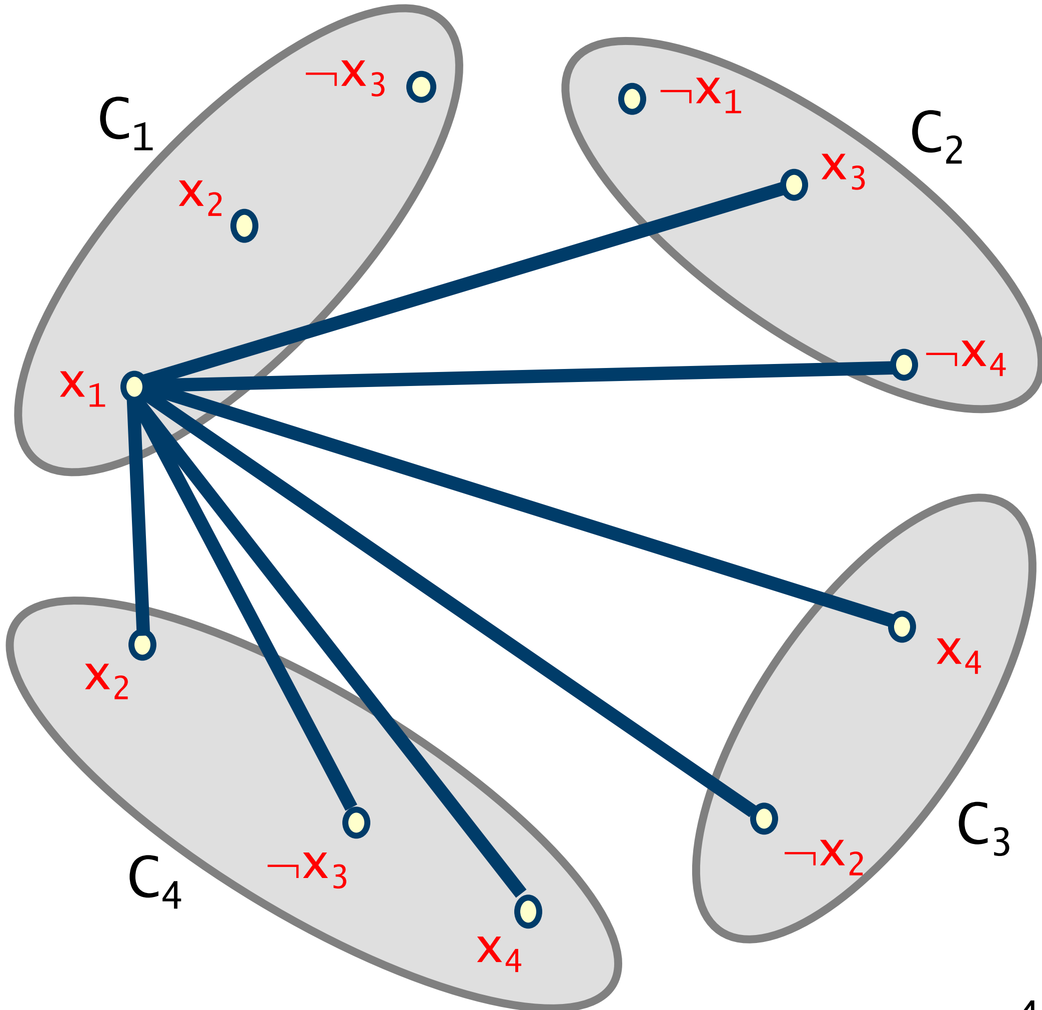
$B = (x_1 \vee x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_3 \vee \neg x_4) \wedge (\neg x_2 \vee x_4) \wedge (x_2 \vee \neg x_3 \vee x_4)$

– there are $K = 4$ clauses

The graph **G**

– vertices of G are pairs (l,C) where l is a literal in clause C

– {(l,C),(m,D)} is an edge if and only if $l \neq \neg m$ and $C \neq D$



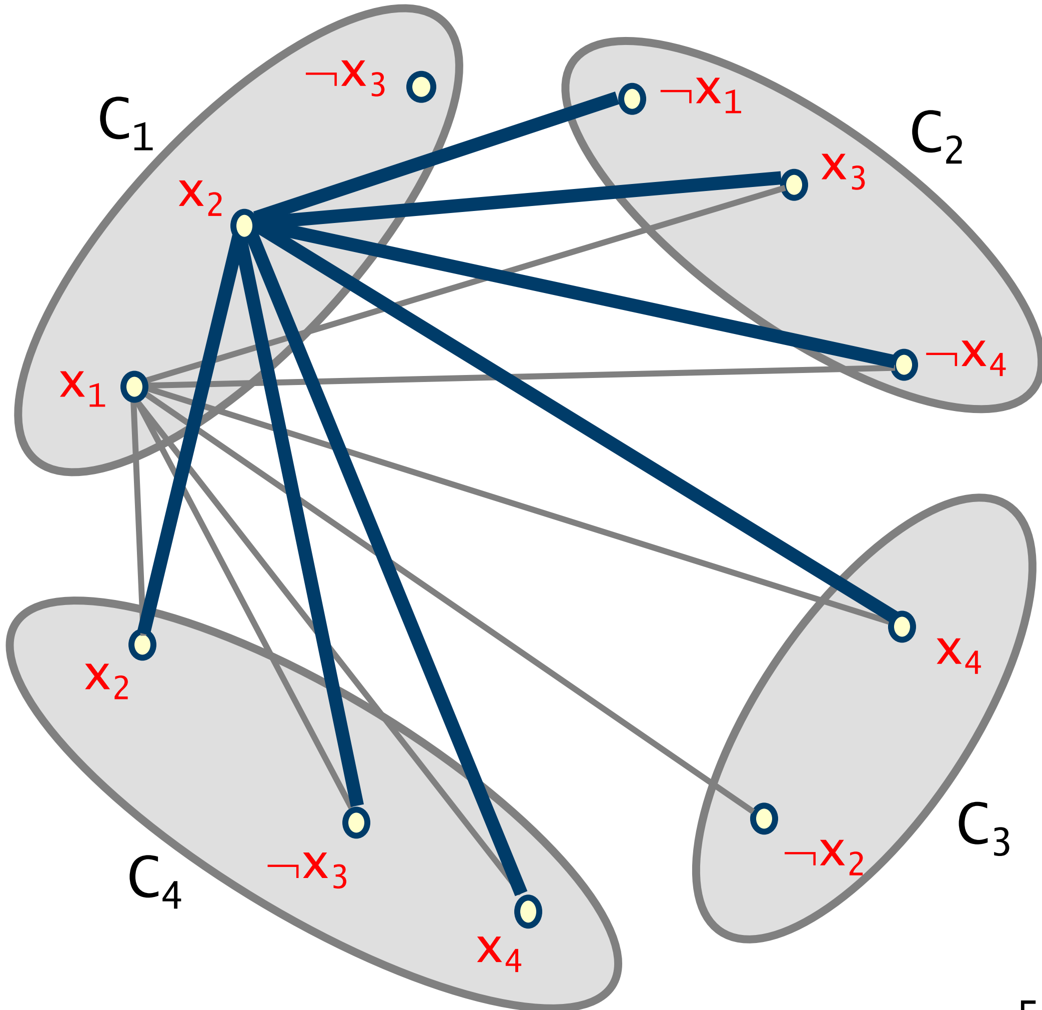
Clique is NP-complete – Example

$B = (x_1 \lor x_2 \lor \neg x_3) \land (\neg x_1 \lor x_3 \lor \neg x_4) \land (\neg x_2 \lor x_4) \land (x_2 \lor \neg x_3 \lor x_4)$

- there are $K = 4$ clauses

The graph G

- vertices of G are pairs (l, C) where l is a literal in clause C
- $\{(l,C),(m,D)\}$ is an edge if and only if $l \neq \neg m$ and $C \neq D$



Clique is NP-complete – Example

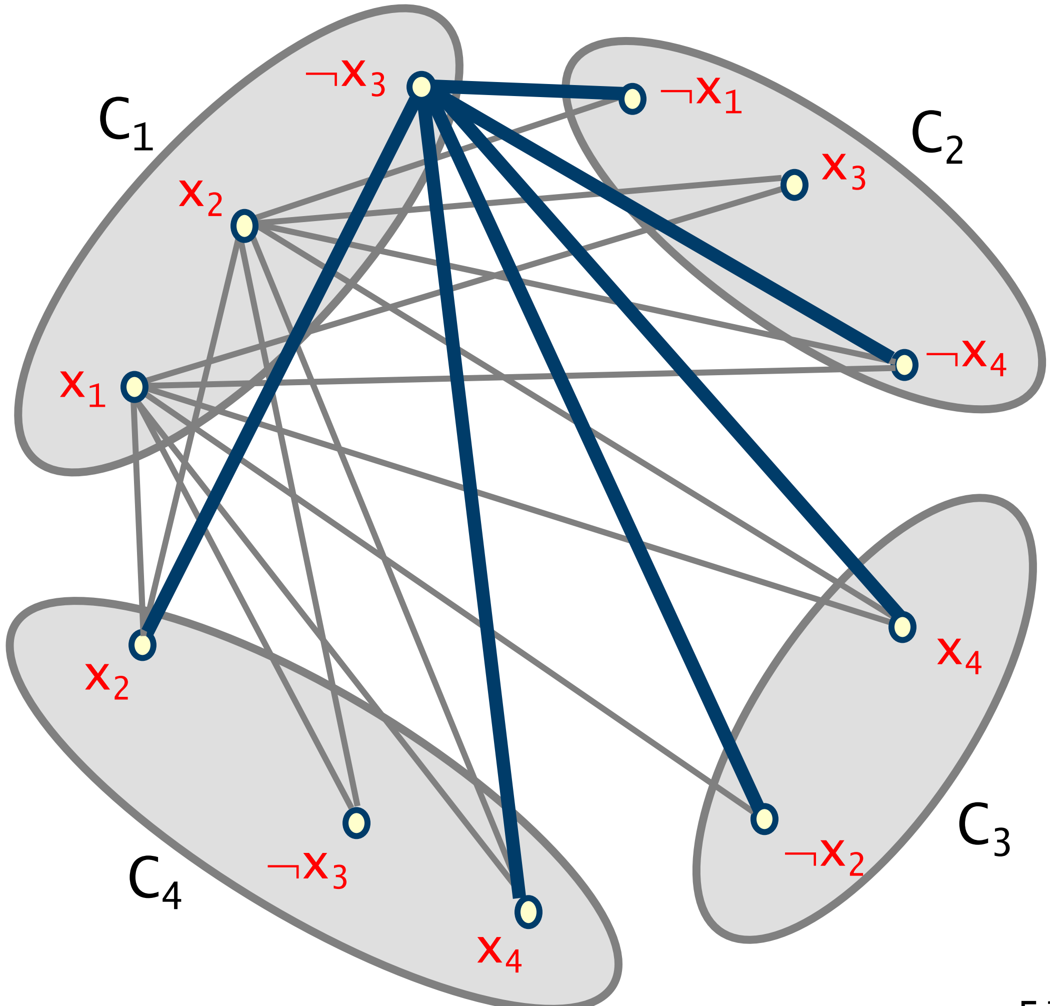
$B = (x_1 \lor x_2 \lor \neg x_3) \land (\neg x_1 \lor x_3 \lor \neg x_4) \land (\neg x_2 \lor x_4) \land (x_2 \lor \neg x_3 \lor x_4)$

– there are $K = 4$ clauses

The graph G

– vertices of G are pairs (l,C) where l is a literal in clause C

– {(l,C),(m,D)} is an edge if and only if $l \neq \neg m$ and $C \neq D$



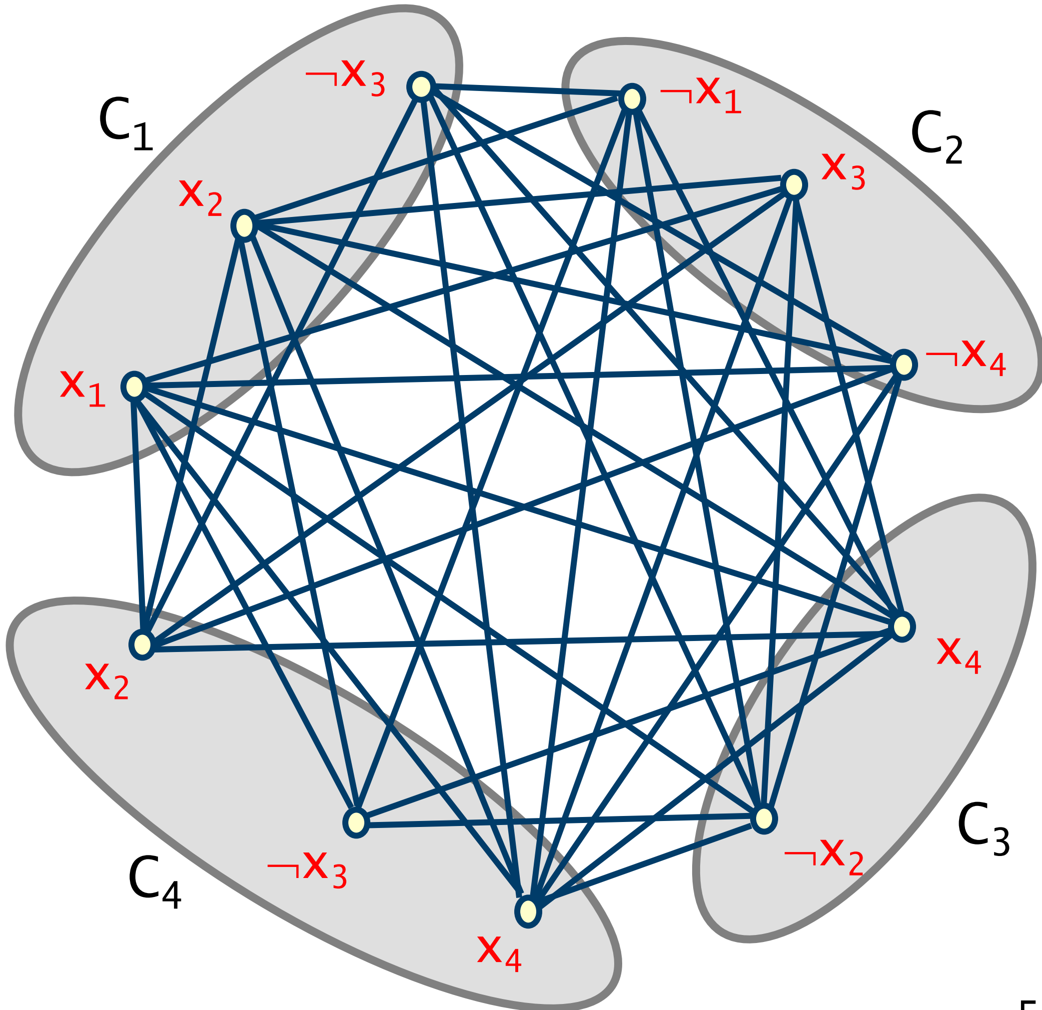
Clique is NP-complete – Example

$B = (x_1 \lor x_2 \lor \neg x_3) \land (\neg x_1 \lor x_3 \lor \neg x_4) \land (\neg x_2 \lor x_4) \land (x_2 \lor \neg x_3 \lor x_4)$

- there are $K = 4$ clauses

The graph G

- vertices of G are pairs (l, C) where l is a literal in clause C
- $\{(l, C), (m, D)\}$ is an edge if and only if $l \neq \neg m$ and $C \neq D$



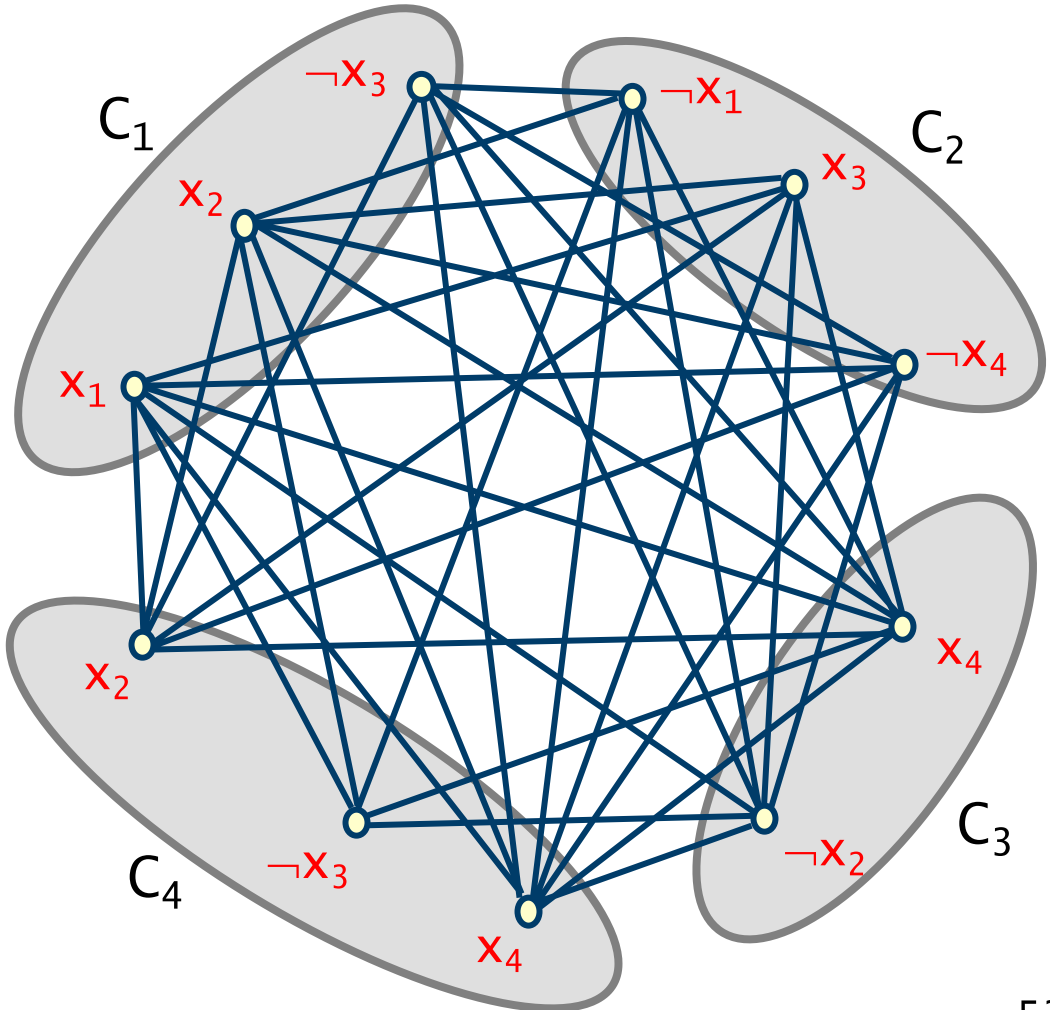
Clique is NP-complete

$B = (x_1 \vee x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_3 \vee \neg x_4) \wedge (\neg x_2 \vee x_4) \wedge (x_2 \vee \neg x_3 \vee x_4)$

– there are $K = 4$ clauses

The graph **G**

G has a clique of size 4
if and only if
B has a satisfying assignment

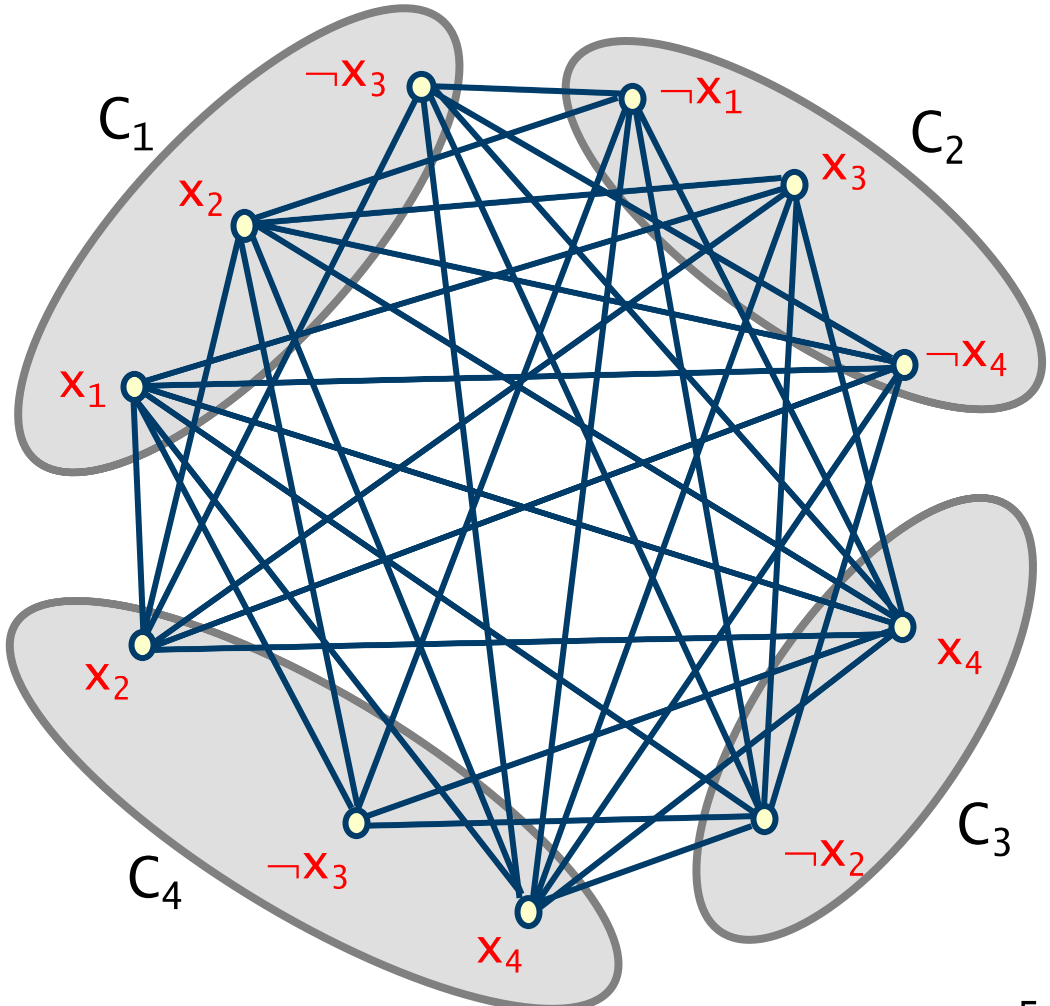


Clique is NP–complete

$B = (x_1 \vee x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_3 \vee \neg x_4) \wedge (\neg x_2 \vee x_4) \wedge (x_2 \vee \neg x_3 \vee x_4)$

– there are $K = 4$ clauses

The graph **G**



G has a clique of size 4
if and only if
B has a satisfying assignment

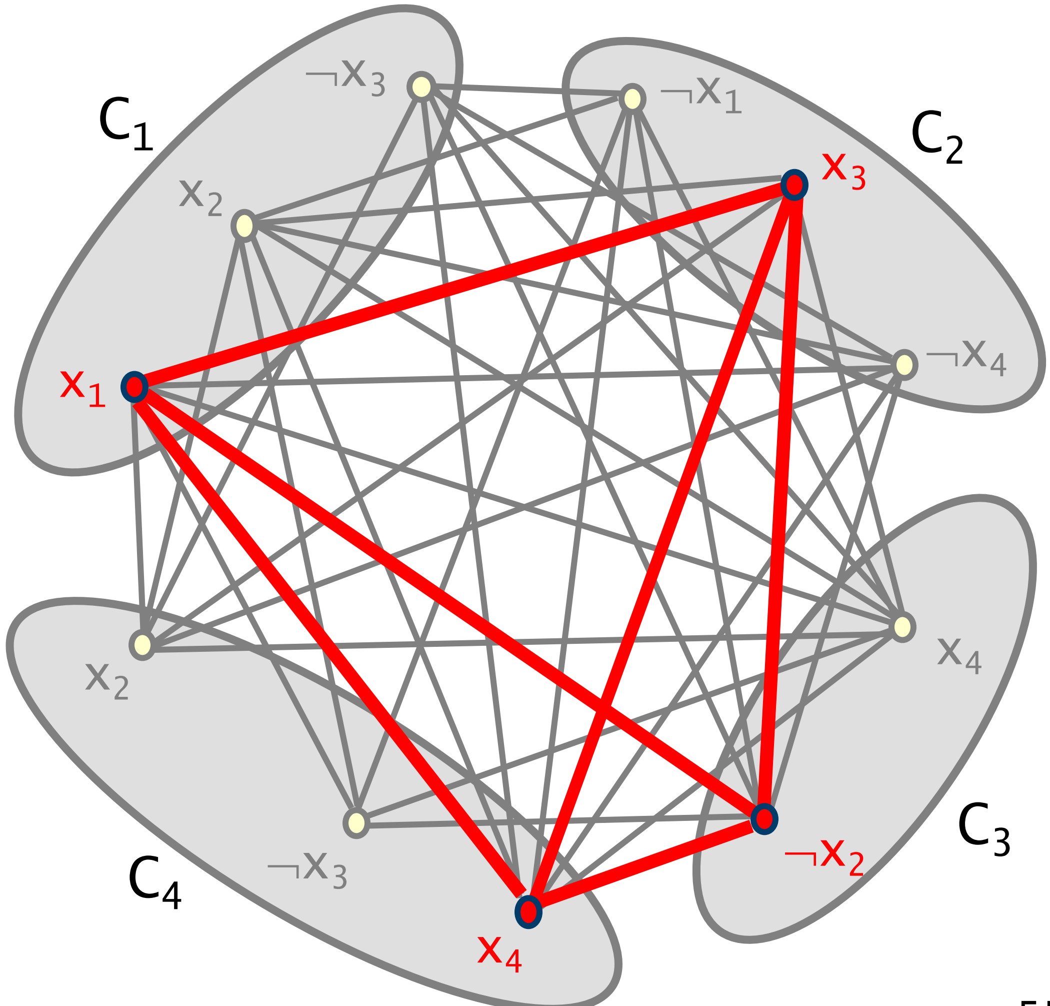
satisfying assignment

Clique is NP-complete

$B = (x_1 \vee x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_3 \vee \neg x_4) \wedge (\neg x_2 \vee x_4) \wedge (x_2 \vee \neg x_3 \vee x_4)$

– there are $K = 4$ clauses

The graph **G**



G has a clique of size 4
if and only if
B has a satisfying assignment

satisfying assignment:
$x_1, \neg x_2, x_3, x_4$ are true
clique of size 4