Lecture 23
Representing Graphs

15-122: Principles of Imperative Computation (Fall 2024)
Frank Pfenning, André Platzer, Rob Simmons,
Penny Anderson, Iliano Cervesato

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.

Additional Resources

- Review slides (`https://cs.cmu.edu/~15122/handouts/slides/review/23-graphs.pdf`)
- Code for this lecture (`https://cs.cmu.edu/~15122/handouts/code/23-graphs.tgz`)

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

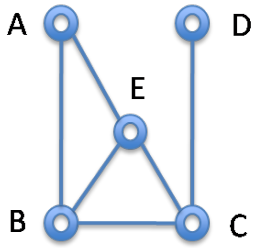
Computational Thinking: We get a taste of the use of graphs in computer science. We note that some graphs are represented explicitly while others are kept implicit.

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.

1 Undirected Graphs

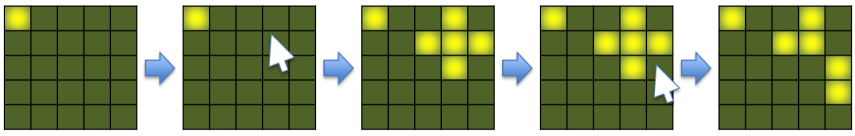
We start with *undirected graphs* which consist of a set V of *vertices* (also called *nodes*) and a set E of *edges*, each connecting two different vertices. The following is a simple example of an undirected graph with 5 vertices (A, B, C, D, E) and 6 edges (AB, BC, CD, AE, BE, CE):



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, no vertex in a graph can have an edge from a node back to itself in this course.

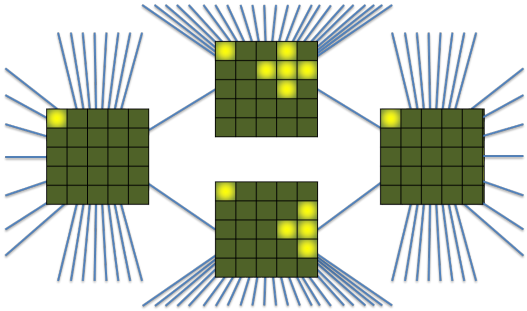
2 Implicit Graphs

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 situation 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. The minimal interface for graphs in Figure 3 allows us to create and free graphs, check whether an edge exists in the graph, add a new edge to the graph, and get, traverse and free the neighbors of a node.

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. The neighbors of a node are given to us as a value of the abstract type `neighbors_t`. We examine

```
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_hasmore_neighbors(neighbors_t nbors);
//@requires nbors  != NULL;

vertex graph_next_neighbor(neighbors_t nbors);
//@requires nbors  != NULL;
//@requires graph_hasmore_neighbors(nbors);

void graph_free_neighbors(neighbors_t nbors);
//@requires nbors  != NULL;
```

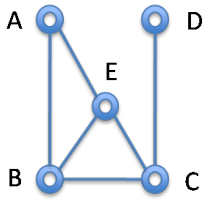
Figure 1: A simple graph interface — `graph.h`

neighbors by mean of the test `graph_hasmore_neighbors` and the *iterator* `graph_next_neighbor`. The function `graph_hasmore_neighbors` returns

`true` if there are more neighbors to examine and `false` otherwise. The function `graph_next_neighbor` returns the next unexamined neighbor — it will return a different neighbor each time it is called.

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 the graph interface in Figure 3 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 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 an $O(e)$ operation. Getting the neighbors of a node would take $O(e)$.

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 data 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.

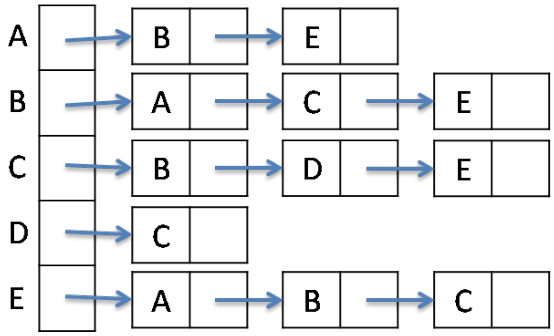
The adjacency matrix representation requires a lot of space: for a graph with v vertices we must allocate space in $O(v^2)$. However, the benefit of the adjacency matrix representation is that adding an edge (`graph_addedge`) and checking for the existence of an edge (`graph_hasedge`) are both $O(1)$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

Are the space requirements for adjacency matrices (which requires space in $O(v^2)$) worse than the space requirements for storing all the edges in a linked list (which 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 graphs, 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 $O(v + e)$ 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. The complexity class $O(v + e)$ is often written as $O(\max(v, e))$ — we leave it as an exercise to check that these two classes are equivalent — and therefore this is the notation we will typically use. 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. Finding the neighbors of a node is immediate with an adjacency list representation as we may simply grab the adjacency list of that node — this has cost $O(1)$. This is in contrast with the adjacency matrix representation where we are forced to check every value on the row of the matrix corresponding to that node.

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.

	Adjacency Matrix	**Adjacency List**
Space	$O(v^2)$	$O(\max(v, e))$
`graph_hasedge`	$O(1)$	$O(\min(v, e))$
`graph_addedge`	$O(1)$	$O(1)$
`graph_get_neighbors`	$O(v)$	$O(1)$

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. It also comes at the expense of `graph_get_neighbors`.

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. We use the vertex list from the graph interface as our adjacency list.

```
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 leave it as an exercise to the reader to define the representation functions

```
bool is_vertex(graph *G, vertex v)
bool is_graph(graph *G)
```

that check that a vertex is valid for a given graph and that a graph itself is valid.

We can allocate the struct for a new graph 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->size = size;
  G->adj = xcalloc(size, sizeof(adjlist*));
  ENSURES(is_graph(G));
  return G;
}
```

Given two vertices, we have to search through the whole adjacency list of 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));
```

```
  for (adjlist *L = G->adj[v]; L != NULL; L = L->next) {
    if (L->vert == w) return true;
  }
  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));  // add w as a neighbor of v
  L->vert = w;
  L->next = G->adj[v];
  G->adj[v] = L;

  L = xmalloc(sizeof(adjlist));  // add v as a neighbor of w
  L->vert = v;
  L->next = G->adj[w];
  G->adj[w] = L;

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

We represent the neighbors of a vertex as a struct with a single field containing a suffix of its adjacency list:

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

Implementing the representation invariant function

```
  bool is_neighbors(neighbors *nbors);
```

that checks that a value of type `neighbors*` is valid, is left to the reader.

As we iterates through the neighbors of a vertex, we move the value pointed to by the field `next_neighbor` along the adjacency list. Writing the representation invariant for values of this type is left as an exercise.

Finding the neighbors of a vertex is just a matter of creating an instance of this struct and initializing its field with the adjacency list of the vertex of interest.

```
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;
}
```

Implementing the emptiness check and the iterator are straightforward:

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

vertex graph_next_neighbor(neighbors *nbors) {
  REQUIRES(is_neighbors(nbors));
  REQUIRES(graph_hasmore_neighbors(nbors));

  vertex v = nbors->next_neighbor->vert;
  nbors->next_neighbor = nbors->next_neighbor->next;
  return v;
}
```

Observe that `graph_next_neighbor` updates the `next_neighbor` field to the next node in the adjacency list (the second precondition guarantees there must be a next node). Had we defined the neighbors of a vertex as its adjacency list (as opposed to a struct that points to its adjacency list), we would have no way do this — do you see why? The vertex that this function returns belongs to the graph and therefore `graph_next_neighbor` should not free its node in the adjacency list.

The function `graph_free_neighbors` frees the `neighbor_header` struct:

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

It is important that it doesn't free the adjacency list nodes pointed to by the `next_neighbor` field (if any): these belong to the graph and may be accessed again by future operations. These nodes will be freed by`graph_free` when the graph itself is not needed anymore.

7 Adjacency Matrix Implementation

We leave writing an adjacency matrix implementation of our graph interface as an exercise.

In the rest of his discussion, we will assume that it implements the operations of the graph interface with the following costs for a graph with v vertices and e edges: `graph_new` has cost $O(v^2)$, while the functions `graph_size`, `graph_hasedge`, `graph_addedge` and `graph_free` have all cost $O(1)$. The function `graph_get_neighbors` has cost $O(v)$, the function `graph_free_neighbors` costs $O(\min(v, e))$, while `graph_hasmore_neighbors` and `graph_next` both have constant cost.

Your implementation should aim for these costs and you should check that it achieves them.

8 Iterating through a Graph

To gain practice with working with our graph interface, we write a function that prints all the edges in a graph. Give it a try and then check your work on the next page. This function has the following prototype:

```
void graph_print(graph_t G)
```

Our implementation is as follows:

```
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_hasmore_neighbors(nbors)) {
      vertex w = graph_next_neighbor(nbors);  // w is a neighbor of v
      printf(" %u,", w);
    }
    graph_free_neighbors(nbors);
    printf("\n");
  }
}
```

The outer loop examines all the vertices in the graph. For each of them, we compute its neighbor list and then go through it using the iterator in the inner loop to print them. We call the function `graph_free_neighbors` to dispose of the neighbor list once we are done with it.

It is interesting to analyze the complexity of `graph_print` on a graph containing v vertices and e edges. We will do so first assuming the adjacency list implementation in Section 6 and then an adjacency matrix implementation that achieves the costs mentioned in Section 7.

The outer loop runs v times. Inside this loop, the following operations take place:

- Some print statements that we may assume have cost $O(1)$.
- A call to `graph_get_neighbors`, whose cost is constant in the adjacency list representation. Up to this point in the code, the cost of our function is $O(v)$.
- The body of the inner loop performs constant cost operations only. In isolation, the body of this loop runs $O(v)$ times since each vertex can have up to $v - 1$ neighbors. Thus, a naive analysis gives us an $O(v^2)$ worst-case complexity for `graph_print` up to this point in the code.

  However, each neighbor corresponds to an edge in the graph. Therefore, the body of the inner loop will be executed exactly $2e$ times total over an entire run of `graph_print` — each edge is examined twice, once from each of its endpoints. Thus the inner loop has cost $O(e)$ *overall*. Adding this to our tally, the cost of `print_graph` to this point in our analysis is $O(\max(v, e))$ — which we recall is the common way of writing $O(v + e)$.

- A call to `graph_free_neighbors`. This has constant cost.

Thus, `graph_print` has cost $O(\max(v, e))$ in the adjacency list representation.

Let's turn to the adjacency matrix implementation. The outer loop still runs v times and the cost of the operations inside this loop is as follows:

- The print statements cost $O(1)$.
- The call to `graph_get_neighbors` now costs $O(v)$ since we are using the adjacency matrix representation. Up to this point in the code, the cost of `graph_print` is now $O(v^2)$ in this representation.
- By the same analysis as in the adjacency list representation the body of the inner loop runs exactly $2e$ times, for a total cost of $O(e)$ *overall*. The cost up to this point using the adjacency matrix representation is therefore $O(\max(v^2, e))$. Since $e \in O(v^2)$ for any graph, this expression simplifies to $O(v^2)$.
- The call to `graph_free_neighbors` costs $O(e)$ *overall* in the adjacency matrix representation.

Summarizing, our analysis tells us that `graph_print` has cost $O(\max(v, e))$ in the adjacency list representation and $O(v^2)$ with the adjacency matrix representation. For a dense graph — where $e \in O(v^2)$ — these two expressions are equivalent. For a sparse graph, the former can be significantly cheaper.

9 Exercises

Exercise 1 (sample solution on page 16)**.** *Define the representation functions* `is_graph` *and* `is_vertex` *(and any other you may need) used in the contracts of the adjacency list implementation in Section Adjacency List Implementation of the graph interface of Section Explicit Graphs and a Graph Interface. Do not worry about making sure the linked lists are acyclic.*

Exercise 2 (sample solution on page 16)**.** *Give an implementation of the graph interface in Section Explicit Graphs and a Graph Interface based on adjacency matrices. Make sure to provide adequate representation functions.*

Exercise 3 (sample solution on page 20)**.** *Write a client-side function which takes in a* `graph_t G`*, and returns a new graph where only edges where both endpoints are even remain. You should try to make your code as efficient as possible.*

Exercise 4 (sample solution on page 21)**.** *Recall this segment from the lecture code graph invariant:*

```
bool is_acyclic(adjlist* start) {
  if (start == NULL) return true;
  adjlist* h = start->next;          // hare
  adjlist* t = start;                // tortoise
  while (h != t) {
    if (h == NULL || h->next == NULL) return true;
    h = h->next->next;
    //@assert t != NULL; // hare is faster and hits NULL quicker
    t = t->next;
  }
  //@assert h == t;
  return false;
}
```

In a v-vertex graph with e edges represented as an adjacency list, what is the complexity of checking that all chains are acyclic, in terms of v and e?

Exercise 5 (sample solution on page 22)**.** *Determine the complexity of the following code in terms of the number of vertices v and number of edges e of the graph* `G`*. Do your analysis for both adjacency matrix and adjacency list implementations.*

```
int f1(graph_t G) {
  unsigned int n = graph_size(G);
  int total = 0;
  for (unsigned int i = 0; i < n; i ++) {
    neighbors_t nbors = graph_get_neighbors(G, i);
```

```
    if (nbors != NULL) {
      total++;
    }
    else {
      graph_addedge(G, i, 0);
    }
    graph_free_neighbors(nbors);
  }
  return total;
}
```

Sample Solutions

Solution of exercise 1 The code for the function `is_vertex` is a fairly simply check, but for `is_graph` we need to make sure there are no `NULL` pointers, self loops, duplicate edges, or invalid edges. We also need to make sure that the graph is undirected, so we have a helper function to help us make sure that edges go in both directions.

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

bool is_in(adjlist* p, vertex v) {
  while (p != NULL) {
    if (p->vert == v) return true;
    p = p->next;
  }
  return false;
}

bool is_graph(graph *G) {
  if (G == NULL) return false;
  if (G->adj == NULL) return false;
  for (unsigned int i = 0; i < G->size; i++) {
    if (!is_acyclic(G->adj[i])) return false;
    for (adjlist *p = G->adj[i]; p != NULL; p = p->next) {
      //Check for valid vertex and self-loops
      if (p->vert == i || !(is_vertex(G, p->vert))) return false;
      //Make sure graph is symmetric
      if (!is_in(G->adj[p->vert], i)) return false;
      //Make sure edge only appears once
      if (is_in(p->next, p->vert)) return false;
    }
  }
  return true;
}
```

The function `is_acyclic`, which checks that a linked list is `NULL`-terminated (by virtue of being acyclic) has been omitted.

Solution of exercise 2 Graph are defined as the type `graph`, a struct consisting of the number of vertices and an array of array of booleans. Neighbors

are defined as in the adjacency list representation. The representation invariant function `is_vertex` stays unchanged with respect to the adjacency list representation. The data structure invariant function `is_graph` is simpler: it checks that neither the input graph `G` nor that the array of array `G->adj` nor any of the arrays in it are `NULL`, it then checks that the elements on the diagonal of the matrix are set to `false` (to verify that there are no self-edges) and that the matrix is symmetric (to ensure that the graph is undirected).

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

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

bool is_graph (graph *G) {
  if (G == NULL || G->adj == NULL) return false;
  for (unsigned int i = 0; i < G->size; i++) {
    if (G->adj[i] == NULL) return false;
    //No self-loops
    if (G->adj[i][i]) return false;
    for (unsigned int j = 0; j < G->size; j++) {
      //Must be undirected
      if (G->adj[i][j] != G->adj[j][i]) return false;
    }
  }
  return true;
}
```

Given these premises, the remaining functions are fairly straightforward. The function `graph_new(numvert)` creates an array of `numvert` arrays, and then populates each of its element with an array of `numvert` booleans. Using `xcalloc` automatically initialized their elements to `false` (since the newly created graph shall have no edges). The function `graph_free` shall free all these arrays of booleans, the array that contains them, and naturally the struct that represents the graph — in this order.

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

graph *graph_new(unsigned int numvert) {
  graph *G= xmalloc(sizeof(graph));
  G->size = numvert;
  bool **matrix = xmalloc(G->size * sizeof(bool*));
  for (unsigned int i = 0; i < G->size; i++) {
    matrix[i] = xcalloc(G->size, sizeof(bool));
  }
  G->adj = matrix;
  ENSURES(is_graph(G));
  return G;
}

bool graph_hasedge(graph *G, vertex v, vertex w) {
  REQUIRES(is_graph(G));
  REQUIRES(v < graph_size(G) && w < graph_size(G));
  return G->adj[v][w];
}

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

  G->adj[v][w] = true;
  G->adj[w][v] = true;

  ENSURES(is_graph(G));
}

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

The type `neighbor_header` is defined exactly as for the adjacency list representation, i.e., as a struct pointing to a linked list of vertices. For clarity, we rename the node type as `neighbor` although it is an adjacency list. The function `graph_get_neighbor` creates the structs and adds a vertex to the list it contains whenever the matrix reads `true` for that vertex.

```
typedef struct neighbor_list neighbor;
struct neighbor_list {
  vertex vert;
  neighbor *next;
};

struct neighbor_header {
  neighbor *neighbors;
};
typedef struct neighbor_header neighbors;

neighbors_t graph_get_neighbors(graph *G, vertex v) {
  REQUIRES(is_graph(G) && v < graph_size(G));
  neighbors *N = xcalloc(sizeof(neighbors), 1);
  for (vertex i = 0; i < G->size; i++) {
    if (G->adj[i][v]) {
      //Insert new neighbor at beginning of list
      neighbor *n = xmalloc(sizeof(neighbor));
      n->vert = i;
      n->next = N->neighbors;
      N->neighbors = n;
    }
  }
  ENSURES(N != NULL);
  return N;
}

bool graph_hasmore_neighbors(neighbors_t nbors) {
  REQUIRES(nbors != NULL);
  return nbors->neighbors != NULL;
}
```

The remaining functions on neighbors are straightforward. In the function `graph_free_neighbors`, we must be careful not to free a node before we set up an alternate way to access it successor.

```
vertex graph_next_neighbor(neighbors_t nbors) {
  REQUIRES(nbors != NULL);
  REQUIRES(graph_hasmore_neighbors(nbors));
  neighbor *n = nbors->neighbors;
  nbors->neighbors = n->next;
  vertex res = n->vert;
  free(n);
  return res;
}

void graph_free_neighbors(neighbors_t nbors) {
  REQUIRES(nbors != NULL);
  neighbor *n = nbors->neighbors;
  while (n != NULL) {
    neighbor *next = n->next;
    free(n);
    n = next;
  }
  free(nbors);
}
```

Solution of exercise 3 Intuitively, we want to create a new graph and add every edge whose endpoints are even. Doing so by looping over all possible pairs of even vertices would work, but that would cost $O(v^2)$, which is high for sparse graphs. A more efficient solution, assuming an adjacency list implementation, loops over the neighbors of the even vertices. The resulting complexity is $O(v + e \min(e, v))$.

```
graph_t even_connected(graph_t G) {
  REQUIRES(G != NULL);
  int n = graph_size(G);
  graph_t G_new = graph_new(n);
  for (int i = 0; i < n; i += 2) {
    if (i % 2 == 0) {
      neighbors_t nbors = graph_get_neighbors(G, i);
      while (graph_hasmore_neighbors(nbors)) {
        vertex w = graph_next_neighbor(nbors);
        if (w % 2 == 0 && !graph_hasedge(G_new, i, w)) {
          graph_addedge(G_new, i, w);
        }
      }
      graph_free_neighbors(nbors);
    }
  }
  ENSURES(G_new != NULL);
  return G_new;
}
```

Note the call to `graph_hasedge` in the innermost conditional. This is necessary because of the preconditions of `graph_addedge` which is specialized to undirected graphs: once we have added the edge (u, w) from u, we shall not add (w, u) from w since that's the same edge again.

The inner loop will run $2e$ times, just like for `graph_print`. The function `graph_hasedge` is called on all vertices both of whose endpoints are even, which in the worst case will be all e edges. Each time it is called, it will inspect one of the adjacency lists, which has cost $O(\min(e, v))$. Therefore, the cost of the inner loop is $O(e \min(e, v))$. Consequently, by reasoning in the same was as for `graph_print`, the cost of the function is $O(v + e \min(e, v))$.

Solution of exercise 4 The overall acyclicity test has cost $O(v + e)$.

In order to check acyclicity on a graph, we have to loop through all of the vertices (which costs $O(v)$), and for each vertex check its neighbors. However, some vertices might have a small number of edges, or none at all, while some vertices might have a lot. Thus, we do not know what the cost of a single `is_acyclic` call will be, but we can figure out the total cost.

Over the course of all of the acyclic checks, we eventually loop over all of the neighbors. The overall number of neighbors visited is $2e$, since each edge appears twice in our graph. Thus, the total cost of checking the chains is $O(v + e)$, since we have to check the chain of each vertex (each index of the adjacency list), even if it is empty, and the total work of checking the

non-empty chains is $O(e)$.

Solution of exercise 5 In both cases, the loop runs v times.

Adjacency list representation: Each of the v calls to `graph_get_neighbors` costs $O(1)$. So, this operation contributes $O(v)$ over all iterations of the loop. The same is true for `graph_free_neighbors`.

The function `graph_addedge` has $O(1)$ cost and it runs at most v times, for a total cost equal to $O(v)$.

Therefore the overall cost of `f1` is $O(v)$ using the adjacency list representation.

Adjacency matrix representation: In this representation, `graph_get_neighbors` costs $O(v)$, for an overall cost of $O(v^2)$ over all iterations of the loop.

Each call to `graph_addedge` costs $O(1)$, for a worst-time overall cost of $O(v)$.

The cost of `graph_free_neighbors` is more interesting. Each call has cost $O(\min(e, v))$, however the overall number of neighbor nodes that are freed is exactly $2e$. Thus, the overall cost over all iterations of the loop is $O(e)$.

The total cost of `f1` is $O(v^2 + v + e)$ which simplifies to $O(v^2)$ since $e \leq v^2$.