Computer Science II

Professor Tom Ellman
Lecture 22
Maze Problem
Maze and Graph Representation
Graph Representation of Maze
Graph

- A pair of sets: (V,E).
- V is a set of vertices: \{v_1,v_2,\ldots,v_n\}
- E is a set of edges: \{(v_i,v_j), \ldots\}
  - Un-Directed Graph:
    - Edges are un-directed.
    - \( (v_i,v_j) = (v_j,v_i) \).
  - Directed Graph:
    - Edges are directed.
    - \( (v_i,v_j) \neq (v_j,v_i) \).
Graphs and Relations

• A graph is a data structure that represents a relation.

• Examples of Relations:
  – Parent, Child, Sibling, Cousin.
  – Roommate, Classmate.
  – Older, Younger.
  – Loves, Hates.
Sibling Relation (Symmetric)

George W. Bush

Jeb Bush

Al Gore

Hillary Clinton
Loves Relation (Asymmetric)

Romeo → Juliet

Pip → Estelle
Maze and Graph Representation
Graph Representation of Maze
Un-Directed Graph Representation of Maze

- \( V = \{v_0, v_1, \ldots, v_{29}\} \)
- \( E = \{(v_0, v_1), (v_0, v_{26}), (v_1, v_2), (v_1, v_{13}), (v_2, v_3), (v_2, v_{11}), (v_3, v_{19}), (v_4, v_9), (v_5, v_{10}), (v_6, v_7), (v_7, v_8), (v_7, v_{12}), (v_8, v_{18}), (v_9, v_{10}), (v_9, v_{14}), (v_{11}, v_{12}), (v_{11}, v_{15}), (v_{13}, v_{14}), (v_{14}, v_{15}), (v_{15}, v_{16}), (v_{16}, v_{17}), (v_{16}, v_{23}), (v_{18}, v_{19}), (v_{18}, v_{24}), (v_{19}, v_{25}), (v_{20}, v_{21}), (v_{21}, v_{27}), (v_{22}, v_{23}), (v_{22}, v_{28}), (v_{23}, v_{24}), (v_{26}, v_{27}), (v_{28}, v_{29})\} \).
Adjacency Lists

• Each vertex $v$ is associated with a list of its neighbors.

• Each edge $(u,v)$ appears on two lists:
  – Vertex $v$ is a neighbor of $u$.
  – Vertex $u$ is a neighbor of $v$. 
Representing Graphs: Adjacency Lists
Adjacency Matrix

• Boolean matrix has one row and one column for each vertex.
• Matrix entry in row $r$, column $c$ is
  – $true$ if $(r, c)$ is an edge.
  – $false$ otherwise.
## Representing Graphs: Adjacency Matrix

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>...</th>
<th>26</th>
<th>27</th>
<th>28</th>
<th>29</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>T</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>T</td>
<td>T</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>T</td>
<td>T</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td>T</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>T</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>26</td>
<td>T</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>T</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>27</td>
<td></td>
<td>T</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>T</td>
<td></td>
<td></td>
</tr>
<tr>
<td>28</td>
<td></td>
<td></td>
<td>T</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>T</td>
<td></td>
</tr>
<tr>
<td>29</td>
<td></td>
<td></td>
<td></td>
<td>T</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>T</td>
</tr>
</tbody>
</table>
Depth-First Search (DFS)

• Explores the current path to completion before backing up and trying another path.

• Guaranteed to find a solution path, if there is one.

• Might not find an optimal (shortest) path.
Recursive Depth-First Search of a Graph

depthFirstSearchRecursive(vertex)
// Begin search by calling this procedure
// on the start vertex.
If (vertex is the goal vertex)
    then Return the path to vertex.
else a. Mark the vertex as “visited”.
b. Call this procedure recursively on each
    unvisited neighbor of vertex, and return
    the resulting path, if one was found.
c. Return null (since no path was found).
Recursive Depth-First Search of a Graph

ArrayList<Integer> depthFirstSearch(Integer vertex) {
    Node startNode = new Node(vertex,null);
    return dfsHelper(startNode);
}

ArrayList<Integer> dfsHelper(Node node) {
    Integer v = node.vertex;
    if (goal(v)) return path(node);
    else {
        visited[v] = true;
        for (Integer w:neighbors.get(v))
            if (!visited[w]) {
                Node newNode = new Node(w,node);
                ArrayList<Integer> solution = dfsHelper(newNode);
                if (solution!=null) return solution;
            }
    }
    return null;
}

Project GraphSearch
class Node {
    Integer vertex;
    Node parent;

    Node(Integer vertex, Node parent) {
        this.vertex = vertex;
        this.parent = parent;
    }
}

A Node object holds a vertex and a pointer to the parent node from which this node was generated.
Nodes Record Path
Back to Start Vertex
```java
ArrayList<Integer> path(Node node) {
    ArrayList<Integer> thePath;
    if (node.parent != null) {
        thePath = path(node.parent);
    } else {
        thePath = new ArrayList<Integer>();
        thePath.add(node.vertex);
    }
    return thePath;
}
```

A method to return the path from the start vertex to a given node. Notice that the correct ordering depends on the fact that the `ArrayList` class `add` method puts the added item at the end of the list.
Iterative Depth-First Search of a Graph

depthFirstSearchIterative(vertex)
// Begin search by calling this procedure
// on the start vertex.
If (vertex is the goal vertex)
    then Return the path to vertex.
else 1. Let s be an empty stack.
    2. Push vertex onto s.
    3. Mark vertex as “visited”.
    4. While (s is not empty)
        a. Let v = Pop(s).
        b. For (w in Neighbors(v))
            If (w has not been visited)
                then If (w is the goal vertex)
                    then Return the path to w.
                else 1. Mark w as “visited”.
                    2. Push w onto s.
Iterative Depth-First Search of a Graph

```java
ArrayList<Integer> depthFirstSearchIterative(Integer vertex) {
    Node startNode = new Node(vertex, null);
    if (goal(vertex)) return path(startNode);
    Stack<Node> stack = new Stack<Node>();
    stack.push(startNode);
    visited[vertex] = true;
    while (!stack.isEmpty()) {
        Node node = stack.pop();
        Integer v = node.vertex;
        for (Integer w : neighbors.get(v))
            if (!visited[w]) {
                Node newNode = new Node(w, node);
                if (goal(w)) return path(newNode);
                else {
                    visited[w] = true;
                    stack.push(newNode);
                }
            }
    }
    return null;
}
```

Project GraphSearch
Breadth-First Search (BFS)

- Explores all possible paths in parallel.

- Guaranteed to find a solution path, if there is one.

- Always finds a shortest path, measured by the number of graph edges traversed.
Breadth-First Search of a Graph

breadthFirstSearch (vertex)
// Begin search by calling this procedure
// on the start vertex.
If (vertex is the goal vertex)
    then Return the path to vertex.
else 1. Let q be an empty queue.
    2. EnQueue vertex onto q.
    3. Mark vertex as “visited”.
    4. While (q is not empty)
        a. Let v = DeQueue(q).
        b. For (w in Neighbors(v))
            If (w has not been visited)
                then If (w is the goal vertex)
                    then Return the path to w.
                else 1. Mark w as “visited”.
                    2. EnQueue w onto q.
Breadth-First Search of a Graph

```
ArrayList<Integer> breadthFirstSearch(Integer vertex) {
    Node startNode = new Node(vertex,null);
    if (goal(vertex)) return path(startNode);
    Queue<Node> queue = new LinkedList<Node>();
    queue.add(startNode);
    visited[vertex] = true;
    while (!queue.isEmpty()) {
        Node node = queue.remove();
        Integer v = node.vertex;
        for (Integer w:neighbors.get(v))
            if (!visited[w]) {
                Node newNode = new Node(w,node);
                if (goal(w)) return path(newNode);
                else {
                    visited[w] = true;
                    queue.add(newNode);
                }
            }
    }
    return null;
}
```
Comparing DFS and BFS

- **DFS:**
  - Time is $O(N)$ where $V$ is the number of vertices in the graph.
  - Space is $O(L)$ where $l$ is the length of the solution path.

- **BFS:**
  - Time is $O(N)$ where $n$ is the number of vertices in the graph.
  - Space is $O(N)$ where $V$ is the number of vertices in the graph.
Accounting for Edge Length

• Suppose we want to find a shortest path, where by “shortest”, we mean the physical length of the path in the maze.

• Let’s assign a length to each edge in the graph.

• How should we change our BFS algorithm?
Best First Search

• Store distance from start in each node.

• Order nodes in a priority queue.

• Test for goal vertex when taking a node out of the queue.