CSE 250

Data Structures

Dr. Eric Mikida
epmikida@buffalo.edu
208 Capen Hall

Lec 17: Intro to Graphs

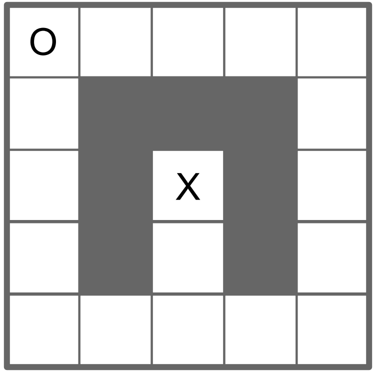
Announcements

- WA3 due this Sunday @ 11:59PM
- Midterm grading happen now, hold off on discussion until grading completes

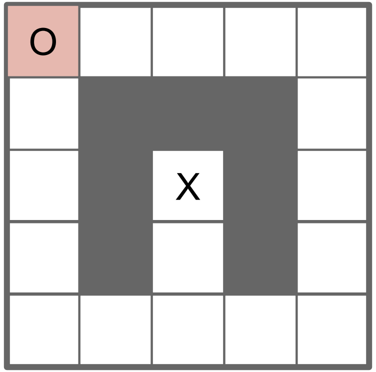
Mazes

```
steps(pos, dest):
  if pos == dest then return 0
  elif is_visited(pos) then return ∞
  elif is_filled(pos) then return ∞
  else
    Mark pos as visited
    min = 1 + min of all 4 steps
    Mark pos as unvisited
    return min
```

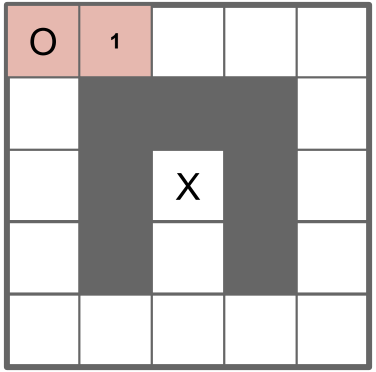
Mazes



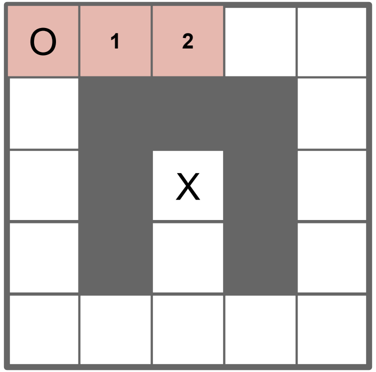
Mazes



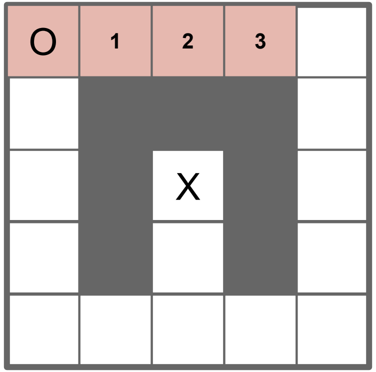
Mazes



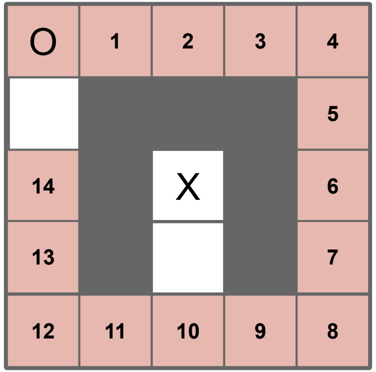
Mazes



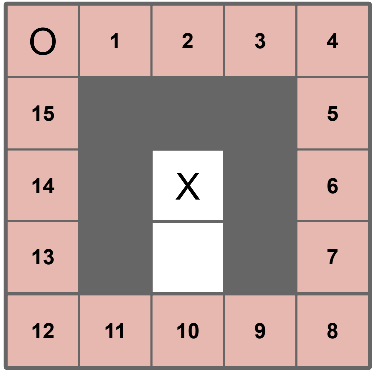
Mazes



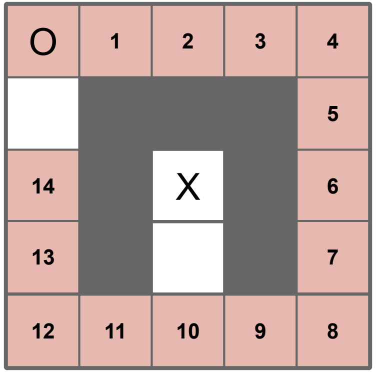
Mazes



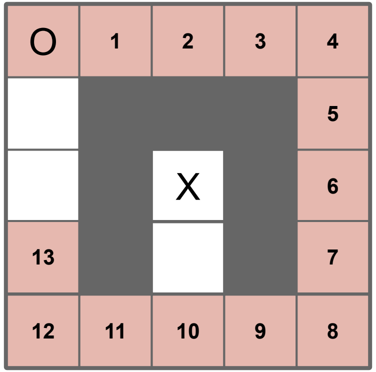
Mazes



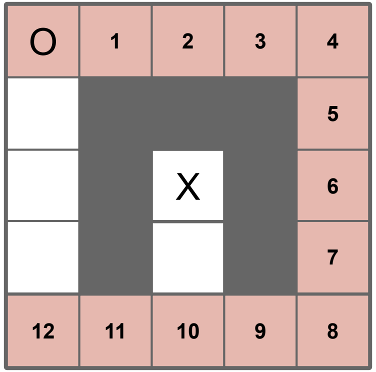
Mazes



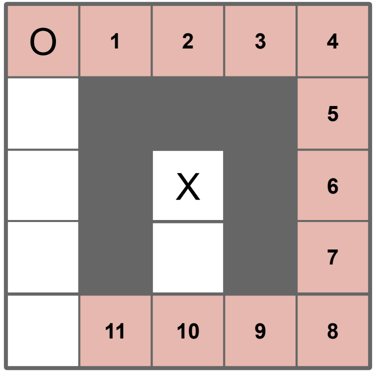
Mazes



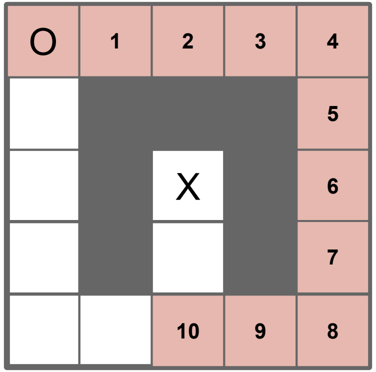
Mazes



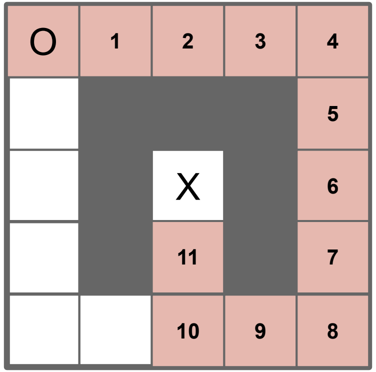
Mazes



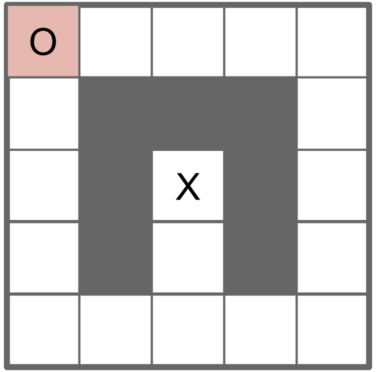
Mazes



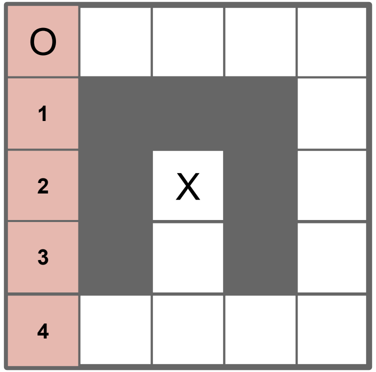
Mazes



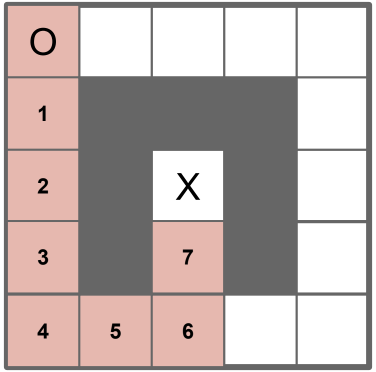
Mazes



Mazes



Mazes



Formalizing Maze-Solving

Inputs:

- The map: an *n* x *m* grid of squares which are either filled or empty
- The **O** is at position *start*
- The **X** is at position *dest*

Goal: Compute `steps(start, dest)`, the minimum number of steps from start to end.

Formalizing Maze-Solving

Inputs:

- The map: an *n* x *m* grid of squares which are either filled or empty
- The **O** is at position *start*
- The **X** is at position *dest*

Goal: Compute `steps(start, dest)`, the minimum number of steps from start to end. ✓

Formalizing Maze-Solving

Inputs:

- The map: an *n* x *m* grid of squares which are either filled or empty
- The **O** is at position *start*
- The **X** is at position *dest*

Goal: Compute `steps(start, dest)`, the minimum number of steps from start to end. ✓

What path did we take?

Mazes

Idea: Keep track of the nodes marked visited…that's our path!

Mazes: Now with…some data structure?

```
steps(pos, dest, visited):
  if pos == dest then return visited.copy()
  elif pos ∈ visited then return no_path
  elif is_filled(pos) then return no_path
  else
    visited.append(pos)
    bestPath = 1 + min of all 4 steps
    visited.removeLast()
    return bestPath
```

Mazes: Now with...some data structure?

```
steps(pos, dest, visited):
  if pos == dest then return visited.copy()
  elif pos ∈ visited then return no_path
  elif is_filled(pos) then return no_path
  else
    visited.append(pos)
    bestPath = 1 + min of all 4 steps
    visited.removeLast()
    return bestPath
```

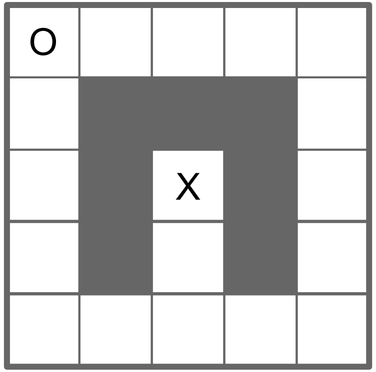
What could this data structure be??

Mazes: Now with...Stacks!

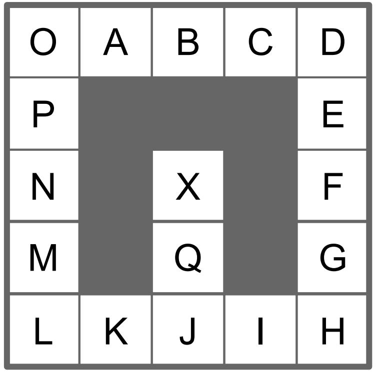
```
steps(pos, dest, visited):
  if pos == dest then return visited.copy()
  elif pos ∊ visited then return no_path
  elif is_filled(pos) then return no_path
  else
    visited.push(pos)
    bestPath = 1 + min of all 4 steps
    visited.pop()
    return bestPath
```

A stack!

Tracing an Example Search



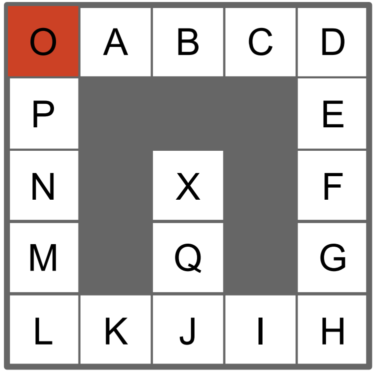
Tracing an Example Search



Call Stack

`visited`

Tracing an Example Search



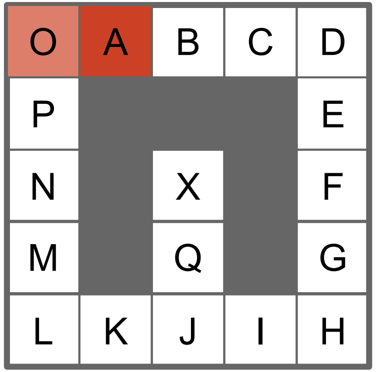

```
steps(O,X):
  steps(moveRight,X)
  steps(moveLeft,X)
  steps(moveUp,X)
  steps(moveDown,X)
```

Call Stack

O

`visited`

Tracing an Example Search



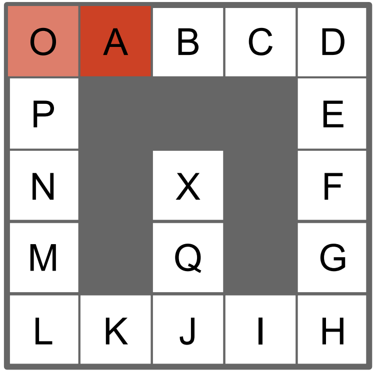

```
steps(A,X):
  steps(moveRight,X)
  steps(moveLeft,X)
  steps(moveUp,X)
  steps(moveDown,X)
steps(O,X):
  steps(moveRight,X)
  steps(moveLeft,X)
  steps(moveUp,X)
  steps(moveDown,X)
```

Call Stack

A
O

`visited`

Tracing an Example Search



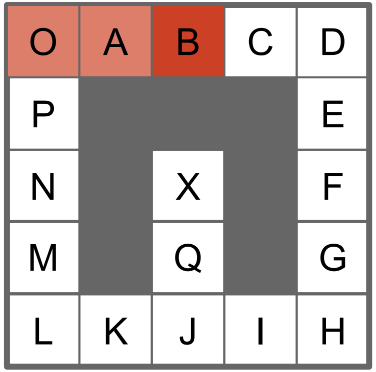

```
steps(A,X):
  steps(moveRight,X)
  steps(moveLeft,X)
  steps(moveUp,X)
  steps(moveDown,X)
steps(O,X)
```

Call Stack

A
O

`visited`

Tracing an Example Search



steps(B,X):
  steps(moveRight,X)
  steps(moveLeft,X)
  steps(moveUp,X)
  steps(moveDown,X)

steps(A,X):
  ~~steps(moveRight,X)~~
  steps(moveLeft,X)
  steps(moveUp,X)
  steps(moveDown,X)

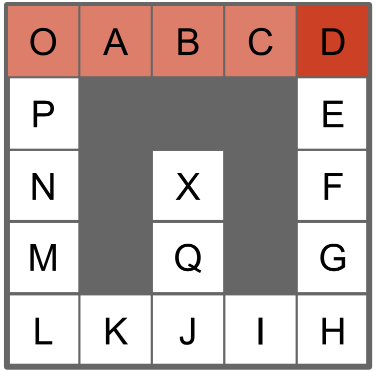
steps(O,X)

Call Stack

B
A
O

visited

Tracing an Example Search



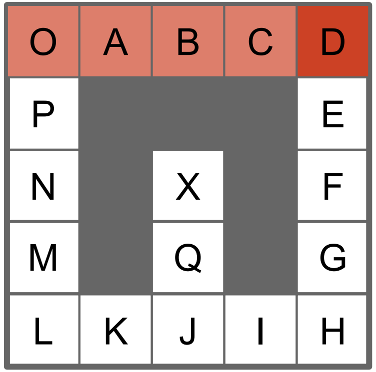

```
steps(D,X):
  steps(moveRight,X)
  steps(moveLeft,X)
  steps(moveUp,X)
  steps(moveDown,X)
steps(C,X)
steps(B,X)
steps(A,X)
steps(O,X)
```

Call Stack

D
C
B
A
O

visited

Tracing an Example Search



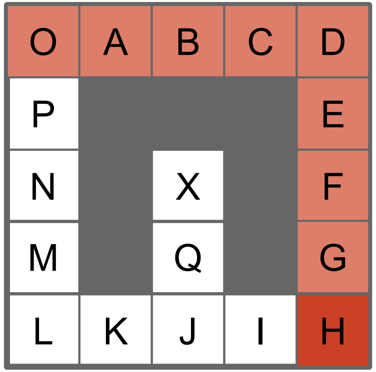
Call Stack
steps(D,X): ~~steps(moveRight,X)~~ ~~steps(moveLeft,X)~~ ~~steps(moveUp,X)~~ steps(moveDown,X)
steps(C,X)
steps(B,X)
steps(A,X)
steps(O,X)

Call Stack

visited
D
C
B
A
O

`visited`

Tracing an Example Search



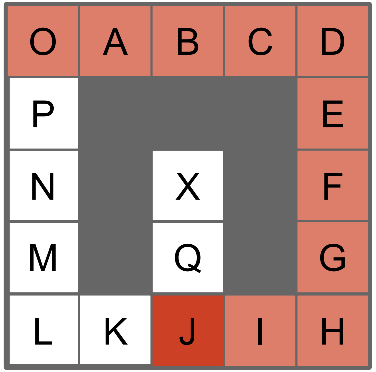
`steps(H,X)`
`steps(G,X)`
`steps(F,X)`
`steps(E,X)`
`steps(D,X)`
`steps(C,X)`
`steps(B,X)`
`steps(A,X)`
`steps(O,X)`

Call Stack

H
G
F
E
D
C
B
A
O

`visited`

Tracing an Example Search



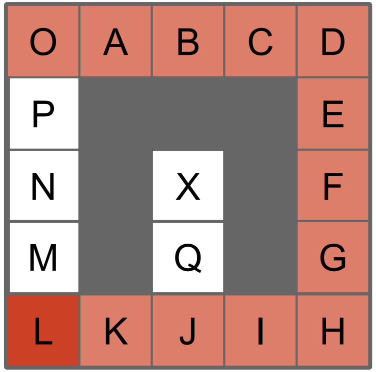

```
steps(J,X):
  ~~steps(moveRight,X)~~
  steps(moveLeft,X)
  steps(moveUp,X)
  steps(moveDown,X)
steps(I,X)
steps(H,X)
…
steps(O,X)
```

Call Stack

J
I
H
…
O

visited

Tracing an Example Search



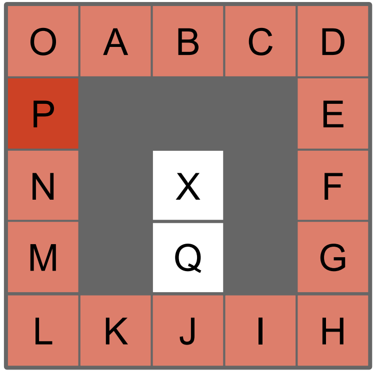
`steps(L,X)`
`steps(K,X)`
`steps(J,X)`
`steps(I,X)`
`steps(H,X)`
…
`steps(O,X)`

Call Stack

L
K
J
I
H
…
O

`visited`

Tracing an Example Search



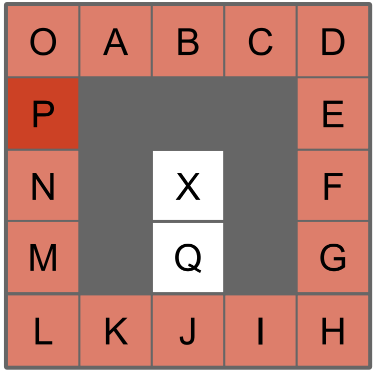

```
steps(P,X):
  steps(moveRight,X)
  steps(moveLeft,X)
  steps(moveUp,X)
  steps(moveDown,X)
steps(N,X)
steps(M,X)
steps(L,X)
...
steps(O,X)
```

Call Stack

P
N
M
L
…
O

visited

Tracing an Example Search



steps(P,X): ~~steps(moveRight,X)~~ ~~steps(moveLeft,X)~~ ~~steps(moveUp,X)~~ ~~steps(moveDown,X)~~
steps(N,X)
steps(M,X)
steps(L,X)
...
steps(O,X)

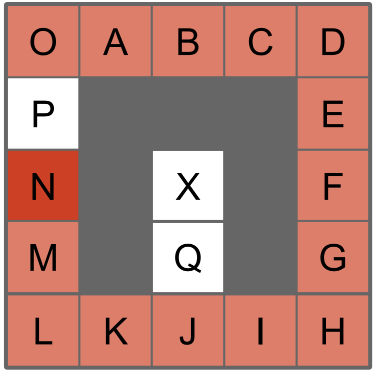
Call Stack

All 4 return no_path, so min is also no_path

P
N
M
L
...
O

visited

Tracing an Example Search



`steps(N,X):`
~~`steps(moveRight,X)`~~
~~`steps(moveLeft,X)`~~
~~`steps(moveUp,X)`~~
`steps(moveDown,X)`

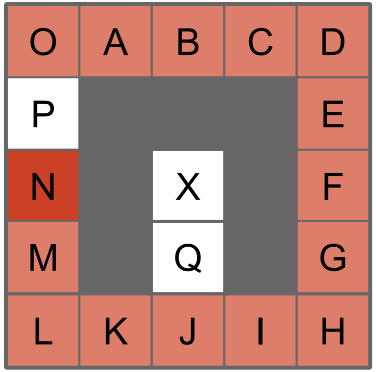
`steps(M,X)`
`steps(L,X)`
…
`steps(O,X)`

Call Stack

N
M
L
…
O

`visited`

Tracing an Example Search



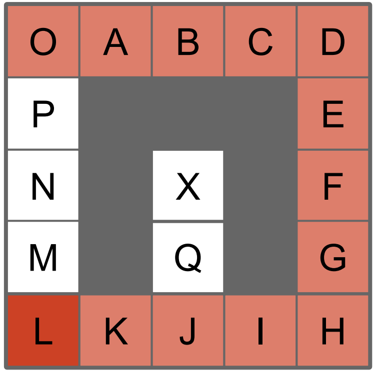
`steps(N,X):` ~~`steps(moveRight,X)`~~ ~~`steps(moveLeft,X)`~~ ~~`steps(moveUp,X)`~~ ~~`steps(moveDown,X)`~~
`steps(M,X)`
`steps(L,X)`
…
`steps(O,X)`

Call Stack

N
M
L
…
O

`visited`

Tracing an Example Search



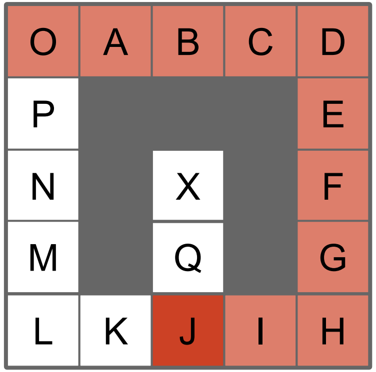
`steps(L,X)`
…
`steps(O,X)`

Call Stack

L
…
O

`visited`

Tracing an Example Search



Call Stack

steps(J,X):
  ~~steps(moveRight,X)~~
  ~~steps(moveLeft,X)~~
  steps(moveUp,X)
  steps(moveDown,X)

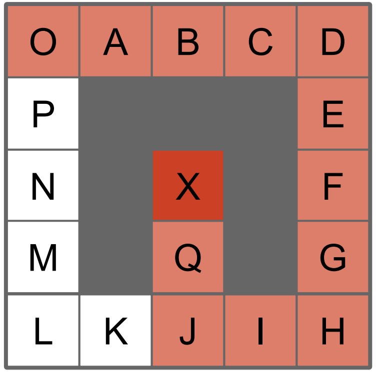
steps(I,X)
steps(H,X)
…
steps(O,X)

Call Stack

visited
J
I
H
…
O

visited

Tracing an Example Search



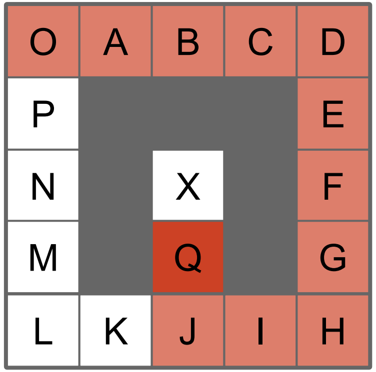
Call Stack
`steps(X,X)` `return visited.copy!`
`steps(Q,X)`
`steps(J,X)`
`steps(I,X)`
`steps(H,X)`
`…`
`steps(O,X)`

Call Stack

visited
X
Q
J
I
H
…
O

`visited`

Tracing an Example Search



returned no_path

returned OABCDEFGHIJQX

```
steps(Q,X):
```

~~steps(moveRight,X)~~
~~steps(moveLeft,X)~~
~~steps(moveUp,X)~~
~~steps(moveDown,X)~~

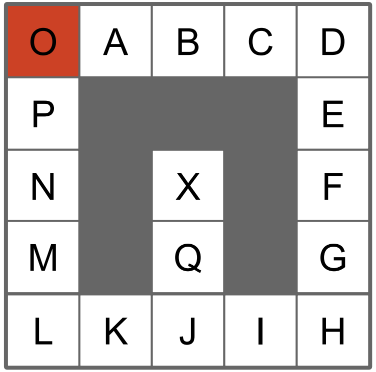
```
steps(J,X)
steps(I,X)
steps(H,X)
...
steps(O,X)
```

Call Stack

Q
J
I
H
…
O

visited

Tracing an Example Search



returned OABCDEFGHIJQX

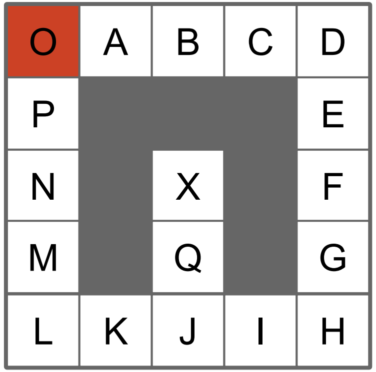
```
steps(O,X):
  ~~steps(moveRight,X)~~
  steps(moveLeft,X)
  steps(moveUp,X)
  steps(moveDown,X)
```

Call Stack

O

`visited`

Tracing an Example Search



returned OABCDEFGHIJQX

returned no_path

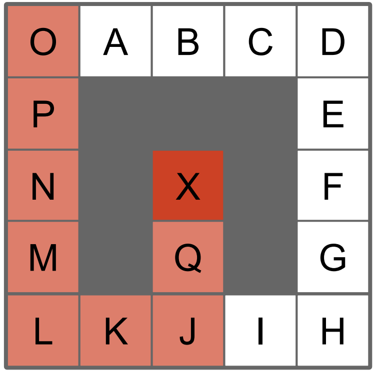
steps(O,X):
  ~~steps(moveRight,X)~~
  ~~steps(moveLeft,X)~~
  ~~steps(moveUp,X)~~
  steps(moveDown,X)

Call Stack

O

visited

Tracing an Example Search



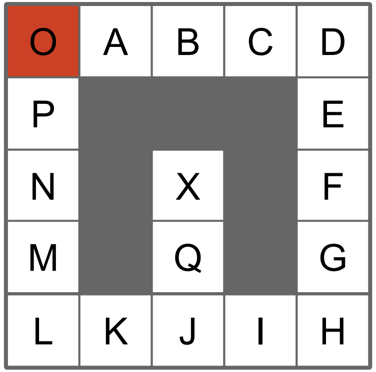
Call Stack
`steps(X,X)` `return visited.copy!`
`steps(Q,X)`
`steps(J,X)`
`steps(K,X)`
`steps(L,X)`
…
`steps(O,X)`

Call Stack

`visited`
X
Q
J
K
L
…
O

`visited`

Tracing an Example Search



returned OABCDEFGHIJQX

returned no_path

returned OPNMLKJQX

`steps(O,X):`

~~`steps(moveRight,X)`~~

~~`steps(moveLeft,X)`~~

~~`steps(moveUp,X)`~~

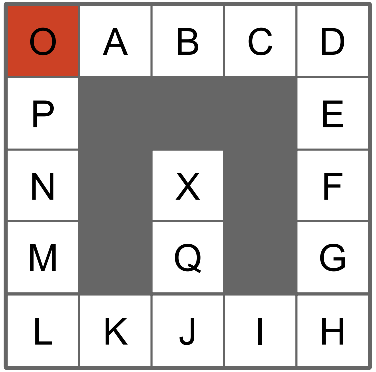
~~`steps(moveDown,X)`~~

Call Stack

O

`visited`

Tracing an Example Search



returned OABCDEFGHIJQX

```
steps(O,X):
```

~~steps(moveRight,X)~~

~~steps(moveLeft,X)~~

~~steps(moveUp,X)~~

~~steps(moveDown,X)~~

returned no_path

returned OPNMLKJQX

Call Stack

O

visited

Queues?

Thought Experiment: Can we do something similar with queues?

Queues?

Thought Experiment: Can we do something similar with queues?

Hold that thought!

Let's Talk About Graphs

A **graph** is a pair **(*V*,*E*)** where:

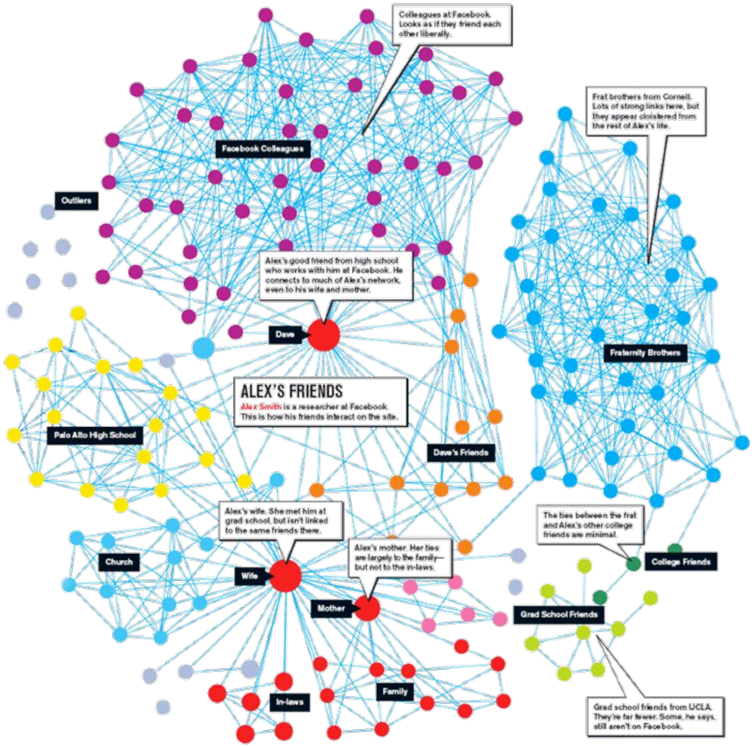
- ***V*** is a set of **vertices**
- ***E*** is a set of vertex pairs called **edges**
- Edges and vertices may also store data **(labels)**

Graphs

Example: A social network

(nodes store users, pictures, tweets, etc)

(edges store interactions)

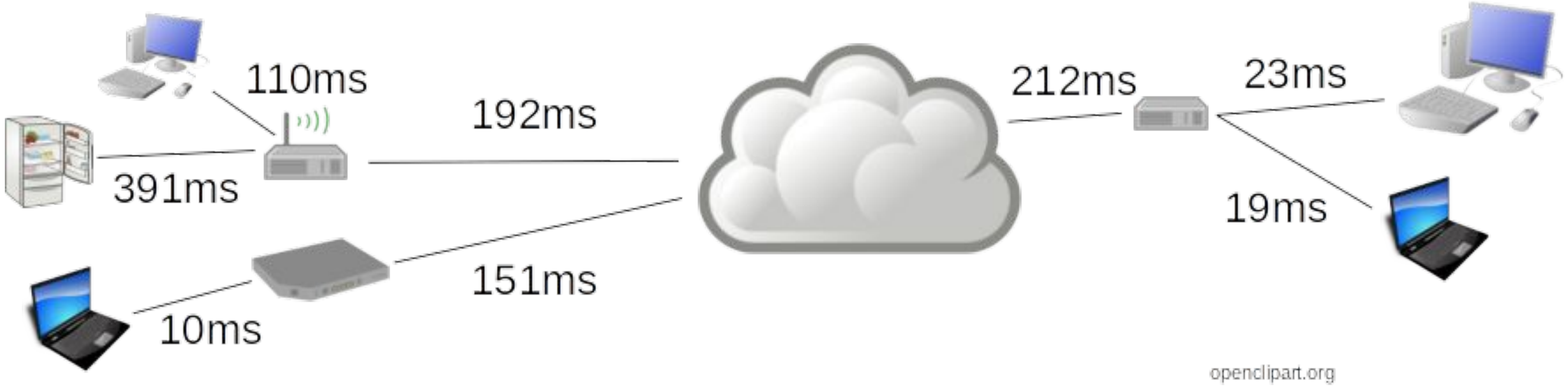


Ref :https://www.pinterest.com/pin/490470215639647556/

Graphs

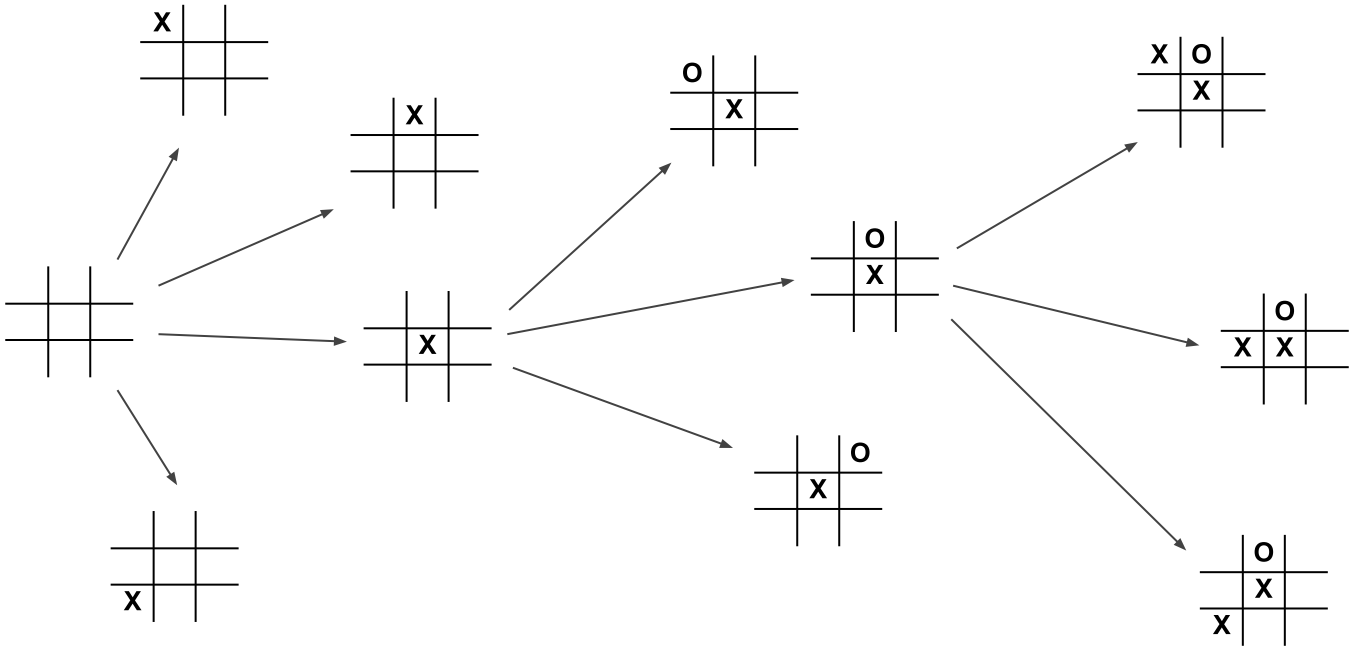
Example: A computer network

(edges store ping, nodes store addresses)



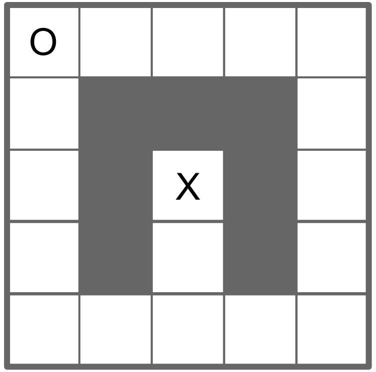
Graphs

Example: Moves in a game



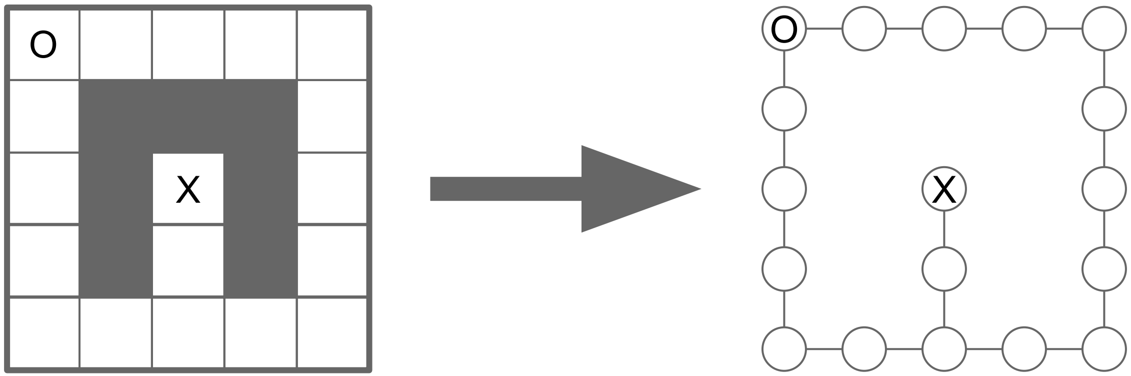
Back to Mazes

How could we represent our maze as a graph?



Back to Mazes

How could we represent our maze as a graph?



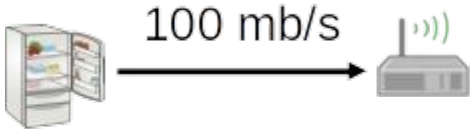
Edge Types

Directed Edge (asymmetric relationship)

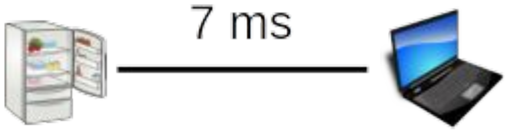
- Ordered pair of vertices (***u*, *v***)
- origin (***u***) →destination (***v***)

Undirected Edge (symmetric relationship)

- Unordered pair of vertices (***u*,*v***)



transmit bandwidth



round-trip latency

Edge Types

Directed Edge (asymmetric relationship)

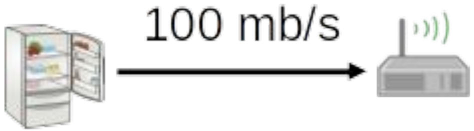
- Ordered pair of vertices (***u*, *v***)
- origin (***u***) →destination (***v***)

Undirected Edge (symmetric relationship)

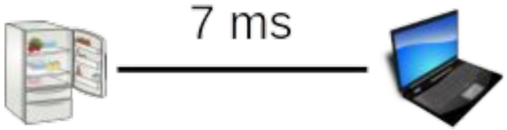
- Unordered pair of vertices (***u*,*v***)

Directed Graph: All edges are directed

Undirected Graph: All edges are undirected



transmit bandwidth



round-trip latency

Terminology

Endpoints of an edge

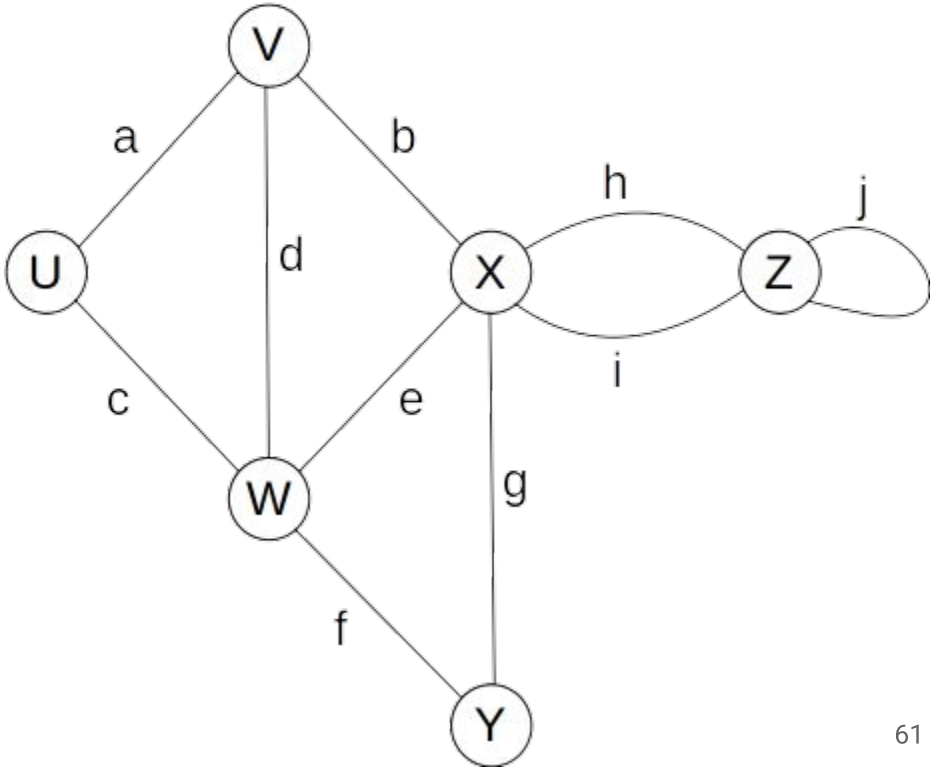
U, *V* are endpoints of *a*

Adjacent Vertices

U, *V* are adjacent

Degree of a vertex

X has degree 5



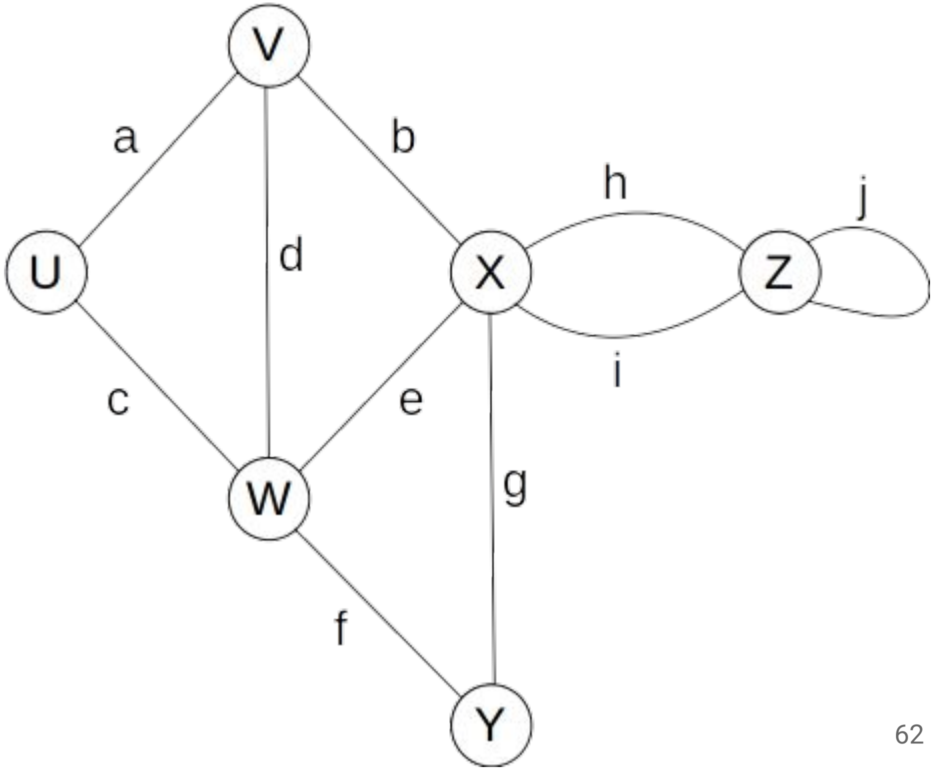
Terminology

Edges indecent on a vertex
a, *b*, *d* are incident on *V*

Parallel Edges
h, *i* are parallel

Self-Loop
j is a self-loop

Simple Graph
A graph without parallel edges or self-loops

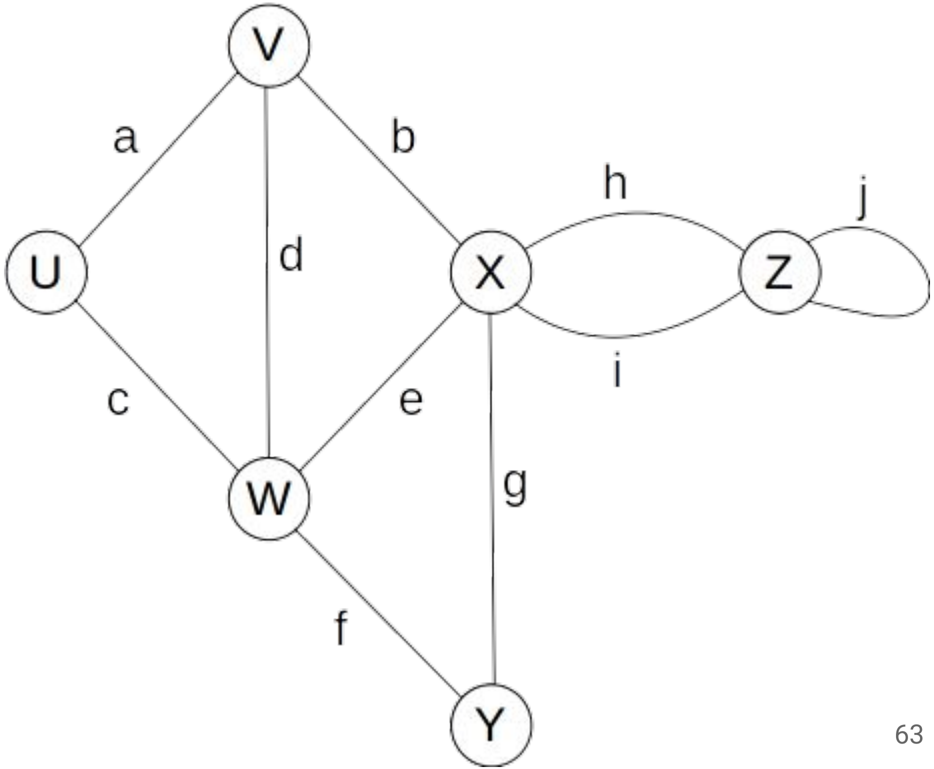


Terminology

Path
A sequence of alternating vertices and edges

- begins with a vertex
- ends with a vertex
- each edge preceded/followed by its endpoints

Simple Path
A path such that all of its vertices and edges are distinct

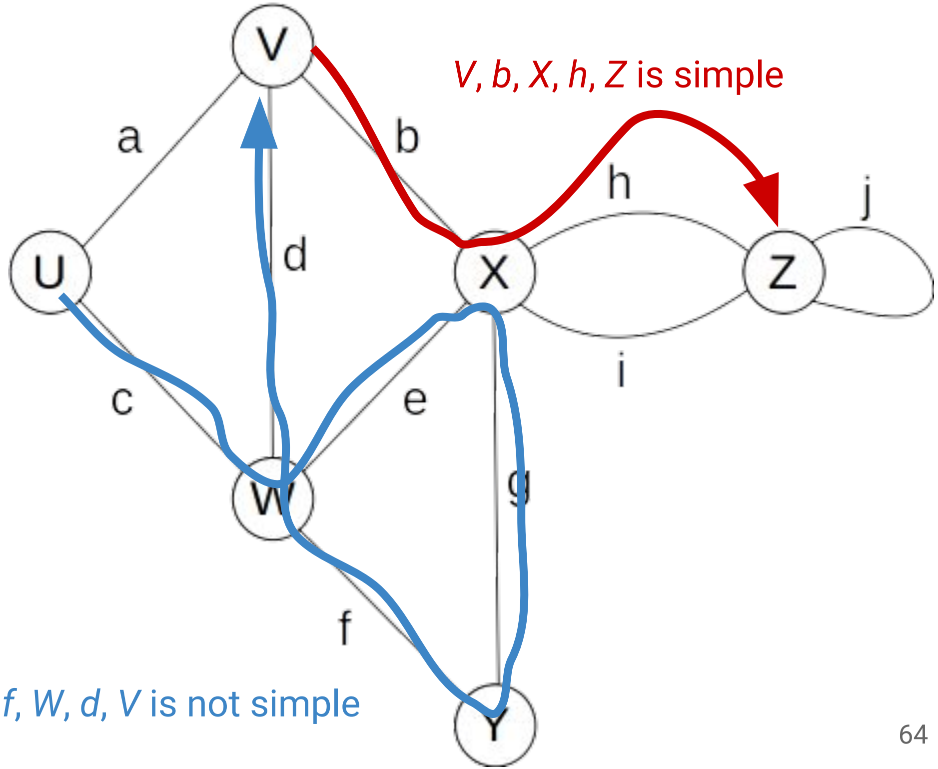


Terminology

Path
A sequence of alternating vertices and edges

- begins with a vertex
- ends with a vertex
- each edge preceded/followed by its endpoints

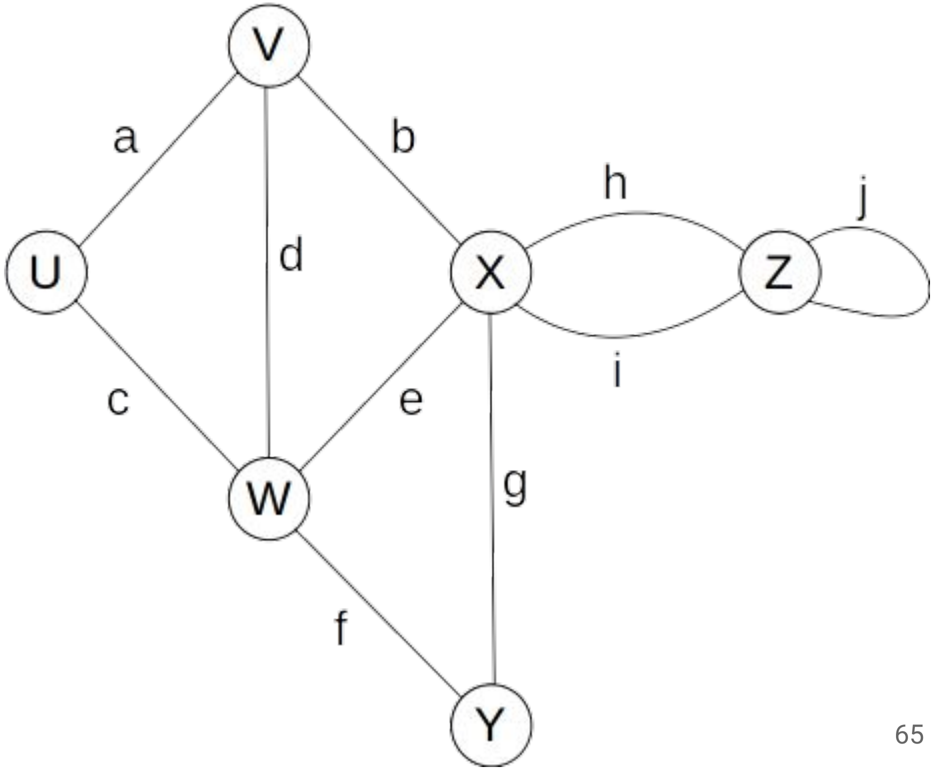
Simple Path
A path such that all of its vertices and edges are distinct



Terminology

Cycle
A path the begins and ends with the same vertex. Must contain at least one edge

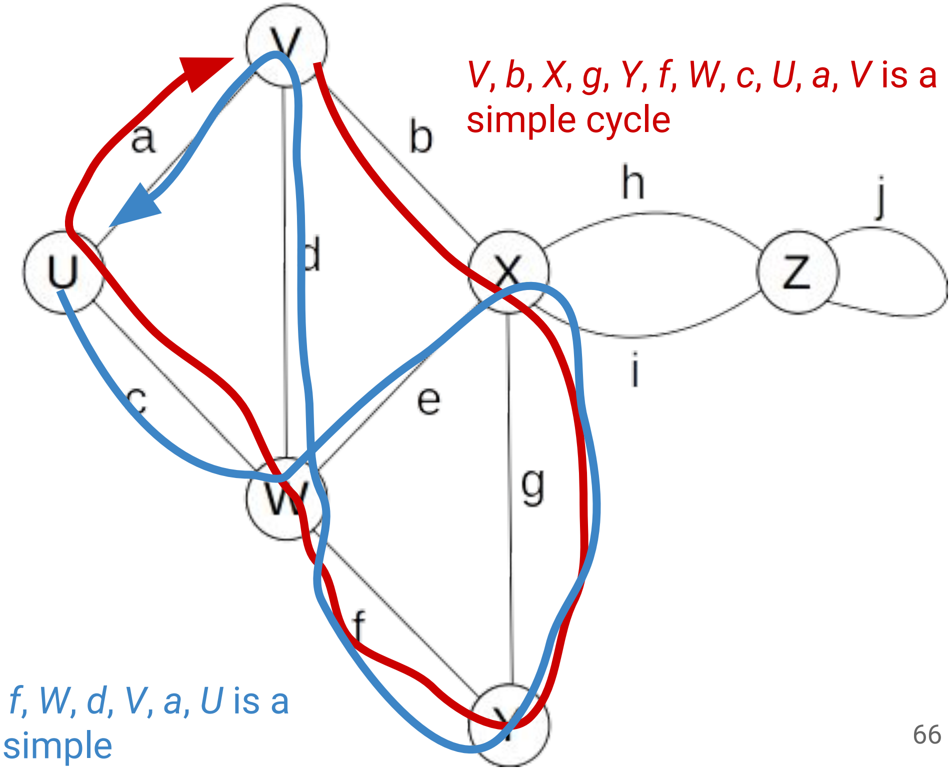
Simple Cycle
A cycle such that all of its vertices and edges are distinct



Terminology

Cycle
A path the begins and ends with the same vertex. Must contain at least one edge

Simple Cycle
A cycle such that all of its vertices and edges are distinct



V, b, X, g, Y, f, W, c, U, a, V is a simple cycle

U, c, W, e, X, g, Y, f, W, d, V, a, U is a cycle that is not simple

Notation

n The number of vertices

m The number of edges

deg(*v*) The degree of vertex ***v***

Graph Properties

$$\sum_{v} deg(v) = 2m$$

Graph Properties

$$\sum_{v} deg(v) = 2m$$

Proof: Each edge is counted twice

Graph Properties

In a directed graph with no self-loops and no parallel edges:

$$m \leq n\,(n - 1)$$

Graph Properties

In a directed graph with no self-loops and no parallel edges:

$$m \leq n\,(n - 1)$$

No parallel edges: each pair is connected at most once

No self-loops: pick each vertex only once

Graph Properties

In a directed graph with no self-loops and no parallel edges:

$$m \leq n\ (n - 1)$$

No parallel edges: each pair is connected at most once

No self-loops: pick each vertex only once

n choices for the first vertex; **(*n* - 1)** choices for the second vertex. Therefore even if there was one edge between every possible pair, we still have at most ***n*(*n* - 1)** edges

A (Directed) Graph ADT

Two type parameters (`Graph[V,E]`)

`V`: The vertex label type

`E`: The edge label type

Vertices

…are elements (like Linked List Nodes)

…store a value of type `V`

Edges

…are also elements

…store a value of type `E`