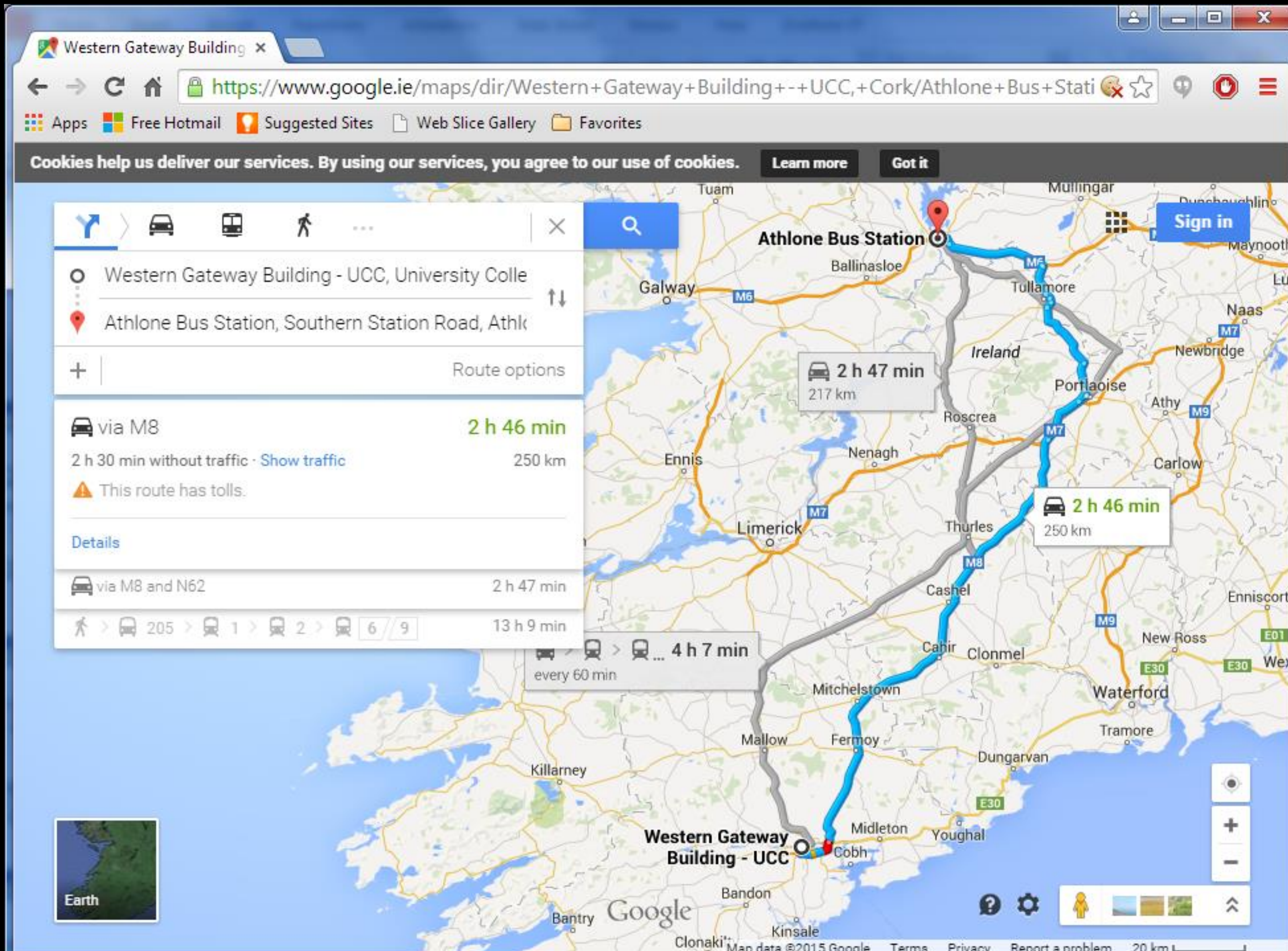
# The Graph ADT and implementations

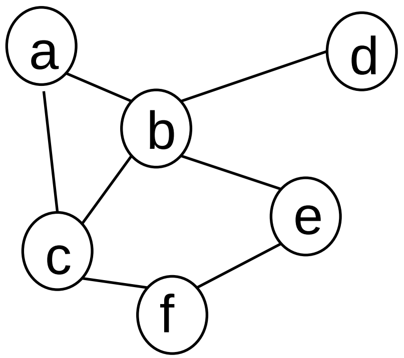


# Graphs

A graph is an abstract representation of the relationships between a set of objects.

We saw graphs in CS1113:

- social network graphs, call graphs, route map graphs, ...
- graph properties
- shortest path algorithms
- spanning trees



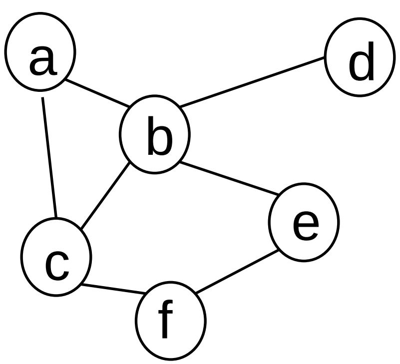

But it was all on paper. We didn't see how to

- implement them efficiently
- implement efficient algorithms for processing them

A *simple graph* is a pair (V,E), where V is a set of *vertices*, and E is a set of *edges*, and each edge is a set {x,y}, where x and y are vertices in V.

The *degree* of a vertex x is the number of edges that contain x.

Two vertices x and y are *adjacent* if there is an edge {x,y}.
Edge {x,y} is *incident* on x (and incident on y).
The *neighbours* of a vertex x are all other vertices adjacent to x




V = {a,b,c,d,e,f}

E = { {a,b}, {a,c}, {b,c}, {b,d}, {b,e},
{c,f}, {e,f} }

degree(b) = 4

neighbours(b) = {a,c,d,e}

In a *multigraph* E is a bag of edges, and so there may be multiple edges {x,y} in E for the same pair of vertices x and y.

In a *directed* graph, each edge is an ordered pair (x,y).

For a directed graph,

- the *out-degree* of a vertex is the number of edges with x as the first element of the pair
- the *in-degree* of a vertex .... x as the second element.

A *weighted* graph has a function w from E to some set, defining a *weight* with the edge.

We can also associate *labels* from some set L with either vertices or edges.

# Vertex and Edge ADTs

Vertex

element(): returns the id of the vertex

Edge

vertices(): returns the pair of vertices the edge is incident on

opposite(x): if the edge is incident on x, return the other vertex;

element(): return the label of the edge

# (Undirected) Graph ADT

| | |
|---|---|
| vertices(): | return a list of all vertices |
| edges(): | return a list of all edges |
| num_vertices(): | return the number of vertices |
| num_edges(): | return the number of edges |
| | |
| get_edge(x,y): | return the edge from x to y |
| | |
| degree(x): | return the degree of vertex x |
| get_edges(x): | return a list of all edges incident on x |
| | |
| add_vertex(elt): | add  a new vertex with element = elt |
| add_edge(x, y, elt) | a new edge between x and y, with element elt |
| | |
| remove_vertex(x): | remove vertex and all incident edges |
| remove_edge(e): | remove edge e |

# Implementing the ADT

The main operations will be retrieving vertices and edges. Updating will be relatively rare.

For the edges, there are 4 main options:

1. a list of edges

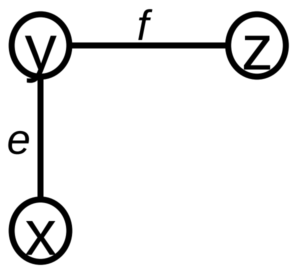
2. adjacency list:
   - for each vertex, store a list of the edges incident on it

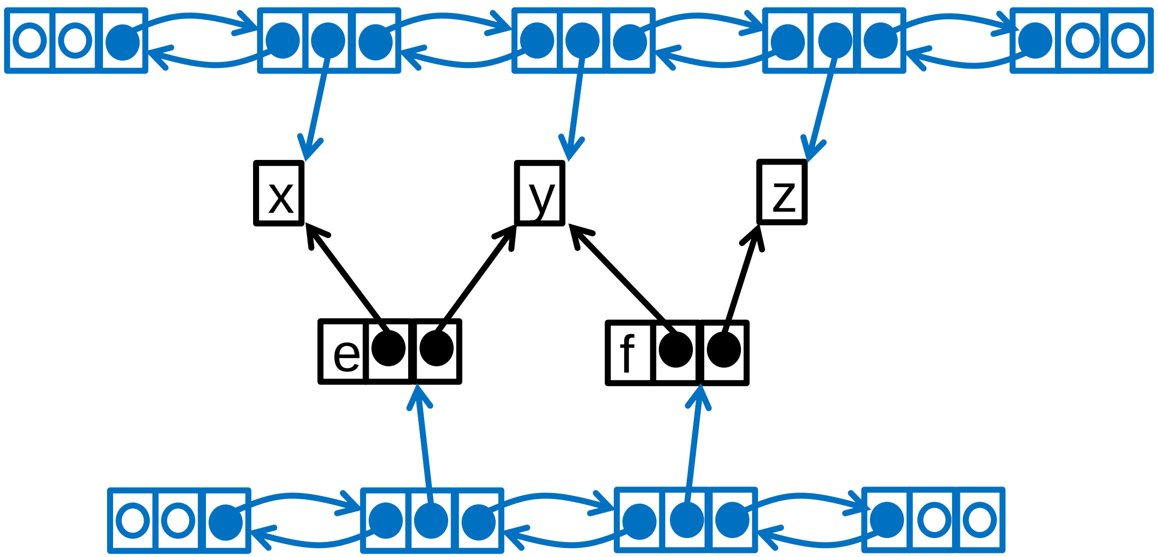
3. adjacency map:
   - for each vertex, store a map of the edges incident on it, using the other vertex as the key

4. adjacency matrix:
   - maintain a 2D array, where matrix[i][j] contains a reference to the edge {i,j}  (or between the ith and jth vertices)

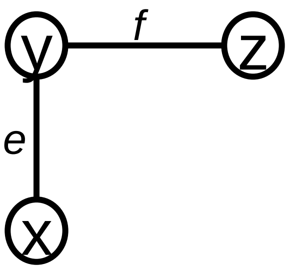
# Edge List

Maintain the vertices and edges in unordered linked lists.



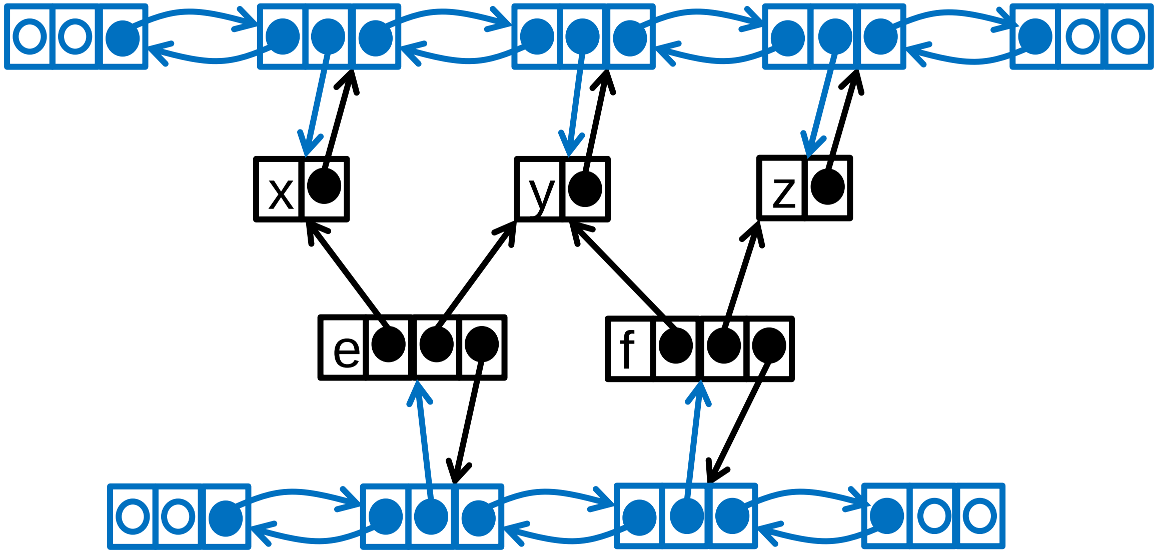
# Edge List (improved)




Maintain the vertices and edges in unordered linked lists.

each vertex maintains a reference back to the list elt

each edge maintains a reference back to the list elt

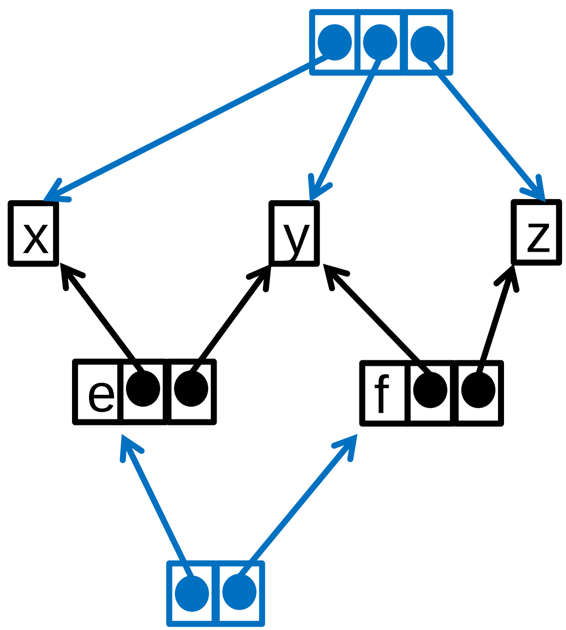


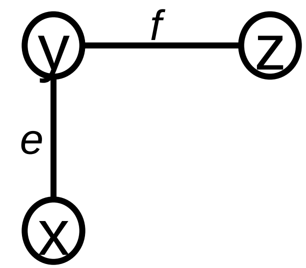
# Edge List: complexity

If n is the number of vertices, and m is the number of edges,

| | |
|---|---|
| Space complexity: | O(n + m) |
| | |
| get_edge(x,y): | O(m) – must check each edge |
| degree(x): | O(m) – must check each edge |
| get_edges(x): | O(m) – must check each edge |
| | |
| add_vertex(elt): | O(1) |
| add_edge(x, y, elt): | O(1) |
| | |
| remove_edge(e): | O(1) |
| | |
| remove_vertex(x): | O(m) - must check each edge |

Would it make a difference if we sorted the lists?

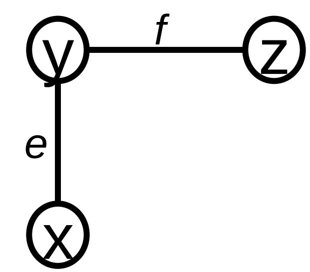

x
y
z
e
f


y
f
z
e
x

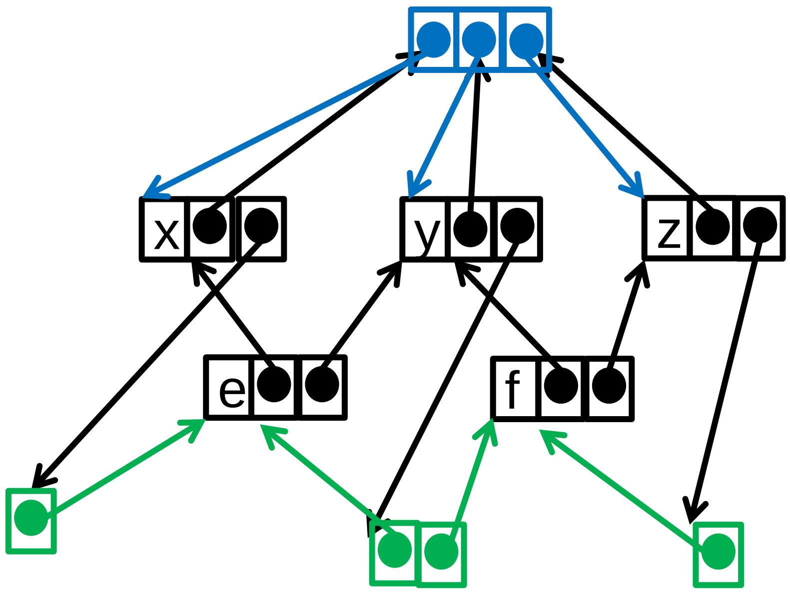
# Adjacency List

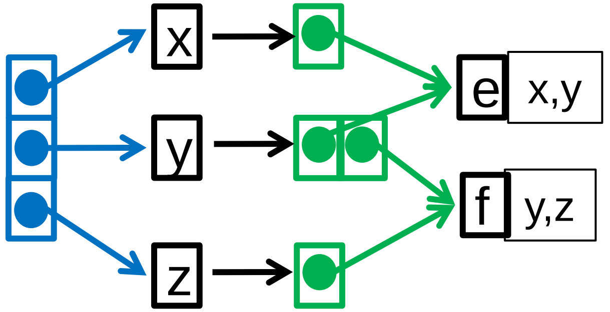
Maintain a list of vertices.

Each vertex points to a list of edges that are incident on it

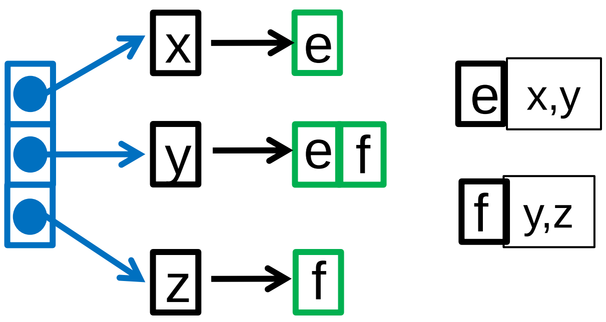



*(now just drawing lists like arrays ...)*




This is the simpler sketch to understand, as long as we remember what references we are hiding ...

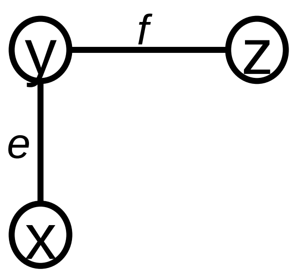


# Adjacency List: complexity

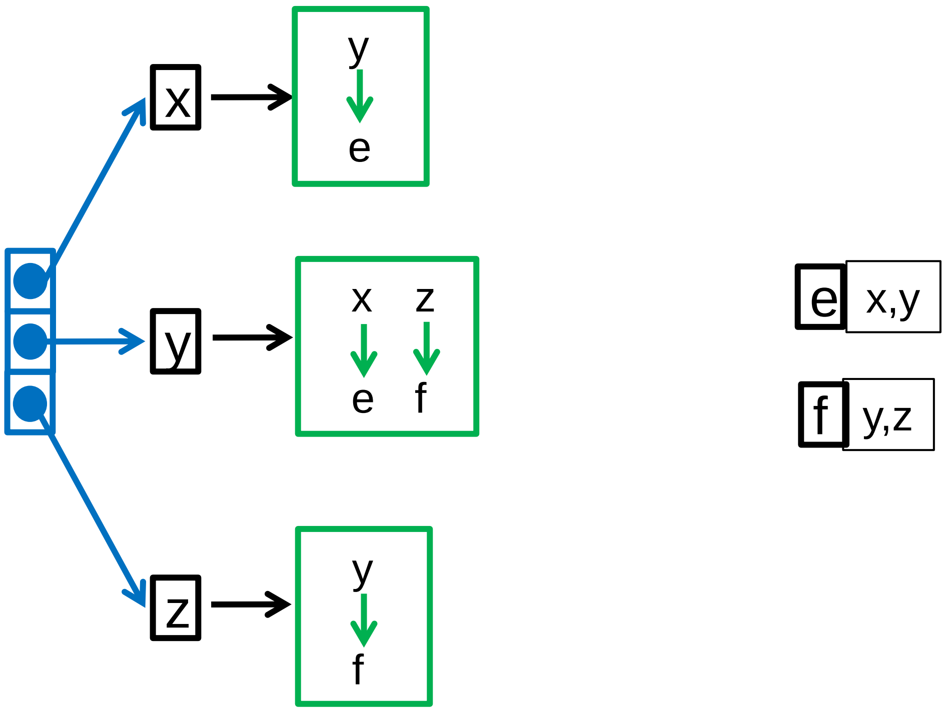
If n is the number of vertices, and m is the number of edges,

| | |
|---|---|
| Space complexity: | O(n + m) |
| | |
| get_edge(x,y): | O(min(degree(x), degree(y))) |
| degree(x): | O(1) |
| get_edges(x): | O(degree(x)) |
| | |
| add_vertex(elt): | O(1) |
| add_edge(x, y, elt): | O(1) |
| | |
| remove_edge(e): | O(1) |
| | |
| remove_vertex(x): | O(degree(x)) |

# Adjacency map




Maintain a list of vertices
Each vertex maintains a hash-map of its edges,
using the other vertices as the key.

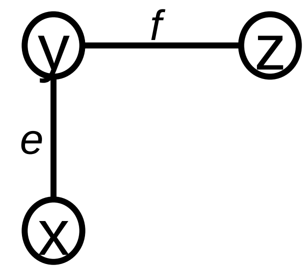


# Adjacency Map: complexity

If n is the number of vertices, and m is the number of edges,

| | | |
|---|---|---|
| Space complexity: | O(n + m) | |
| | | worst case |
| get_edge(x,y): | O(1) *expected* | O(min(degree(x), degree(y))) |
| degree(x): | O(1) | |
| get_edges(x): | O(degree(x)) | |
| add_vertex(elt): | O(1) | |
| add_edge(x, y, elt): | O(1) *expected* | |
| remove_edge(e): | O(1) *expected* | |
| remove_vertex(x): | O(degree(x)) | |

# Adjacency matrix




Associate a unique integer in 0 to n-1 with each vertex

Maintain a 2D array, where cell[i][j] contains a reference to the edge between i and j

| | | 0 | 1 | 2 |
|---|---|---|---|---|
| x → | 0 | | e | |
| y → | 1 | e | | f |
| z → | 2 | | f | |

| | |
|---|---|
| e | x,y |
| f | y,z |

# Adjacency matrix: complexity

If n is the number of vertices, and m is the number of edges,

| | | |
|---|---|---|
| Space complexity: | $O(n^2)$ | Wasteful for sparse graphs |
| get_edge(x,y): | $O(1)$ | |
| degree(x): | $O(n)$ | |
| get_edges(x): | $O(n)$ | |
| add_vertex(elt): | $O(n^2)$ | |
| add_edge(x, y, elt): | $O(1)$ | |
| remove_edge(e): | $O(1)$ | |
| remove_vertex(x): | $O(n^2)$ | |

# Summary

| | edge list | adjacency list | adjacency map | adjacency matrix |
|---|---|---|---|---|
| Space | $O(n + m)$ | $O(n + m)$ | $O(n + m)$ | $O(n^2)$ |
| get_edge | $O(m)$ | $O(\min(\deg(x), \deg(y)))$ | $O(1)$ *expected* | $O(1)$ |
| degree | $O(m)$ | $O(1)$ | $O(1)$ | $O(n)$ |
| get_edges(x) | $O(m)$ | $O(\deg(x))$ | $O(\deg(x))$ | $O(n)$ |
| add_vertex | $O(1)$ | $O(1)$ | $O(1)$ | $O(n^2)$ |
| add_edge | $O(1)$ | $O(1)$ | $O(1)$ *expected* | $O(1)$ |
| remove_edge | $O(1)$ | $O(1)$ | $O(1)$ *expected* | $O(1)$ |
| remove_vertex | $O(m)$ | $O(\deg(x))$ | $O(\deg(x))$ | $O(n^2)$ |

Adding to the underlying structures may change some of these complexities

# Next lecture

Graph traversals