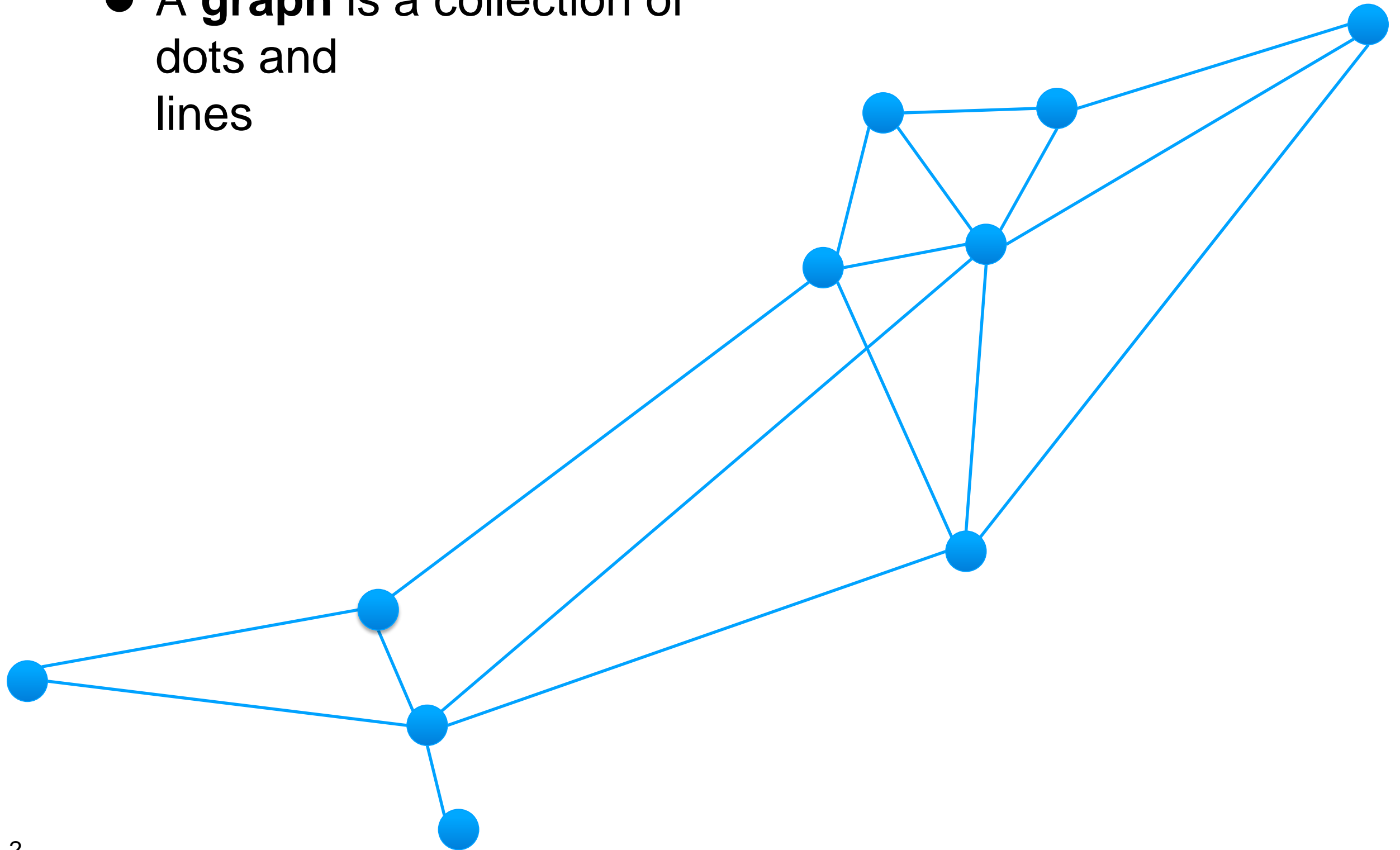


# Graphs

# Graphs

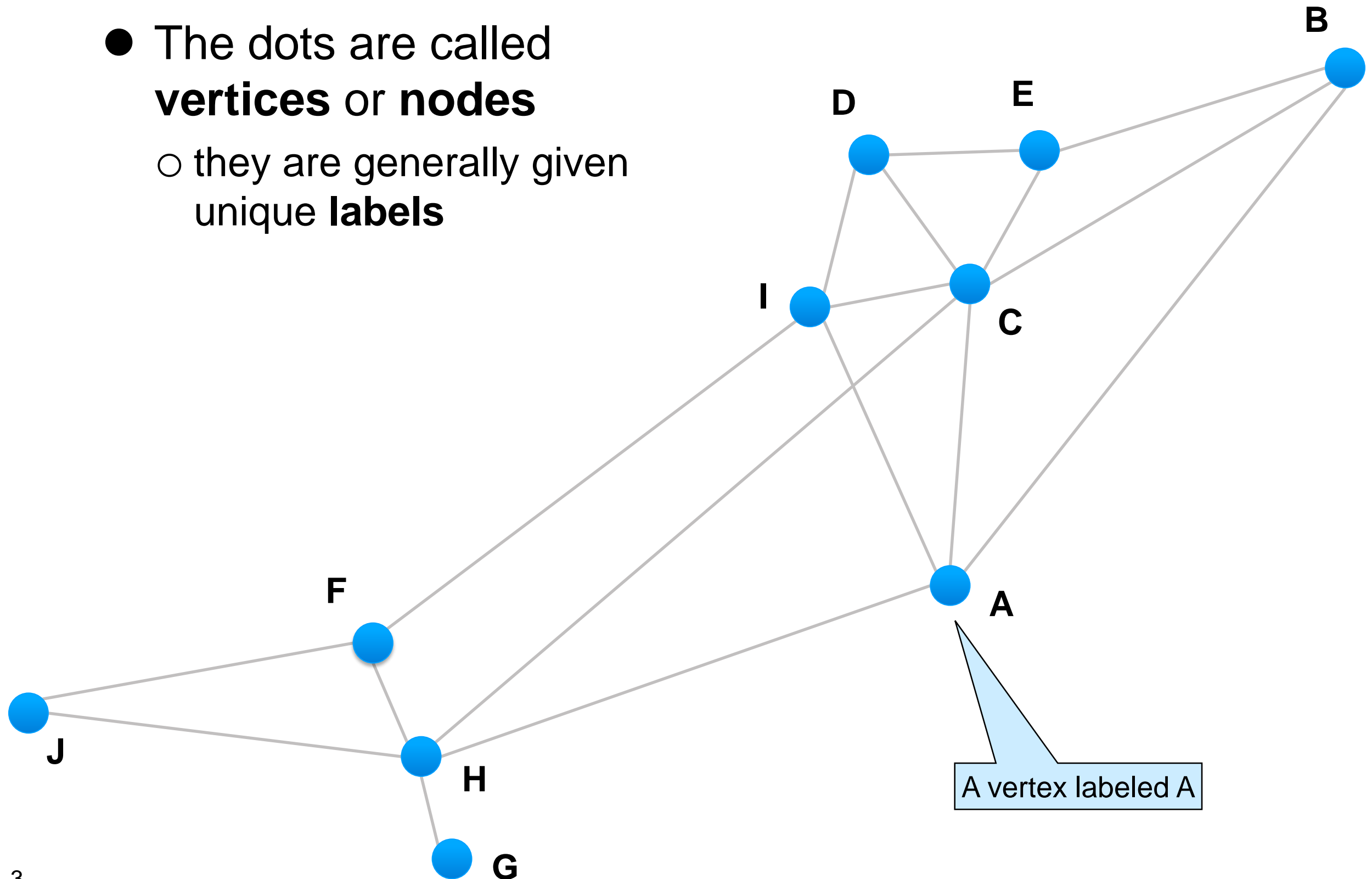
# What is a Graph?

- A **graph** is a collection of dots and lines



# What is a Graph?

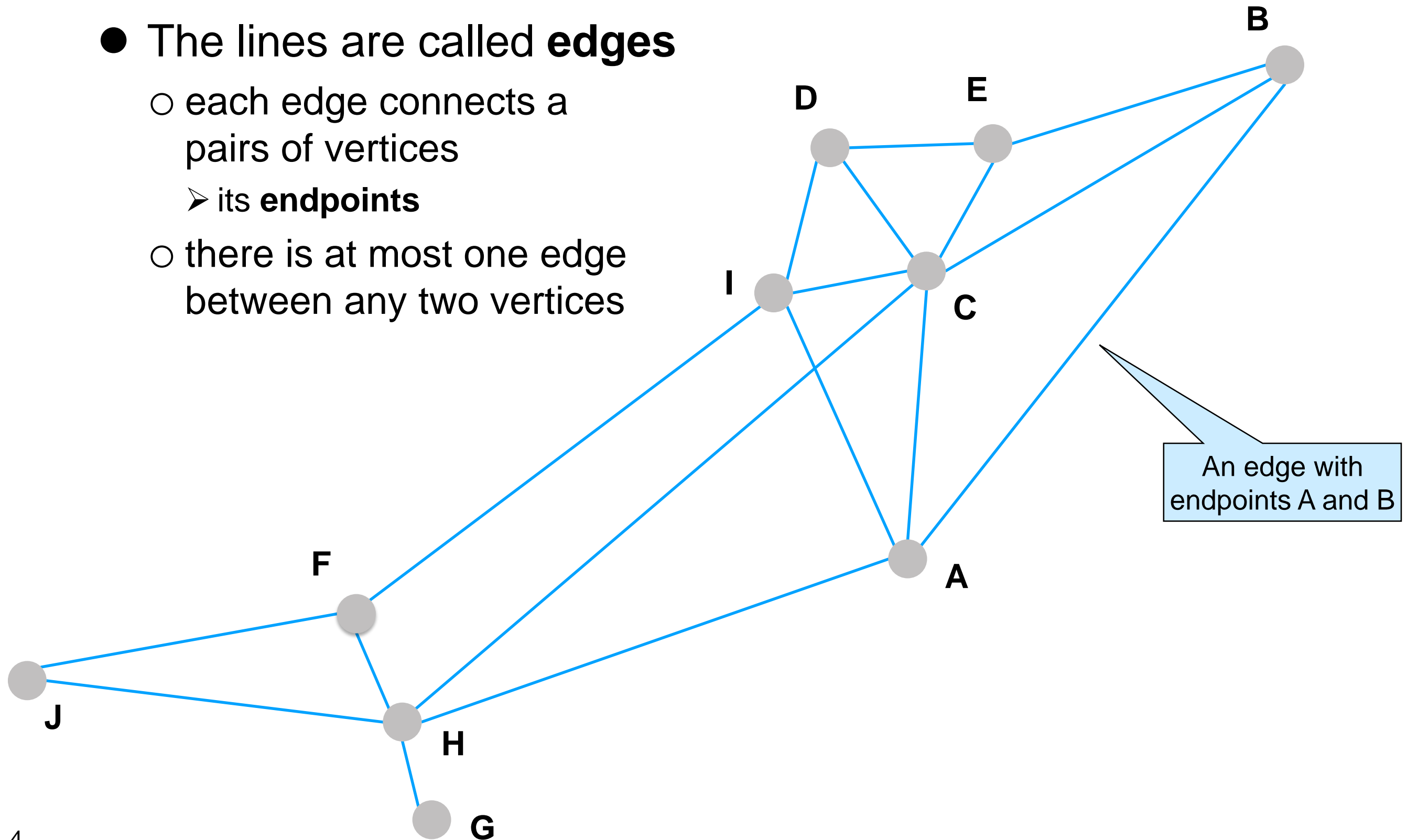
- The dots are called **vertices** or **nodes**
  - they are generally given unique **labels**



# What is a Graph?

- The lines are called **edges**

- each edge connects a pairs of vertices
  - its **endpoints**
- there is at most one edge between any two vertices



# What is a Graph?

- The graphs we will consider

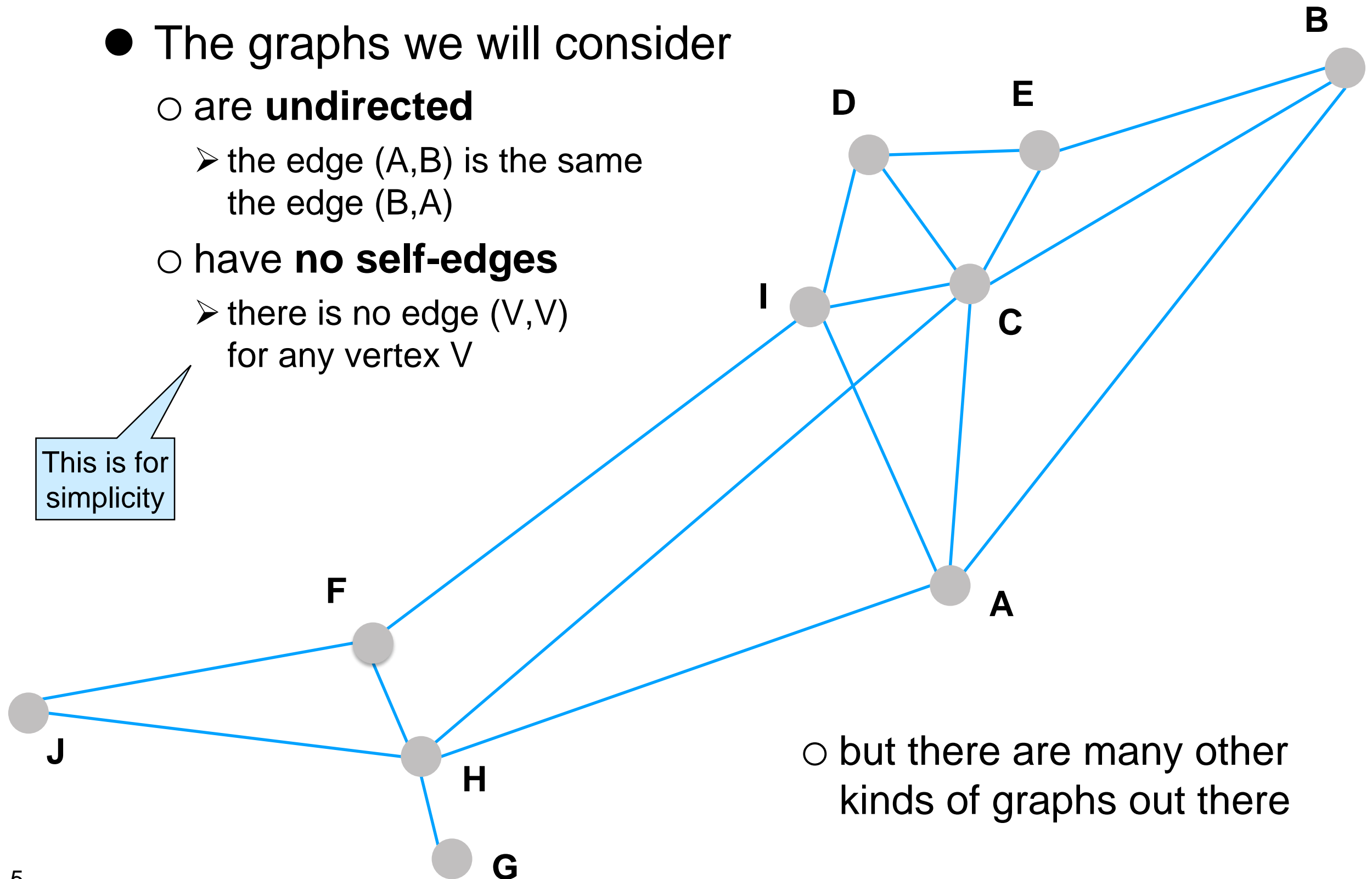
- are **undirected**

- the edge  $(A,B)$  is the same as the edge  $(B,A)$

- have **no self-edges**

- there is no edge  $(V,V)$  for any vertex  $V$

This is for simplicity



- but there are many other kinds of graphs out there

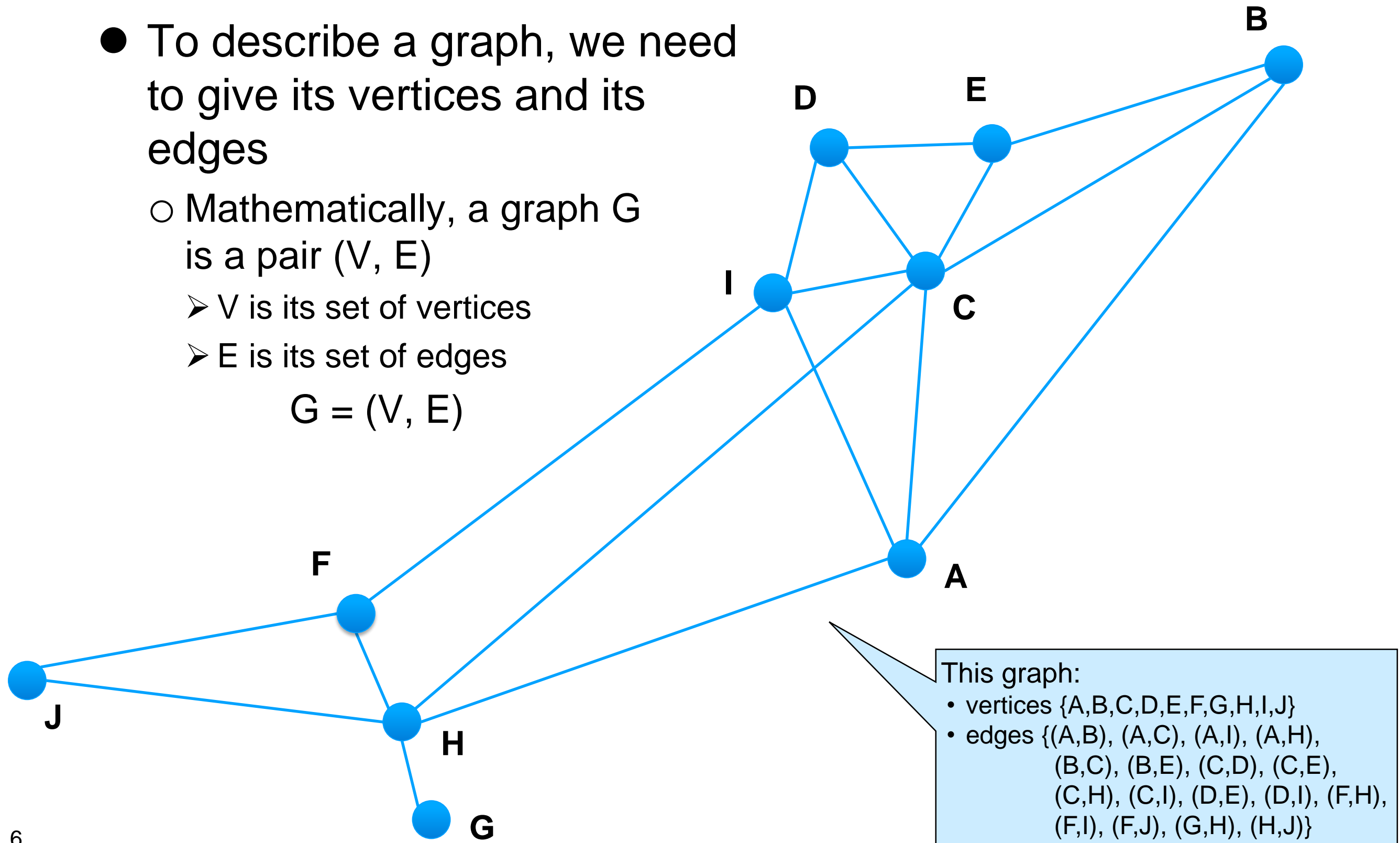
# What is a Graph?

- To describe a graph, we need to give its vertices and its edges

- Mathematically, a graph  $G$  is a pair  $(V, E)$

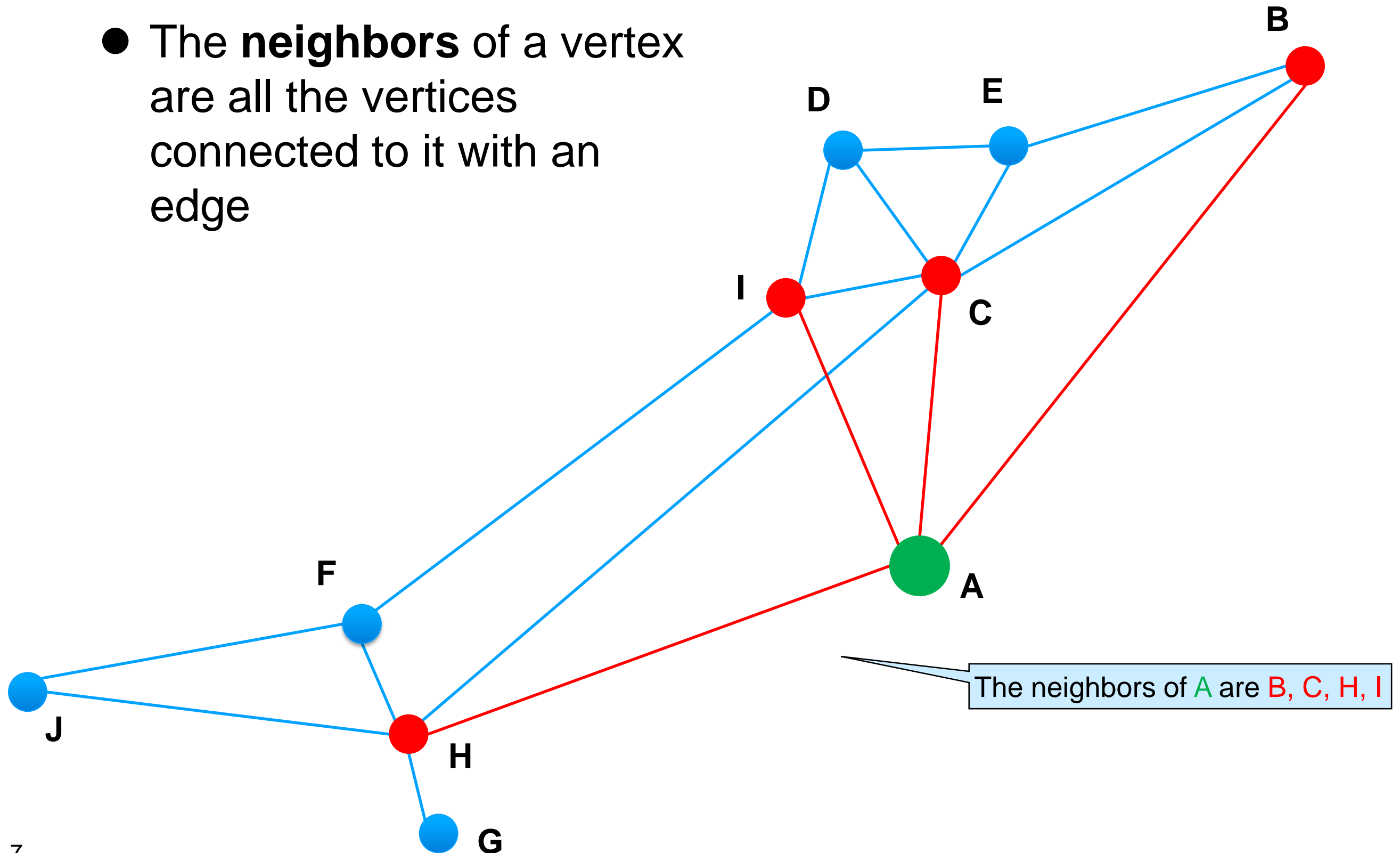
- $V$  is its set of vertices
- $E$  is its set of edges

$$G = (V, E)$$



# What is a Graph?

- The **neighbors** of a vertex are all the vertices connected to it with an edge





# What are Graphs Good for?

- Graphs are a convenient **abstraction** that brings out commonalities between different domains
- Once we understand a problem in term of graphs, we can use **general graph algorithms** to solve it
  - no need to reinvent the wheel every time
- Graphs are everywhere

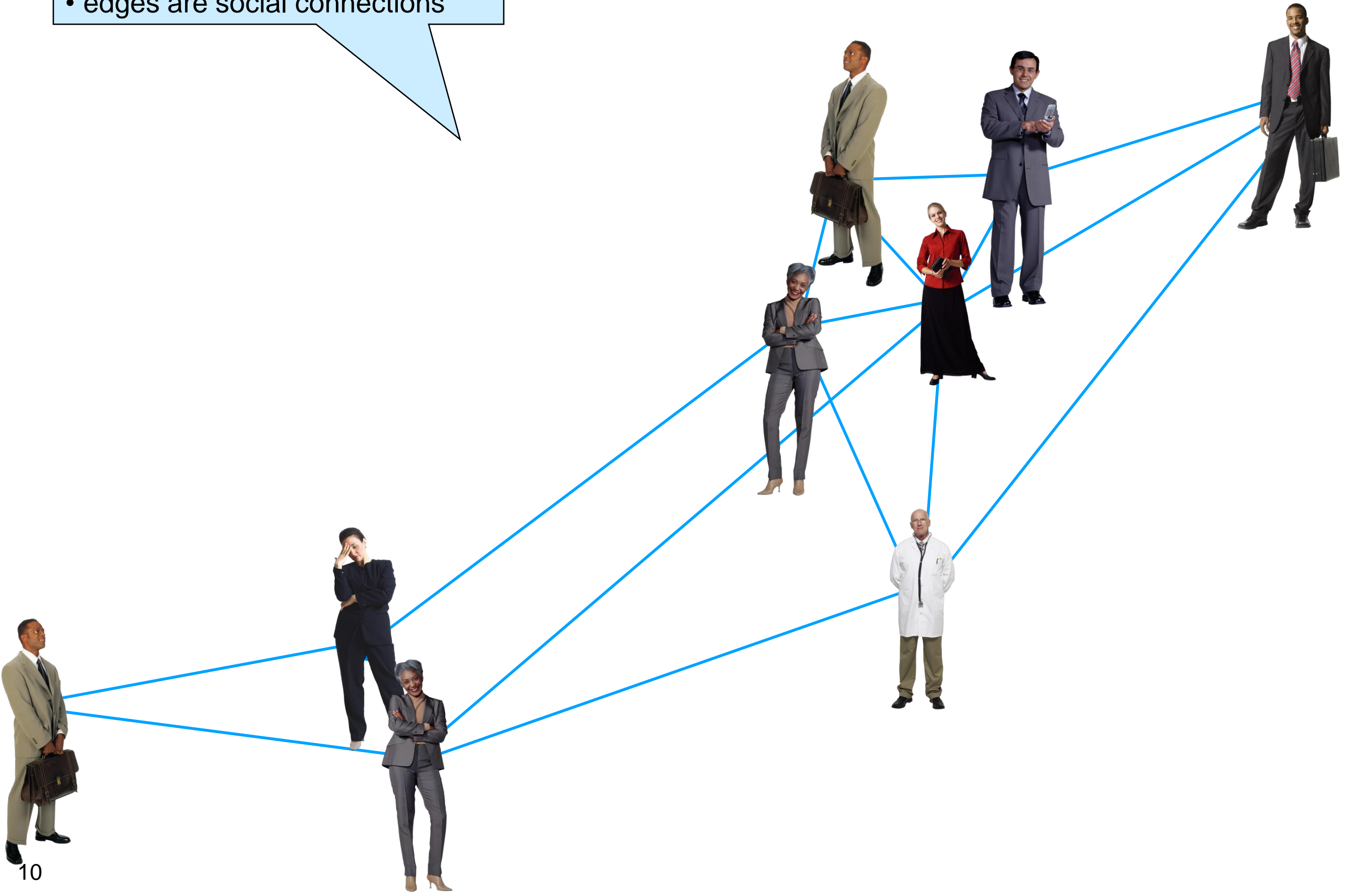
Our graph could represent a road network

- vertices are cities
- edges are major highways



It could represent a social network

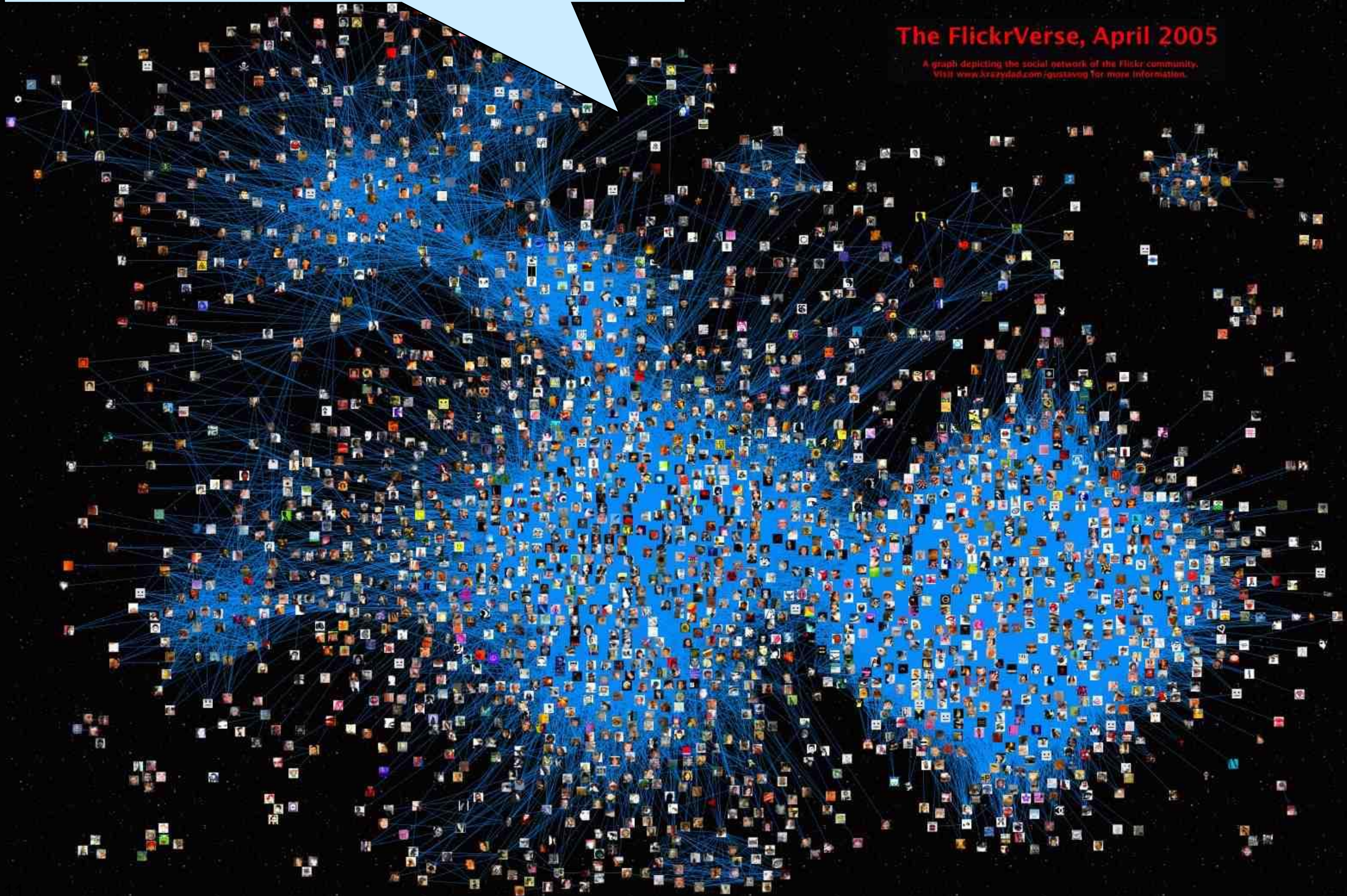
- vertices are people
- edges are social connections





This is what a social network looked like ... in 2005

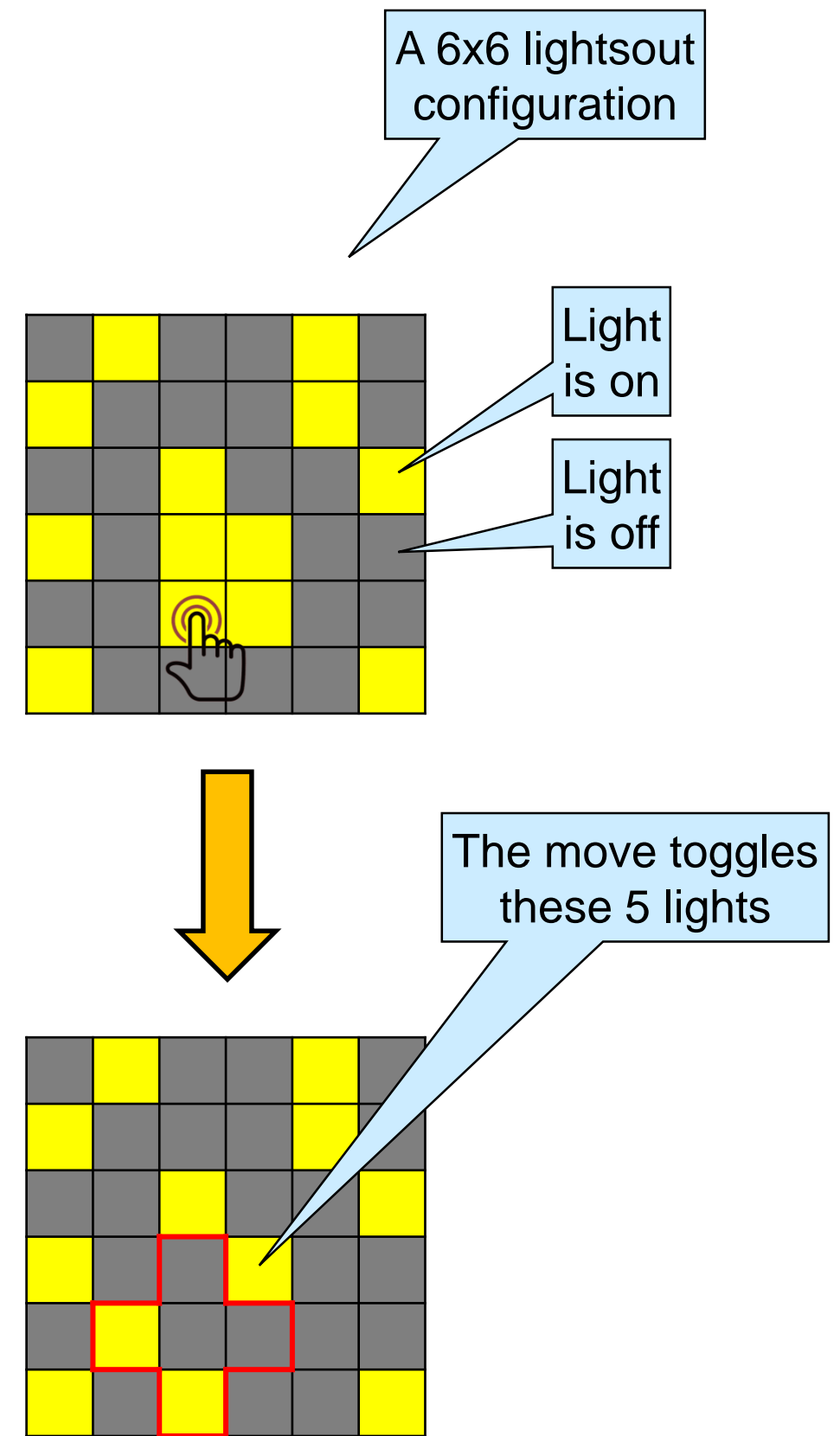
- vertices are people posting photos
- edges are people following the photo stream of others





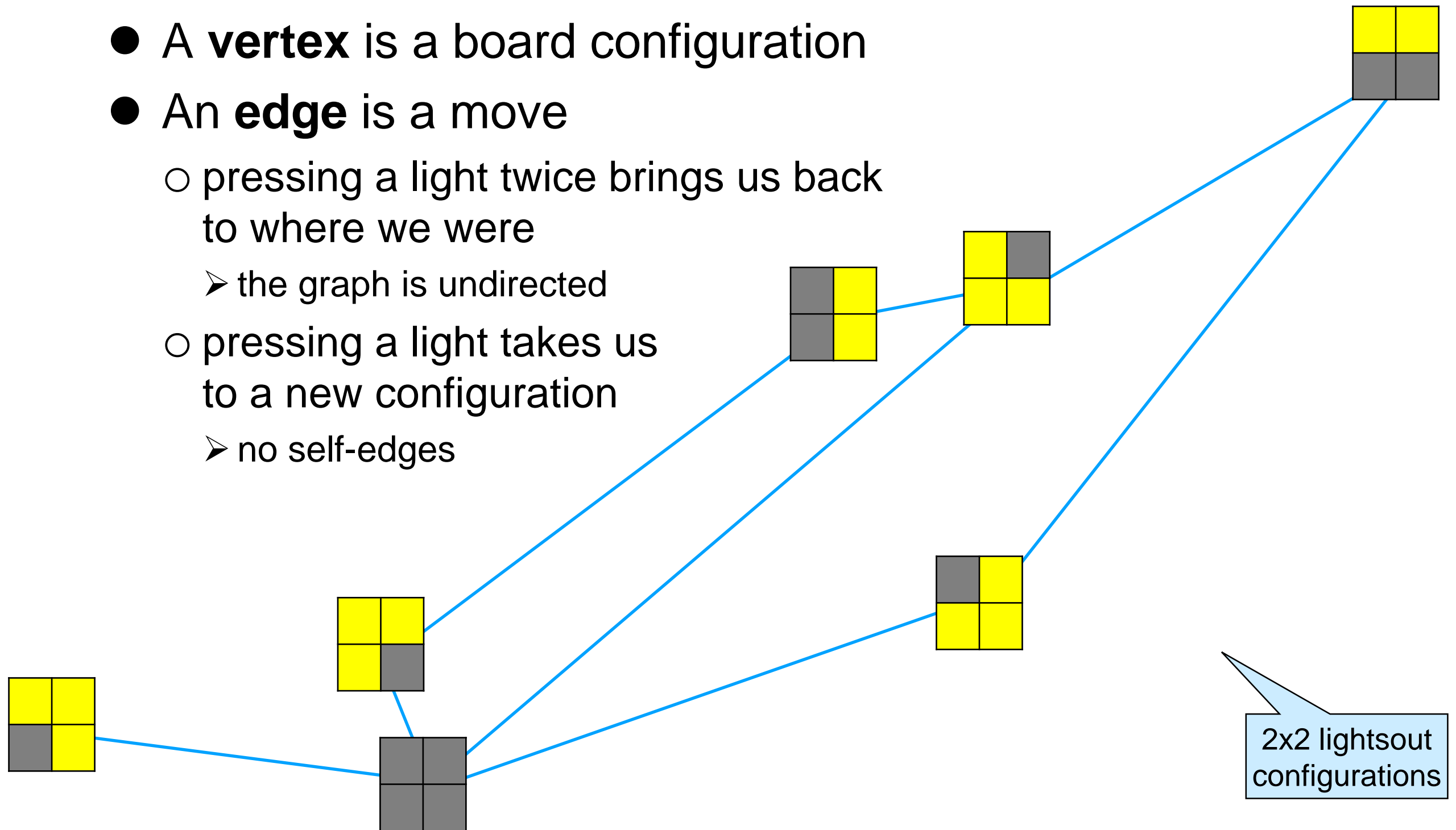
# Lightsout

- Lightsout is a *game* played on boards consisting of  $n \times n$  lights
  - each light can be either on or off
- We make a *move* by pressing a light, which toggles it and its cardinal neighbors
- From a given configuration, the *goal* of the game is to turn off all light



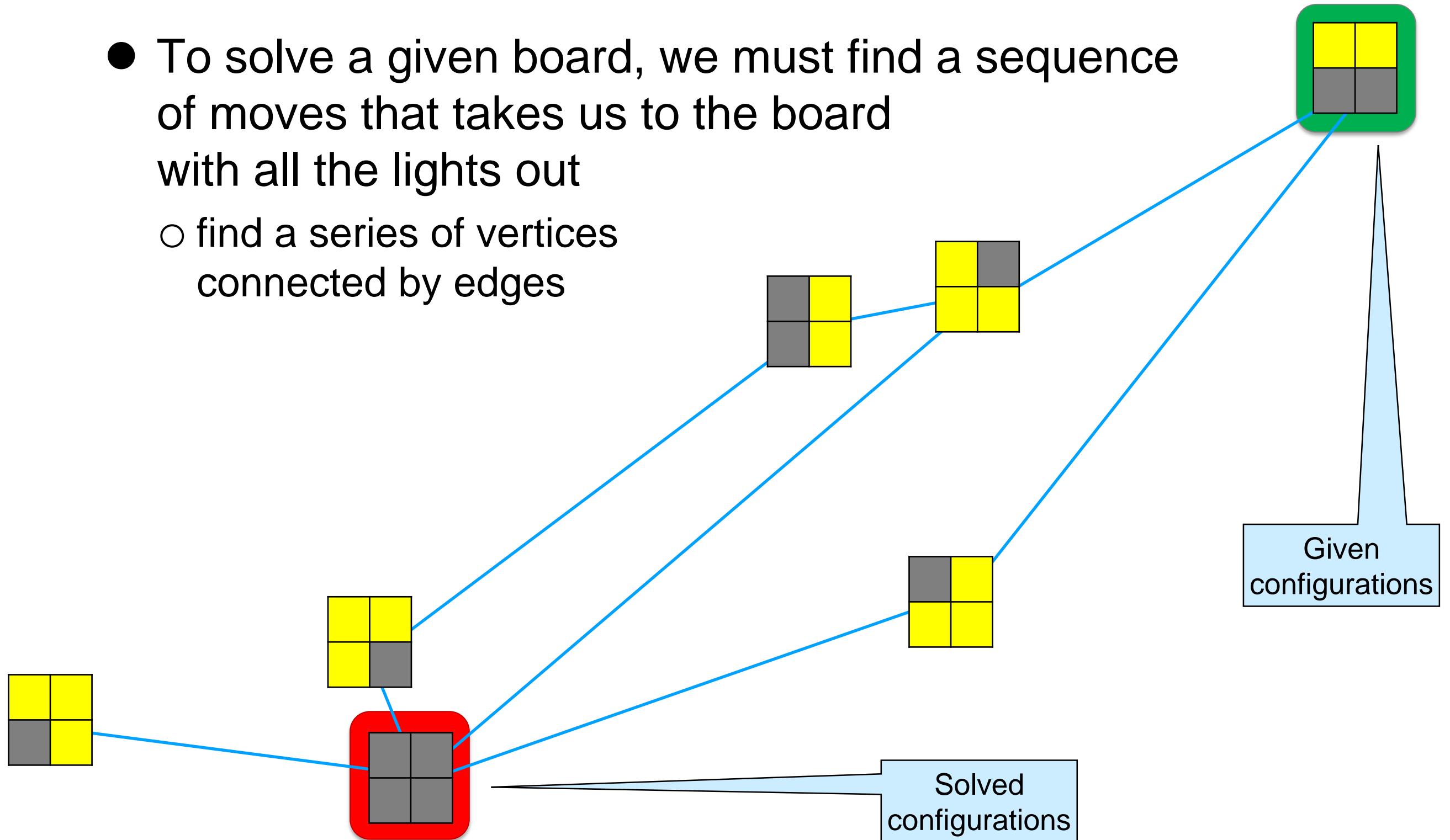
# Lightsout as a Graph

- A **vertex** is a board configuration
- An **edge** is a move
  - pressing a light twice brings us back to where we were
    - the graph is undirected
  - pressing a light takes us to a new configuration
    - no self-edges



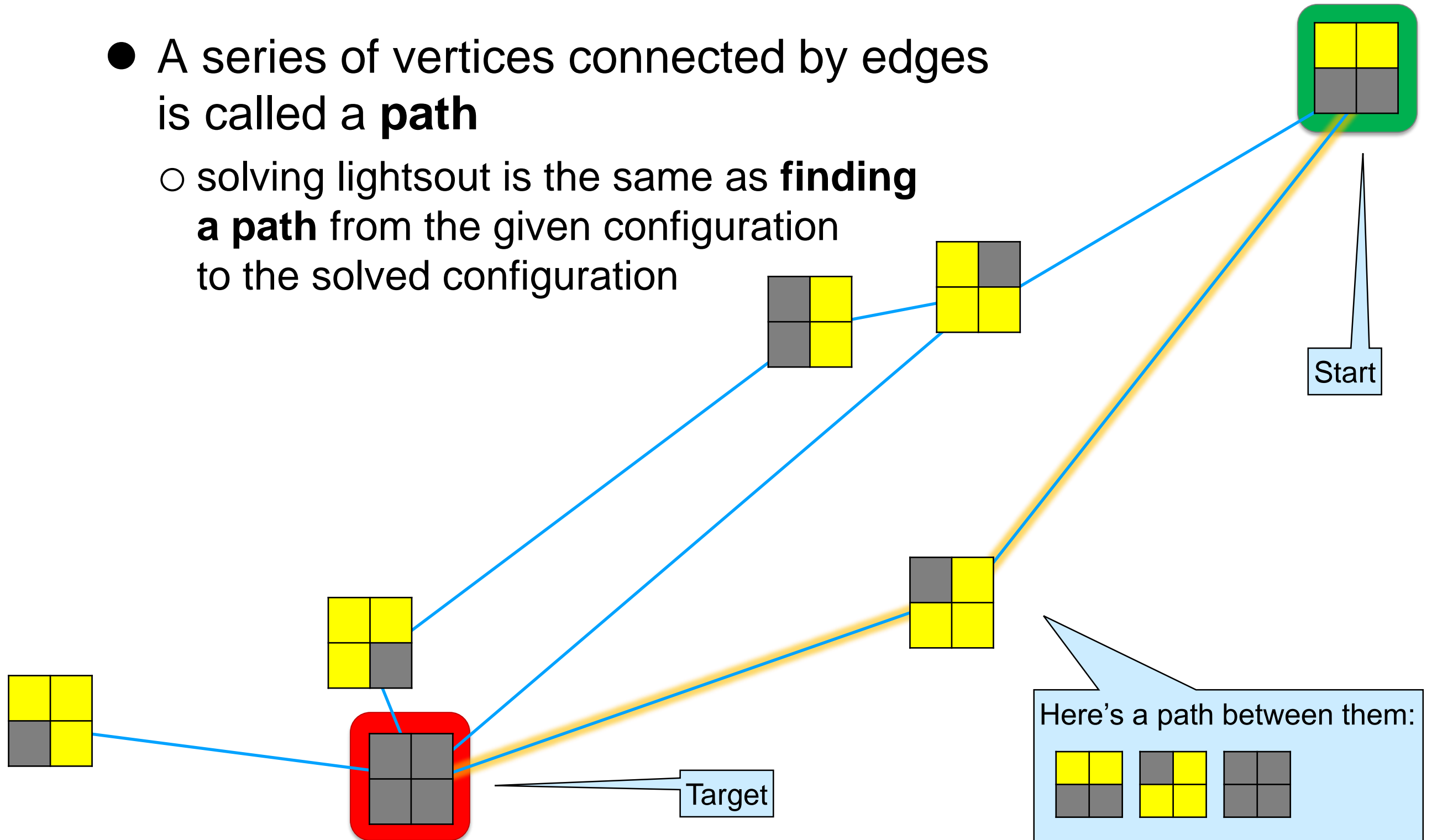
# Lightsout as a Graph

- To solve a given board, we must find a sequence of moves that takes us to the board with all the lights out
  - find a series of vertices connected by edges



# Lightsout as a Graph

- A series of vertices connected by edges is called a **path**
  - solving lightsout is the same as **finding a path** from the given configuration to the solved configuration





# Getting Directions

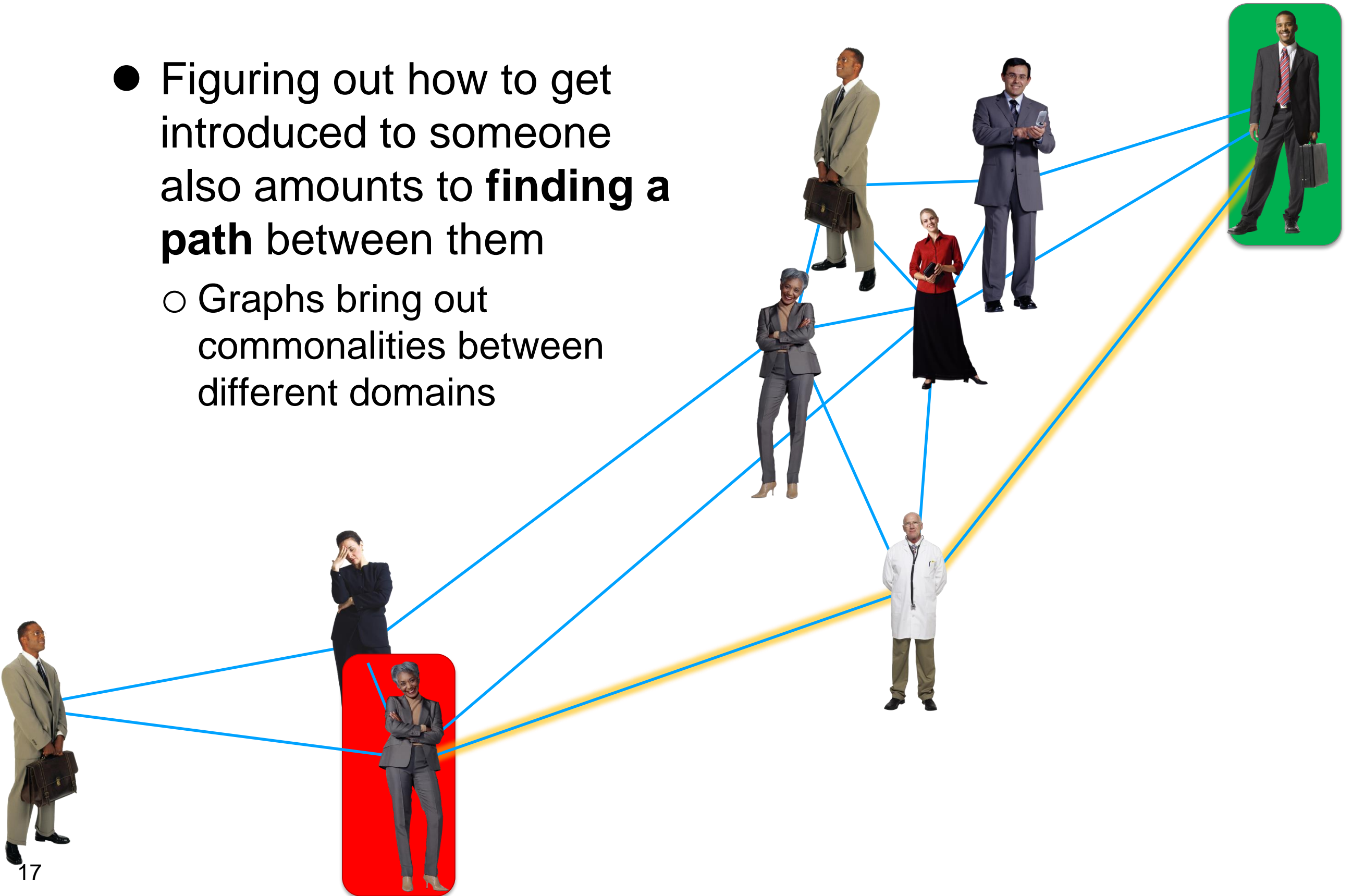
- Figuring out how to go from one place to another also amounts to **finding a path** between them

- *Graphs bring out commonalities between different domains*



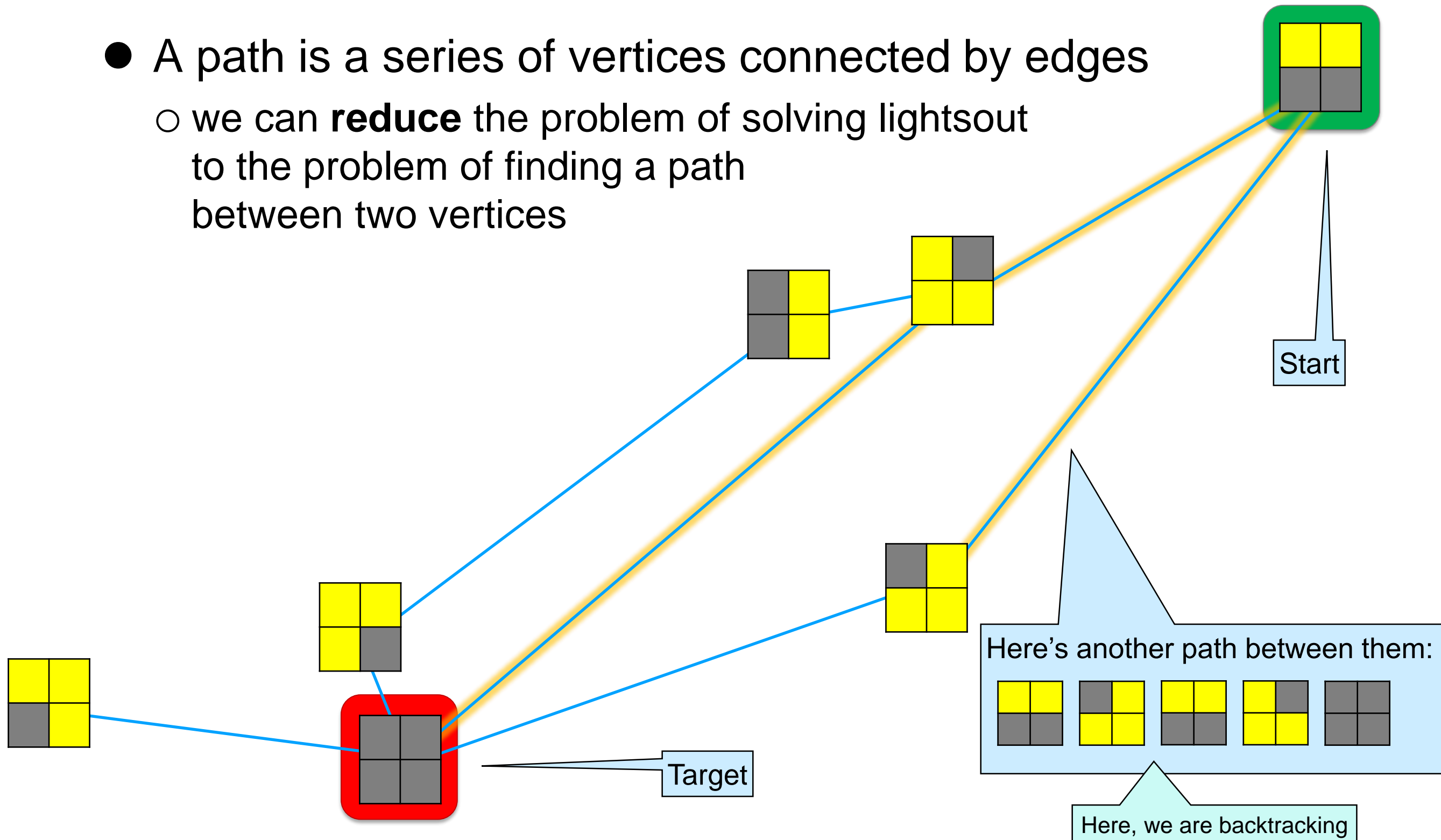
# Getting Introduced

- Figuring out how to get introduced to someone also amounts to **finding a path** between them
  - Graphs bring out commonalities between different domains



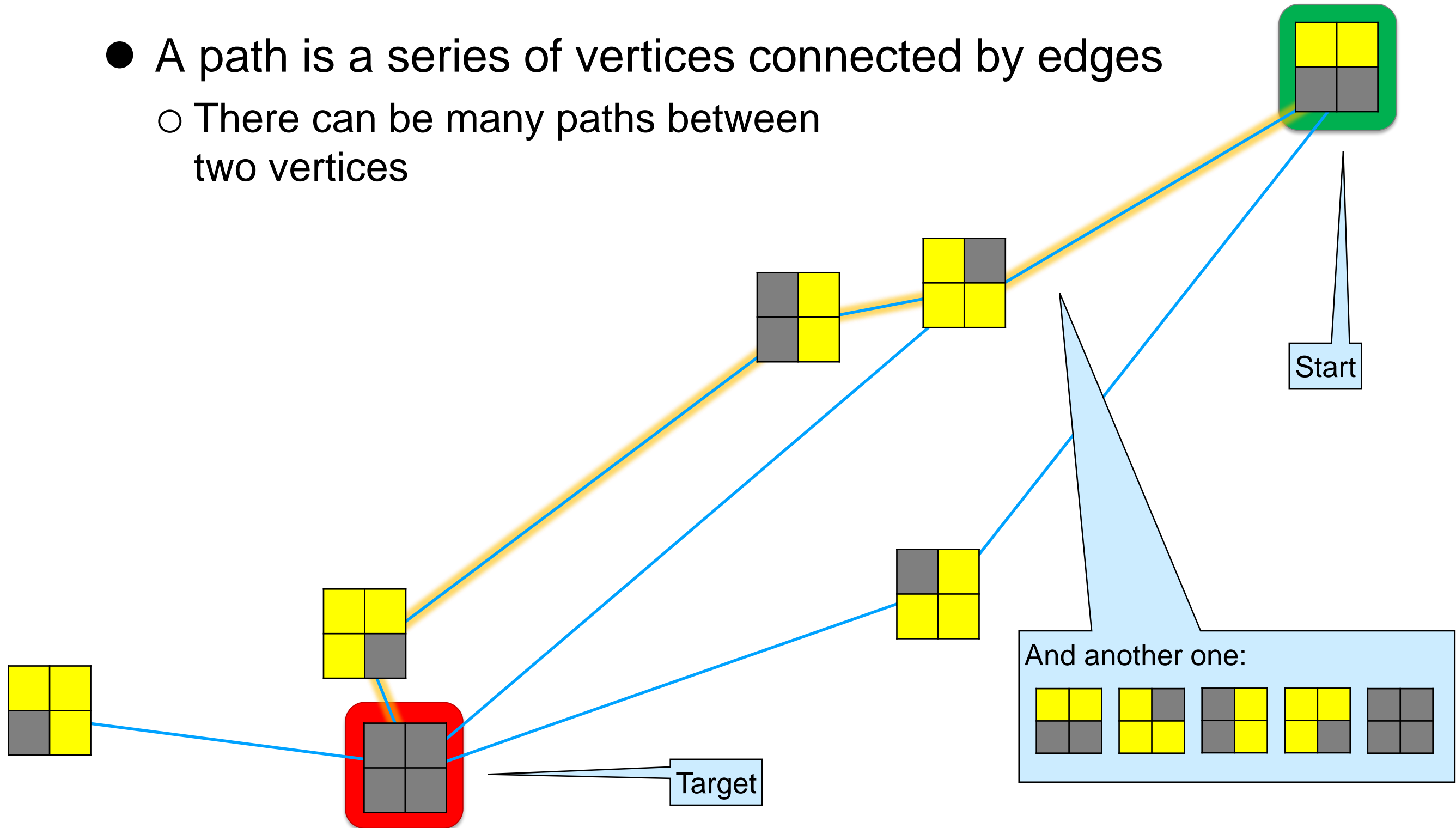
# Lightsout as a Graph

- A path is a series of vertices connected by edges
  - we can **reduce** the problem of solving lightsout to the problem of finding a path between two vertices



# Lightsout as a Graph

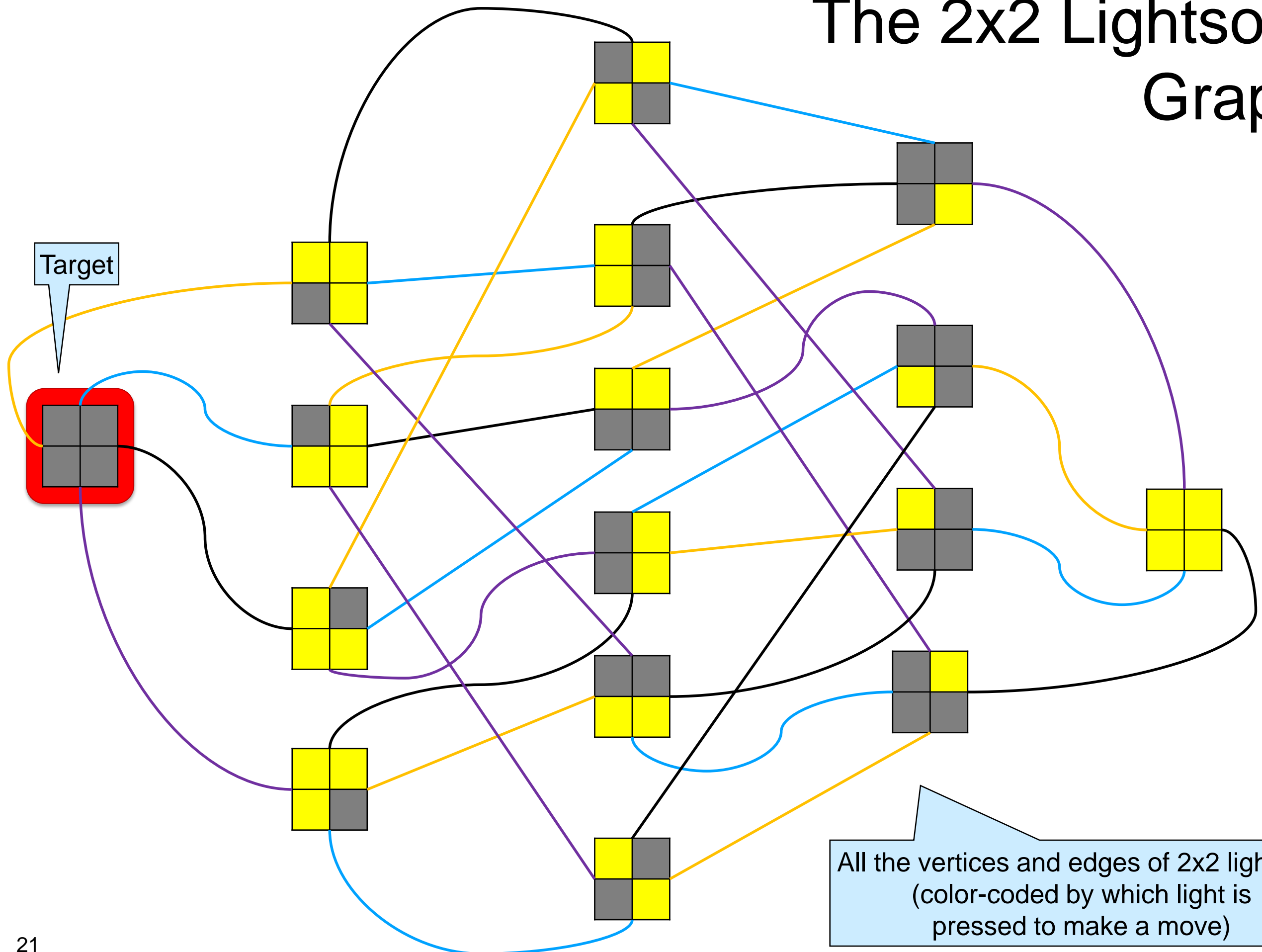
- A path is a series of vertices connected by edges
  - There can be many paths between two vertices



# Lightsout as a Graph

- On  $n \times n$  lightsout,
  - there are  $2^{n^2}$  board configurations
    - each of the  $n^2$  lights can be either on or off
  - from any board, we can make  $n^2$  moves
    - by pressing any one of the  $n^2$  lights
- The graph representing  $n \times n$  lightsout has
  - $2^{n^2}$  vertices
  - $n^2 * 2^{n^2} / 2$  edges
    - there are  $2^{n^2}$  vertices
    - each has  $n \times n$  neighbors
    - but this would count each edge (A,B) twice
      - from A to B and
      - from B to A
    - so we divide by 2

# The 2x2 Lightsout Graph



All the vertices and edges of 2x2 lightsout (color-coded by which light is pressed to make a move)

# Models vs. Data Structures

- A graph can be
  - a conceptual **model** to understand a problem
  - a concrete **data structure** to solve it
- For 2x2 lightsout, it is both
  - Conceptually, it brings out the structure of the problem and highlights what it has in common with other problems
  - Concretely, we can traverse a data structure that represents it in search of a path to the solved board
- Turning 6x6 lightsout into a data structure is not practical
  - each board requires 36 bits
  - we need over 64GB to represent its  $2^{36}$  vertices
  - we need over 2TB to represent its  $36 * 2^{36} / 2$  edges

That's more memory than most computers have

# Implicit Graphs

- We don't need a graph data structure to solve  $n \times n$  lightsout
  - from each board we can **algorithmically** generate all boards that can be reached in one move
  - pick one of them and repeat until
    - we reach the solved board
    - or we reach a previously seen board
      - from it try a different move
- In the process, we are building an **implicit graph**
  - a small portion of the graph exists in memory at any time
    - the boards we have previously seen
      - vertices
    - the moves we still need to try from them
      - edges



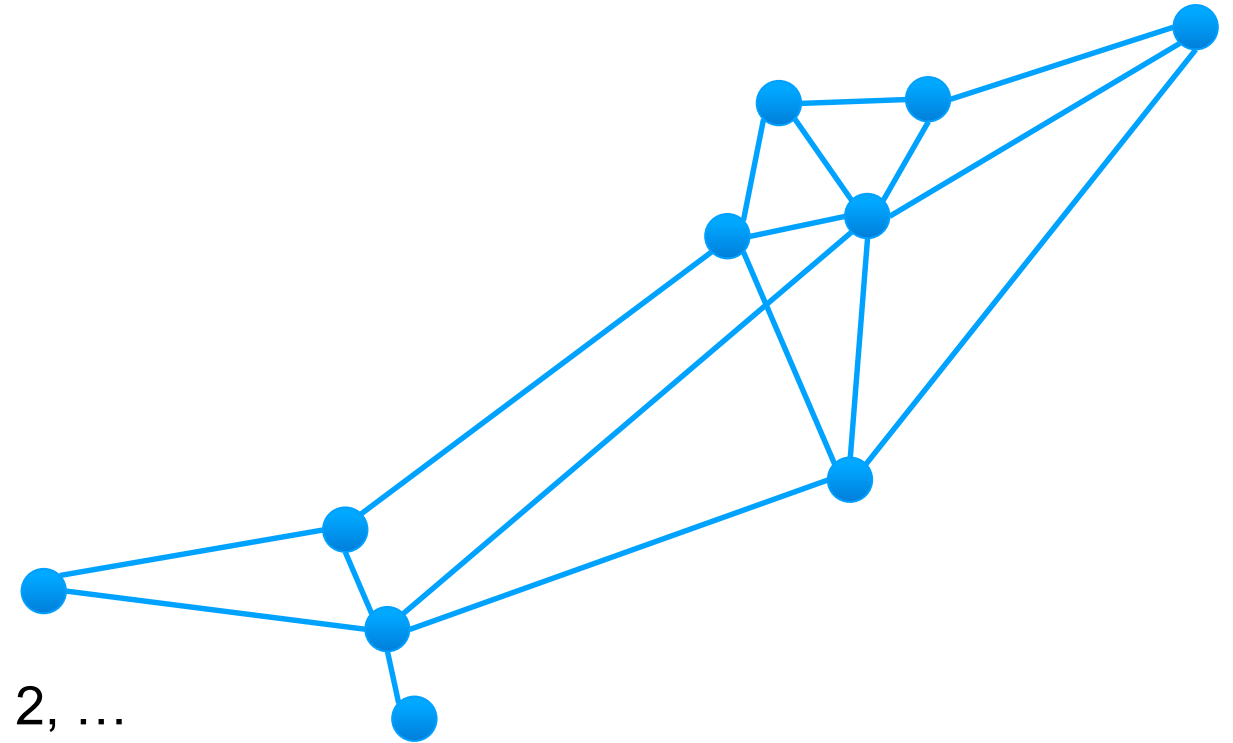
# Explicit Graphs

- For many graphs, there is no algorithmic way to generate their edges
  - roads between cities
  - social networks
  - ...
- We must represent them explicitly as a data structure in memory
- We will now develop a small library for solving problems with these **explicit graphs**

# **A Graph Interface**

# A Minimal Graph Data Structure

- What we need to represent
  - graphs themselves
    - type `graph_t`
  - the vertices of a graph
    - type `vertex`
      - ❑ we label vertices with the numbers 0, 1, 2, ...
        - consecutive integers starting at 0
      - ❑ `vertex` is defined as `unsigned int`
  - the edges of the graph
    - we represent an edge as its endpoints
      - ❑ *no need for an edge type*



# A Minimal Graph Data Structure

- Basic operations on graphs
  - `graph_new(n)` create a new graph with  $n$  vertices
    - we fix the number of vertices at creation time
      - we cannot add vertices after the fact
  - `graph_size(G)` returns the number of vertices in  $G$
  - `graph_hasedge(G, v, w)` checks if the graph  $G$  contains the edge  $(v,w)$
  - `graph_addedge(G, v, w)` adds the edge  $(v,w)$  to the graph  $G$
  - `graph_free(G)` disposes of  $G$
- A realistic graph library would provide a much richer set of operations
  - we can define most of them on the basis of these five

# A Minimal Graph Interface – I

File graph.h

```
typedef unsigned int vertex;
typedef struct graph_header *graph_t;

graph_t graph_new(unsigned int numvert);
//@ensures \result != NULL;

void graph_free(graph_t G);
//@requires G != NULL;

unsigned int graph_size(graph_t G);
//@requires G != NULL;

bool graph_hasedge(graph_t G, vertex v, vertex w);
//@requires G != NULL;
//@requires v < graph_size(G) && w < graph_size(G);

void graph_addedge(graph_t G, vertex v, vertex w);
//@requires G != NULL;
//@requires v < graph_size(G) && w < graph_size(G);
//@requires v != w && !graph_hasedge(G, v, w);

...
```

`vertex` is a concrete type

In a C header file,  
we must define abstract types  
... but we don't need to give the details

This says that `v` and `w`  
must be valid vertices

For simplicity,  
only add **new** edges

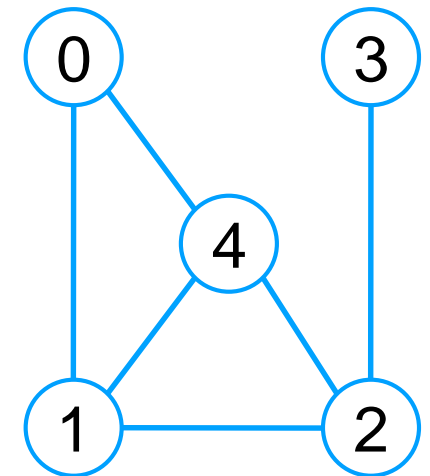
No self-edges

# Example

- We create this graph as

```
graph_t G = graph_new(5);  
graph_addedge(G, 0, 1);  
graph_addedge(G, 0, 4);  
graph_addedge(G, 1, 2);  
graph_addedge(G, 1, 4);  
graph_addedge(G, 2, 3);  
graph_addedge(G, 2, 4);
```

} in any order



We sometimes write the labels inside the vertices

- Then

- `graph_hasedge(G, 3, 2)` returns true, but
- `graph_hasedge(G, 3, 1)` return false
  - there is a path from 3 to 1, but no direct edge

# Neighbors

- It is convenient to handle neighbors explicitly
  - this is not strictly necessary
  - but graph algorithms get better complexity if we do so inside the library
- Abstract type of neighbors
  - `neighbors_t`
- Operations on neighbors
  - `graph_get_neighbors(G, v)`
    - returns the neighbors of vertex  $v$  in  $G$
  - `graph_ismore_neighbors(nbors)`
    - checks if there are additional neighbors
  - `graph_next_neighbor(nbors)`
    - returns the next neighbor
  - `graph_free_neighbors(nbors)`
    - dispose of unexamined neighbors

These allow us to iterate through the neighbors of a vertex

This is called an **iterator**

# A Minimal Graph Interface – II

File graph.h

```
...  
  
typedef struct neighbor_header *neighbors_t;  
  
neighbors_t graph_get_neighbors(graph_t G, vertex v);  
//@requires G != NULL && v < graph_size(G);  
//@ensures \result != NULL;  
  
bool graph_ismore_neighbors(neighbors_t nbors);  
//@requires nbors != NULL;  
  
vertex graph_next_neighbor(neighbors_t nbors);  
//@requires nbors != NULL;  
//@requires graph_ismore_neighbors(nbors);  
  
void graph_free_neighbors(neighbors_t nbors);  
//@requires nbors != NULL;
```

These declarations are part of the same header file

There must be additional neighbors to retrieve the next neighbor



# Example

- We grab the neighbors of vertex 4 as

```
neighbors_t n4 = graph_get_neighbors(G, 4);
```

- n4 contains vertices 0, 1, 2 in some order

```
vertex a = graph_next_neighbor(n4);
```

- say a is vertex 1

- ❑ it could also be 0 or 2

```
vertex b = graph_next_neighbor(n4);
```

- say b is vertex 0

- ❑ it cannot be 1 because we already got that neighbor

- ❑ but it could be 2

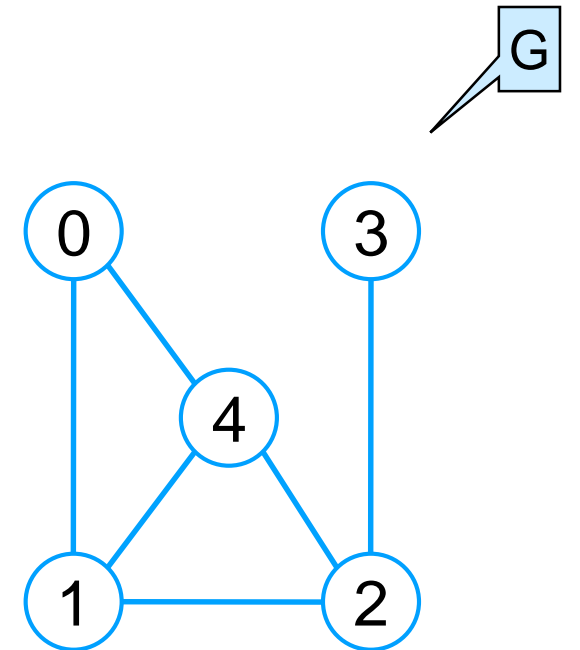
```
vertex c = graph_next_neighbor(n4);
```

- c has to be vertex 2

- ❑ it cannot be 0 or 1 because we already got those neighbors

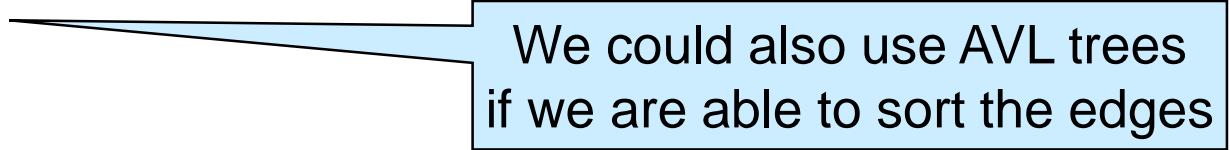
```
graph_ismore_neighbor(n4)
```

- returns false because we have exhausted all the neighbors of 4



# Implementing Graphs

# Implementing Graphs

- How to implement graphs based on what we studied?
  - The main operations are
    - adding an edge to the graph
    - checking if an edge is contained in the graph
      - These are the operations we had for **sets**
    - iterating through the neighbors of a vertex
- Implement graphs as
  - a linked list of edges
  - a hash set

We could also use AVL trees if we are able to sort the edges
- How much would the operations cost?

# Measuring the Cost of Graph Operations

- If a graph has  $v$  vertices, the number  $e$  of edges ranges between

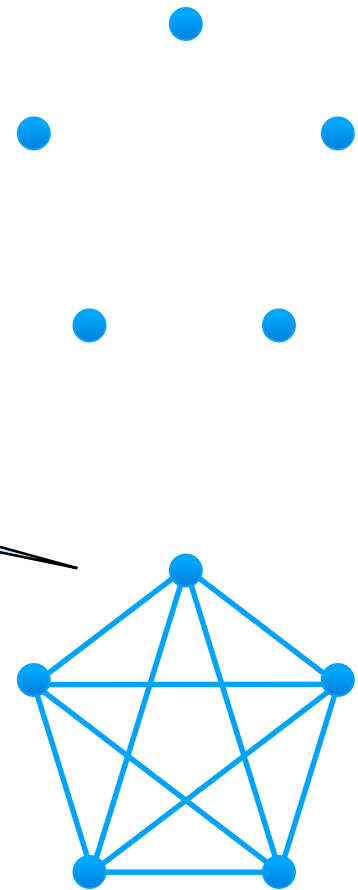
- 0, and

The graph has no edges

- $v(v-1)/2$

This is a complete graph

- there is an edge between each of the  $v$  vertices and the other  $v-1$  vertices, but we divide by 2 so that we don't double-count edges



- So,  $e \in O(v^2)$

- we could do with just  $v$  as a cost parameter,

- but many graphs have far fewer than  $v(v-1)/2$  edges

- using only  $v$  would be overly pessimistic

- Use **both**  $v$  and  $e$  as cost parameters

# Naïve Graph Implementations

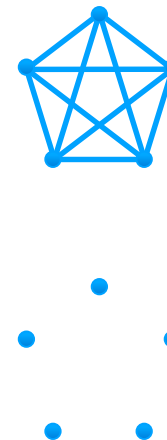
- For implementations based on known data structures, the cost of the basic graph operations are

	<b>Linked list of edges</b>	<b>Hash set of edges</b>
<code>graph_hasedge</code>	$O(e)$	$O(1)$ avg
<code>graph_addege</code>	$O(1)$	$O(1)$ avg+amt

- What about iterating through the neighbors of a vertex?

# Naïve Graph Implementations

- Finding the neighbors of a vertex requires going over all the edges
  - `graph_get_neighbors` has cost  $O(e)$  and  $O(v)$  avg
- How many neighbors are there?
  - at most  $v-1$ 
    - if this vertex has an edge to all other vertices
  - at most  $e$ 
    - there cannot be more neighbors than edges in the graph
- A vertex has  $O(\min(v,e))$  neighbors
  - iterating through the neighbors costs  $O(\min(v,e))$ 
    - times the cost of the operation being performed



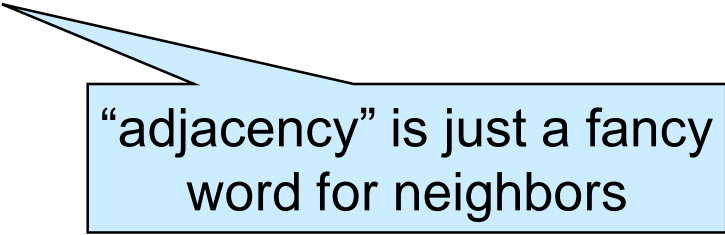
# Naïve Graph Implementations

- In summary

	<b>Linked list of edges</b>	<b>Hash set of edges</b>
<code>graph_hasedge</code>	$O(e)$	$O(1)$ avg
<code>graph_addege</code>	$O(1)$	$O(1)$ avg + amt
<code>graph_get_neighbors</code>	$O(e)$	$O(v)$ avg
<i>Iterating through neighbors</i>	$O(\min(v,e))$	$O(\min(v,e))$

# Classic Graph Implementations

- Can we do better?
- Two representations of graphs are commonly used
  - the adjacency matrix representation
  - the adjacency list representation
- Both give us better cost  
... in different situations ...



“adjacency” is just a fancy word for neighbors

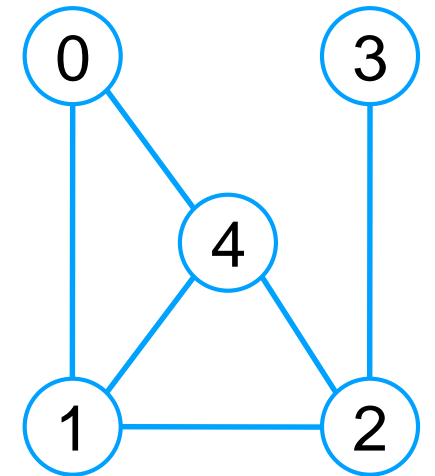


# The Adjacency Matrix Representation

- Represent the graph as a  $v \times v$  matrix of booleans

- $M[i,j] == \text{true}$  if there is an edge between  $i$  and  $j$
- $M[i,j] == \text{false}$  otherwise

$M$  is called the **adjacency matrix**



- Cost of the operations

- `graph_hasedge(G, v, w)`:  $O(1)$

➤ just return  $M[v,w]$

- `graph_addege(G, v, w)`:  $O(1)$

➤ just set  $M[v,w]$  to true

- `graph_get_neighbors(G, v)`:  $O(v)$

➤ go through the row for  $v$  in  $M$

- Space needed:  $O(v^2)$

	0	1	2	3	4
0		✓			✓
1	✓		✓		✓
2		✓		✓	✓
3			✓		
4	✓	✓	✓		

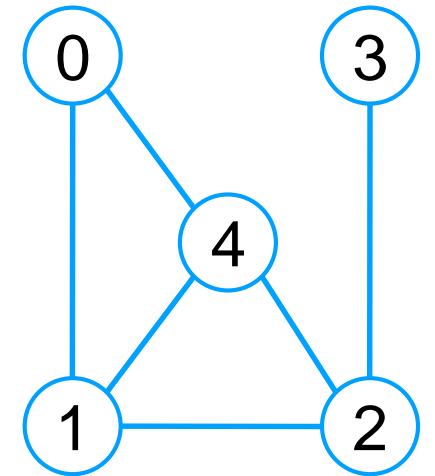
$M[2,4] == \text{true}$   
because  $G$   
contains  
edge  $(2,4)$

For undirected graphs,  
 $M$  is symmetric:  
 $M[i,j] == M[j,i]$

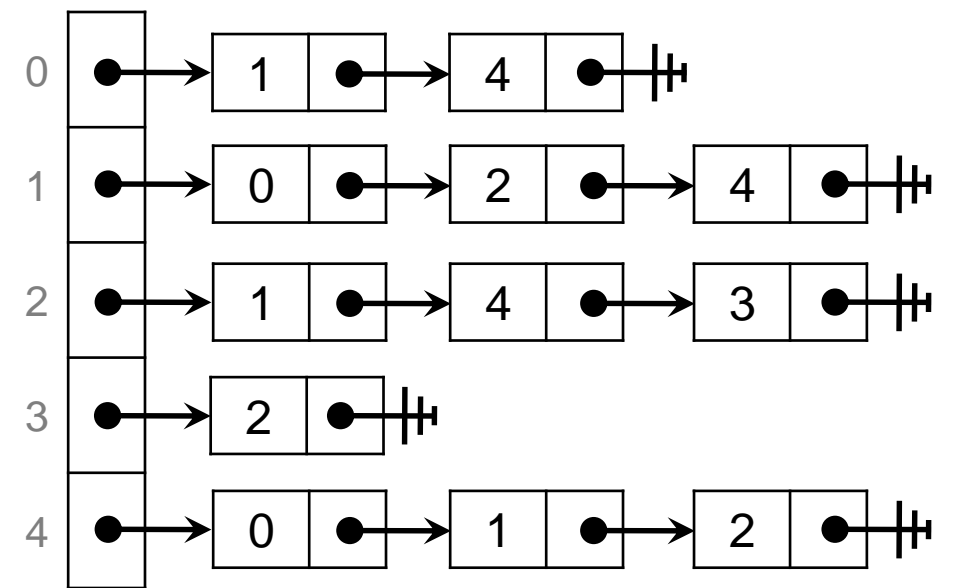
No self-edges,  
so  $M[i,i] == \text{false}$

# The Adjacency List Representation

- For each vertex  $v$ , keep track of its neighbors in a list
  - the **adjacency list** of  $v$
- Store the adjacency lists in a vertex-indexed array



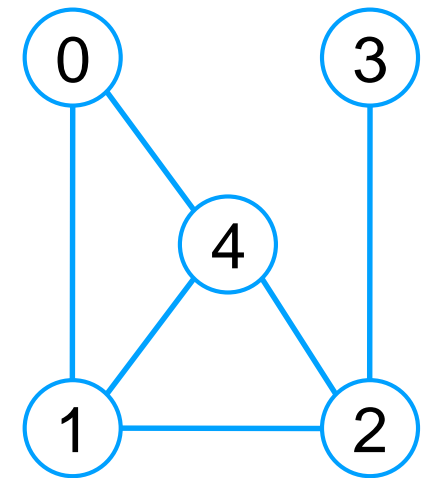
- Cost of the operations
  - `graph_hasedge(G, v, w)`:  $O(\min(v,e))$ 
    - each vertex has  $O(\min(v,e))$  neighbors
    - each adjacency list has length  $O(\min(v,e))$
  - `graph_addedge(G, v, w)`:  $O(1)$ 
    - add  $v$  in  $w$ 's list and  $w$  in  $v$ 's list
  - `graph_get_neighbors(G, v)`:  $O(1)$ 
    - just grab  $v$ 's adjacency list



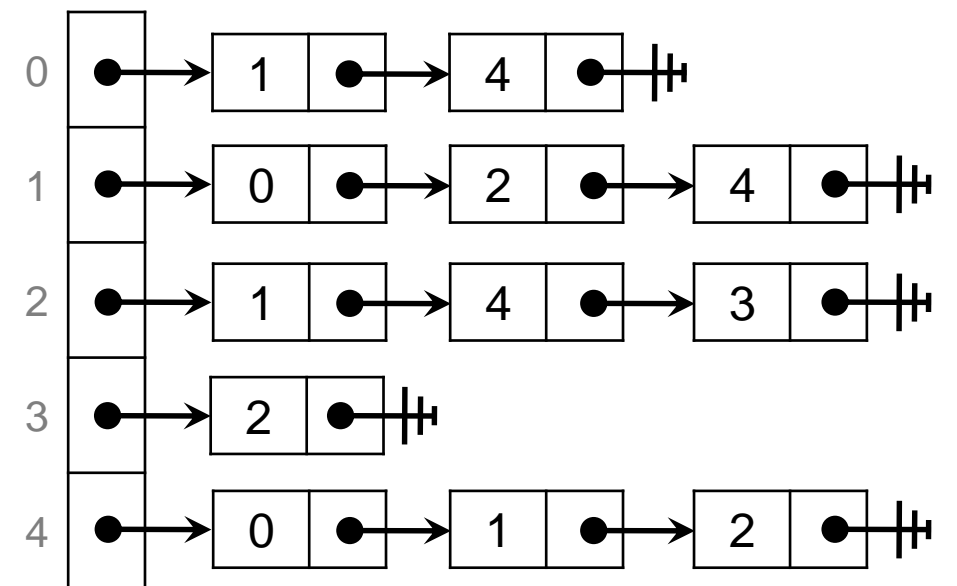
The neighbors of 4 are 0, 1, 2

# The Adjacency List Representation

- For each vertex  $v$ , keep track of its neighbors in a list
  - the **adjacency list** of  $v$
- Store the adjacency lists in a vertex-indexed array



- Space needed:  $O(v + e)$ 
  - a  $v$ -element array
  - $2e$  list items
    - each edge corresponds to exactly 2 list items



- $O(v + e)$  is conventionally written  $O(\max(v,e))$

Why? Note that  $\max(v,e) \leq v+e \leq 2\max(v,e)$

# Adjacency Matrix vs. List

	<b>Adjacency matrix</b>	<b>Adjacency list</b>
Space	$O(v^2)$	$O(v + e)$
graph_hasedge	$O(1)$	$O(\min(v,e))$
graph_addedge	$O(1)$	$O(1)$
graph_get_neighbors	$O(v)$	$O(1)$
<i>Iterating through neighbors</i>	$O(\min(v,e))$	$O(\min(v,e))$

# When to Use What Representation?

- Recall that  $0 \leq e \leq v(v-1)/2$
- A graph is **dense** if it has lots of edges
  - $e$  is on the order of  $v^2$
- A graph is **sparse** if it has relatively few edges
  - $e$  is in  $O(v)$ 
    - at most  $O(v \log v)$
    - but definitely not  $O(v^2)$
  - lots of graphs are sparse
    - social networks
    - roads between cities
    - ...

# Cost in Dense Graphs

- We replace  $e$  with  $v^2$  and simplify

	Adjacency matrix	Adjacency list	
Space	$O(v^2)$	$O(v + e) \rightarrow O(v^2)$	← Same
graph_hasedge	$O(1)$	$O(\min(v,e)) \rightarrow O(v)$	← AM
graph_addege	$O(1)$	$O(1)$	← Same
graph_get_neighbors	$O(v)$	$O(1)$	← AL
<i>Iterating through neighbors</i>	$O(\min(v,e)) \rightarrow O(v)$	$O(\min(v,e)) \rightarrow O(v)$	← Same

# Cost in Dense Graphs

- `graph_hasedge` is faster with AM
  - `graph_get_neighbors` is faster with AL
    - but we typically iterate through the neighbors after grabbing them
  - All other operations are the same
  - The space requirements are the same
  - For dense graphs
    - the two representations have about the same cost
    - but `graph_hasedge` is faster with AM
- the adjacency matrix representation is preferable

# Cost in Sparse Graphs

- We replace  $e$  with  $v$  and simplify

Assume  $e \in O(v)$

	Adjacency matrix	Adjacency list	
Space	$O(v^2)$	$O(v + e) \rightarrow O(v)$	← AL
graph_hasedge	$O(1)$	$O(\min(v,e)) \rightarrow O(v)$	← AM
graph_addege	$O(1)$	$O(1)$	← Same
graph_get_neighbors	$O(v)$	$O(1)$	← AL
<i>Iterating through neighbors</i>	$O(\min(v,e)) \rightarrow O(v)$	$O(\min(v,e)) \rightarrow O(v)$	← Same



# Cost in Sparse Graphs

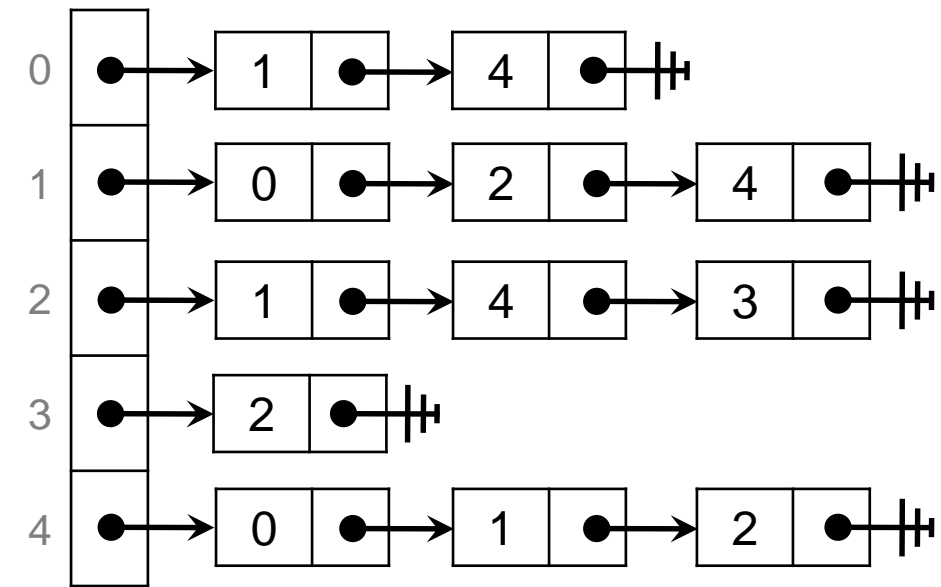
- AL requires **a lot less space**
- `graph_hasedge` is faster with AM
- `graph_get_neighbors` is faster with AL
  - but we typically iterate through the neighbors after grabbing them
- All other operations are the same
- For sparse graphs
  - AL uses substantially less space
  - the two representations have about the same cost
  - but `graph_hasedge` is faster with AM

the adjacency list representation is preferable because it doesn't require as much space

# Adjacency List Implementation

# Graph Types

- An adjacency list is just a NULL-terminated linked list of vertices
- The graph data structure consists of
  - the number  $v$  of vertices in the graph
    - field size
  - a  $v$ -element array of adjacency lists
    - field adjlist



```
typedef struct adjlist_node adjlist;
struct adjlist_node {
    vertex vert;
    adjlist *next;
};

typedef struct graph_header graph;
struct graph_header {
    unsigned int size;
    adjlist **adj;
};
```

adjlist\*[] adj in C0

# Representation Invariants

- The interface defines

```
typedef unsigned int vertex;
```

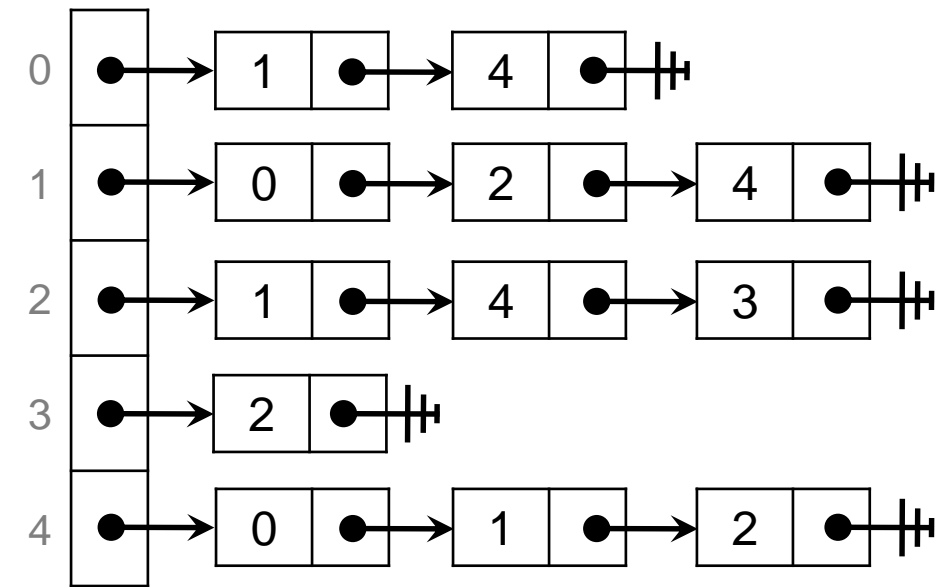
- A vertex is valid if its value is between 0 and the size of the graph

```
bool is_vertex(graph *G, vertex v) {  
    REQUIRES(G != NULL);  
    return v < G->size;  
}
```

$0 \leq v$   
is automatic since  $v$  has  
type `unsigned int`

# Representation Invariants

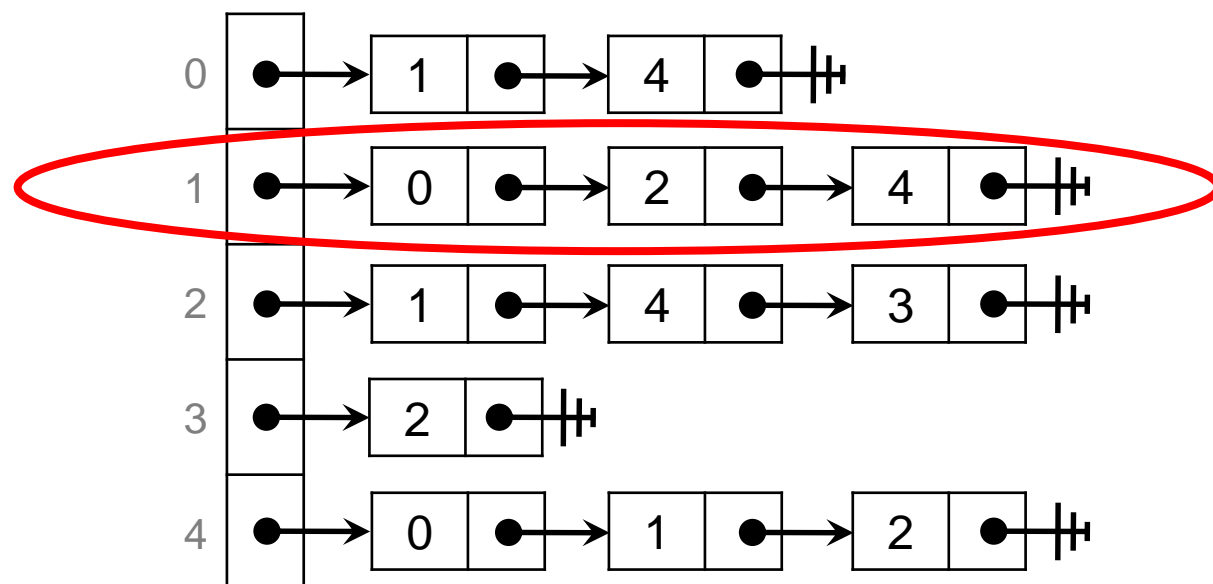
- A graph is valid if
  - it is non-NULL
  - the length of the array of adjacency lists is equal to its size
    - **but we can't check this in C**
  - each adjacency list is valid



```
bool is_graph(graph *G) {  
    if (G == NULL) return false;  
    //@assert(G->size == length(G->adj));  
    for (unsigned int i = 0; i < G->size; i++) {  
        if (!is_adjlist(G, i, G->adj[i])) return false;  
    }  
    return true;  
}
```

# Representation Invariants

- An adjacency list is valid if
  - it is NULL-terminated
  - each vertex is valid
  - there are not self-edges
  - every outgoing edge has a corresponding edge coming back in
  - there are no duplicate edges



```
bool is_adjlist(graph *G, vertex v, adjlist *L) {
    REQUIRES(G != NULL);
    //@requires(G->size == \length(G->adj));
    if (!is_acyclic(L)) return false;

    while (L != NULL) {
        vertex w = L->vert;    // w is a neighbor of v

        // Neighbors are legal vertices
        if (!is_vertex(G, wt)) return false;

        // No self-edges
        if (v == w) return false;

        // Every outgoing edge has a corresponding
        // edge coming back to it
        if (!is_in_adjlist(G->adj[w], v)) return false;

        // Edges aren't duplicated
        if (is_in_adjlist(L->next, w)) return false;

        L = L->next;
    }
    return true;
}
```

# Basic operations

- `graph_size` returns the stored size

- Cost  $O(1)$

```
unsigned int graph_size(graph *G) {  
    REQUIRES(is_graph(G));  
    return G->size;  
}
```

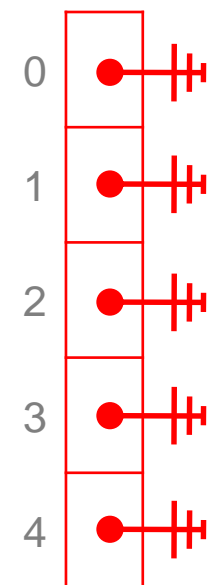
- `graph_new` creates an array of empty adjacency lists

- `calloc` makes it convenient

- Cost  $O(v)$

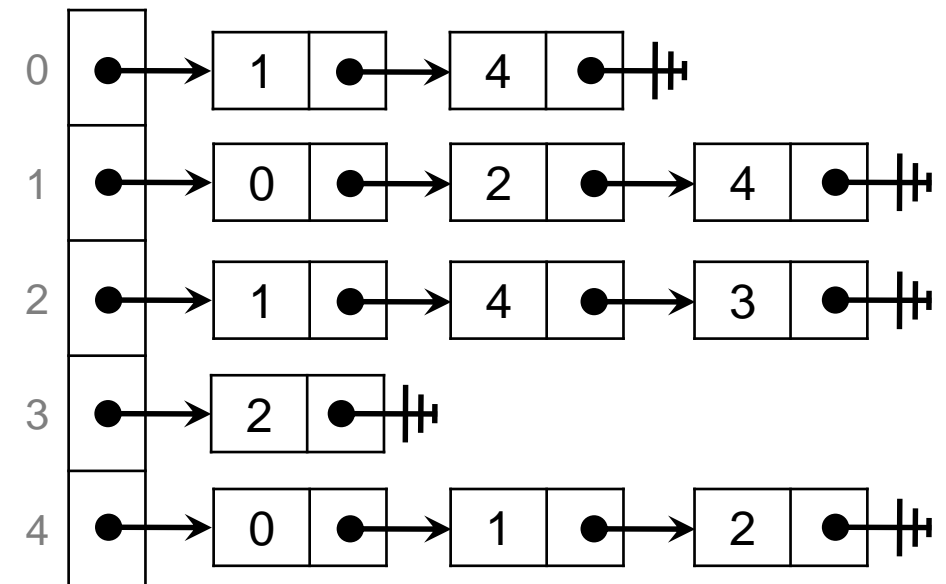
- `calloc` needs to zero out all  $v$  positions

```
graph *graph_new(unsigned int size) {  
    graph *G = xmalloc(sizeof(graph));  
    G->size = size;  
    G->adj = xcalloc(size, sizeof(adjlist*));  
    ENSURES(is_graph(G));  
    return G;  
}
```



# Freeing a Graph

- `graph_free` must free
  - all adjacency lists
  - the array
  - the graph header
- Cost:  $O(v + e)$ 
  - there are  $2e$  nodes to free in the adjacency lists
  - $v$  array positions need to be accessed for that



Free the adjacency list nodes

Free the array

Free the header

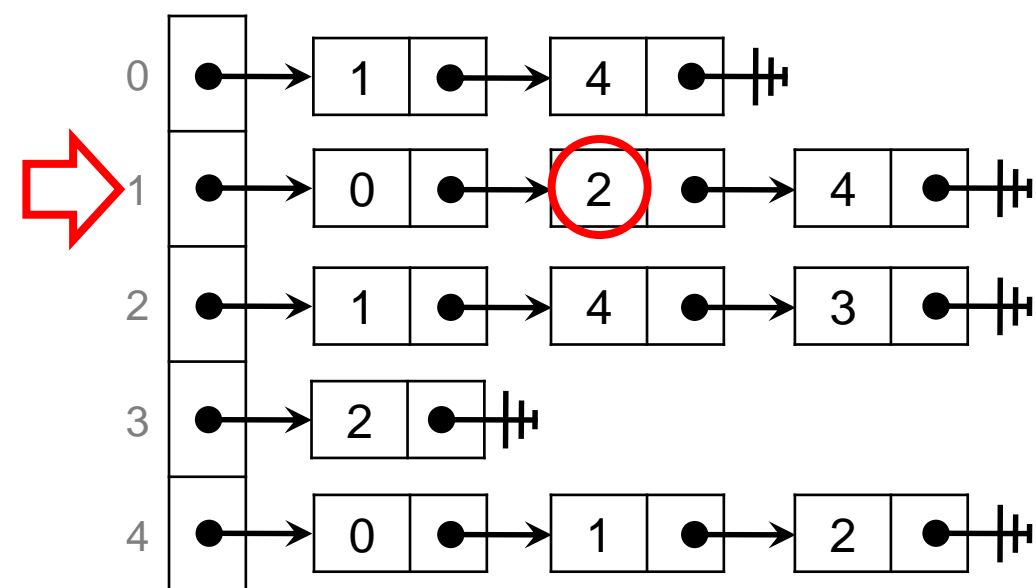
```
void graph_free(graph *G) {  
    REQUIRES(is_graph(G));  
    for (unsigned int i = 0; i < G->size; i++) {  
        adjlist *L = G->adj[i];  
        while (L != NULL) {  
            adjlist *tmp = L->next;  
            free(L);  
            L = tmp;  
        }  
    }  
    free(G->adj);  
    free(G);  
}
```



# Checking Edges

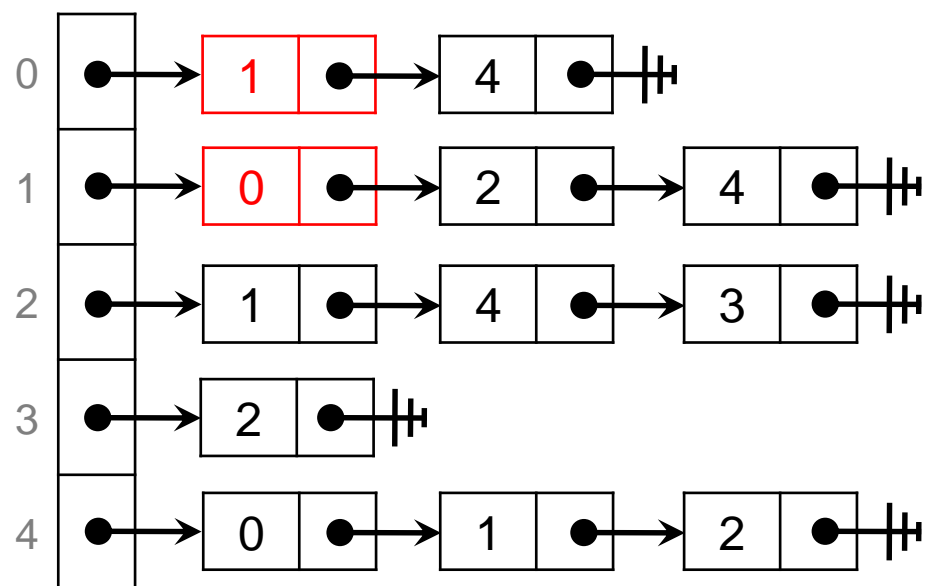
- `graph_hasedge(G, v, w)` does a linear search for `w` in the adjacency list of `v`
  - we could implement it the other way around as well
- Its cost is  $O(\min(v, e))$ 
  - the maximum length of an adjacency list
  - the maximum number of neighbors of a vertex

```
bool graph_hasedge(graph *G, vertex v, vertex w) {  
    REQUIRES(is_graph(G));  
    REQUIRES(is_vertex(G, v) && is_vertex(G, w));  
  
    for (adjlist *L = G->adj[v]; L != NULL; L = L->next) {  
        if (L->vert == w) return true;  
    }  
    return false;  
}
```



# Adding an Edge

- The preconditions exclude
  - self-edges
  - edges already contained in the graph
- `graph_addedge(G, v, w)`
  - adds `w` as a neighbor of `v`
  - and `v` as a neighbor of `w`



```

void graph_addedge(graph *G, vertex v, vertex w) {
    REQUIRES(is_graph(G));
    REQUIRES(is_vertex(G, v) && is_vertex(G, w));
    REQUIRES(v != w && !graph_hasedge(G, v, w));

    adjlist *L;

    L = xmalloc(sizeof(adjlist));
    L->vert = w;
    L->next = G->adj[v];
    G->adj[v] = L;

    L = xmalloc(sizeof(adjlist));
    L->vert = v;
    L->next = G->adj[w];
    G->adj[w] = L;

    ENSURES(is_graph(G));
    ENSURES(graph_hasedge(G, v, w));
}
    
```

add `w` as a neighbor of `v`

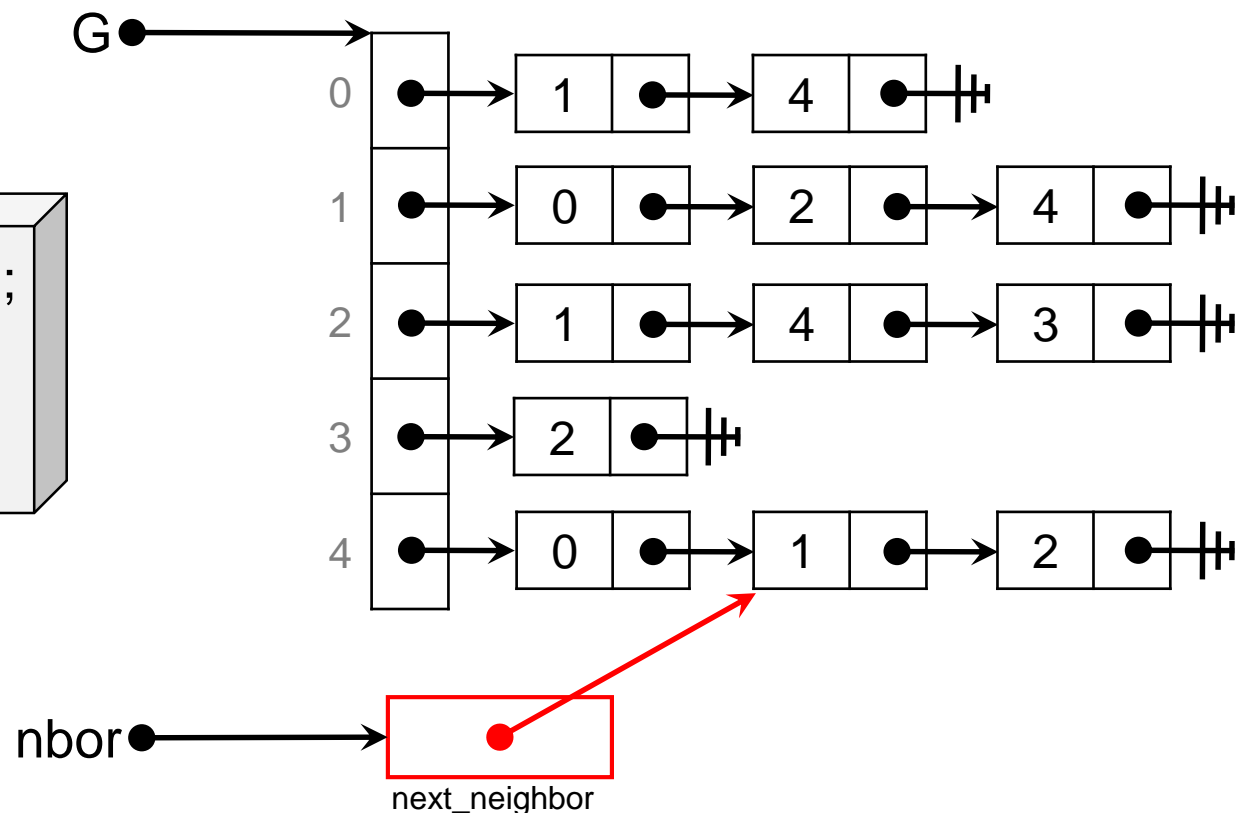
add `v` as a neighbor of `w`

- Constant cost

# Neighbors

- We can use the adjacency list of a vertex as a representation of its neighbors
  - We must be careful however not to modify the graph as we iterate through the neighbors
  - Define a struct with a single field
    - a pointer to the next neighbor to examine

```
typedef struct neighbor_header neighbors;  
struct neighbor_header {  
    adjlist *next_neighbor;  
};
```

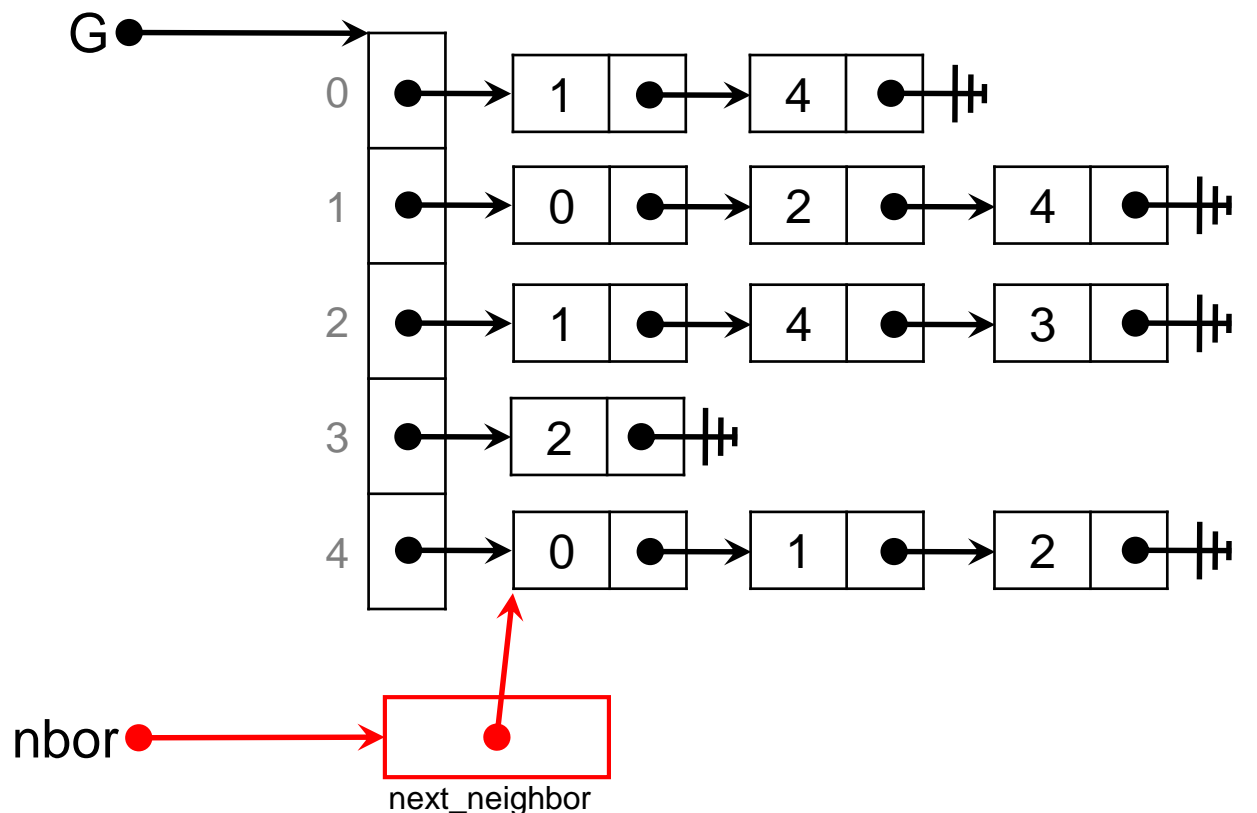


# Neighbors

- `graph_get_neighbors(G, v)`
  - creates a neighbors struct
  - points the `next_neighbor` fields to the adjacency list of `v`
  - returns this struct

```
neighbors *graph_get_neighbors(graph *G, vertex v) {
    REQUIRES(is_graph(G) && is_vertex(G, v));

    neighbors *nbors = xmalloc(sizeof(neighbors));
    nbors->next_neighbor = G->adj[v];
    ENSURES(is_neighbors(nbors));
    return nbors;
}
```



- Constant cost

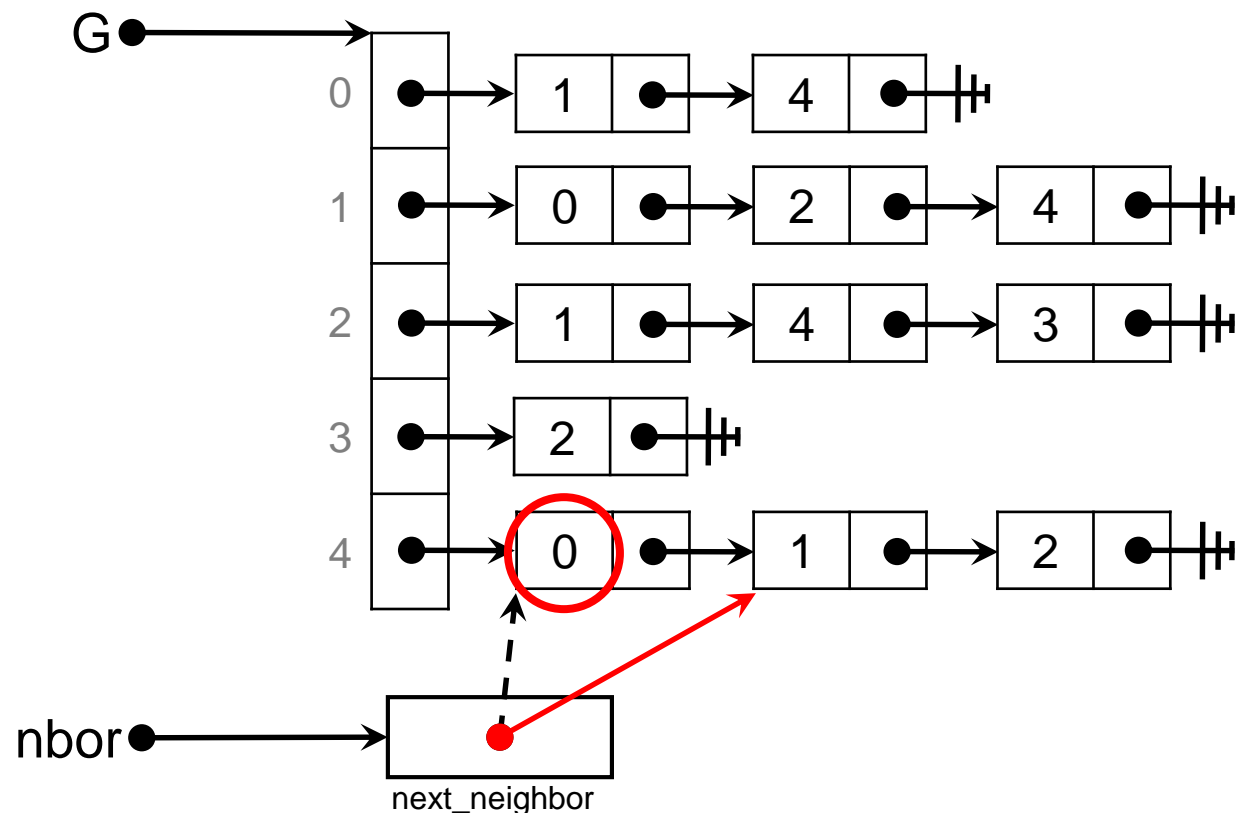
# Neighbors

- `graph_next_neighbor`
  - returns the next neighbor
  - advances the `next_neighbor` field along the adjacency list

It **must not** free that adjacency list node since it is owned by the graph

```
vertex graph_next_neighbor(neighbors *nbors) {  
    REQUIRES(is_neighbors(nbors));  
    REQUIRES(graph_ismore_neighbors(nbors));  
  
    vertex v = nbors->next_neighbor->vert;  
    nbors->next_neighbor = nbors->next_neighbor->next;  
    return v;  
}
```

- Constant cost



# Neighbors

- `graph_ismore_neighbors` checks whether the end of the adjacency list has been reached

```
bool graph_ismore_neighbors(neighbors *nbors) {  
    REQUIRES(is_neighbors(nbors));  
    return nbors->next_neighbor != NULL;  
}
```

- `graph_free_neighbors` frees the neighbor header
  - and **only** the header

It must not free the rest of the adjacency list since it is owned by the graph

```
void graph_free_neighbors(neighbors *nbors) {  
    REQUIRES(is_neighbors(nbors));  
    free(nbors);  
}
```

- Constant time

# Cost Summary

	<b>Adjacency list</b>
Space	$O(v + e)$
graph_new	$O(v)$
graph_free	$O(v + e)$
graph_size	$O(1)$
graph_hasedge	$O(\min(v,e))$
graph_addedge	$O(1)$
graph_get_neighbors	$O(1)$
graph_ismore_neighbors	$O(1)$
graph_next_neighbor	$O(1)$
graph_free_neighbors	$O(1)$

# Using the Graph Interface



# Printing a Graph

- Using the graph interface, write a client function that prints a graph
  - for every vertex
    - print it
    - print every neighbor of this node

```
void graph_print(graph_t G) {  
    for (vertex v = 0; v < graph_size(G); v++) {  
        printf("Vertices connected to %u: ", v);  
        neighbors_t nbors = graph_get_neighbors(G, v);  
        while (graph_ismore_neighbors(nbors)) {  
            vertex w = graph_next_neighbor(nbors);  
            printf(" %u,", w);  
        }  
        graph_free_neighbors(nbors);  
        printf("\n");  
    }  
}
```

w is a neighbor of v

graph.h

```
typedef unsigned int vertex;  
typedef struct graph_header *graph_t;  
  
graph_t graph_new(unsigned int numvert);  
//@ensures \result != NULL;  
  
void graph_free(graph_t G);  
//@requires G != NULL;  
  
unsigned int graph_size(graph_t G);  
//@requires G != NULL;  
  
bool graph_hasedge(graph_t G, vertex v, vertex w);  
//@requires G != NULL;  
//@requires v < graph_size(G) && w < graph_size(G);  
  
void graph_addedge(graph_t G, vertex v, vertex w);  
//@requires G != NULL;  
//@requires v < graph_size(G) && w < graph_size(G);  
//@requires v != w && !graph_hasedge(G, v, w);  
  
typedef struct neighbor_header *neighbors_t;  
  
neighbors_t graph_get_neighbors(graph_t G, vertex v);  
//@requires G != NULL && v < graph_size(G);  
//@ensures \result != NULL;  
  
bool graph_ismore_neighbors(neighbors_t nbors);  
//@requires nbors != NULL;  
  
vertex graph_next_neighbor(neighbors_t nbors);  
//@requires nbors != NULL;  
//@requires graph_ismore_neighbors(nbors);  
//@ensures is_vertex(\result);  
  
void graph_free_neighbors(neighbors_t nbors);  
//@requires nbors != NULL;
```

- We will see other algorithms that follow this pattern

graph_get_neighbors	O(1)
graph_ismore_neighbors	O(1)
graph_next_neighbor	O(1)
graph_free_neighbors	O(1)

# What is the Cost of graph\_print?

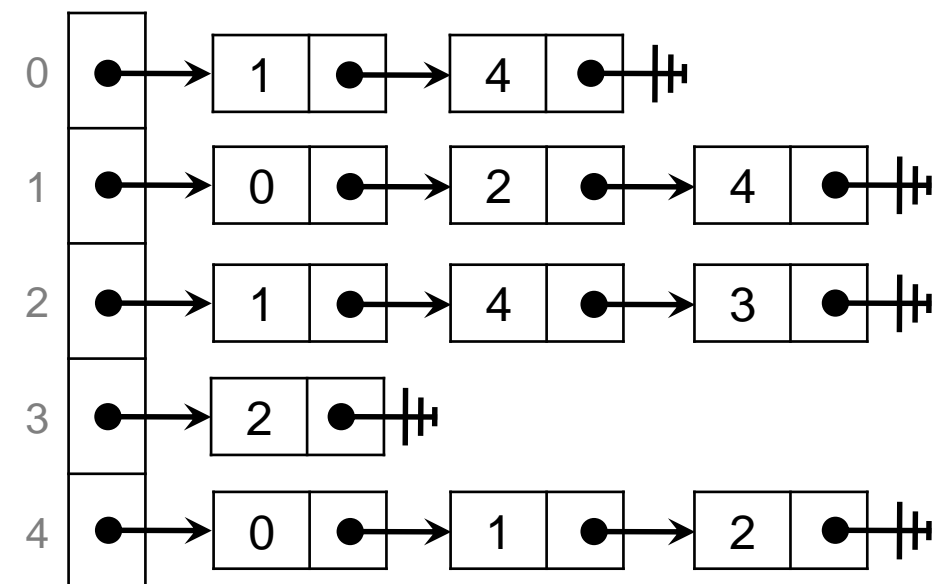
- For a graph with  $v$  vertices and  $e$  edges
- using a library based on the **adjacency list** representation

	Cost	Tally
<code>void graph_print(graph_t G) {</code>		
<code>  for (vertex v = 0; v &lt; graph_size(G); v++) {</code>	$v$ times	
<code>    printf("Vertices connected to %u: ", v);</code>	O(1)	O(v)
<code>    neighbors_t nbors = graph_get_neighbors(G, v);</code>	O(1)	O(v)
<code>    while (graph_ismore_neighbors(nbors)) {</code>	O(min(v,e)) times	O(v min(v,e))
<code>      vertex w = graph_next_neighbor(nbors);</code>	O(1)	O(v min(v,e))
<code>      printf(" %u,", w);</code>	O(1)	O(v min(v,e))
<code>    }</code>		
<code>  graph_free_neighbors(nbors);</code>	O(1)	O(v min(v,e))
<code>  printf("\n");</code>	O(1)	O(v min(v,e))
<code>}</code>		

- So the cost of `graph_print` is  $O(v \min(v, e))$

# What is the Cost of `graph_print`?

- The cost of `graph_print` is  $O(v \min(v, e))$ 
  - for a graph with  $v$  vertices and  $e$  edges using adjacency lists
- Is that right?
  - We assumed every vertex has  $O(\min(v, e))$  neighbors
  - But **overall** `graph_print` visits every edge exactly twice
    - once from each endpoint
    - the body of the inner loop runs  $2e$  times over all iterations of the outer loop
    - the entire inner loop costs  $O(e)$



# What is the Cost of `graph_print`?

- The entire inner loop costs  $O(e)$

	Cost	Tally
<pre>void graph_print(graph_t G) {</pre>		
<pre>  for (vertex v = 0; v &lt; graph_size(G); v++) {</pre>	$v$ times	
<pre>    printf("Vertices connected to %u: ", v);</pre>	$O(1)$	$O(v)$
<pre>    neighbors_t nbors = graph_get_neighbors(G, v);</pre>	$O(1)$	$O(v)$
<pre>    while (graph_ismore_neighbors(nbors)) {</pre>	$O(e)$	
<pre>      vertex w = graph_next_neighbor(nbors);</pre>		
<pre>      printf(" %u,", w);</pre>		$O(v + e)$
<pre>    }</pre>		
<pre>    graph_free_neighbors(nbors);</pre>	$O(1)$	$O(v + e)$
<pre>    printf("\n");</pre>	$O(1)$	$O(v + e)$
<pre>  }</pre>		
<pre>}</pre>		

- The actual cost of `graph_print` is  $O(v + e)$ 
  - for a graph with  $v$  vertices and  $e$  edges **using adjacency lists**

# What is the Cost of `graph_print`?

- Using the adjacency matrix representation
- By the same argument, the entire inner loop costs  $O(e)$ 
  - and `graph_free_neighbors` too

```
void graph_print(graph_t G) {
  for (vertex v = 0; v < graph_size(G); v++) {
    printf("Vertices connected to %u: ", v);
    neighbors_t nbors = graph_get_neighbors(G, v);
    while (graph_ismore_neighbors(nbors)) {
      vertex w = graph_next_neighbor(nbors);
      printf(" %u,", w);
    }
    graph_free_neighbors(nbors);
    printf("\n");
  }
}
```

	Cost	Tally
<code>for (vertex v = 0; v &lt; graph_size(G); v++) {</code>	$v$ times	
<code>printf("Vertices connected to %u: ", v);</code>	$O(1)$	$O(v)$
<code>neighbors_t nbors = graph_get_neighbors(G, v);</code>	$O(v)$	$O(v^2)$
<code>while (graph_ismore_neighbors(nbors)) {</code>	$O(e)$	$O(v^2 + e)$
<code>vertex w = graph_next_neighbor(nbors);</code>		
<code>printf(" %u,", w);</code>		
<code>graph_free_neighbors(nbors);</code>	$O(1)$	$O(v^2 + e)$
<code>printf("\n");</code>		

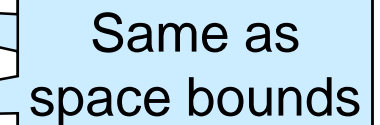
This is  $O(\min(v,e))$  by itself, but there are only  $2e$  neighbors to free

- The actual cost of `graph_print` is  $O(v^2 + e)$ 
  - This is  $O(v^2)$  since  $e \in O(v^2)$  always

# What is the Cost of `print_graph`?

- Adjacency list representation:  $O(v + e)$

- Adjacency matrix representation:  $O(v^2)$



Same as space bounds

- For a dense graph

  - $e \in O(v^2)$

they are the same

- For a sparse graph, AL is better

```
void graph_print(graph_t G) {
    for (vertex v = 0; v < graph_size(G); v++) {
        printf("Vertices connected to %u: ", v);
        neighbors_t nbors = graph_get_neighbors(G, v);
        while (graph_ismore_neighbors(nbors)) {
            vertex w = graph_next_neighbor(nbors);
            printf(" %u,", w);
        }
        graph_free_neighbors(nbors);
        printf("\n");
    }
}
```