# Lecture 23
# Representing Graphs

15-122: Principles of Imperative Computation (Spring 2018)
Frank Pfenning, André Platzer, Rob Simmons, Penny Anderson

In this lecture we introduce *graphs*. Graphs provide a uniform model for many structures, for example, maps with distances or Facebook relationships. Algorithms on graphs are therefore important to many applications. They will be a central subject in the algorithms courses later in the curriculum; here we only provide a very basic foundation for graph algorithms.

With respect to our learning goals we will look at the following notions.

**Computational Thinking:** We get a taste of the use of graphs in computer science. We note that some graphs are represented explicitly while others are kept implicit.

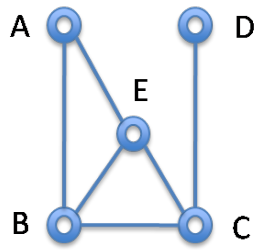
**Algorithms and Data Structures:** We see two basic ways to represent graphs: using adjacency matrices and by means of adjacency lists.

**Programming:** We use linked lists to give an adjacency list implementation of graphs.

 © Carnegie Mellon University 2018

# 1 Undirected Graphs

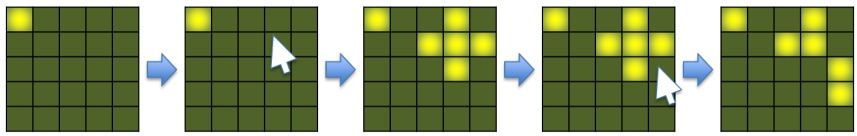
We start with *undirected graphs* which consist of a set $V$ of *vertices* (also called *nodes*) and a set $E$ of *edges*, each connecting two different vertices. The following is a simple example of an undirected graph with 5 vertices ($A, B, C, D, E$) and 6 edges ($AB$, $BC$, $CD$, $AE$, $BE$, $CE$):




We don't distinguish between the edge $AB$ and the edge $BA$ because we're treating graphs as undirected. There are many ways of defining graphs with slight variations. Because we specified above that each edge connects two different vertices, we will say that no vertex in a graph can have an edge from a node back to itself.

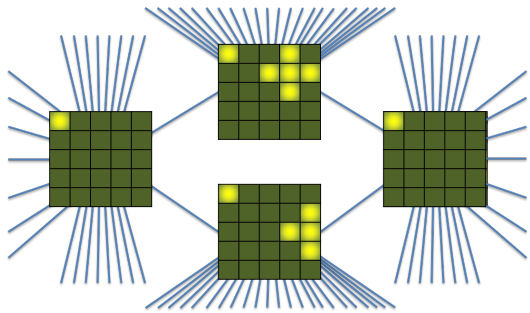
# 2 Implicit Graphs

There are many, many different ways to represent graphs. In some applications they are never explicitly constructed but remain implicit in the way the problem was solved. The game of *Lights Out* is one example of a game that implicitly describes an undirected graph. Lights Out is an electronic game consisting of a grid of lights, usually 5 by 5. The lights are initially pressed in some pattern of on and off, and the objective of the game is to turn all the lights off. The player interacts with the game by touching a light, which toggles its state and the state of all its cardinally adjacent neighbors (up, down, left, right).



We can think of lights out as an implicit graph with $2^{25}$ vertices, one for every possible configuration of the 5x5 lights out board, and an edge between

two vertices if we can transition from one board to another with a single button press. If we transition from one board to another by pressing a button, we can return to the first board by pressing the *same* button. Therefore the graph is undirected.



Each of the $2^{25}$ vertices is therefore connected to 25 different edges, giving us $25 \times 2^{25}/2$ total edges in this graph — we divide by 2 because going to a node and coming back from it are expressed by the same edge. But because the graph is implicit in the description of the Lights Out game, we don't have to actually store all 32 million vertices and 400 million edges in memory to understand Lights Out.

An advantage to thinking about Lights Out as a graph is that we can think about the game in terms of graph algorithms. Asking whether we can get all the lights out for a given board is asking whether the vertex representing our starting board is connected to the board with all the lights out by a series of edges: a *path*. We'll talk more about this *graph reachability* question in the next lecture.

## 3 Explicit Graphs and a Graph Interface

Sometimes we *do* want to represent a graph as an explicit set of edges and vertices and in that case we need a graph datatype. In the C code that follows, we'll refer to our vertices with unsigned integers. A minimal interface for graphs would allow us to create and free graphs, check whether an edge exists in the graph, and add a new edge to the graph.

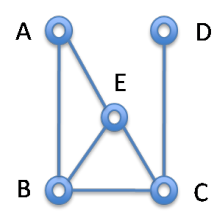
```
typedef unsigned int vertex;
typedef struct graph_header* graph_t;

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);
```

We use the C0 notation for contracts on the interface functions here. Even though C compilers do not recognize the `@requires` contract and will simply discard it as a comment, the contract still serves an important role for the programmer reading the program. For the graph interface, we decide that it does not make sense to add an edge into a graph when that edge is already there, hence the second precondition.

With this minimal interface, we can create a graph for what will be our running example (letting A = 0, B = 1, and so on):

```
graph_t G = graph_new(5);
graph_addedge(G, 0, 1); // AB
graph_addedge(G, 1, 2); // BC
graph_addedge(G, 2, 3); // CD
graph_addedge(G, 0, 4); // AE
graph_addedge(G, 1, 4); // BE
graph_addedge(G, 2, 4); // CE
```




We could implement this graph interface in a number of ways. In the simplest form, a graph with $e$ edges can be represented as a linked list or array of edges. In the simplest linked list implementation, it takes $O(1)$ time to add an edge to the graph with `graph_addedge`, because it can be appended to the front of the linked list. Finding whether an edge exists in a graph with $e$ edges might require traversing the whole linked list, so `graph_hasedge` is a $O(e)$ operation.

Hashtables and balanced binary search trees would be our standard

tools in this class for representing sets of edges more efficiently. Instead of taking that route, we will discuss two classic structures for directly representing graphs.

## 4 Adjacency Matrices

One simple way is to represent the graph as a two-dimensional array that describes its edge relation as follows.

| | A | B | C | D | E |
|---|---|---|---|---|---|
| A | | ✔ | | | ✔ |
| B | ✔ | | ✔ | | ✔ |
| C | | ✔ | | ✔ | ✔ |
| D | | | ✔ | | |
| E | ✔ | ✔ | ✔ | | |

There is a checkmark in the cell at row $v$ and column $v'$ exactly when there is an edge between nodes $v$ and $v'$. This representation of a graph is called an *adjacency matrix*, because it is a matrix that stores which nodes are neighbors.

We can check if there is an edge from B (= 1) to D (= 3) by looking for a checkmark in row 1, column 3. In an undirected graph, the top-right half of this two-dimensional array will be a mirror image of the bottom-left, because the edge relation is symmetric. Because we disallowed edges between a node and itself, there are no checkmarks on the main diagonal of this matrix.

The adjacency matrix representation requires a lot of space: for a graph with $v$ vertices we must allocate space in $O(v^2)$. However, the benefit of the adjacency matrix representation is that adding an edge (`graph_addedge`) and checking for the existence of an edge (`graph_hasedge`) are both $O(1)$ operations.

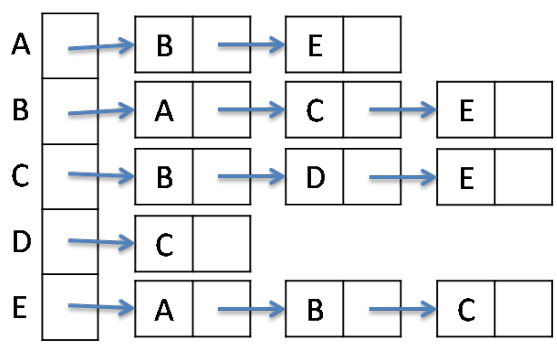
Are the space requirements for adjacency matrices (requires space in $O(v^2)$) worse than the space requirements for storing all the edges in a linked list (requires space in $O(e)$)? That depends on the relationship between $v$, the number of vertices, and $e$ the number of edges. A graph with $v$ vertices has between 0 and $\binom{v}{2} = \frac{v(v-1)}{2}$ edges. If most of the edges exist,

so that the number of edges is proportional to $v^2$, we say the graph is *dense*. For a dense graph, $O(e) = O(v^2)$, and so adjacency matrices are a good representation strategy for dense matrices, because in big-$O$ terms they don't take up more space than storing all the edges in a linked list, and operations are much faster.

## 5 Adjacency Lists

If a graph is not dense, then we say the graph is *sparse*. The other classic representation of a graphs, *adjacency lists*, can be a good representation of sparse graphs.

In an adjacency list representation, we have a one-dimensional array that looks much like a hash table. Each vertex has a spot in the array, and each spot in the array contains a linked list of all the other vertices connected to that vertex. Our running example would look like this as an adjacency list:




Adjacency lists require $O(\max(v, e))$ space to represent a graph with $v$ vertices and $e$ edges: we have to allocate a single array of length $v$ and then allocate two list entries per edge. Adding an edge is still constant time, but lookup (`graph_hasedge`) now takes time in $O(\min(v, e))$, since $\min(v-1, e)$ is the maximum length of any single adjacency list.

Our very simple graph interface doesn't let us take very good advantage of the adjacency list implementation. Using adjacency lists allows us to efficiently check many properties of the graph that, using only the `graph_hasedge` function, would be quite a bit more expensive to check. One example is finding some (unspecified) edge (if there is one) connected to a vertex. This is $O(1)$ if you can access the adjacency lists directly as the implementation, but if you're respecting the interface and only using

`graph_hasedge`, the same check takes time in $O(v)$. (We will add this functionality to the interface when we study search algorithms.) Another example is finding the neighbors of a vertex, which an adjacency list implementation can return in constant time (by simply returning the adjacency list of that vertex). Using only the interface operations, the cost rises to $O(v \min(v, e))$ — the cost calling `hasedge` on this vertex and all other vertices in the graph.

The following table summarizes and compares the asymptotic cost associated with the adjacency matrix and adjacency list implementations of a graph, under the assumptions used in this chapter. We also mention the cost of finding the neighbors of a vertex if we were to implement it as a library-side operation.

| | **Adjacency Matrix** | **Adjacency List** |
|---|---|---|
| Space | $O(v^2)$ | $O(\max(v, e))$ |
| `graph_hasedge` | $O(1)$ | $O(\min(v, e))$ |
| `graph_addedge` | $O(1)$ | $O(1)$ |
| *(finding the neighbors of a vertex)* | $O(v)$ | $O(1)$ |

The cost of `graph_hasedge` can be reduced by storing the neighbors of each node not in a linked list but in a more search-efficient data structure, for example an AVL tree or a hash set. Of course, doing so requires additional space, something that may not be desirable in some applications.

## 6 Adjacency List Implementation

The header for a graph is a struct with two fields: the first is an unsigned integer representing the actual size, and the second is an array of adjacency lists.

```
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;
};
```

We can allocate the struct using `xmalloc`, since we're going to have to initialize both its fields anyway. But we'd definitely allocate the adjacency list itself using `xcalloc` to make sure that it is initialized to array full of `NULL` values: empty adjacency lists.

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

Given two vertices, we have to search through the whole adjacency list for one vertex to see if it contains the other vertex. This is what gives the operation a running time in $O(\min(v, e))$.

```
bool graph_hasedge(graph* G, vertex v, vertex w) {
  REQUIRES(is_graph(G) && is_vertex(G, v) && is_vertex(G, w));

  adjlist *L = G->adj[v];
  while (L != NULL) {
    if (L->vert == w) return true;
    L = L->next;
  }
  return false;
}
```

Because we assume an edge must not already exist when we add it to the graph, we can add an edge in constant time:

```
void graph_addedge(graph* G, vertex v, vertex w) {
  REQUIRES(is_graph(G) && 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));
}
```

## Exercises

**Exercise 1.** *Define the representation functions* `is_graph` *and* `is_vertex` *(and any other you may need) used in the contracts of the adjacency matrix implementation in Section 6 of the graph interface of Section 3.*

**Exercise 2.** *Give an implementation of the graph interface in Section 3 based on adjacency matrices. Make sure to provide adequate representation functions.*