Graphs

Graph Representations

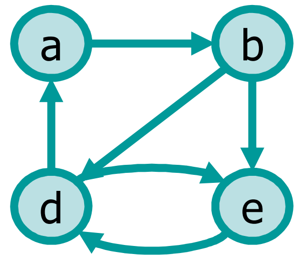
Paolo Camurati and Stefano Quer
Dipartimento di Automatica e Informatica
Politecnico di Torino

Why graph?

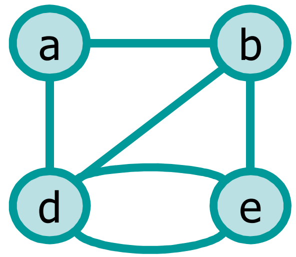
- They are used in
  - Many practical applications
  - Hundreds of algorithms
- They represent an interesting abstraction usable in various domains
  - Connections
  - Cycles
  - Shortest paths
  - Etc.
- Active research area in Computer Science and Discrete Mathematics

Representations of graphs

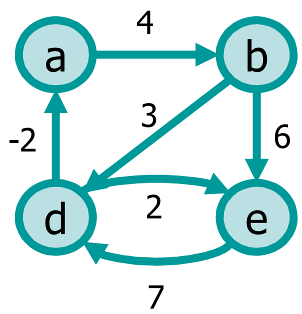
- Representation of graphs G = (V, E)
  - Adjacency matrix
  - Adjacency list
- Both of them can be applied to
  - Directed graphs

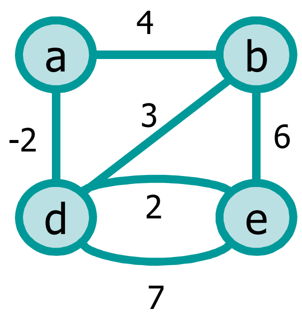


  - Undirected graphs



  - Weighted graphs





Adjacency matrix

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

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

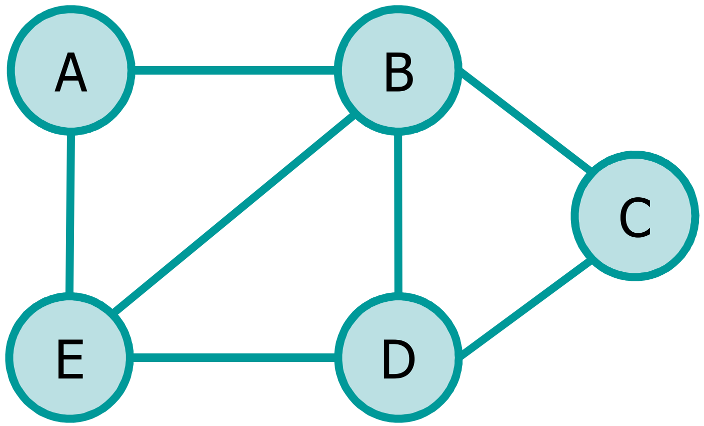
- For
  - Undirected graphs the matrix M is symmetric
  - Weighted graphs the matrix M stores the edges' weight

Example: Undirected graph

	A	B	C	D	E
A	0	1	0	0	1
B	1	0	1	1	1
C	0	1	0	1	0
D	0	1	1	0	1
E	1	1	0	1	0

In the general case we need a way (i.e., a symbol table) to map vertex identifiers to matrix (row and column) indices

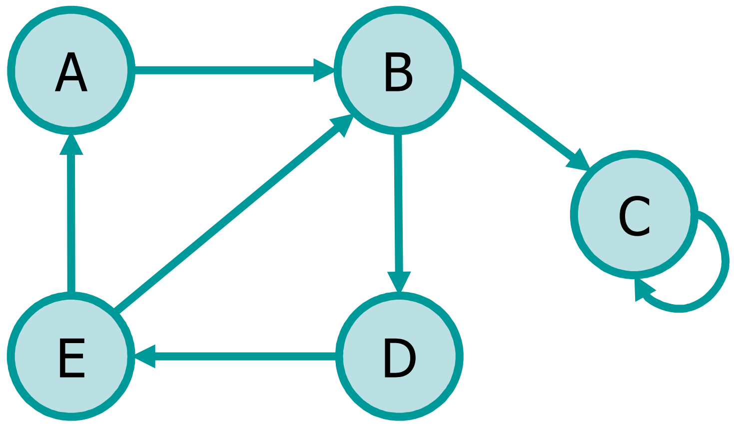
Symmetric



Example: Directed graph

	A	B	C	D	E
A	0	1	0	0	0
B	0	0	1	1	0
C	0	0	1	0	0
D	0	0	0	0	1
E	1	1	0	0	0

In the general case we need a way (i.e., a symbol table) to map vertex identifiers to matrix (row and column) indices

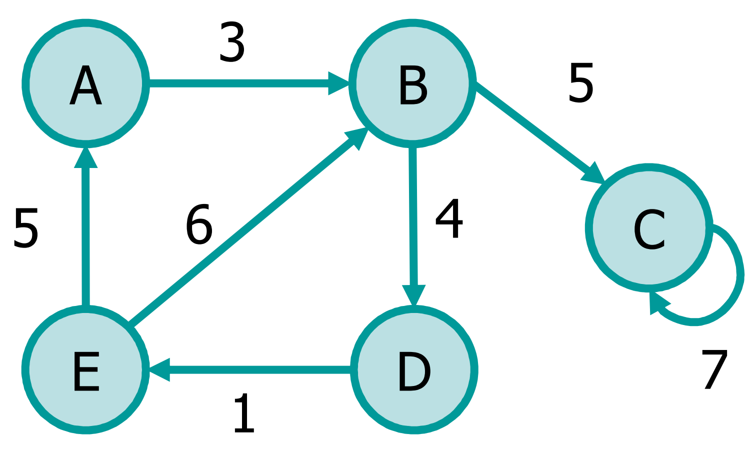


Example: Weighted Directed graph

	A	B	C	D	E
A	0	3	0	0	0
B	0	0	5	4	0
C	0	0	7	0	0
D	0	0	0	0	1
E	5	6	0	0	0

In the general case we need a way (i.e., a symbol table) to map vertex identifiers to matrix (row and column) indices

Integer values, real values, etc.



Graph library (with adjacency matrix)

- Possible implementations
  - Static 2D matrix
    - Either the graph size has to be known at compilation time
    - Or the program incurs into a memory loss
  - 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

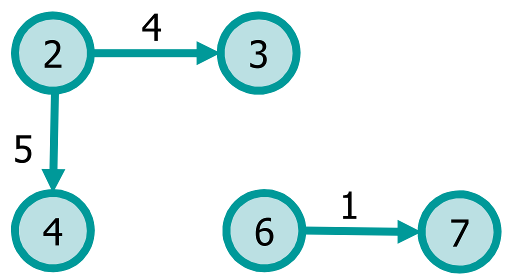
```
nVertex dir/undirected
vertex1  vertex2  weight
vertex1  vertex2  weight
vertex1  vertex2  weight
...
```

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

Unweighted graphs have all weights set equal to 1 **or** the field does not appear

- Example

```
12 1
2 3 4
2 4 5
6 7 1
...
```



Graph library (with adjacency matrix)

```
#define MAX_LINE 100

enum {WHITE, GREY, BLACK};

typedef struct graph_s graph_t;
typedef struct vertex_s vertex_t;

struct graph_s {
  vertex_t *g;
  int nv;
};
```

Enumeration types (or enum) is a user defined data type mainly used to assign names to constants easy to read and maintain
WHITE=0, GREY=1, BLACK=2

Graph Wrapper

Structure declaration with several extra attributes

```
struct vertex_s {
  int id;
  int color;
  int dist;
  int disc_time;
  int endp_time;
  int pred;
  int scc;
  int *rowAdj;
};
```

Graph library (with adjacency matrix)

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

  g = (graph_t *) util_calloc (1, sizeof(graph_t));

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

  g->g = (vertex_t *)
    util_calloc (g->nv, sizeof(vertex_t));
```

Function **util_fopen, util_calloc , etc.** belong to the ADT utility library

Graph library (with adjacency matrix)

```
  for (i=0; i<g->nv; i++) {
    g->g[i].id = i;
    g->g[i].color = WHITE;
    g->g[i].dist = INT_MAX;
    g->g[i].pred = g[i].scc = -1;
    g->g[i].disc_time = g[i].endp_time = -1;
    g->g[i].rowAdj = (int *)util_calloc(g->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->g[i].rowAdj[j] = weight;
    if (dir == 0) g->g[j].rowAdj[i] = weight;
  }
  fclose(fp);
  return g;
}
```

Graph library (with adjacency matrix)

```
void graph_attribute_init (graph_t *g) {
  int i;

  for (i=0; i<g->nv; i++) {
    g->g[i].color = WHITE;
    g->g[i].dist = INT_MAX;
    g->g[i].disc_time = -1;
    g->g[i].endp_time = -1;
    g->g[i].pred = -1;
    g->g[i].scc = -1;
  }

  return;
}
```

Graph library (with adjacency matrix)

It is often necessary to store the correspondence between identifiers and indices of each graph node to access the adjacency matrix

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

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

Trivial implementation (linear cost) !!!

Given the node identifier we get its matrix index and vice-versa

0	1	2	3	4	…
idABC	idXYZ	idFOO	idBAR	…	

st

It is possible to use any type of symbol table (e.g., hash-tables)

Graph library (with adjacency matrix)

Free the graph

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

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

  return;
}
```

Pro's and Con's

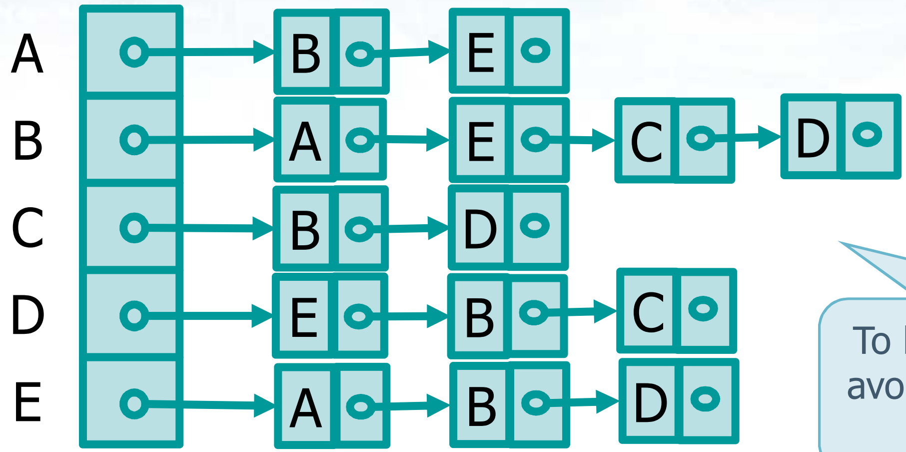
- Space complexity
  - Quadratic in the number of vertices |V|
    - $S(n) = \Theta(|V|^2)$
  - It is advantageous
    - For dense graphs, for which |E| is close to $|V|^2$
    - When we need to be able to tell quickly if there is a connecting edge between two vertices
- No extra costs for storing the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

Boolean versus Integers or Reals

Adjacency list

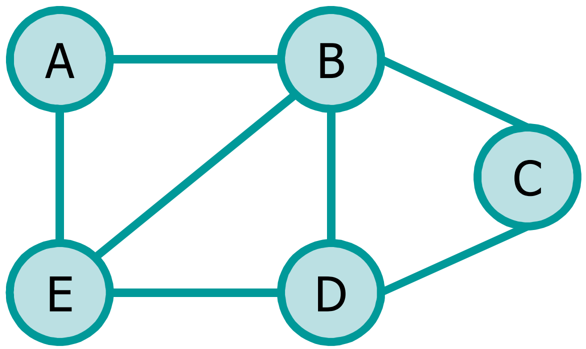
- Given a graph
  - G = (V, E)
- Its adjacency list is formed by
  - A main list representing vertices
  - A secondary list of vertices or edges for each element of the main list
- The list of lists may have different implementations
  - An array of lists
  - A true list of lists
  - A BSTs of BSTs
  - An hash-table of hash-tables

Example: Undirected graph

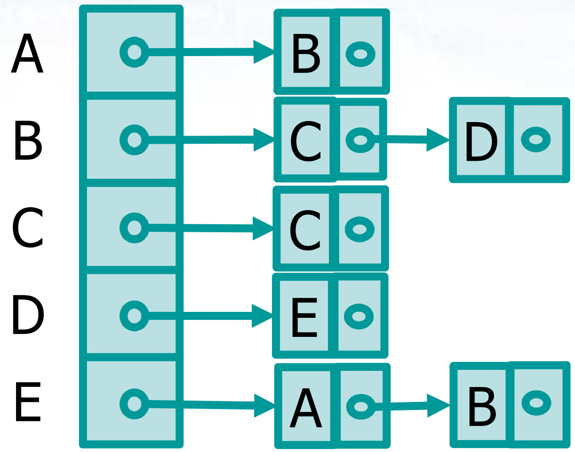


To be efficeint we need a way to avoid linear searches (pointers or efficient symbol tables)

Array of lists

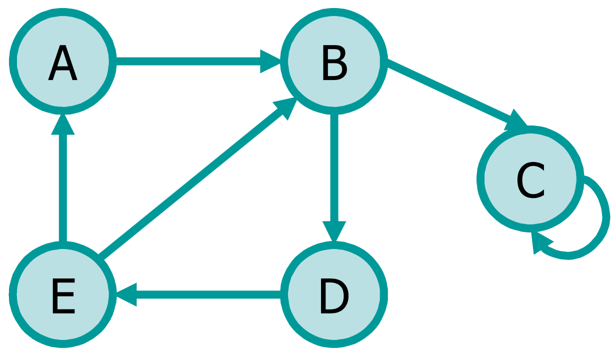


Example: Directed graph

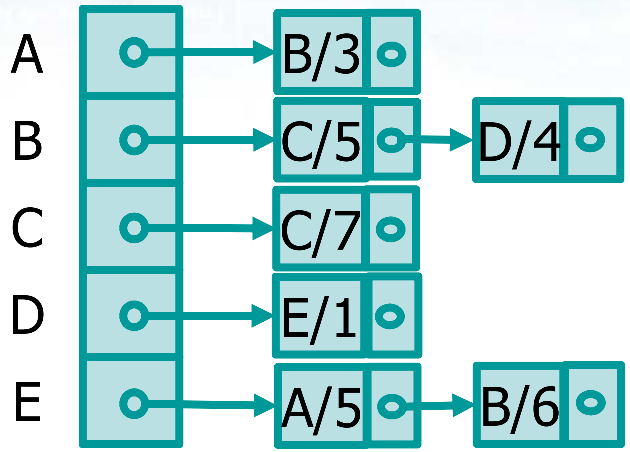


Array of list

To be efficeint we need a way to avoid linear searches (pointers or efficient symbol tables)

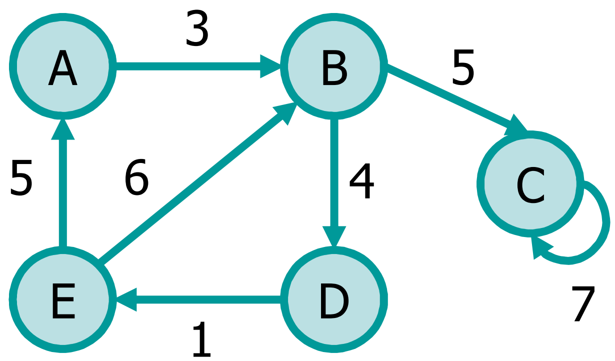


Example: Weighted directed graph



Array of list

To be efficeint we need a way to avoid linear searches (pointers or efficient symbol tables)



Graph library (with adjacency matrix)

```
#define MAX_LINE 100

enum {WHITE, GREY, BLACK};

typedef struct graph_s graph_t;
typedef struct vertex_s vertex_t;
typedef struct edge_s edge_t;

/* graph wrapper */
struct graph_s {
  vertex_t *g;
  int nv;
};
```

Enumeration types (or enum) is a user defined data type mainly used to assign names to constants easy to read and maintain
WHITE=0, GREY=1, BLACK=2

Graph Wrapper

Graph library (with adjacency list)

```
struct edge_s {
  int weight;
  vertex_t *dst;
  edge_t *next;
};

struct vertex_s {
  int id;
  int color;
  int dist;
  int disc_time;
  int endp_time;
  int scc;
  vertex_t *pred;
  edge_t *head;
  vertex_t *next;
};
```

Main list of vertices and seconday lists of edges

Nodes of the edge list

Each edge points to the destination vertex

Nodes of the vertex list

Each vertex has several attributes

Secondary list: Edge list

Main list: Vertex list

Graph library (with adjacency list)

```
graph_t *graph_load (char *filename) {
  graph_t *g;
  char line[MAX_LINE];
  int i, j, weight, dir;
  FILE *fp;
  g = (graph_t *) util_calloc (1, sizeof(graph_t));
  fp = util_fopen(filename, "r");
  fgets(line, MAX_LINE, fp);
  if (sscanf(line, "%d%d", &g->nv, &dir) != 2) {
    sscanf(line, "%d", &g->nv);
    dir = 1;
  }
  /* create initial structure for vertices */
  for (i=g->nv-1; i>=0; i--) {
    g->g = new_node (g->g, i);
  }
```

Creates main list of vertices

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, i, j, weight);
    if (dir == 0) {
      new_edge (g, j, i, weight);
    }
  }
  fclose(fp);

  return g;
}
```

Load edges in secondary lists

Graph library (with adjacency list)

Add a new vertex node into main list

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

  v = (vertex_t *)util_malloc(sizeof(vertex_t));
  v->id = id;
  v->color = WHITE;
  v->dist = INT_MAX;
  v->scc = v->disc_time = n->endp_time = -1;
  v->pred = NULL;
  v->head = NULL;
  v->next = g;
  return v;
}
```

Graph library (with adjacency list)

Add a new edge node into secondary list

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

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

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

Graph library (with adjacency list)

```
void graph_attribute_init (graph_t *g) {
  vertex_t *v;

  v = g->g;
  while (v!=NULL) {
    v->color = WHITE;
    v->dist = INT_MAX;
    v->disc_time = -1;
    v->endp_time = -1;
    v->scc = -1;
    v->pred = NULL;
    v = v->next;
  }

  return;
}
```

Graph library (with adjacency list)

It is often necessary to avoid linear searches (use pointers or efficient symbol tables)

```
vertex_t *graph_find (graph_t *g, int id) {
  vertex_t *v;

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

  return NULL;
}
```

Given the node identifier we get its matrix index and vice-versa

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

st

It is possible to use any type of symbol table (e.g., hash-tables)

Graph library (with adjacency list)

Free list of lists

```
void graph_dispose (graph_t *g) {
  vertex_t *v;
  edge_t *e;

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

Pro's and con's

- Total amount of elements in the lists
  - Undirected graphs: $2 \cdot |E|$
  - Directed graphs: $|E|$
- 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
- Extra memory is needed to represent weights in weighted graphs