BLG 223E - Recitation 5
Graphs

Ali Esad Uğur (ugura20), Enes Erdoğan (erdogane16)

Quiz Time!

Q1

```c
void func(int *ptr_y){
    printf("(3) Address of ptr_y: %p\n", &ptr_y);
    printf("(4) Address stored by ptr_y: %p\n", ptr_y);
}
int main(){
    int x = 5;
    int *ptr_x = &x;
    printf("(1) Address of ptr_x: %p\n", &ptr_x);
    printf("(2) Address stored by ptr_x: %p\n", ptr_x);
    func(ptr_x);
    return 0;
}
```

a) (1) == (3) and (2) == (4)
b) (1) == (3) and (2) != (4)
c) (1) != (3) and (2) == (4)
d) (1) != (3) and (2) != (4)

Q1

```c
void func(int *ptr_y){
    printf("(3) Address of ptr_y: %p\n", &ptr_y);
    printf("(4) Address stored by ptr_y: %p\n", ptr_y);
}
int main(){
    int x = 5;
    int *ptr_x = &x;
    printf("(1) Address of ptr_x: %p\n", &ptr_x);
    printf("(2) Address stored by ptr_x: %p\n", ptr_x);
    func(ptr_x);
    return 0;
}
```

a) (1) == (3) and (2) == (4)
b) (1) == (3) and (2) != (4)
c) (1) != (3) and (2) == (4)
d) (1) != (3) and (2) != (4)

Q2

```c
void func(int *ptr) {
    ptr = (int *)malloc(sizeof(int));
    *ptr = 10;
}
int main() {
    int *ptr = (int *)malloc(sizeof(int));
    func(ptr);
    printf("%d\n", *ptr);
    free(ptr);
    return 0;
}
```

a) 0 & leak
b) 10 & no leak
c) 10 & leak
d) Garbage value & leak

Q2

```c
void func(int *ptr) {
    ptr = (int *)malloc(sizeof(int));
    *ptr = 10;
}
int main() {
    int *ptr = (int *)malloc(sizeof(int));
    func(ptr);
    printf("%d\n", *ptr);
    free(ptr);
    return 0;
}
```

a) 0 & leak
b) 10 & no leak
c) 10 & leak
d) Garbage value & leak

Q3

```c
void func(int *ptr) {
    ptr = (int *)malloc(sizeof(int));
    *ptr = 10;
}
int main() {
    int a = 0;
    int *ptr = &a;
    func(ptr);
    printf("%d\n", *ptr);
    free(ptr);
    return 0;
}
```

a) Prints 0 & Throws an error & leak
b) Prints 10 & No error & no leak
c) Prints 0 & No error & leak
d) Prints 10 & No error & leak

Q3

```c
void func(int *ptr) {
    ptr = (int *)malloc(sizeof(int));
    *ptr = 10;
}
int main() {
    int a = 0;
    int *ptr = &a;
    func(ptr);
    printf("%d\n", *ptr);
    free(ptr);
    return 0;
}
```

```
enerd@eneserdo:~/ds24/recit_graph$ gcc quiz.c -Wall -Werror && ./a.out
0
free(): invalid pointer
Aborted (core dumped)
```

Q4

```c
void func(int *ptr_y){
    int y = 10;
    ptr_y = &y;
}
int main() {
    int x = 5;
    int *ptr_x = &x;

    func(ptr_x);
    printf("x: %d\n", *ptr_x);
    return 0;
}
```

a) 5
b) 10
c) Segmentation fault (core dumped)

Q4

```c
void func(int *ptr_y){
    int y = 10;
    ptr_y = &y;
}
int main() {
    int x = 5;
    int *ptr_x = &x;

    func(ptr_x);
    printf("x: %d\n", *ptr_x);
    return 0;
}
```

a) 5

b) 10

c) Segmentation fault (core dumped)

Some correct ways to manipulate memory inside a function

```c
void func(int **double_ptr) {
    *double_ptr = (int *)malloc(sizeof(int));
    **double_ptr = 10;
}
int main() {
    int *ptr;
    func(&ptr);
    printf("%d\n", *ptr);
    free(ptr);
    return 0;
}
```

```c
void func(int *ptr_y){
    *ptr_y = 10;
}
int main(){
    int *ptr_x = (int *)malloc(sizeof(int));
    func(ptr_x);
    printf("%d\n", *ptr_x);
    free(ptr_x);
    return 0;
}
```

```c
int* func() {
    int *ptr = (int *)malloc(sizeof(int));
    *ptr = 10;
    return ptr;
}
int main() {
    int *ptr;
    ptr = func();
    printf("%d\n", *ptr);
    free(ptr);
    return 0;
}
```

Q5

```c
int* func(){
    int arr_x[5] = {1, 2, 3, 4, 5};
    return arr_x;
}
int main() {
    int *ptr_x = func();
    printf("x: %d\n", ptr_x[0]);
    return 0;
}
```

a) 1
b) Undefined Behaviour
c) 0

Q5

```c
int* func(){
    int arr_x[5] = {1, 2, 3, 4, 5};
    return arr_x;
}
int main() {
    int *ptr_x = func();
    printf("x: %d\n", ptr_x[0]);
    return 0;
}
```

a) 1

b) Undefined Behaviour (I got seg fault in gcc, and got 1 in clang)

c) 0

What does *Undefined Behaviour (UB)* means?

UB happens when the code does something that the C standard does not define. This means the program might work differently depending on the compiler, operating system, or hardware. It is just unpredictable and unreliable.

- Division by zero
- Out-of-bounds array access
- Using an uninitialized variable
- Dereferencing a null pointer

Q6

```c
void func(int **local_ptr) {
    local_ptr = (int **)malloc(sizeof(int *));
    *local_ptr = (int *)malloc(sizeof(int));
    **local_ptr = 10;
    printf("Address 2: %p\n", &local_ptr);
}
int main() {
    int **ptr = (int **)malloc(sizeof(int *));
    printf("Address 1: %p\n", &ptr);
    func(ptr);
    printf("%d\n", **ptr);
    free(ptr);
    free(*ptr);
    return 0;
}
```

a) Address 1 and 2 are same & prints 10 & no leak
b) Address 1 and 2 are same & prints 10 & leak
c) Address 1 and 2 are different & throws error & leak
d) Address 1 and 2 are different & prints 10 & no leak

Q6

```c
void func(int **local_ptr) {
    local_ptr = (int **)malloc(sizeof(int *));
    *local_ptr = (int *)malloc(sizeof(int));
    **local_ptr = 10;
    printf("Address 2: %p\n", &local_ptr);
}
int main() {
    int **ptr = (int **)malloc(sizeof(int *));
    printf("Address 1: %p\n", &ptr);
    func(ptr);
    printf("%d\n", **ptr);
    free(ptr);
    free(*ptr);
    return 0;
}
```

```
enerd@eneserdo:~/ds24/recit_graph$ gcc quiz.c -Wall -Werror && ./a.out
Address 2: 0x7fff397d13a0
Address 1: 0x7fff397d1378
Segmentation fault (core dumped)
```

Q7

```c
void func(int *func_ptr) {
    printf("(3) Size of func_ptr: %lu\n", sizeof(func_ptr));
}
int main() {
    int arr[5] = {1, 2, 3, 4, 5};

    int *ptr = arr;
    printf("(1) Size of arr: %lu\n", sizeof(arr));
    printf("(2) Size of ptr: %lu\n", sizeof(ptr));
    func(ptr);
}
```

a) (1) == (2) and (2) == (3)
b) (1) == (2) and (2) != (3)
c) (1) != (2) and (2) == (3)

Q7

```c
void func(int *func_ptr) {
    printf("(3) Size of func_ptr: %lu\n", sizeof(func_ptr));
}
int main() {
    int arr[5] = {1, 2, 3, 4, 5};

    int *ptr = arr;
    printf("(1) Size of arr: %lu\n", sizeof(arr));
    printf("(2) Size of ptr: %lu\n", sizeof(ptr));
    func(ptr);
}
```

a) (1) == (2) and (2) == (3)
b) (1) == (2) and (2) != (3)
c) (1) != (2) and (2) == (3)

Quick Review: Graphs

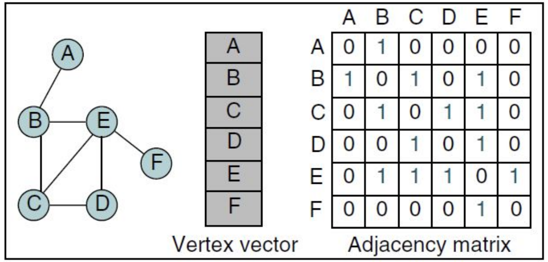
As opposed to previous data structures, a node may have multiple predecessors as well as multiple successors

Graphs are very useful structures to represent some complex real-world problems:

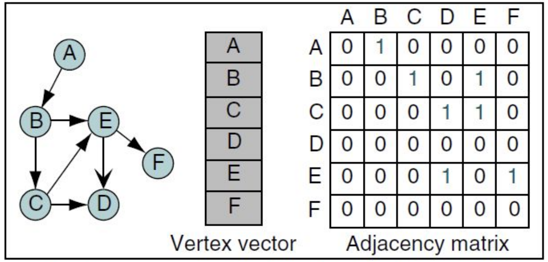
- Social networks
- Transportation networks
- Web pages
- Electric grids

Graph Storage Structures

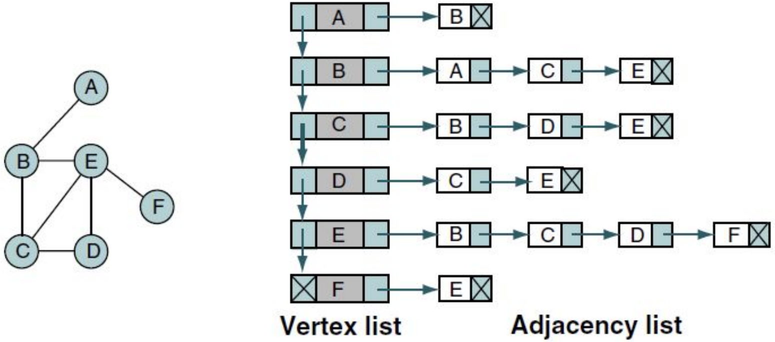
To represent a graph, we need to store two sets: **vertices** and **edges/arcs**.



(a) Adjacency matrix for nondirected graph

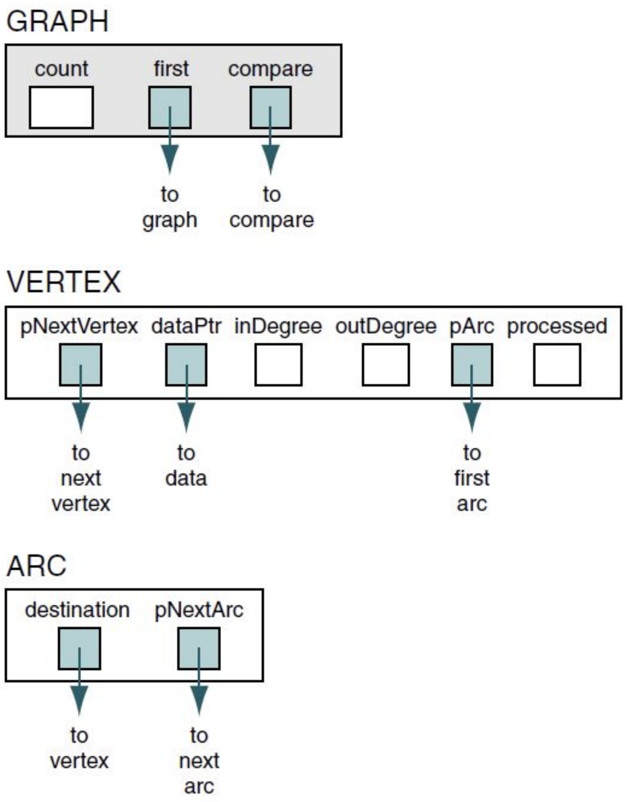


(b) Adjacency matrix for directed graph



Major limitation of adj matrix: The number of vertices must be known in advance, Otherwise, we need to do a lot of copying as size increases.

How the reference book implements the graph




```
typedef struct
   {
    int              count;
    struct vertex* first;
    int (*compare)
           (void* argu1,
            void* argu2);
   } GRAPH;

typedef struct vertex
 {
  struct vertex*  pNextVertex;
  void*           dataPtr;
  int             inDegree;
  int             outDegree;
  short           processed;
  struct arc*     pArc;
 } VERTEX;

typedef struct arc
 {
  struct vertex*  destination;
  struct arc*     pNextArc;
 } ARC;
```

Our implementation

```c
typedef struct Edge
{
    struct Vertex *dst;
    int weight;
} Edge;

typedef struct Vertex
{
    void *data;
    struct Edge **adjacents;
    int num_adj;
    int capacity;
    int processed;
} Vertex;

typedef struct Graph
{
    Vertex **vertices;
    int num_vertices;
    int capacity;
    int (*compare)(void *, void *);
} Graph;
```

What are the advantages and disadvantages?

Our implementation

```c
typedef struct Edge
{
    struct Vertex *dst;
    int weight;
} Edge;

typedef struct Vertex
{
    void *data;
    struct Edge **adjacents;
    int num_adj;
    int capacity;
    int processed;
} Vertex;

typedef struct Graph
{
    Vertex **vertices;
    int num_vertices;
    int capacity;
    int (*compare)(void *, void *);
} Graph;
```

It is definitely way simpler, but if there are frequent additions/deletions, there might be many copying and big allocations (depending on the capacity increment policy)

Each implementation has its own pros and cons!

Make sure you understand both of them

Fundamental Operations

- Creating a graph for a generic data type i.e. void*
- Adding or removing a vertex
- Adding or removing an edge between the vertices
- Destroying a graph
- BFS/DFS

Problem 1: Maze with DFS

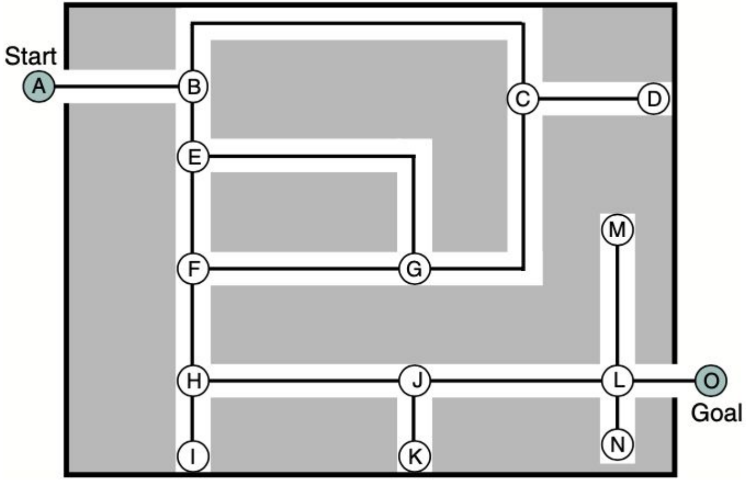
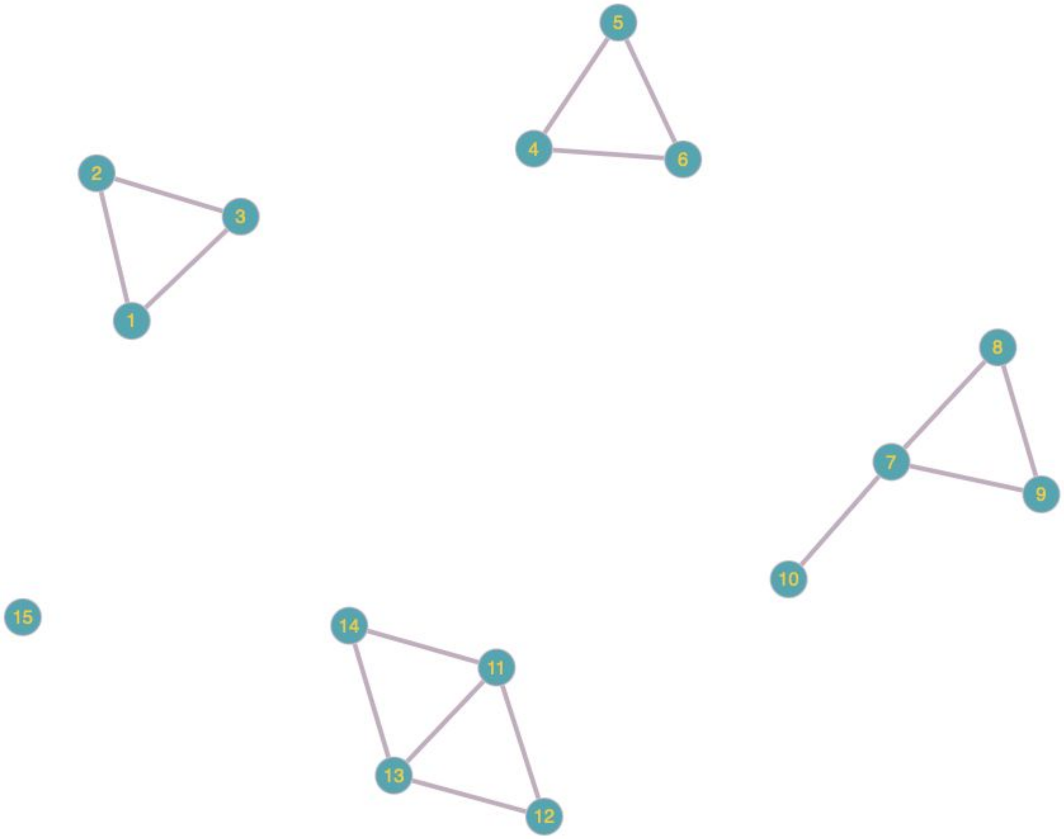


FIGURE 11-27 Graph Maze for Project 27

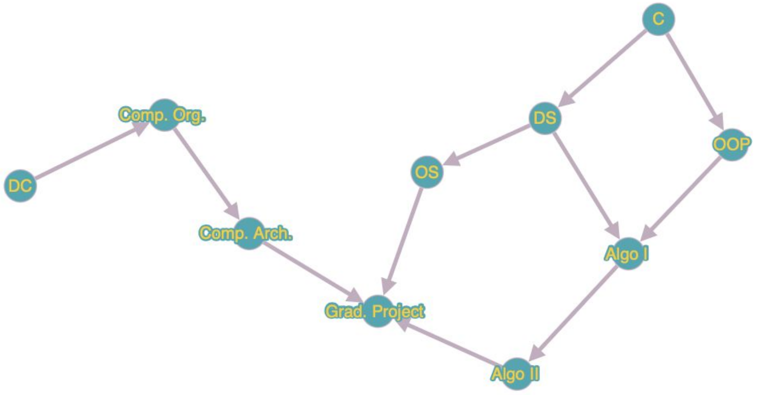
Write a program that simulates a mouse's movement through the maze, using a graph and a depth-first traversal. When the program is complete, print the path through the maze.

Problem 2: Counting Connected Components with BFS



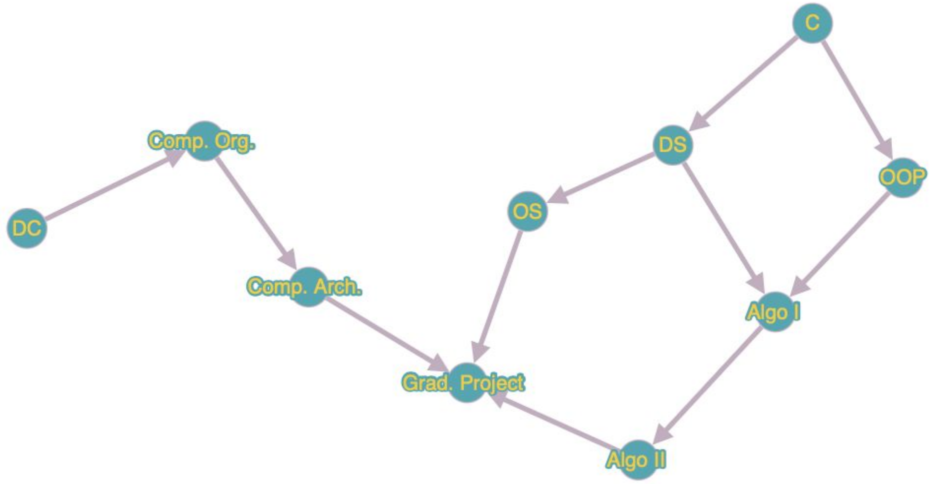
Problem 3: Topological Sort with Courses

- Directed Acyclic Graph (DAG)
- A linear order of vertices such that for every directed edge u-v, vertex u comes before vertex v in the ordering
- Implement topologicalSort function



Problem 3: Topological Sort

DC -> CompOrg -> CompArch-> C -> OOP -> DS -> OS -> Algo I -> Algo II -> Grad. Project



Problem 4: Shortest Path - Dijkstra's Algorithm

- Start from an initial vertex and find the shortest paths to all other vertices from the initial one

Check this for step by step visualizations: https://www.cs.usfca.edu/~galles/visualization/Dijkstra.html