GRAPHS, PART 2

cs2420 | Introduction to Algorithms and Data Structures | Spring 2016
administrivia...
- assignment 8 due tonight

- assignment 9 is out and due next Wednesday
  - it is another pair-programming assignment

- Prof. Meyer’s office hours for this week are moved to Friday 10am-12pm
last time...
- trees are a **subset** of graphs

- a graph is a set of **nodes** connected by edges
  - an edge is just a link between two nodes
  - nodes don’t have a parent-child relationship
  - links can be bi-directional

- graphs are used **EXTENSIVELY** throughout CS
-graphs have no root; must store all nodes

```java
class Graph<E> {
    List<Node> nodes;
    ...
}
```

-implementation is more general than a tree

```java
class Node{
    E Data;
    List<Node> neighbors;
    ...
}
```

-the order in which neighbors appear in the list is unspecified
  -a different order still make the same graph!
pathfinding

- there may be more than one path from one node to another

- we are often interested in the path length

- finding the shortest (or cheapest) path between two nodes is a common graph operation
- depth-first search (just like a tree) — DFS

- breadth-first search — BFS

- if there exists a path from one node to another these algorithms will find it
  - the nodes on this path are the steps to take to get from point A to point B

- if multiple such paths exist, the algorithms may find different ones
depth-first search
- look at the first edge going out of the start node
- recursively search from the new node
- upon returning, take the next edge
- if no more edges, return

- when visiting a node, mark it as visited so we don’t get stuck in a cycle
  - skip already visited nodes during traversal

- for each node visited, save a reference to the node where we came from to reconstruct the path
breadth-first search
- instead of visiting deeper nodes first, visit shallower nodes first
  - visit nodes closest to the start point first, gradually get further away

- create an empty queue
- put the starting node in the queue
- while the queue is not empty
  - dequeue the current node
  - for each unvisited neighbor of the the current node
    - mark the neighbor as visited
    - put the neighbor into the queue

- notice it is not recursive… it just runs until the queue is empty!
what path will BFS find from B to C?

A) B E C
B) B E A D C
C) B E D C
D) none
what path will DFS find from A to D?

A) A B E D
B) A D
C) none
D) this is a trick question
what is true of DFS, searching from a start node to a goal node?

A) if a path exists, it will find it
B) it is guaranteed to find the shortest path
C) it is guaranteed to not find the shortest path
D) it must be careful about cycles
E) a, b, and d
F) a, c, and d
G) a and d
what is true of BFS, searching from a start node to a goal node?
A) if a path exists, it will find it
B) it is guaranteed to find the shortest path
C) it is guaranteed to not find the shortest path
D) it must be careful about cycles
E) a, b, and d
F) a, c, and d
G) a and d
topological sort

- consider a graph with no cycles

- a topological sort orders nodes such that...
  - if there is a path from node A to node B, then A appears before B in the sorted order

- example: scheduling tasks
  - represent the tasks in a graph
  - if task A must be completed before task B, then A has an edge to B
today...
- BFS and the homework
- weighted graphs
- dijkstra’s algorithm
- in this graph, every spot is a node
- nodes have edges with adjacent spots
- expand out equally in all directions from starting node
- when any path reaches the goal node... done!

what type of search are we doing here?
- in this graph, every spot is a node
- nodes have edges with adjacent spots
- expand out equally in all directions from starting node
- when any path reaches the goal node... done!

what type of search are we doing here?
- in this graph, every spot is a node
- nodes have edges with adjacent spots
- expand out equally in all directions from starting node
- when any path reaches the goal node... done!

what type of search are we doing here?
- in this graph, every spot is a node
- nodes have edges with adjacent spots
- expand out equally in all directions from starting node
- when any path reaches the goal node... done!

what type of search are we doing here?
- in this graph, every spot is a node
- nodes have edges with adjacent spots
- expand out equally in all directions from starting node
- when any path reaches the goal node... done!

what type of search are we doing here?
- put the starting node in the queue and mark it as visited

- while the queue is not empty:
  - dequeue the current node
  - if current == goal, done!
  - otherwise, mark current's neighbors as visited and add them to the queue
-put the starting node in the queue and mark it as visited

-while the queue is not empty:
  - dequeue the current node
  - if current == goal, done!
  - otherwise, mark current's neighbors as visited and add them to the queue

```
queue:
  current:
  visited
  came from
```
- put the starting node in the queue and mark it as visited

- while the queue is not empty:
  - dequeue the current node
  - if current == goal, done!
  - otherwise, mark current's neighbors as visited and add them to the queue

```
current: 8
queue: 8
```

![Graph showing the queue and visited nodes]
- put the starting node in the queue and mark it as visited

- while the queue is not empty:
  - dequeue the current node
  - if current == goal, done!
  - otherwise, mark current's neighbors as visited and add them to the queue

```
current:  8
queue:   15  9
```

![Diagram showing a grid with nodes labeled from 0 to 23, and an arrow indicating the path from node 8 to node 9, with a node marked as visited.](image)
- put the starting node in the queue and mark it as visited

- while the queue is not empty:
  - dequeue the current node
  - if current == goal, done!
  - otherwise, mark current's neighbors as visited and add them to the queue

current: 15
queue: 9 17

visited

came from
- Put the starting node in the queue and mark it as visited.

- While the queue is not empty:
  - Dequeue the current node.
  - If current == goal, done!
  - Otherwise, mark current's neighbors as visited and add them to the queue.

Current: 9
Queue: 17 10

Visited: came from

Diagram:

```
0 1 2 3 4 5 6
7
8 9 10 11 12 13 14
15
17 18 19 20 21 22 23
```
- put the starting node in the queue and mark it as visited

- while the queue is not empty:
  - dequeue the current node
  - if current == goal, done!
  - otherwise, mark current's neighbors as visited and add them to the queue

current: 17
queue: 10 18
- put the starting node in the queue and mark it as visited

- while the queue is not empty:
  - dequeue the current node
  - if current == goal, done!
  - otherwise, mark current's neighbors as visited and add them to the queue

```plaintext
current: 10
queue: 18 11
```

![Diagram showing a grid with nodes 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, with node 10 highlighted and the visited node marked with a dot. The arrow indicates the direction from node 8 to node 10.]
- put the starting node in the queue and mark it as visited

- while the queue is not empty:
  - dequeue the current node
  - if current == goal, done!
  - otherwise, mark current's neighbors as visited and add them to the queue

**current:** 18
**queue:** 11 19

![Diagram with visited nodes and connections](image)
-put the starting node in the queue and mark it as visited

-while the queue is not empty:
  - dequeue the current node
  -if current == goal, done!
  -otherwise, mark current's neighbors as visited and add them to the queue
- Put the starting node in the queue and mark it as visited

- While the queue is not empty:
  - Dequeue the current node
  - If current == goal, done!
  - Otherwise, mark current's neighbors as visited and add them to the queue

```
current: 19  goal!
queue:  7  12
```

```
0 1 2 3 4 5 6
7

8 9 10 11 12 13 14
15

17 18 19 20 21 22 23
```
-put the starting node in the queue and mark it as visited

-while the queue is not empty:
  - dequeue the current node
  - if current == goal, done!
  - otherwise, mark current's neighbors as visited and add them to the queue

current: 19 goal!
queue: 7 12

-reconstruct the path…

visited
came from
- **shortest pathfinding problem**

- **the problem:**
  - input: a simple text file describing the maze
    - includes wall locations, start point, and end point
  - output: a similar text file with the shortest path from start to end indicated in the maze

- **represent all possible moves with a graph, then do a breadth first search**
input

```
5 10
xxxxxxxxxxx
X  S   X
X  X
X    G  X
xxxxxxxxxxx
```

output

```
5 10
xxxxxxxxxxx
X  S....  X
X  .  X
X    G  X
xxxxxxxxxxx
```

- **X** wall segment
- **S** starting point
- **G** goal
- an open space
- solution path indicator
<table>
<thead>
<tr>
<th>input</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>5 10</td>
<td>5 10</td>
</tr>
<tr>
<td>XXXXXXXXXX</td>
<td>XXXXXXXXXX</td>
</tr>
<tr>
<td>X S</td>
<td>X S......X</td>
</tr>
<tr>
<td>XXXXXXXX X</td>
<td>XXXXXXXX.X</td>
</tr>
<tr>
<td>X G</td>
<td>X G.......X</td>
</tr>
<tr>
<td>XXXXXXXXX</td>
<td>XXXXXXXXXX</td>
</tr>
</tbody>
</table>

- **`X`** wall segment
- **`S`** starting point
- **`G`** goal
- an open space
- solution path indicator
input
10 19
XXXXXXXXXXXXXXXXXXXXXXXX
X S X
X X
X X
XX X X X
X X XXX X X
X G XX X
X X
X
XXXXXXXXXXXXXXXXXXXXXXXX

output
10 19
XXXXXXXXXXXXXXXXXXXXXXXX
X S. X
X . X
X . X
XX X . X X
X X XXX . X X
X G..XX . X X
X .... X X
X X
XXXXXXXXXXXXXXXXXXXXXXXX

x wall segment
s starting point
g goal
an open space
solution path indicator
can any node have an edge to any other node?

how do we represent walls?
for this specific problem, we can store the graph as a 2D array

Node nodes[][];
nodes = new Node[5][10];

Node class doesn’t need a list of edges
- neighbors are implied (up, down, left right)
- ie. for node N, the up neighbor would be:
  nodes[N.row-1][N.col]

walls are null
- ie. no neighbor if null
-while reading the input:
  -for every character that makes up the maze [i][j]
    -if it is a wall
      nodes[i][j] = null
    -else
      nodes[i][j] = new Node(...)

-make sure to handle the start and goal nodes
rules

- the path cannot go through or on top of walls
- the path must be connected (no skips or jumps)
- diagonally-adjacent spaces are not connected
  - only up, down, left, right
- if no path exists, the output file will have no dots
- if multiple shortest paths exist, any of them are valid
- must produce output in exact format specified
- all you have to do is read in a file and produce a new file

- BUT, we are providing a program to read in your solution and display it as a Pacman game board

5 10
XXXXXXXXXXXX
X  S......X
XXXXXXXXXX.X
X  G......X
XXXXXXXXXXXX
no solution
file output

try
{
  PrintWriter output = new PrintWriter(
      new FileWriter("example.txt"));

  output.print("G");
  output.print("X");
  output.println();

  output.close();
}

file will contain "GX" and a newline
reading numbers

int height, width;

String[] dimensions =
    input.readLine().split(" ");

try {
    height = Integer.parseInt(dimensions[0]);
    width = Integer.parseInt(dimensions[1]);
}
weighted graphs
- sometimes it makes sense to associate a cost with traversing an edge

- we can add a weight to each edge
  - this is just a number!

- a higher weight indicates a more costly step

- weighted path length is the sum of all edge weights on a path
  - this is NOT the same as path length!
- What is the shortest path from SLC to Paris?
- What is the cheapest path from SLC to Paris?
- Cheapest is not always the shortest!
- Will regular BFS find the cheapest path?
- make a new `Edge` class, which contains the reference and the weight

- instead of nodes having direct references to neighbors

```java
class Node{
    E data;
    List<Edge> neighbors;
}

class Edge{
    Node otherEnd;
    double weight;
}
```
dijkstra’s algorithm
- Dijkstra’s algorithm finds the *cheapest* path

- keep track of the total path cost from start node to the current node

- cost of path to next node is total cost so far plus weight of edge to next node

- instead of traversing nodes in the order they were encountered, traverse in order of cheapest total cost first
we want to find a path from A to C

dis this time we use a priority queue. mark nodes after removal from the queue.
we want to find a path from A to C

A\_costSoFar = 0
we want to find a path from \textbf{A} to \textbf{C}

dqueue \textbf{A}(0), and enqueue \textbf{A}'s neighbors with \textbf{A}'s cost-so-far plus the edge weight

\begin{align*}
\text{B}.\text{costSoFar} &= \text{A}.\text{costSoFar} + 3 \\
\text{D}.\text{costSoFar} &= \text{A}.\text{costSoFar} + 9 \\
\text{B}.\text{cameFrom} &= \text{A} \\
\text{D}.\text{cameFrom} &= \text{A}
\end{align*}

\textbf{priority queue:} \begin{tabular}{|c|c|}
\hline
\textbf{B}(3) & \textbf{D}(9) \\
\hline
\end{tabular}
we want to find a path from A to C
dequeue B (3), and enqueue B’s neighbors with B’s cost-so-far plus the edge weight

E.costSoFar = B.costSoFar + 4
E.cameFrom = B

priority queue: E(7) D(9)
we want to find a path from A to C

Dequeue E(7), and enqueue E’s neighbors with E’s cost-so-far plus the edge weight

// A visited, so skip
// cheaper path to D found!
C.costSoFar = E.costSoFar + 4
D.costSoFar = E.costSoFar + 1
D.cameFrom = E

priority queue: D(8) C(11)

visited unvisited
we want to find a path from A to C
dequeue D (8), and enqueue D’s neighbors with D’s cost-so-far plus the edge weight

// cheaper path to C found!
C.costSoFar = D.costSoFar + 2
C.cameFrom = D

priority queue: C(10)

visited
unvisited
we want to find a path from A to C
dequeue C(10). We found our goal! Final cost is 10. Reconstruct path.

A — B — E — D — C

priority queue: visited unvisited
Dijkstra(Node start, Node goal)
{
    initialize all nodes’ cost to infinity

    PQ.enqueue(start)
    while(!PQ.empty())
    {
        curr = PQ.dequeue()
        if(curr == goal) {return} \ done!
        curr.visited = true
        foreach unvisited neighbor n of curr:
        {
            if(n.cost > curr.cost + edgeweight)
            {
                PQ.enqueue(n) || update n’s position in PQ
                n.cameFrom = curr
                n.cost = curr.cost + edgeweight
            }
        }
    }
}
what path will Dijkstra’s find from A to C?

A) A B E C
B) A D C
C) A B E D C
next time...
**-reading**
  - chapter 20: Hash Tables in book

**-homework**
  - assignment 8 due tonight
  - assignment 9 due next Wednesday