Analysis of Algorithms

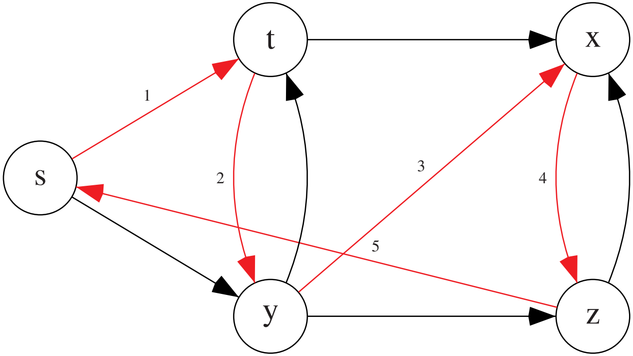
Class NP

Hikmat Farhat

April 7, 2020

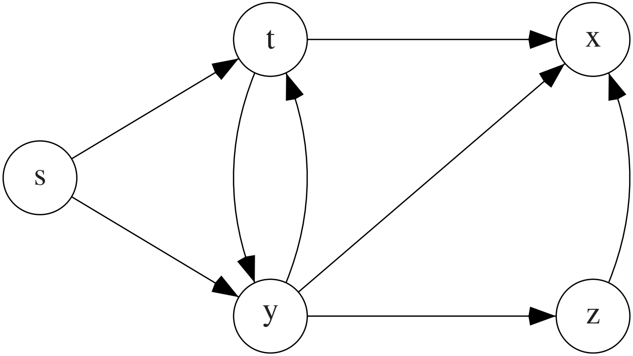
Hamiltonian Cycle

- A Hamiltonian cycle in a graph $G = (V, E)$ is a sequence of non-repeating vertices (except the first and last) such that each pair of vertices are connected in the graph.
- For example one Hamiltonian cycle in the graph below is the sequence s, t, y, x, z, s (there are others, e.g. s, y, t, x, z, s)



Decision Problems

- While the graph in the previous slide contains multiple Hamiltonian cycles, the one below does not contain **any**



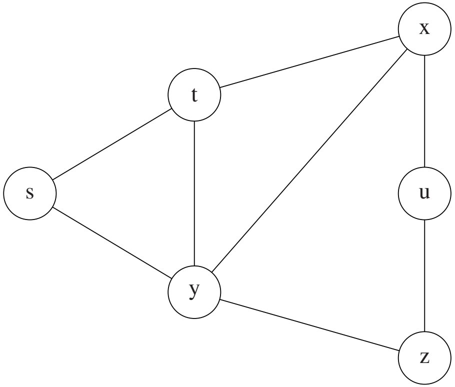
- The question of whether a graph contains a Hamiltonian cycle is called a **decision problem**.

Formally

- Define the set H as:
- $H = \{$the set of all graphs $G | G$ has a Hamiltonian cycle$\}$
- Then given a graph x we need to decide if $x \in H$ or $x \notin H$.
- To do so we need an algorithm A that takes a graph x as input and outputs **yes or no** depending on whether $x \in H$ or not:
  1. $A(x) = yes$ then $x \in H$
  2. $A(x) = no$ then $x \notin H$
- One can find an algorithm A to solve the problem but, unfortunately, no one has been able to find an **efficient** one.
- In fact there are many interesting(decision and optimization) problems that no one has found an efficient algorithm for yet.

Decision vs Optimization problems

- Some problems are not decision problems, rather, they can be classified as optimization problems, e.g. the maximal independent set.
- Given a graph $G = (V, E)$ an independent set is a subset $S \subseteq V$ of vertices such that for all $u, v \in S$ we have $(u, v) \notin E$.
- In the graph below $\{t, z\}$ is I.S. as is $\{y, x\}$ but we want the **largest** one i.e. $\{s, x, z\}$



- Any optimization problem, like the maximal independent set, can be recast as a sequence of decision problems.
- In the case of the I.S. in the previous graph we convert it into a sequence of decision problems:
- We know there is always an IS of size 1
- Is there an IS of size 2? yes
- Is there an IS of size 3?yes
- IS there an IS of size 4? no
- Therefore the maximal independent set has size 3.
- Note: since the size of the independent set cannot be more that $|V| = n$ then if the decision problem has a polynomial time algorithm then the optimization problem will also have a polynomial time algorithm.

Verifiers

- As mentioned before, many of the decision problems do not (yet) have an efficient algorithm.
- Can we find a common characteristic for some of these problems?
- It turns out that for many of them, if we are given a potential solution, we can **efficiently verify** that it is indeed a solution.

- Let H be the set of graphs that have a Hamiltonian cycle.
- Let x be an encoding of a graph and s be a list of vertices of x (certificate).
- $B(x, s)$ is called a **verifier** of H if the following two conditions hold:
  1. for all $x \in H$ $\exists s$ such that $B(x, s)$=yes
  2. for all $x \notin H$ and $\forall s$, $B(x, s)$ =no
- If B is a polynomial time algorithm and $|s| = O(|x|^c)$ for some c then B is said to be an **efficient verifier of** H

Class NP

- We define the class of problems NP (Non-deterministic polynomial time) as all the problems that have **efficient verifiers**
- As an example, the following polynomial algorithm is an efficient verifier for H.
- Given a graph $G = (V, E)$ and a list of vertices L
- First check that the first and the last element of L are the same, if not return false
- Then check that no vertex is repeated in L (except the first and last)
- Finally check that consecutive vertices are connected in G
- The pseudo-code for the above is given in the next slide.

Hamiltonian Cycle has efficient verifier: $HC \in NP$

```
Verify($G, L$)
/* check first and last vertex are equal */
if $L[1] \neq L[n]$ then
    return false
/* check that no vertex is repeated (except first and
   last) */
for $i = 1$ to $n - 1$ do
    for $j = i + 1$ to $n - 1$ do
        if $L[i] = L[j]$ then
            return false
/* check that consecutive vertices are connected */
for $i = 1$ to $n - 1$ do
    if $(L[i], L[i + 1]) \notin E$ then
        return false
/* passed all checks, $L$ is a Hamiltonian cycle */
return true
```

IS has efficient verifier: $IS \in NP$

- Given a graph $G = (V, E)$ the following polynomial time algorithm verifies that L is an independent set of size k

```
Verify($G, L, size$)

/* check that size is $k$                                   */
if $size \neq k$ then
    return false
/* check that no vertex is repeated and no two are
   connected                                                */
for $i = 1$ to $n - 1$ do
    for $j = i + 1$ to $n$ do
        if $L[i] = L[j]$ then
            return false
        if $(L[i], L[j]) \in E$ then
            return false
/* passed all checks, $L$ is an independent set of size $k$
   */
return true
```

Boolean Satisfiability

- A boolean formula is an expression made of boolean variables that can be assigned the values of TRUE or FALSE (0 or 1), boolean operators $\wedge$ (conjunction),$\vee$ (negation),$\neg$ (disjunction) and possibly parenthesis.
- A **literal** is a variable or a negation of a variable (i.e. x or $\neg x$).
- A **clause** is a disjunction of literals (or a single literal).
- A boolean formula is said to be satisfiable if there is an assignment to the variables such that the expression evaluates to true.
- The **boolean satisfiability problem** (**SAT**) is the problem of deciding whether a boolena formula is satisfiable.
- A special case of SAT is **3SAT**, when the formula is formed by a conjunction of clauses, where each clause has at most three literals

$SAT, 3SAT \in NP$

- Let n be the number of variables and k be the number of clauses. The double array $F[i][j]$ gives the index of variable j in clause i.
- For example $F[2][3] = 7$ means that the third variable in clause 2 is x_7
- $A[i]$ is the value (0 or 1) assigned to variable x_i
- The following is an efficient verifier for 3SAT where A is an array of size n containing the value of the n boolean variables (assignment)

Efficient verifier for 3*SAT*

```
Verify($F, A$)
/* check that each clause is satisfied                    */
for $i = 1$ to $k$ do
    $sum \leftarrow 0$
    for $j = 1$ to 3 do
        $sum \leftarrow sum + A[F[i][j]]$
    if $sum = 0$ then
        return false
/* all clauses are satisifed                              */
return true
```

Example

- Consider the following $3SAT$ formula with three clauses and five variables

$$(x_1 \vee x_2 \vee \neg x_5) \wedge (x_3 \vee \neg x_2 \vee \neg x_4) \wedge (\neg x_4 \vee \neg x_3 \vee \neg x_2)$$

- The formula is satisfied with the assignment $x_1 = 1, x_2 = 0$ (regardless of the values of x_3,x_4 and x_5)
- One can convince you that the formula is satisfiable by supplying a **short** proof which in this case the assignment of the variables.
- You can check (in polynomial time, refer to the previous slides) that the supplied assignment indeed makes the formula satisfiable.
- Whereas the converse is not true. Consider the formula below

Unsatisfiable formula

- How can one convince you that the formula below is unsatisfiable? Can they supply a **short** proof?
- You have to check all possibilities

$$(x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \neg x_2) \wedge (x_2 \vee \neg x_3) \wedge (x_3 \vee \neg x_1) \wedge \quad (\neg x_1 \vee \neg x_2 \vee \neg x_3)$$

Using Z3

- Even though SAT is NP-complete, which means no polynomial time algorithm exists yet for solving it, many SAT solvers are very efficient solvers of SAT instances
- They use backtracking (later) and heuristics.
- We will give examples using **Z3**
- The input to Z3 is a set of disjunctive clauses. This means that the input formula is a conjunctive normal form (CNF).

Example1

- The satisfiable formula below

$$(x_1 \vee x_2 \vee \neg x_5) \wedge (x_3 \vee \neg x_2 \vee \neg x_4) \wedge (\neg x_4 \vee \neg x_3 \vee \neg x_2)$$

- Is input into a file, say formula.cnf, for Z3 as follows

```
(declare-const x1 Bool)
(declare-const x2 Bool)
(declare-const x3 Bool)
(declare-const x4 Bool)
(declare-const x5 Bool)
(assert (and
          (or x1 x2 (not x5))
          (or x3 (not x2) (not x4))
          (or (not x4) (not x3) (not x2))
        )
)
(check-sat)
(get-model)
```

- run "z3 formula.cnf". The first line says "sat" which means the formula is satisfiable and then it prints the assignment.

```
sat
(model
  (define-fun x3 () Bool
    false)
  (define-fun x2 () Bool
    false)
  (define-fun x1 () Bool
    false)
  (define-fun x5 () Bool
    false)
  (define-fun x4 () Bool
    false)
)
```

- The unsatisfiable formula

$$(x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \neg x_2) \wedge (x_2 \vee \neg x_3) \wedge (x_3 \vee \neg x_1) \wedge \quad (\neg x_1 \vee \neg x_2 \vee \neg x_3)$$

- Is written as

```
(declare-const x1 Bool)
(declare-const x2 Bool)
(declare-const x3 Bool)
(assert (and
          (or x1 x2 x3)
          (or x1 (not x2) )
          (or x2 (not x3) )
          (or  x3 (not x1))
          (or (not x1) (not x2) (not x3))
        )
)
(check-sat)
(get-model)
```

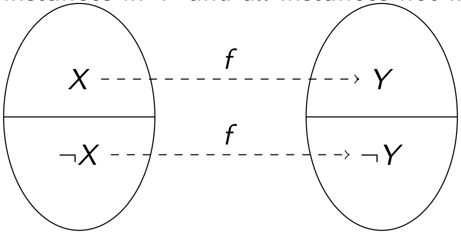
- when we run z3 on the above input we get

```
unsat
(error "line 13 column 10: model is not available")
```

Polynomial time reductions

- Let Y and X be decision problems.
- We say that Y is polynomial time reducible to X, and write $Y \leq_p X$ if
- If there exists a function (or procedure) f such that
  1. f is polynomial time
  2. $v \in Y \Leftrightarrow f(v) \in X$
- The importance of reduction is that if one has a polynomial time algorithm for X it can be used to solve instances of Y in polynomial time.

As shown in the figure below, a reduction f maps all instances in X to instances in Y and all instances not in X to instances not in Y.



- Theorem: If $Y \leq_p X$ and $X \in P$ then $Y \in P$.
- **Proof.**
- $X \in P$ then there exists a polynomial time algorithm B that decides X. This means that for any u, $B(u) = True \Leftrightarrow u \in X$.
- $Y \leq_p X$ then there exists a polynomial time function f such that $v \in Y \Leftrightarrow f(v) \in X$.
- Let v be any instance. Compute $f(v) = u$ in polynomial time.
- Decide u in polynomial time using B
- $B(u) =$true then $f(v) = u \in X$ and therefore $v \in Y$ return true
- $B(u) =$false then $f(v) \notin X$ and therefore $v \notin Y$ return false
- The above is a procedure to decide in polynomial time any instance y of Y and thus $Y \in P$.
- We are actually more interested in the **contrapositive**

Hard problems

- Theorem: If $Y \leq_p X$ and $Y \notin P$ then $X \notin P$.
- Proof.
- By contradiction. Assume that $X \in P$ then there exists a polynomial time algorithm to decide X.
- Since $Y \leq_P X$ then using the previous theorem we know there exists a polynomial time algorithm to decide Y which is a contradiction since $Y \notin P$.
- The above theorem gives us a notion of "hardness" of a problem.
- $Y \leq_P X$ can be interpreted that X is at least as "hard" to solve as Y.
- This observation lead us to the important class of problems called NP-complete.

NP-complete Problems

- A problem X is said to be NP-complete if
  1. $X \in NP$
  2. For all $Y \in NP$ we have $Y \leq_p X$.
- So in a sense NP-complete problems are the "hardest" problems in NP.
- So far we have shown the following relations
  1. $P \subseteq NP$
  2. NP-complete $\subseteq NP$.
- But are P and NP-complete proper susbsets of NP ?
- Most researchers believe so, but it hasn't been proved yet!

SAT is NP-complete

- Theorem (Cook 1971): SAT is NP-complete.
- We will use 3SAT as our starting point to show some of well known problems to be NP-complete

SAT to 3SAT

- Any formula ϕ in CNF can be transformed into a formula ψ in CNF with each clause having at most three literals such that ψ is satisfiable iff ϕ is satisfiable.
- The transformation works as follows:
  1. Given a clause $C = (l_1 \vee l_2 \vee A)$ where l_1 and l_2 are literals and A is a disjunction of k literals.
  2. Introduce a new variable y and convert C into $C' = (l_1 \vee l_2 \vee y) \wedge (\bar{y} \vee A)$
  3. Note we started with a clause that has $k + 2$ literals into two clauses of size 3 and $k + 1$.

Correctness

- Suppose that $(l_1 \vee l_2 \vee A)$ is satisfiable then
  1. Either $(l_1 \vee l_2)$ is satisfiable then set $y = 0$ so $(l_1 \vee l_2 \vee y) \wedge (\bar{y} \vee A)$ is satisfiable.
  2. Or A is satisfiable then set $y = 1$ so $(l_1 \vee l_2 \vee y) \wedge (\bar{y} \vee A)$ is satisfiable.
- Suppose that $(l_1 \vee l_2 \vee y) \wedge (\bar{y} \vee A)$ is satisfiable then
  1. l_1, l_2 and A cannot all be false, i.e. one of them is true so $(l_1 \vee l_2 \vee A)$ is satisfiable.

- Let $A = l_3 \vee B$ then we have (B has $k-1$ literals)

$$(l_1 \vee l_2 \vee y) \wedge (\bar{y} \vee l_3 \vee B)$$

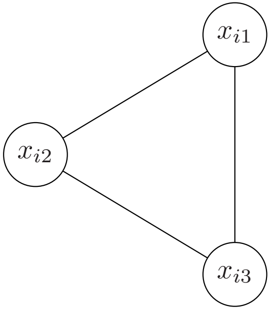
- We repeat our construction one more time by adding a variable z

$$(l_1 \vee l_2 \vee y) \wedge (\bar{y} \vee l_3 \vee z) \wedge (\bar{z} \vee B)$$

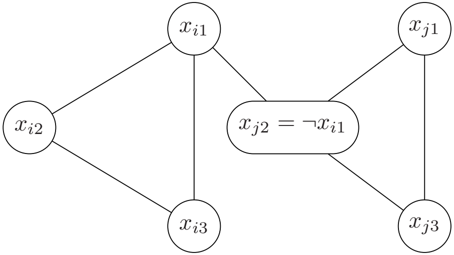
- We repeat the above procedure until no clauses has more than 3 literals.
- Since in the original formula each clause has at most n literals we need at most $O(n)$ operations to reduce a clause to at most 3 literals.
- Starting from k clauses our procedure is $O(k \cdot n)$, clearly polynomial in the number of variables.
- Since SAT is NP-complete and we have a polynomial time reduction from SAT to 3SAT then 3SAT is NP-complete.

3SAT$\leq_p$ Independent Set

- We prove that a k-clause 3SAT decision reduces to k-independent set decision.
- For every literal x_{ij}, $i = 1, \dots, k, j = 1, 2, 3$, create a graph vertex labeled x_{ij}.
- Nodes representing literals belonging to the same clause are mutually connected (see below for clause i)



- For every literal in clause i, x_{il}, if its negation occurs in clause $j \neq i$ then the vertices representing these two variables are joined with an edge.
- Example below where variable x_{l2} in clause l is the negation of x_{i1} in clause i



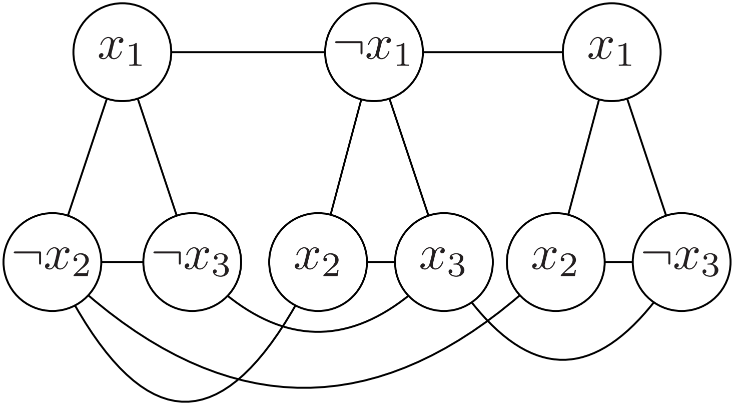
- Suppose that a k-clause 3SAT instance is satisfiable. Then each clause has at least one variable set to true. Select from each clause one variable that is set to true to obtain k variables (not necessarily distinct but from different clauses) set to true. Now consider the corresponding set of nodes. Since the 3SAT instance is satisfiable the set of variables cannot contain a variable and its complement then nodes corresponding to different clauses are not connected. Furthermore, since we chose one node (corresponding to a variable) from every clause implies that the chosen set is independent and has size k.
- Suppose that an independent set of k nodes exists. Then the variables corresponding to the nodes do not contain any variable and its negation together therefore it is safe to set the variables to true which implies that each clause of the 3SAT problem has at least one true variable, thus the 3SAT instance is satisfiable

Example

- Consider the 3SAT formula

$$\phi = (x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee \neg x_3)$$

- It is reduced to an instance of IS as shown below



Independent Set $\leq_p$ Vertex Cover

- Given a graph $G = (V, E)$ a subset of vertices $S \subseteq V$ is said to be a vertex cover iff for every edge $(u, v) \in E$ either $u \in S$ or $v \in S$.
- Usually we look for the **smallest** vertex cover in a graph.
- The corresponding decision problem is: given a graph $G = (V, E)$ is there a vertex cover of size k. Clearly V is a vertex cover of size n.
- We show that deciding vertex cover of size k is NP-complete by reducing it to independent set of size $n - k$.

Proof

- Suppose that S is a vertex cover. We need to show that $V - S$ is an independent set.
- Assume otherwise, then there exists two vertices $u, v \in V - S$ and $(u, v) \in E$. This implies that both ends of the edge (u, v) do not belong to S which means that S is not a vertex cover. A contradiction. Therefore $V - S$ is an independent set.
- Conversely assume $V - S$ is an independent set. We need to show that S is a vertex cover.
- Assumed otherwise, then there exists an edge $(u, v) \in E$ such that both $u \notin S$ and $v \notin S$ hence $u \in V - S$, $v \in V - S$ and $(u, v) \in E$ therefore $V - S$ is not an independent set. A contradiction.

Independent Set $\leq_p$ Clique

- Given a graph $G = (V, E)$ a clique is a subset of vertices $S \subset V$ such that for any $u, v \in S$ we have $(u, v) \in E$.
- Next we describe a polynomial time reduction from Independent set of size k to a clique of size k.
- Let $G = (V, E)$ be a graph and construct from G a complement graph $G^c = (V, E^c)$ such that $(u, v) \notin E$ iff $(u, v) \in E^c$.
- G has an independent set of size k iff G^c has a clique of size k.
- **Proof:** Suppose S is an independent set of G of size k we show that S is a clique of G^c of size k. Let $(u, v) \in S$, since S is independent set of G then $(u, v) \notin E$ and therefore $(u, v) \in E^c$

3SAT $\leq_p$ Clique

- Another way of showing that clique is NP complete is a reduction from 3SAT that is similar to the reduction to independent set.
- We will use the reduction from independent set to clique as a guide.
- For every literal in every clause we create a vertex. Vertices corresponding to literals in the same clause are not connected (opposite for IS). Every node representing a literal in clause i is connected to every node representing literal in clause $j \neq i$ unless these literals are complements to each other (again the opposite for IS).
- The above construction gives us that $3SAT$ with k clauses is satisfiable iff the corresponding graph has a k clique.

Useful SAT encoding

- Because SAT solvers are very efficient it convenient to reduce many problems to SAT.
- In the previous section we concentrated on reduce SAT (or 3SAT) to other problems to show them NP-complete.
- When looking for a solution usually we would like to perform the opposite reduction.
- To do that it is useful to review some convenient SAT encoding of common situations

At least one, two, three,...

- Given n boolean variables $x_1, \ldots, x_n$.
- One of the common situations is to require at least k of them to be true.
- We start with a small example and then we generalize.
- Suppose we have three variables x_1, x_2, x_3
- If we require that *at least one* is true can be written as a single clause $(x_1 \vee x_2 \vee x_3)$.
- what about at least two?three?
- At least two $(x_1 \vee x_2) \wedge (x_1 \vee x_3) \wedge (x_2 \vee x_3)$
- At least three is just $x_1 \wedge x_2 \wedge x_3$.

Generalization

- For n variables $x_1, \ldots x_n$ at least k requires all combinations of $n - k + 1$ variables.
- At least two requires all combinations of $n - 2 + 1$
- For x_1, x_2, x_3, x_4 this means all combinations of 3 variables

$$(x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee x_4) \wedge (x_1 \vee x_3 \vee x_4) \wedge (x_2 \vee x_3 \vee x_4)$$

- At least three requires all combinations of $4 - 3 + 1$ variables

$$\begin{aligned} &(x_1 \vee x_2) \wedge (x_1 \vee x_3) \wedge (x_1 \vee x_4) \\ \wedge &(x_2 \vee x_3) \wedge (x_2 \vee x_4) \wedge (x_3 \vee x_4) \end{aligned}$$

Python itertools

- itertools is a python package that is useful in listing different combinations of variables
- for example to list all combination of 2 and 3 variables from a set of 4 variables we can write

```
import itertools
vars=[1,2,3,4]
for (i,j) in itertools.combinations(vars,2):
    print(i,j)

for (i,j,k) in itertools.combinations(vars,3):
    print(i,j,k)
```

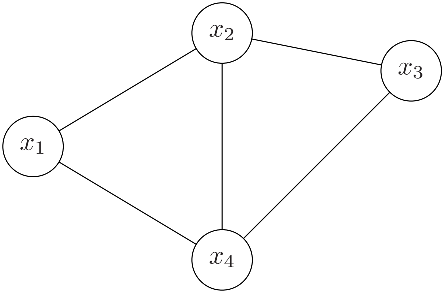
Solving Indepent set

- Given a graph (V, E) with $|V| = n$ we would like to determine if there is an independent set of size $k \leq n$.
- We can reduce this problem to SAT as follows
  1. We assign a variable x_i to each node i.
  2. Since we are looking for size k then at least k variables should be true
  3. For each edge (x_i, x_j) add a constraint that both of them cannot be true

Example

- The graph below has 4 nodes so we create 4 boolean variables x_1, x_2, x_3, x_4.
- We are looking for an independent set of size 2 so we need at least two variables to be true: all combinations of 4-(2-1)=3

$$(x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee x_4) \wedge (x_1 \vee x_3 \vee x_4) \wedge (x_2 \vee x_3 \vee x_4)$$



- We also need to add the edge constraints

$$(\bar{x}_1 \vee \bar{x}_2) \wedge (\bar{x}_1 \vee \bar{x}_4) \wedge (\bar{x}_2 \vee \bar{x}_3) \wedge (\bar{x}_2 \vee \bar{x}_4) \wedge (\bar{x}_3 \vee \bar{x}_4)$$

- The z3 input will look like

```
(declare-const x1 Bool)
(declare-const x2 Bool)
(declare-const x3 Bool)
(declare-const x4 Bool)
(assert (or x1 x2 x3))
(assert (or x1 x2 x4))
(assert (or x1 x3 x4))
(assert (or x2 x3 x4))
(assert (or (not x1)(not x2)))
(assert (or (not x1)(not x4)))
(assert (or (not x2)(not x3)))
(assert (or (not x2)(not x4)))
(assert (or (not x3)(not x4)))
(check-sat)
(get-model)
```

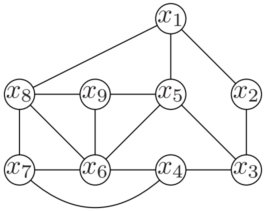
- We run z3

```
z3 independent.txt

sat
(model
  (define-fun x3 () Bool
    true)
  (define-fun x2 () Bool
    false)
  (define-fun x1 () Bool
    true)
  (define-fun x4 () Bool
    false)
)
So x1 and x3 are selected as an independent set of size 2.
```

Another IS example

- Running the same algorithm on figure below produces a solution of $\{x_1, x_3, x_7, x_9\}$. The Python code can be found at https://github.com/hikmatfarhat-ndu/Python-exercises
- How would you find another solution? Add a condition that not all the above are true at the same time. This would give another solution $\{x_2, x_4, x_5, x_8\}$.



k-Coloring to SAT

- Given a graph $G = \langle V, E \rangle$ and k-colors
- is it possible to assign a color to every vertex such that no two neighbors have the same color?
- This is called the k-coloring problem
- Later we will show that 3-coloring (and thus k-coloring) is NP-complete by reducing 3SAT to 3-Coloring.
- Now we need to do the opposite: reduce k-coloring to SAT to be able to solve k-coloring instances.

Exactly one is true

- A common occurrence is when we have k variables and we want **exactly one** of them to be true.
- From our previous discussion this is equivalent to

$$(x_1 \vee x_2 \vee \ldots \vee x_k) \bigwedge_{i \neq j} (\bar{x}_i \vee \bar{x}_j)$$

- Let x_{ij}, $1 \leq i \leq n$, $1 \leq j \leq k$ be a boolean variable such that $x_{ij} = 1$ iff node i has color j.
- Since a node can be assigned exactly one color then: at least one is true and at most one is true
- Thus for every node i we have a clause:

$$(x_{i1} \vee x_{i2} \vee \ldots \vee x_{ik}) \bigwedge_{l \neq m} (\bar{x}_{il} \vee \bar{x}_{im})$$

- Also for every two neighbors i, j cannot have the same color so

$$(\bar{x}_{i1} \vee \bar{x}_{j1}) \wedge \ldots \wedge (\bar{x}_{ik} \vee \bar{x}_{ik})$$

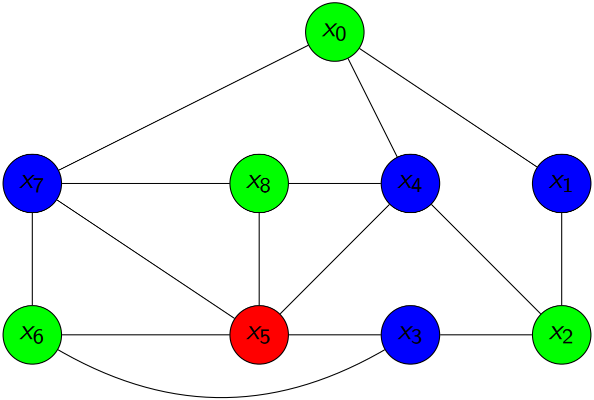
Encoding

- Given n nodes and k-colors we encode the variables as follows
- the first node we associate nodes $1, \ldots, k$ for the k colors.
- The second node $x_{k+1}, \ldots, x_{2k}$
- In general for node i we associate the indices $k \cdot i - k + 1, k \cdot i - k + 2, \ldots, k \cdot i - 1$
- In Python, let colors be an array of colors then the function below returns the variable number for node i color j

```
def varnum(i,j):
  return len(colors)*i-j+1
```

Example

- Consider again the graph we used to test for independent set and check if it admits 3-coloring. Using z3 we get the solution in the figure below.



Mini sudoku

- Before we use a SAT solver to solve sudoku we try semi-manually to solve a mini version of sudoku
- The mini sudoku is just a 2x2 grid.
- As an example consider the grid below

1	
2	

- clearly the answer is setting the top white cell to 2 and the bottom to 1
- We will see how we encode it as a SAT problem

mini sudoku encoding

- The rules are
  1. All cells in the same row cannot have the same value
  2. All cells in the same column cannot have the same value
  3. Each cell can contain only one value
- To implement the above we introduce variables $x_{ijk} = 1$ if cell (i, j) contains the value k.
- For example in the grid above we have $x_{111} = 1$ and $x_{212} = 1$.

- All cells in row 1 cannot have the same value translates into

$$(x_{111} \vee x_{121}) \wedge (\bar{x}_{111} \vee \bar{x}_{121})$$
$$\wedge(x_{112} \vee x_{122}) \wedge (\bar{x}_{112} \vee \bar{x}_{122})$$

- All cells in row 2 cannot have the same value translates into

$$(x_{211} \vee x_{221}) \wedge (\bar{x}_{211} \vee \bar{x}_{221})$$
$$\wedge(x_{212} \vee x_{222}) \wedge (\bar{x}_{212} \vee \bar{x}_{222})$$

- All cells in column 1 cannot have the same value translates into

$$(x_{111} \vee x_{211}) \wedge (\bar{x}_{111} \vee \bar{x}_{211})$$
$$\wedge(x_{112} \vee x_{212}) \wedge (\bar{x}_{112} \vee \bar{x}_{212})$$

- All cells in column 2 cannot have the same value translates into

$$(x_{121} \vee x_{221}) \wedge (\bar{x}_{121} \vee \bar{x}_{221})$$
$$\wedge(x_{122} \vee x_{222}) \wedge (\bar{x}_{122} \vee \bar{x}_{22})$$

- Finally, each cell cannot have both values

$$
\begin{gathered}
(x_{111} \vee x_{112}) \wedge (\bar{x}_{111} \vee \bar{x}_{112}) \\
(x_{121} \vee x_{122}) \wedge (\bar{x}_{121} \vee \bar{x}_{122}) \\
(x_{211} \vee x_{212}) \wedge (\bar{x}_{211} \vee \bar{x}_{212}) \\
(x_{221} \vee x_{222}) \wedge (\bar{x}_{221} \vee \bar{x}_{222})
\end{gathered}
$$

- Also $x_{111} = 1$ and $x_{212} = 1$.
- We enter these clauses as input to minisat by renaming the 8 variables as : $x_{111} = 1, x_{121} = 2, x_{112} = 3, x_{122} = 4$ $x_{211} = 5, x_{221} = 6, x_{212} = 7, x_{222} = 8$.
- The encoding is in the file mini-sudoku.txt on blackboard.

Solving Sudoku

- Unlike the mini sudoku it is almost impossible to write the needed clauses by hand
- We will write a Python code to generate the needed clauses
- A central computation that we will need often is when exactly one variable is true
- Also we need an automatic encoding of the variables x_{ijk}.
- We define a function that returns the variable number given (i, j, k) as $100 * i + 10 * j + k$.
- In Python

```
def varnum(i,j,k):
    return 100*i+10*j+k
```

- We define a function that takes an input a list of literals and adds the clauses that guarantees that only one of them is true.
- Note that below we use the itertools.combinations which returns all combinations of 2 variables from a list of literals.

```
def exactly_one_of(literals):
    # at least one is true
    solver.add(z3.Or([l for l in literals]))
    # no two can be true
    for u,v in itertools.combinations(literals,2):
        solver.add(z3.Or(z3.Not(u),z3.Not(v)))
```

- Using the above we can add the constraints
    1. For each cell (i, j) exactly one value is true
    2. For each row value k appears exactly once
    3. For each column value k appears exactly once
    4. For each 3x3 block value k appears exactly once
    5. Finally, we add the clauses that make the filled cells to be true

Dealing with 3x3 blocks

- Note the "origin" of each 3x3 block below.
- From the "origin" x the block is defined as $(x+\delta_i, x+\delta_j)$ where $\delta_i = 0, 1, 2$ and $\delta_j = 0, 1, 2$.

	1	2	3	4	5	6	7	8	9
1	x_{11}			x_{14}			x_{17}		
2									
3									
4	x_{41}			x_{44}			x_{47}		
5									
6									
7	x_{71}			x_{74}			x_{77}		
8									
9									

- The Python code for the sudoku is on black board

SAT Modulo Theories

- A extension of SAT that allows us to reason about other than boolean variables
- SMT can be regarded as a constraint satisfaction problem
- For example if x, y, z are real numbers or integers
- are the following constraints satisfiable?
- $x + y + z = 10$
- $2 \leq x \leq 7$
- $y - z = 2$ etc.

Example: Job Scheduling

- We will solve an instance of job scheduling using the z3 smt solver
- z3 can be used stand alone with input a file in smt-lib format
- It also has a binding for other languages C++, C#, Python

Example: Job Scheduling

- Suppose that we have 6 jobs of length (arbitrary units)
- $L = [4, 5, 6, 7, 8, 9]$
- Further suppose that you have three workers (A,B and C) referred to as 1,2 and 3 to complete the jobs
- Let $s[i], e[i], p[i]$ be the starting time, ending time and the worker that performed job i
- We have the following constraints
- $e[i] - s[i] = L[i]$.
- Suppose that job 3 cannot start until jobs 2 and 6 finish
- No worker can perform more than one job at a time
- Jobs 1 and 4 require the special skills of worker 2

Example: using z3

- Z3 uses the smt-lib syntax which is very similar to Lisp
- First we declare all the needed variables s, e, p

```
(declare-const s1 Int)
...
(declare-const s6 Int)
(declare-const e1 Int)
...
(declare-const p6 Int)
```

- Make sure the length of job 1 is 4 and job 2 is 5 etc..

```
(assert (= e1 (+ s1 4)))
(assert (= e2 (+ s2 5)))
...
```

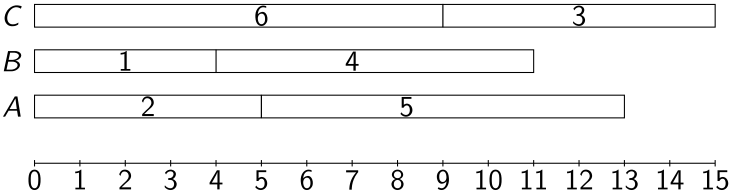
- If jobs 1 and 2 are performed by the same worker make sure that they are not done concurrently

```
(assert (=> (= p1 p2) (or (>= s1 e2) (>= s2 e1))))
....
```

- Make sure job 3 cannot start until jobs 2 and 6 finish

```
(assert (and (>= s3 e2) (>=s3 e6)))
```

Solution


C
6
3
B
1
4
A
2
5
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15