Lecture 30

Graphs II

- Announcements
- Graph Traversals
  - Mazes
  - Basic Ideas
  - Breadth-First Search
  - Depth-First Search
- Shortest Paths
  - Dijkstra's Algorithm
- structure Package
Announcements
Congratulations to Daniel Zhang!
Winner of the 2021 Darwin contest!

```plaintext
BigBaller
gray
ifenemy 8
ifwall 6
ifsame 6
hop
go 1
left
go 1
infect
go 1
```

Congrats to [Daniel Zhang](#)! Winner of the 2021 Darwin contest!
Note: The lowest lab scores will be dropped!
Lab 9 is based on Dijkstra’s algorithm and airline travel.
Graph Traversal
Mazes
Perfect Mazes

A paper-and-pencil maze that has no loops and no accessible areas is called a "perfect maze".

Perfect Maze

Imperfect Maze
Solving and Traversing Perfect Mazes

Perfect mazes can be solved using the right-hand rule. (The left-hand rule also works and explores the maze in the opposite order.)

"Always keep your right hand on the wall."
— Right-Hand Rule

If you don't take the exit then you will traverse the entire maze before returning to the entrance.
Perfect Mazes as Trees

Perfect mazes are spanning trees of a grid graph with two vertices specified as entrance and exit.

- When there is an edge between two nodes remove the corresponding wall.
- If the entrance or exit is along an exterior wall then remove one of these exterior walls.
Perfect Mazes as Trees

Perfect mazes are spanning trees of a grid graph with two vertices specified as entrance and exit.

- When there is an edge between two nodes remove the corresponding wall.
- If the entrance or exit is along an exterior wall then remove one of these exterior walls.
Perfect Mazes as Trees

Perfect mazes are spanning trees of a grid graph with two vertices specified as entrance and exit.

- When there is an edge between two nodes remove the corresponding wall.
- If the entrance or exit is along an exterior wall then remove one of these exterior walls.

You can create random mazes by generating random spanning trees of a grid graph.

- In Kruskal's or Prim's algorithm assign all edges the same cost, and break ties randomly.
Traversing Mazes vs Graphs

Perfect mazes are easier to traverse than arbitrary graphs since the underlying graph is acyclic. The right-hand rule will not work for imperfect graphs or for arbitrary graphs.

When solving mazes humans "see" the entire maze all at once and can use their intuition. Computer programs work one step at a time and only "see" one thing.
Pitfalls and Perspective

We want to be able to traverse graphs even when they have cycles.

- This means that we must avoid getting ourselves into infinite loops.

Furthermore, we need to consider algorithms from first-person rather than top-down perspective.

One difference is that our algorithms can jump back to previous places (i.e. nodes) that they have remembered. Remembering where we have been also helps us avoiding looping endlessly.
Basic Ideas
Traversing General Graphs

When traversing general graphs (as opposed to trees) we need to be more careful.

- There are cycles so we need to avoid looping around them endlessly.

We need to avoid looping around this cycle.

We will present two iterative algorithms.

- The algorithms will traverse over each edge exactly twice.
Vertex States
To avoid the looping problem we will keep track of some data. Each vertex can be thought of as being in one of three different states.

1. A vertex is *undiscovered* if the algorithm has not seen the vertex yet.
2. A vertex is *discovered* if the algorithm has seen the vertex but we have not followed all of its incident edges.
3. A vertex is *processed* if the algorithm has seen the vertex and followed all of its incident edges.
To-Do List

We need to keep track of vertices that have been discovered but not yet processed. Whenever we encounter an undiscovered vertex we add it to a *to-do list*.

![Diagram showing the to-do list process]

- Initially the to-do list contains the single vertex where we start.
- When processing we examine a vertex's neighbors clockwise from 12 o'clock,
  - While processing vertex x we discover vertex z, and we rediscover vertex y.
  - We have previously discovered y, so it is already in the to do list (or it has already been fully processed).
  - We have not previously discovered y so it is added to the to-do list.

Our algorithms repeatedly examine vertices in the to-do list until there are none.
Generic Algorithm for Traversing a Graph using Iteration

The following algorithm will correctly and efficiently explore the entire graph using iteration.

- It is efficient since each edge will be examined twice.

```plaintext
function traverse(G, start)
    for all u in vertices of G
        state[u] = "undiscovered"
    state[start] = "discovered"
    todo = new_list()
    add_list(todo, start)
    while todo is not empty
        current_vertex = remove_list(todo)
        for all u in neighbors(G, current_vertex)
            if state[u] == "undiscovered" then
                state[u] = "discovered"
                add_list(todo, u)
        state[current_vertex] = "processed"
    traverse(G, start)  // start is a vertex in G
```

Pseudocode for our generic graph search algorithm using iteration.

Notice that we have not specified the order in which the to-do edges are processed.

- There will be two special cases that provide canonical algorithms.
Breadth-First Search
Breadth-First Search

In breadth-first search (BFS) we process the to-do list as a queue.

- Recall that a queue uses First-In First-Out (FIFO) order.

```python
function BFS(G, start)
    for all u in vertices of G
        state[u] = "undiscovered"
        state[start] = "discovered"
    todo = new_queue()
    enqueue(todo, start)
    while todo is not empty
        current_vertex = dequeue(todo)
        for all u in neighbors(G, current_vertex)
            if state[u] == "undiscovered" then
                state[u] = "discovered"
                enqueue(todo, u)
        state[current_vertex] = "processed"
    BFS(G, start)  // start is a vertex in G
```

Pseudocode for BFS using iteration.

The highlighted lines show how this implementation of BFS differs from the generic search.
Example: BFS

Breadth-first search starting from vertex g.
Example: BFS

Breadth-first search starting from vertex g.
Example: BFS

Breadth-first search starting from vertex g.

<table>
<thead>
<tr>
<th>color</th>
<th>state</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>undiscovered</td>
</tr>
<tr>
<td></td>
<td>discovered</td>
</tr>
<tr>
<td></td>
<td>processed</td>
</tr>
</tbody>
</table>

current_vertex: g
todo: e, h, l, j
Example: BFS

Breadth-first search starting from vertex g.

color | state
--- | ---
undiscovered | discovered | processed

current_vertex  todo
  | e | h | l | j |
Example: BFS

Breadth-first search starting from vertex g.

current_vertex: e  todo: h l j...
Example: BFS

Breadth-first search starting from vertex g.
Example: BFS

Breadth-first search starting from vertex g.
Example: BFS

Breadth-first search starting from vertex g.
Example: BFS

Breadth-first search starting from vertex g.

```
color    state
undiscovered
discovered
processed
```
Example: BFS

Breadth-first search starting from vertex g.

```
color   state
undiscovered
discovered
processed
```

```
current_vertex  todo  l  j  a  c  i  p  f
```

Diagram of the graph with vertices and edges labeled.
Example: BFS

Breadth-first search starting from vertex g.
Example: BFS

Breadth-first search starting from vertex g.

```
color  state
undiscovered
discovered
processed
```

```
current_vertex  todo
j  a  c  i  p  f  m

l
```
Example: BFS

Breadth-first search starting from vertex g.
Example: BFS

Breadth-first search starting from vertex g.

color | state
--- | ---
undiscovered

discovered

processed

current_vertex | j
todo | a c i p f m
Example: BFS

Breadth-first search starting from vertex g.

color | state
------|-------
undiscovered
processed

current_vertex | j

todo | a c i p f m o n
Example: BFS

Breadth-first search starting from vertex g.
Example: BFS

Breadth-first search starting from vertex g.

We have processed every vertex that has distance one from g.

Discovered vertices in the queue are distance two from g.

<table>
<thead>
<tr>
<th>color</th>
<th>state</th>
</tr>
</thead>
<tbody>
<tr>
<td>undiscovered</td>
<td>discovered</td>
</tr>
<tr>
<td>processed</td>
<td></td>
</tr>
</tbody>
</table>

current_vertex  todo  a  c  i  p  f  m  o  n
BFS Notes

1. Breadth-first search will always visit closer nodes first. In other words, a node at distance \( d \) from the starting node will be visited before any node at distance \( d+1 \). (Here \textit{distance} refers to the minimum number of edges on a path from the starting node to the node.)

2. We can create a tree that includes a shortest path from the starting node to every other node. To build the tree, we just add every edge that is used when first discovering a new node. Furthermore, we can make it into a rooted tree, where the starting node is the root, by setting the parent and child of each edge to be the discoverer and discovered, respectively.

3. The order that the nodes are discovered is the same order that the nodes are processed. Both of these orders can be obtained by the todo list by crossing out the values instead of erasing them.
Depth-First Search
Depth-First Search (Iterative)

In depth-first search (DFS) we process the to-do list as a stack.

- Recall that a stack uses Last-In First-Out (LIFO) order.

```
function DFS(G, start)
    for all u in vertices of G
        state[u] = "undiscovered"
    state[start] = "discovered"
    todo = new_stack()
    stack_push(todo, start)
    while todo is not empty
        current_vertex = stack_pop(todo)
        for all u in neighbors(G, current_vertex)
            if state[u] == "undiscovered"
                state[u] = "discovered"
                stack_push(todo, u)
        state[current_vertex] = "processed"
    DFS(G, start)  // start is a vertex in G
```

Pseudocode for DFS using iteration.

The highlighted lines show how this implementation of BFS differs from the generic search.
Example: DFS

Depth-first search starting from vertex g.

<table>
<thead>
<tr>
<th>color</th>
<th>state</th>
</tr>
</thead>
<tbody>
<tr>
<td>discovered</td>
<td></td>
</tr>
<tr>
<td>processed</td>
<td></td>
</tr>
</tbody>
</table>

current_vertex | todo |
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>g</td>
</tr>
</tbody>
</table>
Example: DFS

Depth-first search starting from vertex g.

current_vertex: g, todo: [ ]
Example: DFS

Depth-first search starting from vertex g.

<table>
<thead>
<tr>
<th></th>
<th>color</th>
<th>state</th>
</tr>
</thead>
<tbody>
<tr>
<td>g</td>
<td>discovered</td>
<td></td>
</tr>
<tr>
<td>h</td>
<td>processed</td>
<td></td>
</tr>
<tr>
<td>i</td>
<td>undiscovered</td>
<td></td>
</tr>
</tbody>
</table>

current_vertex: g
todo: j, l, h, e
Example: DFS

Depth-first search starting from vertex g.
Example: DFS

Depth-first search starting from vertex g.

current_vertex: j  
todo: l h e
Example: DFS

Depth-first search starting from vertex g.
Example: DFS

Depth-first search starting from vertex g.

current_vertex  todo

```
k  n  o  a  l  h  e
```
Example: DFS

Depth-first search starting from vertex g.

color | state
-------|------
undiscovered
discovered
processed

current_vertex | k

todo | n o a l h e
Example: DFS

Depth-first search starting from vertex g.
Example: DFS

Depth-first search starting from vertex g.

current_vertex  todo  |  n  o  a  l  h  e

color  |  state
--- | ---
undiscovered
discovered
processed
Example: DFS

Depth-first search starting from vertex g.

```
current_vertex  todo
    n  o  a  l  h  e
```
Example: DFS

Depth-first search starting from vertex g.

color | state
--- | ---
| undiscovered
| discovered
| processed

current_vertex | n
todo | o a l h e
Example: DFS

Depth-first search starting from vertex g.
Example: DFS

Depth-first search starting from vertex g.
Example: DFS

Depth-first search starting from vertex g.

current_vertex: o
depth-first search state:
- a: undiscovered
- b: undiscovered
- c: undiscovered
- d: undiscovered
- e: undiscovered
- f: undiscovered
- g: discovered
- h: processed
- i: undiscovered
- j: undiscovered
- k: undiscovered
- l: undiscovered
- m: undiscovered
- n: undiscovered
- o: discovered
- p: processed
- q: undiscovered

todo: p m a l h e
Example: DFS

Depth-first search starting from vertex g.
Example: DFS

Depth-first search starting from vertex g.
Example: DFS

Depth-first search starting from vertex g.

```
current_vertex: p  todo: q i m a l h e
color | state
undiscovered
discovered
processed
```
Example: DFS

Depth-first search starting from vertex g.

```
color  state
undiscovered
discovered
processed
```
Example: DFS

Depth-first search starting from vertex g.

Some vertices of distance two from g have been processed.

Some vertices of distance one have only been discovered.

color  |  state
-------|-------
undiscovered
discovered
processed

current_vertex  todo
q  i  m  a  l  h  e
Depth-First Search (Recursive)

Depth-first search can also be implemented recursively.

- This implementation implicitly replaces the todo stack with the call stack.

Recursive pseudocode for DFS.

```python
function DFS(G, start)
    state[start] = "discovered"
    for all u in neighbors(start)
        if state[u] == "undiscovered" then
            state[u] = "discovered"
            DFS(G, u)
    state[start] = "processed"

for all u in vertices of G
    state[u] = "undiscovered"
DFS(G, start)  // start is a vertex in G
```

Note: The order of processed is different in the iterative and recursive implementations. Why?
Two examples of recursive depth-first search from the textbook. Both methods are focused on exploring from an initial (or current) vertex.

- `reachableFrom` sets the `visit` property on vertices reached from the initial vertex. The initial vertex is the `vertexLabel` argument.
- `DFS` adds the vertices that are reached to a list. The initial vertex is the `n` argument.

In both cases, the `neighbors` iterator is used to check the neighbors of the current vertex. Recursion is applied on a neighbor only when that neighbor has not already been visited.

We’ll look more at the `structure` package’s implementation of `Graph` shortly.
Exploration Strategies

What is your strategy for exploring video games?

In Super Mario World all ghost houses and stages marked by a red dot have two exits.

Some people fully explore areas before moving on, and others move forward and return later.

These strategies will be called "breadth-first" and "depth-first" respectively.

In Super Mario 3D World all stages have 3 stars and 1 stamp.
BFS or DFS?
Which type of graph traversal is being used when exploring these mazes?

Green indicates discovering areas.
Red indicates backtracking over previously discovered areas.
Which traversal algorithm is Pro-ZD using?
Shortest Paths
Getting From A to B

What is the best way to get from A to B? The answer depends on the context.

- Google Maps wants to provide you the fastest route in terms of time.
- Google Flights wants to minimize a mixture of costs and waiting time and stopovers.
- In Super Mario Bros. you may want to minimize time, or maximize points, coins, 1UPs.
Example

What is the minimum weight route from A to B in the undirected graph below?
Example

What is the minimum weight route from A to B in the directed graph below?

- Directed edges can only be used in their forward direction.
Dijkstra’s Algorithm

We’ll look at the theory of the algorithm.

Why does everyone hate Java? It’s very verbose and repetitive

Reddit link

```java
public static
    Map<String, ComparableAssociation<Integer, Edge<String, Integer>>>
dijkstra(Graph<String, Integer> g, String start)
```
Dijkstra's Algorithm Overview

Recall that BFS finds a spanning tree containing all shortest paths from a root node. That is, BFS processes nodes that have paths of length 1, 2, ... from a root. Dijkstra's algorithm will process the nodes of minimum cost 1, 2, ... from a root.

Edge weights are non-negative, so this order ensures that nodes do not have a smaller weight paths when processed. How do we process nodes in this order?
Rediscovery and Updates

In BFS the edge that discovers a node will always be on a shortest path to that node. This is not true for minimum cost paths. Therefore, we can 'rediscover' nodes, and update the minimum cost to the node. We also need to change the associated parent and spanning tree edge.

- How can we update the minimum cost priorities efficiently?

The approach taken in the book (and next slide) is to add the node to the priority queue again. One approach is to use an enhanced priority queue that allows you to decrease a key's value (see CSCI 256).
Dijkstra's Algorithm

BFS with a priority queue and a min_cost array for shortest path lengths.

- The queue entries are (cost, node) pairs prioritized by cost.

```
function Dijkstra(G, start)
    for all u in vertices of G
        state[u] = “undiscovered”
        min_cost[u] = ∞
    state[start] = “discovered”
    min_cost[start] = 0
    todo = new_priority_queue()
    enqueue(todo, (0, start))
    while todo is not empty
        (cost_v, v) = dequeue(todo)  // cost_v will be the final min_cost[v]
        if state[current_vertex] ≠ “processed”
            for all u in neighbors(G, v) // directed or undirected neighbors
                if state[u] == “undiscovered”
                    state[u] = “discovered”
                    min_cost[u] = min_cost[v] + w_v,u
                else
                    min_cost[u] = min(min_cost[u], min_cost[v] + w_v,u)
                enqueue(todo, (min_cost[u], u))
        state[current_vertex] = “processed”
```

How would we create the actual min cost paths? Use the parent child idea on the next slide (and Lab 9)

This approach can enqueue a node multiple times. Non-optimal cases are ignored by the first if statement.
Example: Dijkstra's Algorithm

Find the min-cost paths from the root node C. (In directed graphs only update using one direction.)

A weighted graph.

At each step we provide a table showing the current min-costs. We also track the current parent of each node (use a pencil!). The final parents form a rooted spanning tree of minimum cost shortest paths to C.
Example: Dijkstra's Algorithm

The green node is being processed. Blue edges are current and red are finalized.

Initial state

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>∞</td>
<td>∞</td>
<td>0</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
</tr>
</tbody>
</table>
Example: Dijkstra's Algorithm

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>20</td>
<td>0</td>
<td>10</td>
<td>25</td>
<td>15</td>
<td>∞</td>
</tr>
</tbody>
</table>
Example: Dijkstra's Algorithm
Example: Dijkstra's Algorithm

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
</tr>
</thead>
<tbody>
<tr>
<td>A processed</td>
<td>5</td>
<td>15</td>
<td>0</td>
<td>10</td>
<td>25</td>
<td>15</td>
<td>∞</td>
</tr>
</tbody>
</table>
Example: Dijkstra's Algorithm

```
<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>5</td>
<td>15</td>
<td>0</td>
<td>10</td>
<td>25</td>
<td>15</td>
<td>∞</td>
</tr>
</tbody>
</table>
```
Example: Dijkstra's Algorithm

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
</tr>
</thead>
<tbody>
<tr>
<td>D processed</td>
<td>5</td>
<td>15</td>
<td>0</td>
<td>10</td>
<td>25</td>
<td>15</td>
<td>∞</td>
</tr>
</tbody>
</table>
Example: Dijkstra's Algorithm

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>5</td>
<td>15</td>
<td>0</td>
<td>10</td>
<td>25</td>
<td>15</td>
<td>∞</td>
</tr>
</tbody>
</table>
Example: Dijkstra's Algorithm
Example: Dijkstra's Algorithm

\[
\begin{array}{c|c|c|c|c|c|c|c}
A & B & C & D & E & F & G \\
5 & 15 & 0 & 10 & 25 & 15 & \infty \\
\end{array}
\]
Example: Dijkstra's Algorithm

\[
\begin{array}{|c|c|c|c|c|c|c|c|}
\hline
 & A & B & C & D & E & F & G \\
\hline
F processed & 5 & 15 & 0 & 10 & 20 & 15 & 40 \\
\hline
\end{array}
\]
Example: Dijkstra's Algorithm

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>5</td>
<td></td>
<td></td>
<td>10</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td></td>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>20</td>
<td></td>
<td>20</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>25</td>
<td>25</td>
<td>20</td>
<td>40</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td>10</td>
<td>20</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>15</td>
<td>15</td>
<td>15</td>
<td>10</td>
<td></td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>G</td>
<td>5</td>
<td>15</td>
<td>0</td>
<td></td>
<td>20</td>
<td>15</td>
<td>40</td>
</tr>
</tbody>
</table>
Example: Dijkstra's Algorithm

A | B | C | D | E | F | G
---|---|---|---|---|---|---
5  | 15| 0 | 10| 20| 15| 30
E processed
Example: Dijkstra's Algorithm

\[
\begin{array}{|c|c|c|c|c|c|c|c|}
\hline
A & B & C & D & E & F & G \\
\hline
5 & 15 & 0 & 10 & 20 & 15 & 30 \\
\hline
\end{array}
\]
Example: Dijkstra's Algorithm

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>15</td>
<td>0</td>
<td>10</td>
<td>20</td>
<td>15</td>
<td>30</td>
</tr>
</tbody>
</table>

G processed
Analysis of Dijkstra's Algorithm

Unlike BFS, the nodes enqueue in a min-priority queue so that is $O(n \log n)$-time. The $O(n^2)$-time from edges dominates, so overall it is still worst-case $O(n^2)$-time.

```
function Dijkstra(G, start)
    for all u in vertices of G
        state[u] = “undiscovered”
        min_cost[u] = ∞
    state[start] = “discovered”
    min_cost[start] = 0
    todo = new_priority_queue()
    enqueue(todo, (0, start))
    while todo is not empty
        (cost_v, v) = dequeue(todo)  // cost_v will be the final min_cost[v]
        if state[current_vertex] ≠ “processed” then
            for all u in neighbors(G, v) // directed or undirected neighbors
                if state[u] == “undiscovered” then
                    state[u] = “discovered”
                    min_cost[u] = min_cost[v] + w_{v,u}
                else
                    min_cost[u] = min(min_cost[u], min_cost[v] + w_{v,u})
                enqueue(todo, (min_cost[u], u))
            state[current_vertex] = “processed”
```

This approach can enqueue a node multiple times. Non-optimal cases are ignored by the first if statement.
Negative Weight Edges

When Dijkstra's algorithm finalizes a path to a node, it never needs to update it, or the other paths that use it. This is not true when there are negative weight edges.

Dijkstra's Algorithm from root A has finalized its min-weight paths to nodes B, C, D, F, H.

For example, how does the -25 edge effect the rest of the calculation?

- There is a shorter path from A to B (via D, C), so all finalized shortest paths through B are incorrect.