Topic 10

Graph and Circuit: From CS to EE applications

資料結構與程式設計
Data Structure and Programming

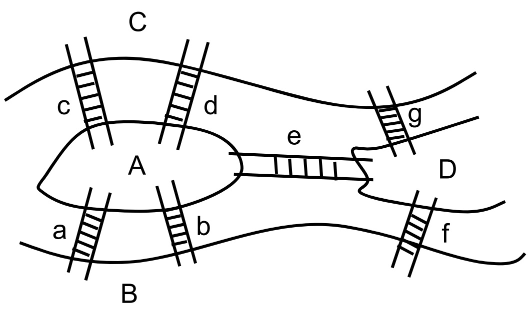
Sep, 2011

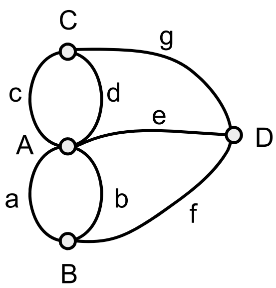
CS to EE? What does that mean?

- Most people think that “Data Structure” is a CS class
  - A “must” subject for CS entrance exam
- In EE area, many problems can be either mapped as graphic problems, or resolved by graphic algorithms
  - e.g. Circuit netlist, network, etc.
  - Understanding graphic data structure and algorithms will be very helpful

The First Use of Graph

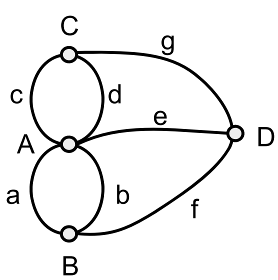
- Köigsberg Bridge Problem
  - Leonhard Euler, 1736
  - Starting at one land, is it possible to walk across all bridges exactly once and returning to the starting land area?





Eulerian Theorem

- There is a walk starting at any vertex, going through each edge exactly once and terminating at the starting vertex, iff the degree of each vertex is even.
  - ➔ Eulerian walk

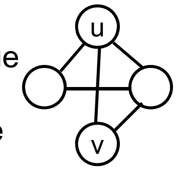


No Eulerian walk, since all 4 vertices are of odd degree.

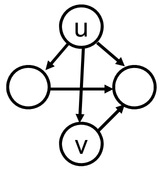
Definition of a Graph

- A graph, G(V, E)
  - V: a finite, nonempty set of vertices → V(G)
  - E: a set of pairs of vertices
    these pairs are called edges → E(G)

1. Undirected Graph
   - If every pair of vertices representing any edge is unordered
   - i.e. (u, v) and (v, u) represent the same edge



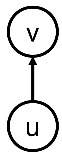
2. Directed Graph (Digraph)
   - Order of the pair of vertices matters
   - <u, v>: ‘u’ is the tail and ‘v’ is the head
   - e.g. A circuit is a directed graph



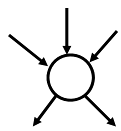
Terminologies

- Given 2 nodes u, v, and an edge (u, v)
  - u and v are called *adjacent*
  - The edge (u, v) is *incident* on vertices u and v

➔ If <u, v> is a directed edge
  - u is *adjacent to* v, and v is *adjacent from* u



- *Degree* of a vertex
  - The number of edges incident to it



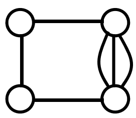
➔ If the graph is directed
  - *In-degree*
    - The number of edges for which the vertex is the head
  - *Out-degree*
    - The number of edges for which the vertex is the tail

Terminologies

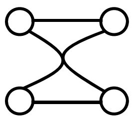
- *Path*
  - A sequence of vertices in which each vertex is adjacent to the next one
    - e.g. { n_1, e_1, n_2, e_2, n_3, e_3, ..., e_{k-1}, n_k }
- *Simple path*
  - All vertices in a path are distinct
- *Length of a path*
  - The number of edges in a path
- *Loop (self-edge)*
  - An edge with 2 identical end-points
- *Cycle*
  - A path with identical start and end points

Graph Properties

- Subgraph G(V', E') of G(V, E)
  - V' $\subseteq$ V; E' $\subseteq$ E
- Simple graph
  - No loops and no two edges link the same vertex pair

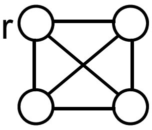


- Multigraph
  - Not simple graph
- Weighted graph
  - Each edge is associated with some weight
- Hypergraph
  - An extension of a graph where edges may be incident to any number of vertices



Complete Graph

- Complete graph
  - Each vertex is adjacent to all the other vertices in the graph
  - For complete graph with n vertices
    - #edges = n (n – 1) / 2

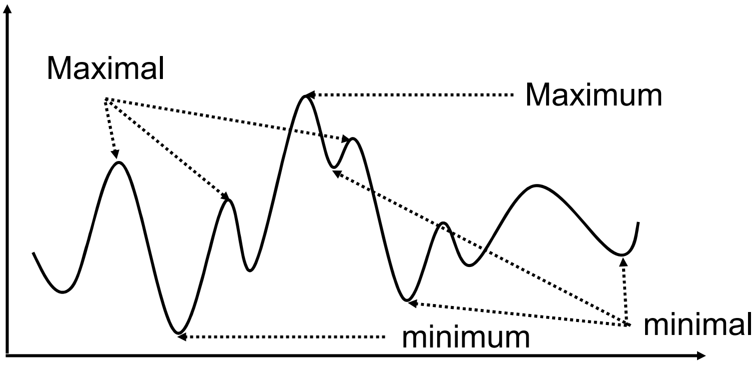


- Clique of a graph
  - Complete subgraph
- Complement G(V', E') of a graph G(V, E)
  - $V' = V; E \cap E' = \varnothing$
  - $G(V, E \cup E')$ is a complete graph

Undirected Graph Properties

- Two vertices u and v are said to be connected
  - Iff there a path from u to v
- A graph is said to be connected
  - Iff every vertex pair is connected
  - ➔ A tree is a connected acyclic graph
- A connected component (or simply *component*) of a graph
  - A maximal connected subgraph

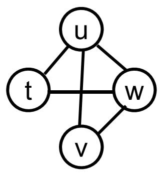
(FYI) Maximal vs. Maximum

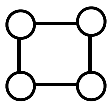


- In many problems, finding Maximum/minimum is very hard
  - Finding maximal/minimal is the only possibility
- How to find a better Maximal/minimal?

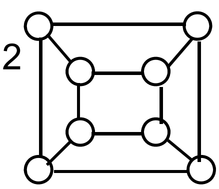
Undirected Graph Properties

- Cutset
  - A minimal set of edges whose removal from the graph makes the graph disconnected



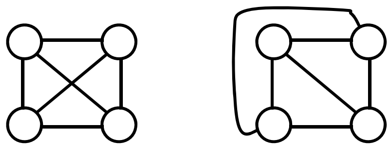


- Bipartite graph
  - Vertex set can be partitioned into 2 subsets such that each edge has end-points in different subsets

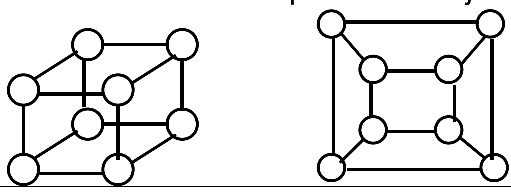


Undirected Graph Properties

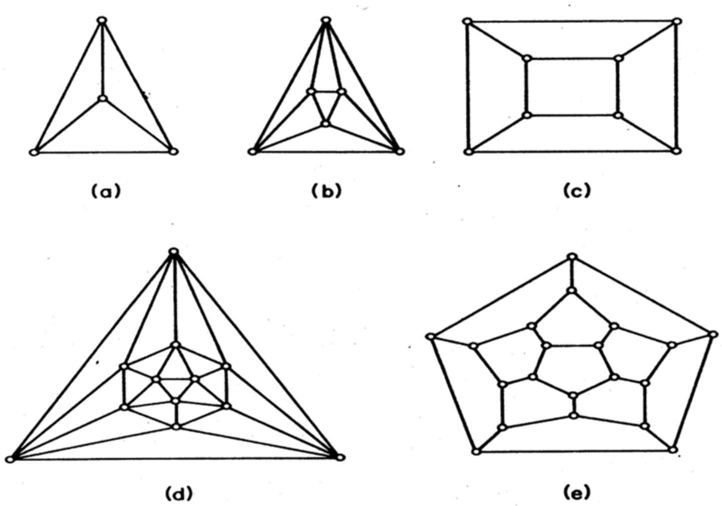
- Plannar graph
  - A diagram on a plane surface such that no two edges cross



- Two graphs are isomorphic
  - There is a one-to-one correspondence between their vertex sets and preserves adjacency



Some interesting plannar graphs



(*a*) The tetrahedron; (*b*) the octahedron; (*c*) the cube; (*d*) the icosahedron; (*e*) the dodecahedron

Undirected Graph Properties

- Each undirected graph can be characterized by four numbers

1. *Clique number* ω(G)
   - The cardinality of its largest clique, called *clique number*
2. *Chromatic number* χ(G)
   - The minimum number of colors needed to color the vertices, such that no edge has end-points with the same color
   - e.g. A bipartite graph is a 2-colorable graph

Property: $\omega(G) \leqq \chi(G)$

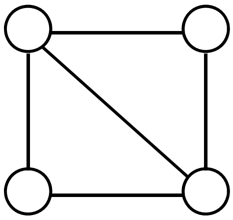
Undirected Graph Properties

3. *Clique cover number* κ(G)
   - A graph is said to be *partitioned into cliques* if its vertex set is partitioned into (disjoint) subsets, each one including a clique
   - The cardinality of a minimum clique partition is called *Clique cover number*
4. *Stability number* α(G)
   - A *stable set*, or *independent set*, is a subset of vertices with the property that no two vertices in the stable set are adjacent
   - The *stability number* is the cardinality of its largest stable set
   - A *coloring* of a graph is a partition of the vertices into subsets, such that each is a stable set

Property: $\alpha(G) \leqq \kappa(G)$

Perfect Graph

- A graph is said to be perfect iff
  - $\omega(G) = \chi(G)$ (clique = chromatic)
  - $\alpha(G) = \kappa(G)$ (stability = clique covering)



$\omega(G) = \chi(G) = 3$

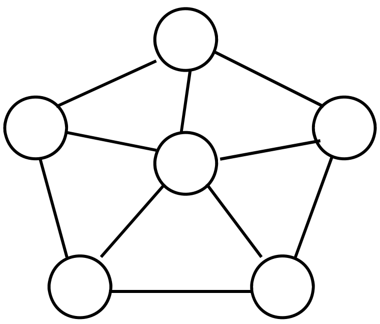
$\alpha(G) = \kappa(G) = 2$

Appendix

- Example:

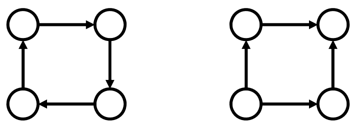
  Clique number $\omega(G)$ ≦ *Chromatic number* $\chi(G)$

  Stability number $\alpha(G)$ ≦ *Clique cover number* $\kappa(G)$



Directed Graph Properties

- A digraph is said to be strongly connected
  - Iff for every pair of distinct vertices u and v, there is a path from u to v, also from v to u



- Strongly connected component (SCC)
  - Maximal subgraph that is strongly connected
  - If a graph is strongly connected, it has only one SCC
  - Linear time algorithm for finding SCCs: **Robert E. Tarjan**, *Depth-first search and linear graph algorithms*, SIAM Journal on Computing, 1(2):146-160, 1972.

Directed Acyclic graph (DAG)

- A directed graph that has no cycle
- Can represent *partially ordered set*
  - A vertex v is a successor (descendant) of a vertex u
    - If there is a path from u to v
    - Called direct successor if the path is an edge
  - Predecessor (ancestor)
- Polar DAG
  - A DAG with 2 distinguished vertices
    - A source and a sink
  - All vertices are reachable from the source
  - Sink is reachable from all the vertices
  - A generic polar DAG may have multiple sources and sinks

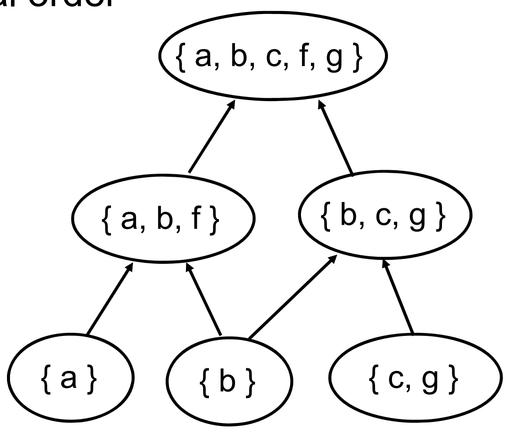
Partially vs. Totally Ordered Set

- A relation "≤" is a partial order on a set *S* if it has:
  1. Reflexivity: $a \leq a$ for all $a \in S$
  2. Antisymmetry: $a \leq b$ and $b \leq a$ implies $a = b$.
  3. Transitivity: $a \leq b$ and $b \leq c$ implies $a \leq c$

- A relation "≤" is a total order on a set *S* if it has the above 3 properties and the following:
  4. Comparability (trichotomy law):
     For any $a, b \in S$, either $a \leq b$ or $b \leq a$

A Partial Order Example

- The "containment" relation of a set is a partial order



Graphic Algorithms

- The importance of learning "graphs" is that many practical problems can be modeled and then solved by standard/well-known graphic algorithms
  1. Breadth-First Search and Depth-First Search
  2. Topological Sort
  3. Strongly Connected Component
  4. Shortest and Longest Path Algorithms
  5. Minimum Spanning Tree
  6. Maximum Flow and Minimum Cut
- Please refer to "Algorithm" book or class for more information
  - We may cover some of them if we have time...

Graph Traversal

- In many graph (DAG) applications, it is important to go through every vertex in certain order
  - e.g. checkSum(), simulate(), etc
- Topological order
  - An order sorted by certain relationship of adjacent vertices
  - e.g.
    - For each vertex, it has higher order than all of its predecessors, and lower order than all of its successors

Depth-First Traversal (Take 1)

```
void
Graph::dfsTraversal(const List<Node*>& srcList)
{
   for_each_source(node, srcList)
      node->dfsTraversal(_dfsList);
}
// post order traversal
void Node::dfsTraversal(List<Node *>& dfsList)
{
   for_each_successor(next, _successors)
      next->dfsTraversal(dfsList);
   dfsList.push_back(this);
}
```

Any Problem??

Depth-First Traversal (Take 2)

```
void
Graph::dfsTraversal(const List<Node*>& srcList)
{
   for_each_source(node, srcList)
      node->dfsTraversal(_dfsList);
}
// post order traversal
void Node::dfsTraversal(List<Node *>& dfsList)
{
   for_each_successor(next, _successors)
      if (!next->isMarked()) {
         next->setMarked();
         next->dfsTraversal(dfsList);
      }
   dfsList.push_back(this);
}
```

Any Problem??

Depth-First Traversal (Take 3)

```
void
Graph::dfsTraversal(const List<Node*>& srcList)
{
   for_each_source(node, srcList)
      node->dfsTraversal(_dfsList);
   for_each_node(node, _dfsList)
      node->unsetMarked();
}
// post order traversal
void Node::dfsTraversal(List<Node *>& dfsList)
{
   for_each_successor(next, _successors)
      if (!next->isMarked()) {
         next->setMarked();
         next->dfsTraversal(dfsList);
      }
   dfsList.push_back(this);
}
```

Any Problem??

Depth-First Traversal (Take 4)

- Remember: use “static data member in a class”

```
class T
{
   static unsigned   _globalRef;
   unsigned          _ref;

public:
   T() : _ref(0) {}
   bool isGlobalRef(){ return (_ref == _GlobalRef); }
   void setToGlobalRef(){ _ref = _globalRef; }
   static void setGlobalRef() { _globalRef++; }
}
```

- Use this method to replace “setMarked()” functions in graph traversal problems

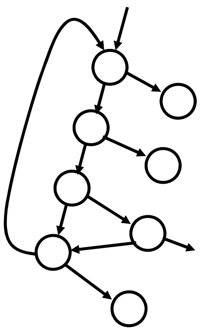
Depth-First Traversal (Take 4)

```
void
Graph::dfsTraversal(const List<Node*>& srcList)
{
   Node::setGlobalRef();
   for_each_source(node, srcList)
      node->dfsTraversal(_dfsList);
}
// post order traversal
void Node::dfsTraversal(List<Node *>& dfsList)
{
   for_each_successor(next, _successors)
      if (!next->isGlobalRef()) {
         next->setToGlobalRef();
         next->dfsTraversal(dfsList);
      }
   dfsList.push_back(this);
}
```

Any Problem??

Depth-First Traversal (Take 5)

```
void
Graph::dfsTraversal(const List<Node*>& srcList)
{
   Node::setGlobalRef();
   for_each_source(node, srcList)
      node->dfsTraversal(_dfsList, _fbList);
}
// post order traversal
void Node::dfsTraversal
(List<Node *>& dfsList, list<Node*>& fbList)
{
   for_each_sucecessor(next, _successors)
      if (!next->isGlobalRef()) {
         next->setToGlobalRef();
         next->setActive();
         next->dfsTraversal(dfsList, fbList);
         next->unsetActive();
      }
      else if (next->isActive()) fbList.push_back(this, next);
   dfsList.push_back(this);      // not push_back(next); why?
}
```



Breath-First Traveral

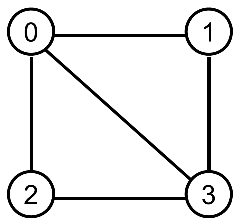
```
algorithm levelOrder(TreeNode t)
   Input: a tree node (can be considered to be a
      tree)
   Output: None.

   Let Q be a Queue
   Q.enqueue(t)
   while the Q is not empty
      tree = Q.dequeue()
      Visit node tree
      if tree has a left child
            Q.enqueue(left child of tree)
      if tree has a right child
            Q.enqueue(right child of tree)
```

How about the “marked” and “loop” Issues ??

Graph Implementation (1)

- Adjacency Matrix

```
class Graph
{
   bool _adjacency[n][n];
};
```



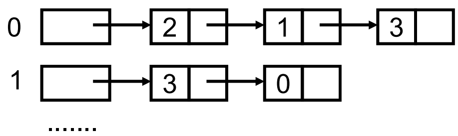
- For undirected graph → upper triangle
- How to perform traversal?
- Difficult to implement various graphic algorithms
- Could be a sparse matrix
- Complexity can be as high as $O(n^2)$

$$\begin{array}{c c} & \begin{array}{cccc} 0 & 1 & 2 & 3 \end{array} \\ \begin{array}{c} 0 \\ 1 \\ 2 \\ 3 \end{array} & \begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix} \end{array}$$

Graph Implementation (2)

- Adjacency List

```
class Graph
{
   List<int>   *_headNodes;
   int          _numNodes;
};
```



$$\begin{array}{c|cccc} & 0 & 1 & 2 & 3 \\ \hline 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 1 \\ 2 & 1 & 0 & 0 & 1 \\ 3 & 1 & 1 & 1 & 0 \end{array}$$

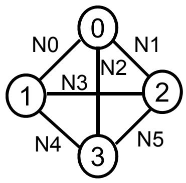
- Better for sparse matrix
- Require n _headNodes and 2*e ListNodes
- (u, v) and (v, u) redefined
- Some operations may still be as expensive as O(n + e)

Graph Implementation (3)

- Adjacency Multilist

```
class Edge
{
   bool     _visited;
   int      _vertex1, vertex2;
   Edge    *_path1, *_path2;
};
class Graph
{
   Edge**   _headNodes;
   int      _numNodes;
};
```

		_vertex1	_vertex2	_path1	_path2
N0		0	1	N1	N3
N1		0	2	N2	N3
N2		0	3	0	N4
N3		1	2	N4	N5
N4		1	3	0	N5
N5		2	3	0	0

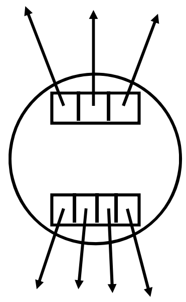


- Same memory requirement as “adjacent list” (except for _visited field))
- Not very intuitive to understand

Graph Implementation (4)

- Two dynamic arrays

```
class Node
{
   Array<Node *>  _successors;
   Array<Node *>  _predecessors;
};
class Graph
{
   Array<Node *>     _nodes;
   // Array<Node *>  _sinks;
   // Array<Node *>  _sources;
};
```



- Memory usage is about the same (n + 2 * e)
- A more intuitive implementation

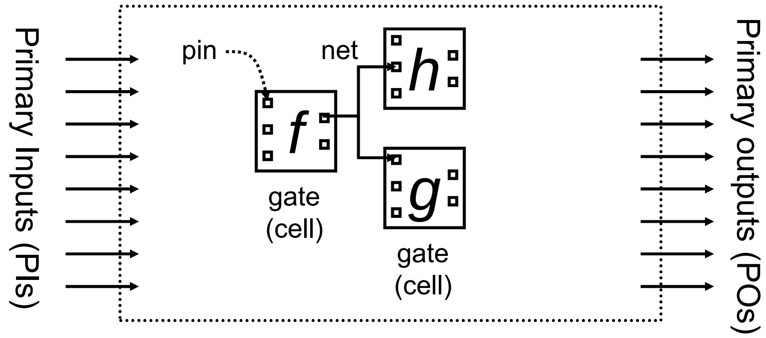
Graph Implementation (5)

- To contain data in nodes

```
template <class T>
class Node
{
   T                 _data;
   Array<Node<T> *>  _successors;
   Array<Node<T> *>  _predecessors;
};
template <class T>
class Graph
{
   Array<Node<T> *>  _nodes;
};
```

Circuit

- A directed diagram for representing the current flow of an electronic design

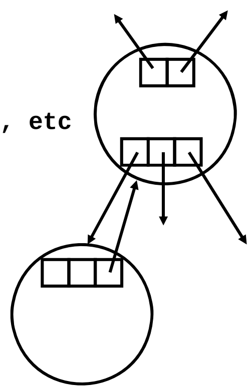


- h and g are f's *fanouts*
- f is h's and g's *fanin*

Circuit Implementation (1)

- Cell-based implementation (1)

```
class Gate
{
   GateType        _type;
   GateFlag        _flag; // visited, etc
   Array<Gate *>   _faninList;
   Array<Gate *>   _fanoutList;
};
class Circuit
{
   Array<Gate *>   _piList;
   Array<Gate *>   _poList;
   Array<Gate *>   _totalList;
};
```



- Gate::_type is to distinguish different functionalities
  - Drawback: usually need a BIG switch in codes

Circuit Implementation (2)

- Cell-based implementation (2)

```
class Gate
{
   ~~GateType        _type;~~
   GateFlag        _flag;
   Array<Gate *>   _faninList;
   Array<Gate *>   _fanoutList;
};
class And : public Gate
{
};
class Circuit
{
   Array<Gate *>   _piList;
   Array<Gate *>   _poList;
   Array<Gate *>   _totalList;
};
```

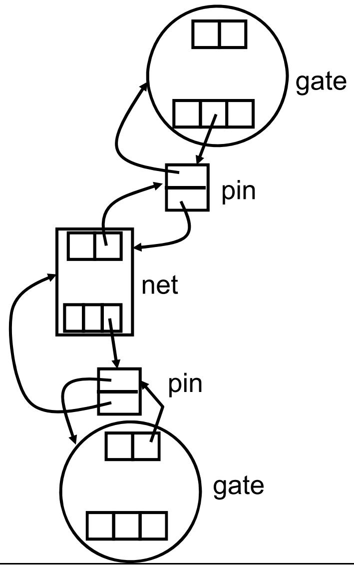

Virtual Functions for Different Types of Gates

```
class Gate {
   virtual bool simulate() = 0;
};
class And : public Gate {
   bool simulate() { ... }
};

===============================
void Circuit::constructNetlist() {
   Gate* newGate = new And;
   ...
}
bool Circuit::simulate() {
   for_each_gate(thisGate, _totalList)
      thisGate->simulate(); // Will call And::simulate()
   ...
}
```

Circuit Implementation (2)

- Net-based implementation

```
class Net
{
   Array<Pin *>    _inPinList;
   Array<Pin *>    _outPinList
};
class Pin
{
   Gate*           _gate;
   Net*            _net;
};
class Gate
{
   GateFlag        _flag;
   Array<Pin *>    _inPinList;
   Array<Pin *>    _outPintList;
};
```



Circuit Implementation (3)

- **AND-Inverter Graph (AIG)**
- All the Boolean functions can be represented by "And: $\wedge$" and "Inverter: $\neg$"
  - e.g. OR(a, b) = $\neg(\neg a \wedge \neg b)$
- As for circuit implementation, it is better to have simpler data structure
  - AIG is enough
  - Two classes: AndGate and InvGate?
    - InvGate is kind of unnecessary...
  - One class: NandGate?
    - Still need an object to represent an Inverter

    Solution: AndGate with (optional) inverted inputs

AIG Implementation

```
class AigGate {
   Array<AigGateV>  _faninList;
   size_t           _ref;
   static size_t    _globalRef_s
};

class AigGateV {
   AigGateV(AigGate* g, size_t phase):
      _gateV(size_t(g) + phase) { }
   AigGate* gate() const {
      return (AigGate*)(_gateV & 0xFFFFFFFC); }
   bool isInv() const { return (_gateV & 0x1); }

   size_t           _gateV;
};
```


AIGER Format

- An simplified, well-accepted AIG format
  - Documents and source codes available at: http://fmv.jku.at/aiger/
- Two versions
  - ASCII format: text format
  - Binary format: more compact representation

  ➔ In HW#6 and final project, we will handle ASCII format only

AIGER ASCII Format

- ASCII format contains several sections
  - Header
  - Inputs
  - Latches
  - Outputs
  - ANDs
  - Symbols
  - Comments
- Except for header, any of the above sections can be omitted if it is not necessary
  - However, their relative order cannot be altered

AIGER ASCII Format

- Header
  - [Syntax] aag M I L O A
    - aag: specify ASCII AIG format
      - [cf] aig: specify binary format
    - M: maximal variable index
    - I, L, O, A: number of inputs, latches, outputs, AND gates
  - [Example] aag 7 2 0 2 3
  - [Note]
    - Exact ONE space before M, I, L, O, A
    - “A” must be immediately followed by a “new line” char.
    - If all variables are used and there are no unused AND gates, then M = I + L + A.

AIGER ASCII Format

- Variables and Literals
  - Each input, latch, and AND gate is assigned with a distinct variable ID (i.e. an unsigned number)
    - Between [1, M]
    - Variable *0* means constant FALSE.
    - The input, latch, and AND variable IDs can be arbitrary. No one is necessary bigger/smaller than the other.
  - A "literal" is a positive or negative form of a variable
    - Let *v* be the ID of a variable, than the literal (2*v*) and (2*v*+1) stands for the positive and negative forms of the variable, respectively
    - e.g. Literal 12 is the positive form of variable 6

      Literal 1 stands for constant TRUE

AIGER ASCII Format

- Inputs
  - [Syntax] <inputLiteralID>
  - [Example] 2
  - [Note]
    - Each line defines exactly one input, which is represented as a literal ID
    - Inputs are non-negative, so the literal IDs must be even numbers

AIGER ASCII Format

- Latches
  - [Syntax] <currStateLiteralID> <nextStateLiteralID>
  - [Example] `8 15`
  - [Note]
    - Each line defines exactly one latch, which containts the current state literal ID followed by the next state ID
    - Currnet states are non-negative (as inputs), so their literal IDs must be even numbers
    - Next states can be inverted (as outputs), so their literal IDs can be positive or negative

AIGER ASCII Format

- Outputs
  - [Syntax] <outputLiteralD>
  - [Example] `9`
  - [Note]
    - Each line defines exactly one output, which is represented as a literal ID
    - Outputs can be inverted, so their literal IDs can be positive or negative

AIGER ASCII Format

◆ AND gates
- [Syntax] <LHS> <RHS1> <RHS2>
- [Example] `12 7 15`
- [Note]
  - Each line defines exactly one AND gate, which containts the LHS literal followed by exactly two RHS literals
  - LHS literals must be non-negative, and the RHS literals can be positive or negative (i.e. non-inverted or inverted)

AIGER ASCII Format

◆ Symbols
- [Syntax] [ilo]<position> <symbolicName>
- [Example] `i0 reset`
  `o1 done`
- [Note]
  - Each line defines exactly one symbolic name for inputs, latches, or outputs
  - There is at most ONE symbolic name for each input, latch, or output
  - <position> denotes the position of the corresponding input/latch/output is defined in it section. It counts from 0.
  - Symbolic name can contain any printable character, except for “new line”
    - [Note] White space and numbers are allowed in names

AIGER ASCII Format

- Comments
  - [Syntax] c
    [anything]…
  - [Example] c
    Game over!!
  - [Note]
    - The comment section starts with a *c* character followed by a new line. The following lines are comments.

Notes on AIGER Format

- No leading or trailing spaces in each line
- No empty line
- "New line" character must present at the end of each line
- All parsed tokens in the same line, except for comments, must be separated by exactly ONE space character
- Identifying undefined literals and checking for cyclic dependencies have to be done explicitly in ASCII format parser

AIGER Examples

1. Empty circuit

```
aag 0 0 0 0 0          // header
```

2. And gate

```
aag 3 2 0 1 1          // header
2                      // input 0
4                      // input 1
6                      // output 0
6 2 4                  // AND gate 0 = 1 & 2
```

3. Or gate

```
aag 3 2 0 1 1
2                      // input 0
4                      // input 1
7                      // output 0 = !(!1 & !2)
6 3 5                  // AND gate 0 = !1 & !2
```


AIGER Examples

4. Half Adder

```
aag 7 2 0 2 3          // header line
2                      // input 0      1st addend bit 'x'
4                      // input 1      2nd addend bit 'y'
6                      // output 0     sum bit    's'
12                     // output 1     carry      'c'
6 13 15                // AND gate 0   x ^ y
12 2 4                 // AND gate 1   x & y
14 3 5                 // AND gate 2   !x & !y
i0 x                   // symbol
i1 y                   // symbol
o0 s                   // symbol
o1 c                   // symbol
c                      // comment header
half adder             // comment
```

Some notes about HW#6

- Topic: An AIGER parser
  - Parse an AIGER netlist file into a circuit data structure (a DAG)
  - Check for floating/undefined variables
  - Check for cyclic conditions
  - Report circuit statistics
  - Report gate connections
  - Perform logic simulations