GAME2001 Data Structures and Algorithms
Fall 2020

Week 14

Graphs

Graphs

- a tree is a type of graph
- nodes of the data structure represent much more abstract objects used for solving different types of problems
- shaped by the data, and not the algorithms
- nodes are shaped to represent a physical or abstract set of objects

Graphs

- Similarities between graphs and trees:
  - nodes encapsulate objects
  - have edges
  - nodes can have multiple other nodes related to them
  - allow node traversal
    - which is often used by algorithms

Graphs

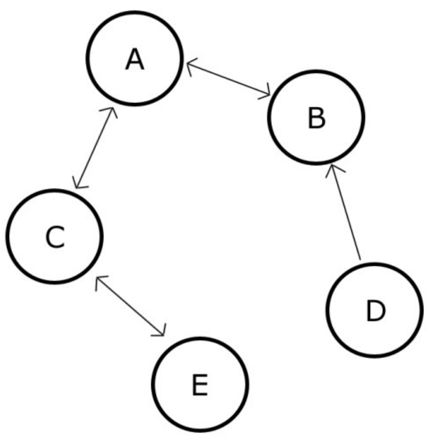
- Differences between trees and graphs:
  - The relationship of each node is more abstract in a graph than in a tree
  - In graphs nodes are called vertices
  - Vertices that are connected are called adjacent vertices
  - Graphs have no keys
    - binary trees use as a way of structuring the tree
  - The edges of a graph's node can go one way (directed) or both ways (nondirected)

Graphs

- The edges of a graph are often represented as an adjacency matrix (adjacency list)
  - the tree uses references to objects or array indexes
- The nodes of a graph can be unweighted or weighted
- Different vertices can have different numbers of children without limit
- The relationship between vertices is not a parent-child relationship
  - much more abstract and often takes on a completely different form

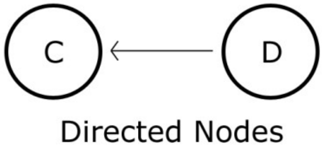
Graphs

- Connected Graph
  - there is a path from all vertices to all other vertices, either directly or indirectly
- Path
  - a sequence of edges that can be taken to get to a destination vertex from a starting vertex



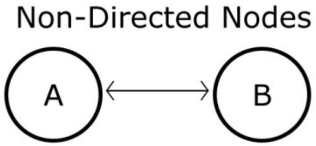
Graphs

- Directed Graph
  - Edges have a direction
  - Edges might go from one vertex to another but not in reverse



Graphs

- Nondirected Graph
  - edges do not have a specific direction
  - you can travel back and forth from connected vertices



Graphs

- Vertices of a graph
  - are its nodes
  - an object that encapsulates what is being represented abstractly
  - can have various properties like weight

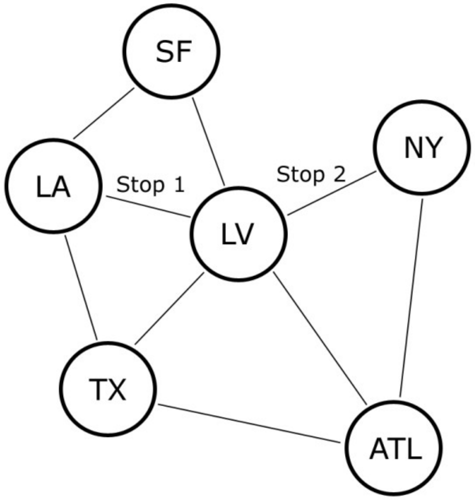
Graphs

- Adjacency Matrix
  - Represent edges from one vertex to another
  - A 2D array (size = N * N)

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

Graphs

- Searching
  - done to find which vertices can be reached from a starting vertex by following a path along the edges

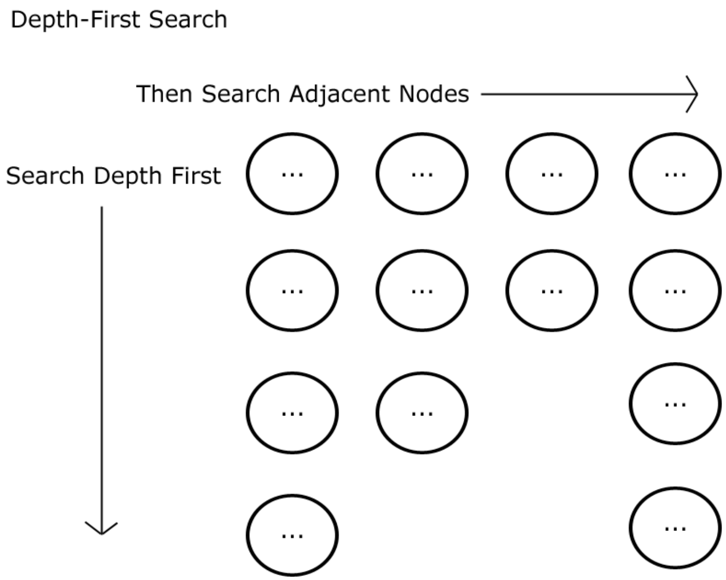


Graphs

- Searching
  - start at a vertex and visit every vertex between it and a destination vertex
  - finds any paths that exist between two vertices
    - If a path exists, the path data, which is the order of the edges needed to get from start to finish, is built and stored for some meaningful purpose

Graphs

- Depth-First Search
  - uses a stack data structure to start at a starting vertex and move until it reaches the destination



Graphs

1) Choose a starting vertex and make it the current vertex
2) Push the current vertex onto the stack and mark it with a flag that tells us it was checked
3) If the current vertex is the destination, then the code is done; otherwise, continue
4) Visit the first adjacent vertex after the current one that was not marked as visited and make it the current vertex
5) Repeat steps 2 through 4 until the algorithm can't go any further along one path
6) If the destination vertex was not found, pop the current vertex off the stack and visit the next adjacent vertex that is not marked as visited
7) Continue steps 2 through 6 until all vertices have been marked visited, which means a path was not found, or until the destination vertex is reached

Graphs

```
#include <vector>
#include <stack>
#include <cassert>

using namespace std;

template<typename T>
class GraphVertex
{
public:
    GraphVertex(T node) : m_node(node) { }

    T GetNode() { return m_node; }

private:
    T m_node;
};
```

Graphs

```
template<typename T>
class Graph
{
public:
    Graph(int numVerts)
        : m_maxVerts(numVerts),
        m_adjMatrix(NULL)
    {
        assert(numVerts > 0);

        m_vertices.reserve(m_maxVerts);

        m_adjMatrix = new char*[m_maxVerts];
        assert(m_adjMatrix != NULL);

        m_vertVisits = new char[m_maxVerts];
        assert(m_vertVisits != NULL);

        memset(m_vertVisits, 0, m_maxVerts);

        for(int i = 0; i < m_maxVerts; i++)
        {
            m_adjMatrix[i] = new char[m_maxVerts];
            assert(m_adjMatrix[i] != NULL);

            memset(m_adjMatrix[i], 0, m_maxVerts);
        }
    }
```

```
    ~Graph()
    {
        if(m_adjMatrix != NULL)
        {
            for(int i = 0; i < m_maxVerts; i++)
            {
                if(m_adjMatrix[i] != NULL)
                {
                    delete[] m_adjMatrix[i];
                    m_adjMatrix[i] = NULL;
                }
            }

            delete[] m_adjMatrix;
            m_adjMatrix = NULL;
        }

        if(m_vertVisits != NULL)
        {
            delete[] m_vertVisits;
            m_vertVisits = NULL;
        }
    }
```

Graphs

```
bool push(T node)
{
    if((int)m_vertices.size() >= m_maxVerts)
        return false;

    m_vertices.push_back(GraphVertex<T>(node));
    return true;
}

void attachEdge(int index1, int index2)
{
    assert(m_adjMatrix != NULL);

    m_adjMatrix[index1][index2] = 1;
    m_adjMatrix[index2][index1] = 1;
}

void attachDirectedEdge(int index1, int index2)
{
    assert(m_adjMatrix != NULL);

    m_adjMatrix[index1][index2] = 1;
}
```

Graphs

```
int getNextUnvisitedVertex(int index)
{
    assert(m_adjMatrix != NULL);
    assert(m_vertVisits != NULL);

    for(int i = 0; i < (int)m_vertices.size(); i++)
    {
        if(m_adjMatrix[index][i] == 1 &&
           m_vertVisits[i] == 0)
        {
            return i;
        }
    }

    return -1;
}
```

Graphs

```
bool DepthFirstSearch(int startIndex, int endIndex)
{
    assert(m_adjMatrix != NULL);
    assert(m_vertVisits != NULL);

    m_vertVisits[startIndex] = 1;

    // FOR OUTPUT PURPOSES OF THE DEMOS.
    cout << m_vertices[startIndex].GetNode();

    stack<int> searchStack;
    int vert = 0;

    searchStack.push(startIndex);

    while(searchStack.empty() != true)
    {
        vert = getNextUnvisitedVertex(searchStack.top());

        if(vert == -1)
        {
            searchStack.pop();
        }
        else
        {
            m_vertVisits[vert] = 1;

            // FOR OUTPUT PURPOSES OF THE DEMOS.
            cout << m_vertices[vert].GetNode();

            searchStack.push(vert);
        }

        if(vert == endIndex)
        {
            memset(m_vertVisits, 0, m_maxVerts);
            return true;
        }
    }

    memset(m_vertVisits, 0, m_maxVerts);
    return false;
}
```

Graphs

```
private:
    vector<GraphVertex<T> > m_vertices;
    int m_maxVerts;

    char **m_adjMatrix;
    char *m_vertVisits;
};
```

Depth First Search Example

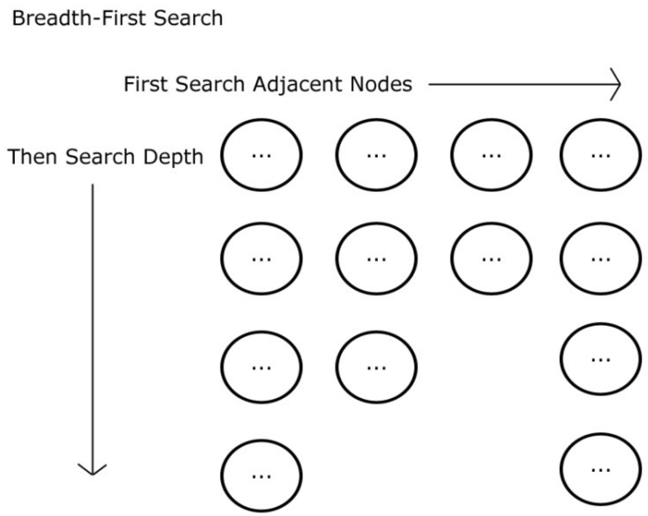
```cpp
#include <iostream>
#include "Graphs.h"

using namespace std;

int main(int args, char **argc)
{
   cout << "Graphs - Depth First Search" << endl;
   cout << endl;

   Graph<char> demoGraph(6);

   demoGraph.push('A');
   demoGraph.push('B');
   demoGraph.push('C');
   demoGraph.push('D');
   demoGraph.push('E');
   demoGraph.push('F');
```

```
Graphs - Depth First Search
DepthFirstSearch Nodes Visited: ACFD
Path from A to D found!
```

```cpp
   // Attach A to C and C to A.
   demoGraph.attachEdge(0, 2);

   // Attach A to D and D to A.
   demoGraph.attachEdge(0, 3);

   // Attach B to E and E to B.
   demoGraph.attachEdge(1, 4);

   // Attach C to F and F to C.
   demoGraph.attachEdge(2, 5);

   // Perform depth first search for a path from A to D.
   cout << "DepthFirstSearch Nodes Visited: ";

   int result = demoGraph.DepthFirstSearch(0, 3);
   cout << endl << endl;

   if(result == 1)
      cout << "Path from A to D found!";
   else
      cout << "Path from A to D NOT found!";

   cout << endl << endl;

   return 1;
}
```

Graphs

- Breadth First Search
  - uses a queue
  - all adjacent vertices to the current vertex are checked before the algorithm moves forward



Graphs

1) Push the starting vertex into a queue and then start a loop that will execute while the queue is not empty
2) Once inside the loop, pop a vertex from the queue and make it the current vertex
3) Place all unchecked vertices adjacent to the current vertex onto the queue and mark them as checked
4) If there are no more vertices adjacent to the current vertex, check if the current vertex is the destination
5) If the destination is found, the algorithm is done
6) If the algorithm did not find the destination, repeat steps 2 through 5

Graphs

```
#include <queue>

template<typename T>
class Graph
{
public:
    bool BreadthFirstSearch(int startIndex, int endIndex)
    {
        assert(m_adjMatrix != NULL);
        assert(m_vertVisits != NULL);

        m_vertVisits[startIndex] = 1;

        // FOR OUTPUT PURPOSES OF THE DEMOS.
        cout << m_vertices[startIndex].GetNode();

        queue<int> searchQueue;
        int vert1 = 0, vert2 = 0;

        searchQueue.push(startIndex);

        while(searchQueue.empty() != true)
        {
            vert1 = searchQueue.front();
            searchQueue.pop();

            if(vert1 == endIndex)
            {
                memset(m_vertVisits, 0, m_maxVerts);
                return true;
            }

            while((vert2 = getNextUnvisitedVertex(vert1)) != -1)
            {
                m_vertVisits[vert2] = 1;

                // FOR OUTPUT PURPOSES OF THE DEMOS.
                cout << m_vertices[vert2].GetNode();

                searchQueue.push(vert2);
            }
        }

        memset(m_vertVisits, 0, m_maxVerts);

        return false;
    }
};
```

Breadth First Search Example

```cpp
#include <iostream>
#include "Graphs.h"

using namespace std;

int main(int args, char **argc)
{
    cout << "Graphs - Breadth First Search" << endl;
    cout << endl;

    Graph<char> demoGraph(6);

    demoGraph.push('A');
    demoGraph.push('B');
    demoGraph.push('C');
    demoGraph.push('D');
    demoGraph.push('E');
    demoGraph.push('F');

    // Attach A to C and C to A.
    demoGraph.attachEdge(0, 2);

    // Attach A to D and D to A.
    demoGraph.attachEdge(0, 3);

    // Attach B to E and E to B.
    demoGraph.attachEdge(1, 4);

    // Attach C to F and F to C.
    demoGraph.attachEdge(2, 5);

    // Perform depth first search for a path from A to D.
    cout << "BreadthFirstSearch Nodes Visited: ";

    int result = demoGraph.BreadthFirstSearch(0, 3);
    cout << endl << endl;

    if(result == 1)
        cout << "Path from A to D found!";
    else
        cout << "Path from A to D NOT found!";

    cout << endl << endl;

    return 1;
}
```

```
Graphs - Breadth First Search
BreadthFirstSearch Nodes Visited: ACDF
Path from A to D found!
```