INF01056
Aula 05/06
Graphs 1

Prof. João Comba

Baseado no Livro Programming Challenges and Competitive Programming

Graph Types $G = (V, E)$

- Vertices/Nodes
- Edges
- Un/Weighted
- Un/Directed
- In/Out Degree
- Self-Loop/Multiple Edges (Multigraph) vs Simple Graph
- Sparse/Dense
- Path, Cycle
- Isolated, Reachable

- (Strongly) Connected Component
- Sub Graph
- Complete Graph
- Directed Acyclic Graph
- Tree/Forest
- Euler/Hamiltonian Path/Cycle
- Bipartite Graph

Data Structures

- Adjacency Matrix
- Adjacency Lists in Lists
- Adjacency Lists in Matrices
- Table of Edges

Breadth-First Search

```
bool processed[MAXV];/* which vertices have been processed */
bool discovered[MAXV];  /* which vertices have been found */
int parent[MAXV];  /* discovery relation */

bool finished = FALSE; /* if true, cut off search immediately */

initialize_search(graph *g) {

  int i; /* counter */

  for (i=1; i<=g->nvertices; i++) {
    processed[i] = discovered[i] = FALSE;
    parent[i] = -1;
  }
}
```

Breadth-First Search

```
bfs(graph *g, int start) {
   queue q;        /* queue of vertices to visit */
   int v;          /* current vertex */
   int i;          /* counter */

   init_queue(&q);
   enqueue(&q,start);
   discovered[start] = TRUE;

   while (empty(&q) == FALSE) {
      v = dequeue(&q);
      process_vertex(v);
      processed[v] = TRUE;
      for (i=0; i<g->degree[v]; i++)
          if (valid_edge(g->edges[v][i]) == TRUE) {
             if (discovered[g->edges[v][i]] == FALSE) {
                 enqueue(&q,g->edges[v][i]);
                 discovered[g->edges[v][i]] = TRUE;
                 parent[g->edges[v][i]] = v;
             }
             if (processed[g->edges[v][i]] == FALSE)
                 process_edge(v,g->edges[v][i]);
          }
   }
}
```

Breadth-First Search

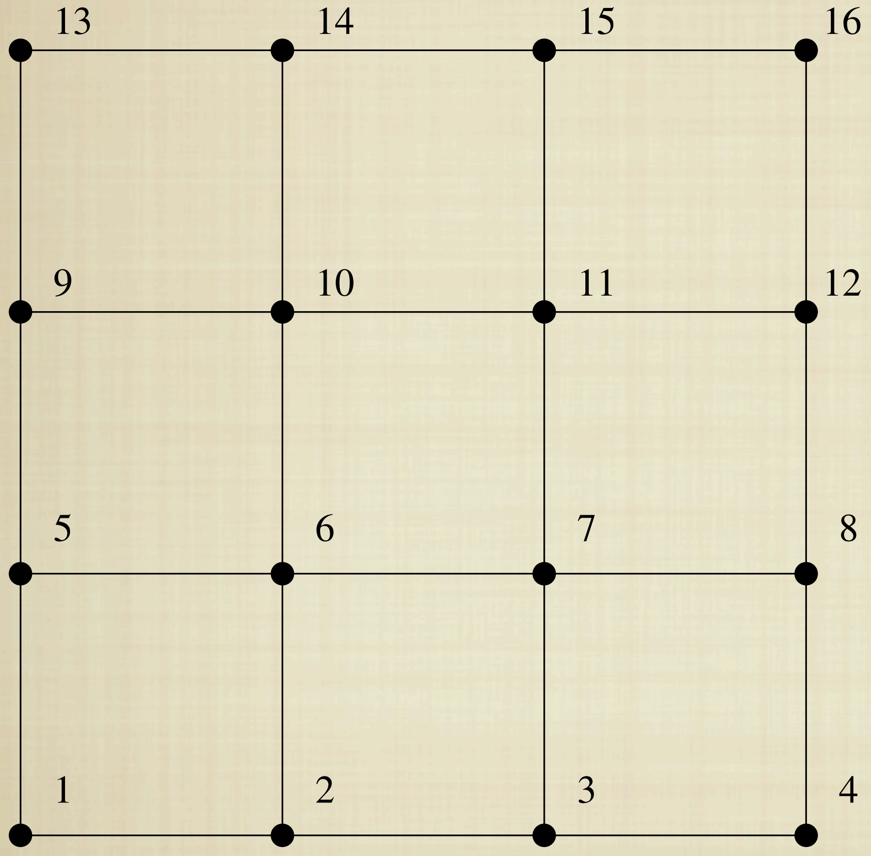
```
extern bool processed[];   /* which vertices have been processed */
extern bool discovered[];  /* which vertices have been found */
extern int parent[];    /* discovery relation */

process_vertex(int v) {
   printf("processed vertex %d\n",v);
}

process_edge(int x, int y) {
        printf("processed edge (%d,%d)\n",x,y);
}

bool valid_edge(int e) {
        return (TRUE);
}
```

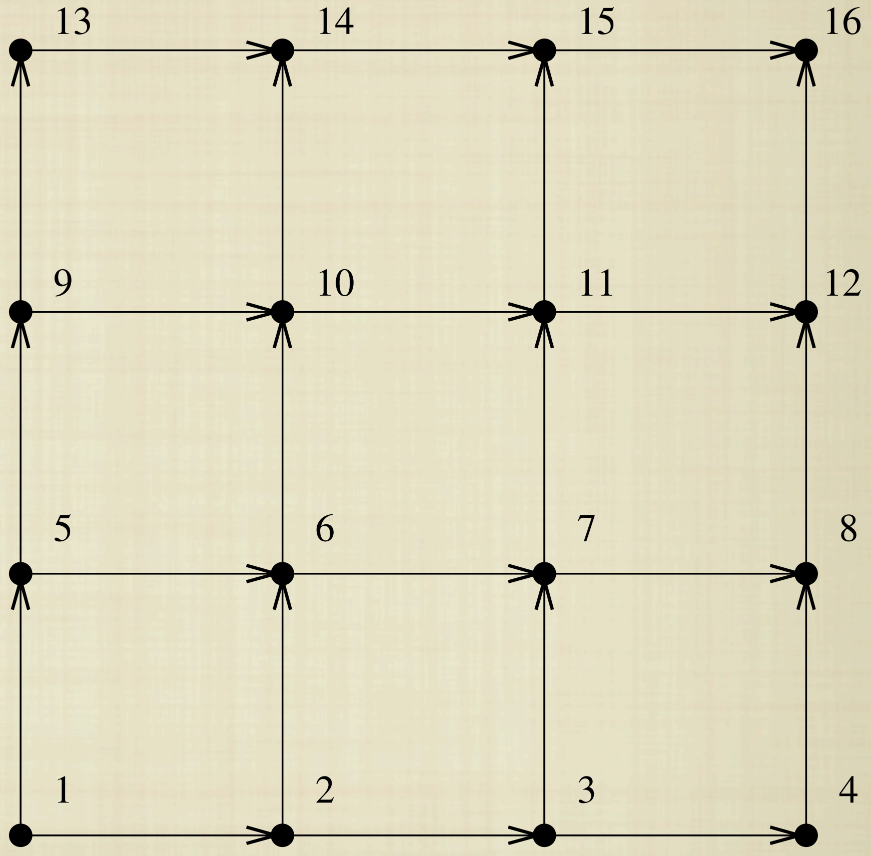
Breadth-First Search

```
main() {
  graph g;
  int i;

  read_graph(&g,FALSE);
  print_graph(&g);
  initialize_search(&g);
  bfs(&g,1);
  for (i=1; i<=g.nvertices; i++)
    printf(" %d",parent[i]);
  printf("\n");

  for (i=1; i<=g.nvertices; i++)
    find_path(1,i,parent);
  printf("\n");
}
```

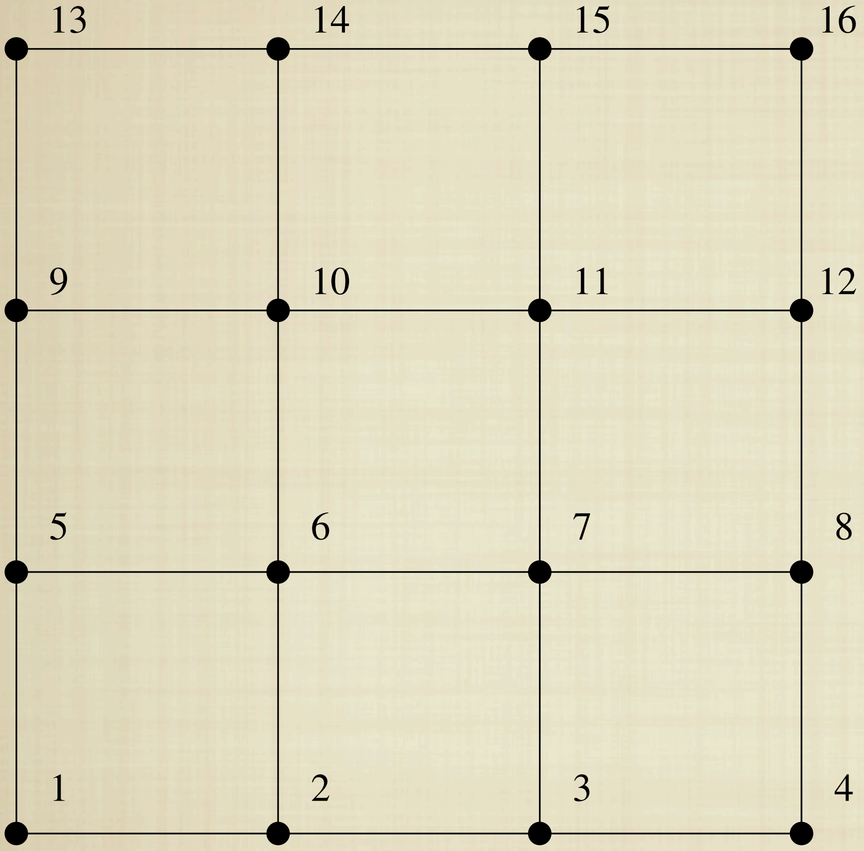
```
find_path(int start, int end, int parents[])
{
        if ((start == end) || (end == -1))
                printf("\n%d",start);
        else {
                find_path(start,parents[end],parents);
                printf(" %d",end);
        }
}
```

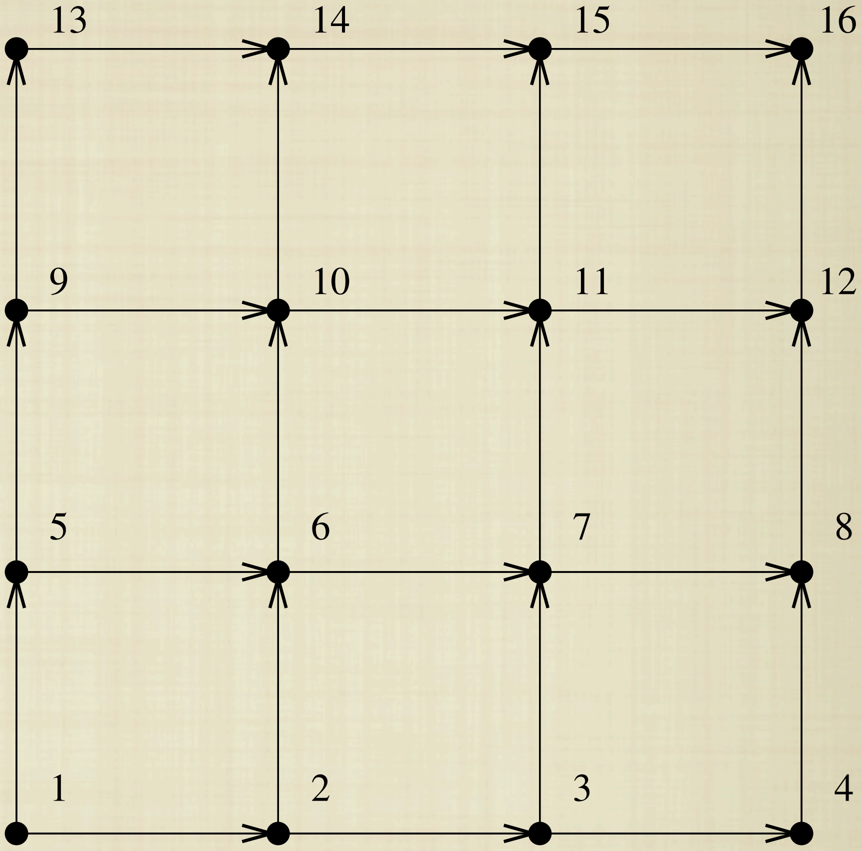
Breadth-First Search





Breadth-First Search





vertex	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
parent	-1	1	2	3	1	2	3	4	5	6	7	8	9	10	11	12

Depth-first search

```
typedef pair<int, int> ii; typedef vector<ii> vi;

void dfs(int u) { // DFS for normal usage
  printf(" %d", u); // this vertex is visited
  dfs_num[u] = DFS_BLACK; // mark as visited
  for (int j = 0; j < (int)AdjList[u].size; j++) {
    ii v = AdjList[u][j]; // try all neighbors v of vertex u
    if (dfs_num[v.first] == DFS_WHITE) // avoid cycle
      dfs(v.first); // v is a (neighbor, weight) pair
  }
}
```

Breadth-first search (using STL)

```
map<int, int> dist; dist[source] = 0;
queue<int> q; q.push(source); // start from source

while (!q.empty()) {
  int u = q.front(); q.pop(); // queue: layer by layer!
  for (int j = 0; j < (int)AdjList[u].size(); j++) {
    ii v = AdjList[u][j]; // for each neighbours of u
    if (!dist.count(v.first)) {
      dist[v.first] = dist[u] + 1; // unvisited + reachable
      q.push(v.first); // enqueue v.first for next steps
    }
  }
}
```

Depth-First Search

```
dfs(graph *g, int v) {
   int i;               /* counter */
   int y;               /* successor vertex */

   if (finished) return;        /* allow for search termination */

   discovered[v] = TRUE;
   process_vertex(v);

   for (i=0; i<g->degree[v]; i++) {
      y = g->edges[v][i];
      if (valid_edge(g->edges[v][i]) == TRUE) {
         if (discovered[y] == FALSE) {
            parent[y] = v;
            dfs(g,y);
         } else
            if (processed[y] == FALSE)
              process_edge(v,y);
      }
      if (finished) return;
   }
   processed[v] = TRUE;
}
```

Finding Cycles

```
extern bool processed[];   /* which vertices have been processed */
extern bool discovered[];  /* which vertices have been found */
extern int parent[];    /* discovery relation */

extern bool finished;   /* flag for early search cutoff */


process_vertex(int v) { }

process_edge(int x, int y) {
   if (parent[x] != y) {/* found back edge! */
     printf("Cycle from %d to %d:",y,x);
                       find_path(y,x,parent);
     printf("\n\n");
     finished = TRUE;
  }
}

bool valid_edge(int e) {
        return (TRUE);
}
```

Connected Components

```
process_vertex(int v) {
   printf(" %d",v);
}

process_edge(int x, int y) { }

bool valid_edge(int e) {
        return (TRUE);
}

connected_components(graph *g) {
   int c;              /* component number */
   int i;              /* counter */

   initialize_search(g);

   c = 0;
   for (i=1; i<=g->nvertices; i++)
      if (discovered[i] == FALSE) {
         c = c+1;
         printf("Component %d:",c);
                  dfs(g,i);
         printf("\n");
      }
}
```

Connected Components

```
int numComp = 0;
dfs_num.assign(V, DFS_WHITE);
REP (i, 0, V - 1) // for each vertex i in [0..V-1]
  if (dfs_num[i] == DFS_WHITE) { // if not visited yet
    printf("Component %d, visit", ++numComp);
    dfs(i); // one component found
    printf("\n");
  }
printf("There are %d connected components\n", numComp);
```

Topological Sort

```
#include "graph.h"
#include "queue.h"

compute_indegrees(graph *g, int in[]) {
  int i,j;      /* counters */

  for (i=1; i<=g->nvertices; i++) in[i] = 0;

  for (i=1; i<=g->nvertices; i++)
    for (j=0; j<g->degree[i]; j++) in[g->edges[i][j]]++;
}
```

Topological Sort

```
topsort(graph *g, int sorted[]) {
   int indegree[MAXV];    /* indegree of each vertex */
   queue zeroin;       /* vertices of indegree 0 */
   int x, y;        /* current and next vertex */
   int i, j;        /* counters */

   compute_indegrees(g,indegree);
   init_queue(&zeroin);
   for (i=1; i<=g->nvertices; i++)
      if (indegree[i] == 0) enqueue(&zeroin,i);

   j=0;
   while (empty(&zeroin) == FALSE) {
      j = j+1;
      x = dequeue(&zeroin);
      sorted[j] = x;
      for (i=0; i<g->degree[x]; i++) {
         y = g->edges[x][i];
         indegree[y] --;
         if (indegree[y] == 0) enqueue(&zeroin,y);
      }
   }
   if (j != g->nvertices)
      printf("Not a DAG -- only %d vertices found\n",j);
}
```