Graphs

Graph Representations

Stefano Quer
Dipartimento di Automatica e Informatica
Politecnico di Torino

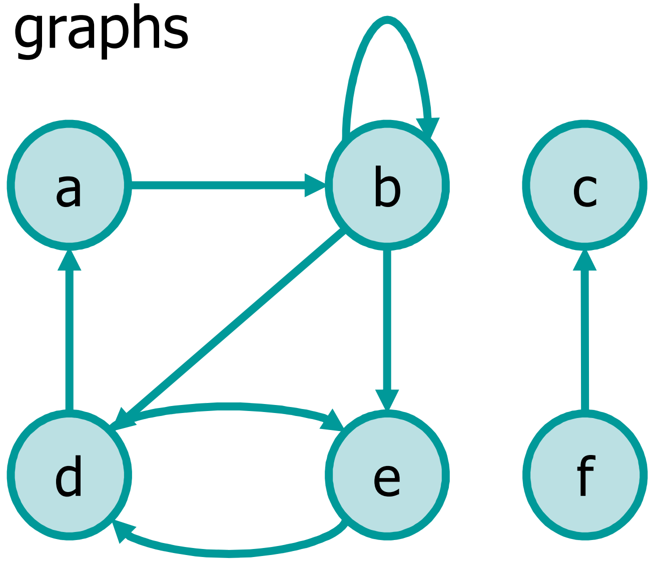
Why graph?

- Many practical applications
- Hundreds of algorithms
- Interesting abstraction usable in various domains
  - Connections
  - Cycles
  - Shortest paths
  - Etc.
- Active research area in Computer Science and Discrete Mathematics

Representations of graphs

- Representation of graphs
  - Adjacency matrix
  - Adjacency list
- Both of them can be applied to directed, undirected and weighterd graphs

G=(V, E)



Adjacency matrix

- Given G = (V, E), its adjacency matrix is
  - A matrix M of |V| x |V| elements

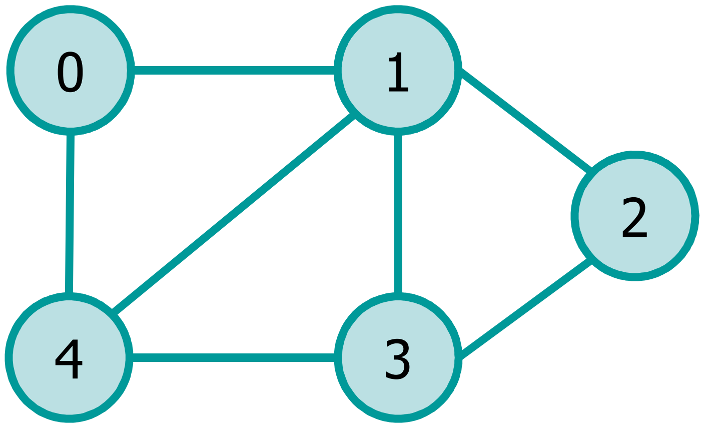
$$M[i, j] = \begin{cases} 1 & \text{if}(i, j) \in E \\ 0 & \text{if}(i, j) \notin E \end{cases}$$

- For undirected graphs A is symmetric

Example: Undirected graph

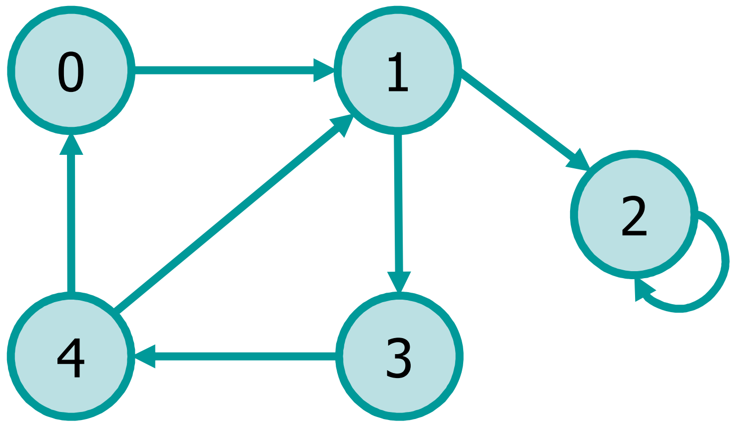
	0	1	2	3	4
0	0	1	0	0	1
1	1	0	1	1	1
2	0	1	0	1	0
3	0	1	1	0	1
4	1	1	0	1	0

Symmetric



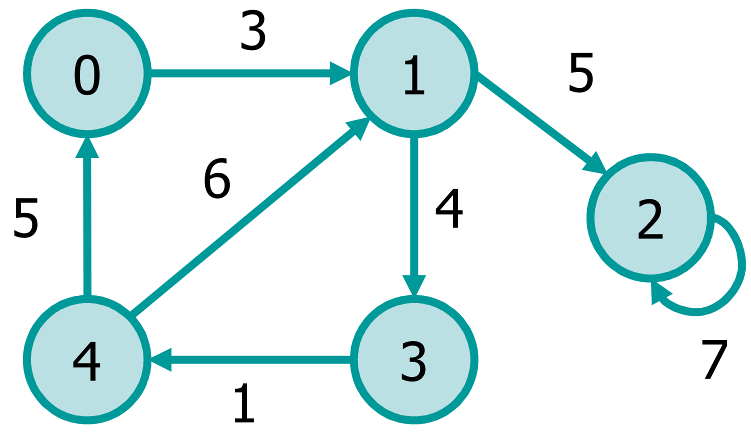
Example: Directed graph

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



Example: Weighted Directed graph

	0	**1**	**2**	**3**	**4**
0	0	3	0	0	0
1	0	0	5	4	0
2	0	0	7	0	0
3	0	0	0	0	1
4	5	6	0	0	0



Graph library (with adjacency matrix)

- Possible implementations
  - Static 2D matrix
  - Dynamic 2D matrix
    - Array of pointers to arrays, i.e., vertex array of structures with dynamic array of vertices
    - Use a struct when it is necessary to store edge/vertex attributes

Graph library (with adjacency matrix)

- Input file format

  nVertex dir/undirected

  vertex1 vertex2 weight

  …

- Example

  12 1

  2 3 4

  2 4 5

  6 7 1

If 0 → undirected graph
If it is not present → directed grph

Unweighted graph have weights equal to 1

Graph library (with adjacency matrix)

```
/* constant declaration */
#define MAX_LINE 100
enum {WHITE, GREY, BLACK};

/* type declarations */
typedef struct vertex graph_t;

/* array (vertices) with rows of adjacency matrix */
struct vertex {
  int id, color, scc;
  int td, tq, dist, pred;
  int *rowAdj;
};
```

Structure declaration with several extra attributes

Graph library (with adjacency matrix)

```
graph_t *graph_load(char *filename, int *nv) {
  graph_t *g;
  char line[MAX_LINE];
  int i, j, weight, dir;
  FILE *fp;

  fp = util_fopen(filename, "r");
  fgets(line, MAX_LINE, fp);
  if (sscanf(line, "%d%d", nv, &dir) != 2) {
    sscanf(line, "%d", nv);
    dir = 1;
  }

  g = (graph_t *)util_calloc(*nv, sizeof(graph_t));
```

Graph library (with adjacency matrix)

```
  for (i=0; i<*nv; i++) {
    g[i].id = i;
    g[i].color = WHITE;
    g[i].dist = INT_MAX;
    g[i].pred = g[i].scc = -1;
    g[i].td = g[i].tq = -1;
    g[i].rowAdj = (int *)util_calloc(*nv, sizeof(int));
  }
   while (fgets(line, MAX_LINE, fp) != NULL) {
    if (sscanf(line, "%d%d%d", &i, &j, &weight) != 3) {
      sscanf(line, "%d%d", &i, &j);
      weight = 1;
    }
    g[i].rowAdj[j] = weight;
    if (dir == 0) g[j].rowAdj[i] = weight;
  }
  fclose(fp);
  return g;
}
```

Graph library (with adjacency matrix)

It is often necessary to store a vertex id to matrix index correspondence

```
int graph_find(graph_t *g, int nv, int id){
  int i;

  for (i=0; i<nv; i++) {
    if (g[i].id == id) {
      return i;
    }
  }
  return -1;
}
```

It is possible to use a symbol table

st

0	1	2	3	4	...
idABC	idXYZ	idFOO	idBAR	...	

Graph library (with adjacency matrix)

Free the graph

```
void graph_dispose(graph_t *g, int nv) {
  int i;

  for (i=0; i<nv; i++) {
    free(g[i].rowAdj);
  }
  free(g);
}
```

Pro's and Con's

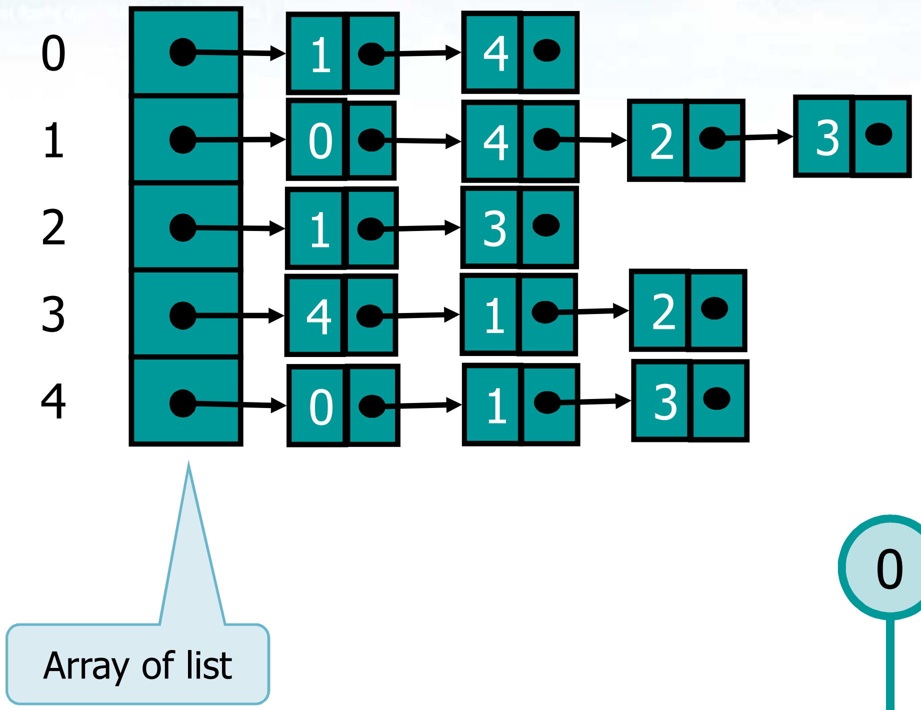
- Space complexity
  - $S(n) = \Theta(|V|^2)$
  - It is advantageous
    - For dense graphs, for which $|E|$ is close to $|V|^2$
    - When we need to be able to tell quickly if there is a connecting edge between two vertices
- No extra costs for storing the weights in a weighted graph
- Efficient access to graph topology

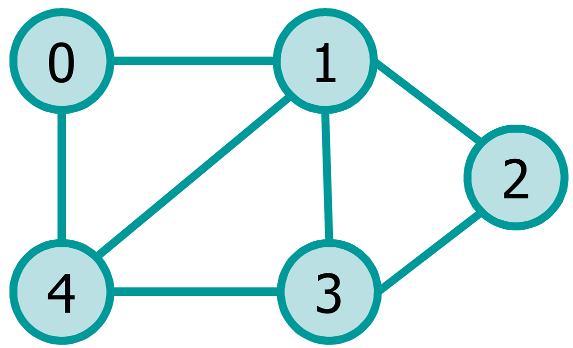
Boolean versus Integers

Adjacency list

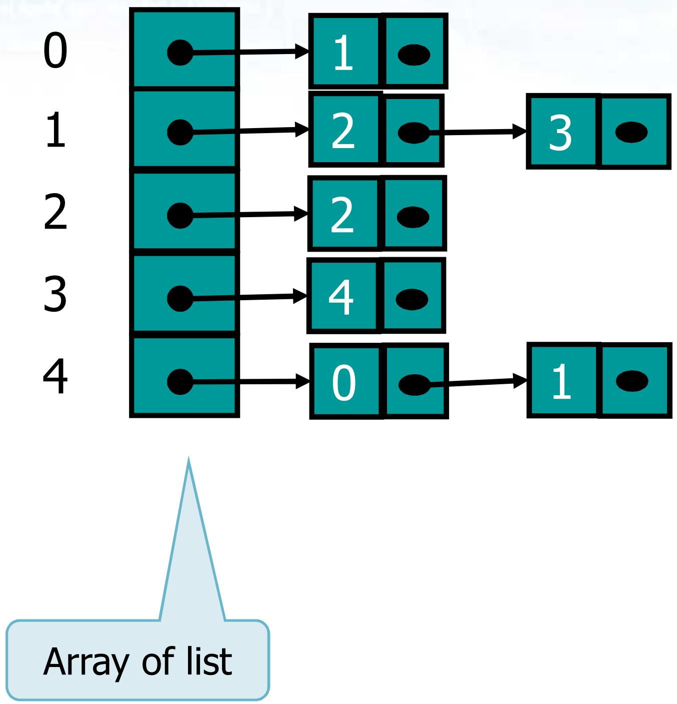
- Given G = (V, E), its adjacency list is
  - A main list representing vertices
  - A secondary list of vertices or edges for each element of the main list
- The list of list may have different implementations
  - A true list of lists
  - An array of lists
  - Etc.

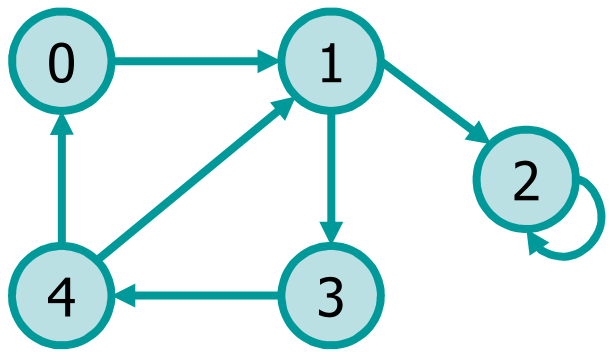
Example: Undirected graph



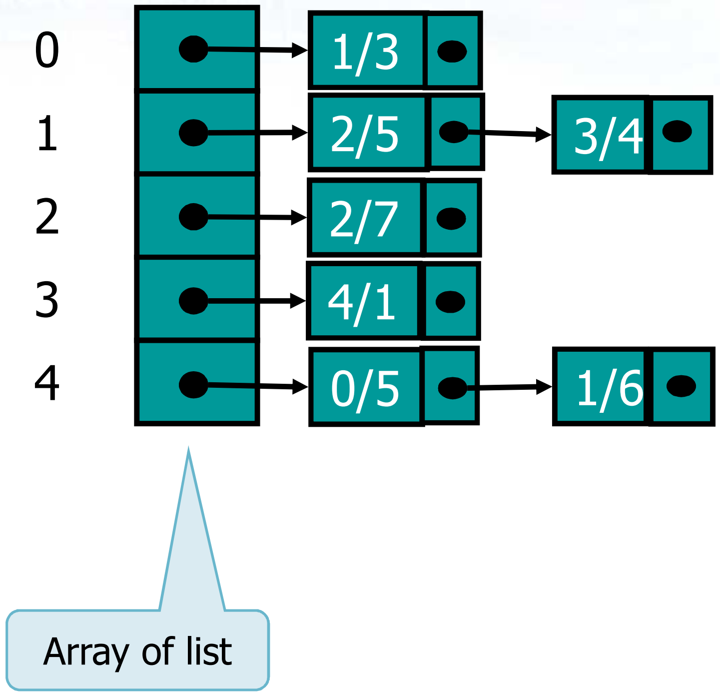


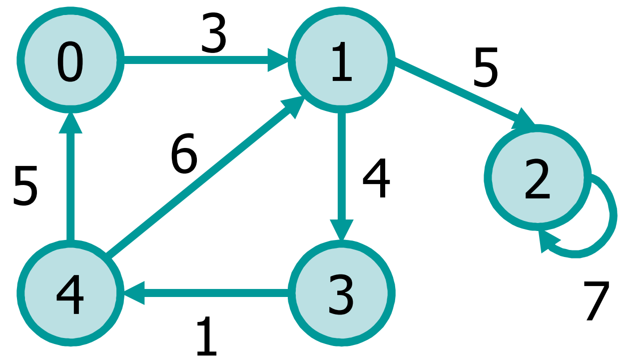
Example: Directed graph





Example: Weighted directed graph





Graph library (with adjacency list)

```
typedef struct edge edge_t;
typedef struct vertex graph_t;
struct edge {
  int weight;
  struct vertex *dst;
  struct edge *next;
};
struct vertex {
  int id, color, scc;
  int td, tq, dist;
  struct vertex *pred;
  struct edge *head;
  struct vertex *next;
};
```

Edges of the vertex list of edge list

Vertices of the vertex list of edge lilst

Graph library (with adjacency list)

```
graph_t *graph_load(char *filename, int *nv) {
  graph_t *g=NULL;
  char line[MAX_LINE];
  int i, j, weight, dir;
  FILE *fp;

  fp = util_fopen(filename, "r");
  fgets(line, MAX_LINE, fp);
  if (sscanf(line, "%d%d", nv, &dir) != 2) {
    sscanf(line, "%d", nv);
    dir = 1;
  }

  /* create initial structure for vertices */
  for (i=*nv-1; i>=0; i--) {
    g = new_node(g, i);
  }
```

Graph library (with adjacency list)

```
  /* load edges */
  while (fgets(line, MAX_LINE, fp) != NULL) {
    if(sscanf(line, "%d%d%d", &i, &j, &weight)!= 3) {
      sscanf(line, "%d%d", &i, &j);
      weight = 1;
    }
    new_edge(g, *nv, i, j, weight);
    if (dir == 0) {
      new_edge(g, *nv, j, i, weight);
    }
  }
  fclose(fp);

  return g;
}
```

Graph library (with adjacency list)

```
static graph_t *new_node(graph_t *g, int id) {
  graph_t *n;

  n = (graph_t *)util_malloc(sizeof(graph_t));
  n->id = id;
  n->color = WHITE;
  n->dist = INT_MAX;
  n->pred = NULL;
  n->scc = n->td = n->tq = -1;
  n->head = NULL;
  n->next = g;
  return n;
}
```

Graph library (with adjacency list)

```
static void new_edge(
  graph_t *g, int nv, int i, int j, int weight) {
  graph_t *src, *dst;
  edge_t *e;

  src = graph_find(g, nv, i);
  dst = graph_find(g, nv, j);

  e = (edge_t *)util_malloc(sizeof(edge_t));
  e->dst = dst;
  e->weight = weight;
  e->next = src->head;
  src->head = e;
}
```

Graph library (with adjacency list)

It is often necessary to store a vertex id to matrix index correspondence

```
graph_t *graph_find(graph_t *g, int nv, int id) {
  graph_t *n = g;

  while (n != NULL) {
    if (n->id == id) {
      return n;
    }
    n = n->next;
  }

  return NULL;
}
```

It is possible to use a symbol table

st

0	1	2	3	4	…
idABC	idXYZ	idFOO	idBAR	…	

Graph library (with adjacency list)

```
void graph_dispose(graph_t *g, int nv) {
  graph_t *n;
  edge_t *e;

  while (g != NULL) {
    n = g;
    while (n->head != NULL) {
      e = n->head;
      n->head = e->next;
      free(e);
    }
    g = n->next;
    free(n);
  }
}
```

Pro's and con's

- Undirected graphs
  - Total amount of elements in the lists = 2|E|
- Directed graphs
  - Total amount of elements in the lists = |E|
- Space complexity
  - $S(n) = O(max(|V|, |E|)) = O(|V+E|)$
  - It is advantageous for sparse graphs for which |E| is much less than $|V|^2$
- Verifying the existence of edge (u,v) requires scanning the adjacency list of u
- Memory needed to represent weights in weighted graphs