Principled Programming

Introduction to Coding in Any Imperative Language

Tim Teitelbaum

Emeritus Professor
Department of Computer Science
Cornell University

Graphs and Depth-First Search

Graphs are an abstract mathematical structure of great utility. When your problem can be cast as question about a graph, you have the opportunity to abstract away from details, and apply one of the known general-purpose graph algorithms that answer such questions.

Depth-First Search is a way to systematically enumerate elements of a graph. You can terminate the enumeration prematurely if you find an example of what you are looking for.

Think of graphs and depth-first search as an higher-level pattern that you should master and use. The problem of Running a Maze has served us well as a pedagogical example, but it's now time to reveal the "double cross": A maze is easily represented as a graph, and finding a path from one maze cell to another is easily done by depth-first search. Seize the opportunity when analysis reveals that such a problem reduction is available.

Sets, Pairs, and Relations:

Let S and T be two sets.

A *relation* between S and T is a set of ordered pairs, $\langle s,t \rangle$, where s is an element of S and t is an element of T.

Set T need not be distinct from set S, i.e., we can have relations between a set and itself.

Example: **has-child**

$$\{ \langle \text{Adam}, \text{Cain} \rangle, \langle \text{Adam}, \text{Abel} \rangle, \langle \text{Eve}, \text{Cain} \rangle, \langle \text{Eve}, \text{Abel} \rangle$$

Example: **has-parent**

$$\{ \langle \text{Cain}, \text{Adam} \rangle, \langle \text{Abel}, \text{Adam} \rangle, \langle \text{Cain}, \text{Eve} \rangle, \langle \text{Abel}, \text{Eve} \rangle \}$$

Directed Graphs:

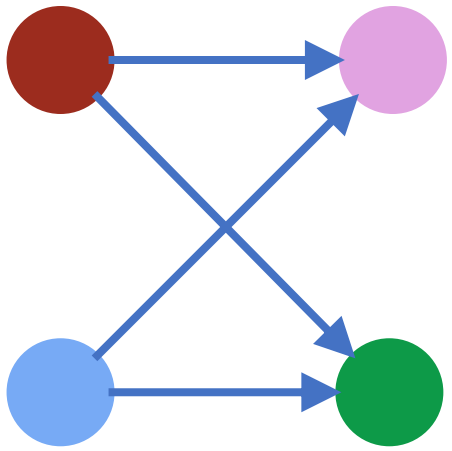
It is convenient to visualize a relation between a set *S* and itself as a collection of *nodes* and *edges*.

The elements of *S* are nodes, and an edge from node *m* to node *n* represents the existence of the pair ⟨*m*,*n*⟩ in the relation.

Such a visualization is known as a *directed graph*.

Example: **has-child**

{ ⟨Adam,Cain⟩, ⟨Adam,Abel⟩, ⟨Eve,Cain⟩, ⟨Eve, Abel⟩ }



Directed Graphs:

It is convenient to visualize a relation between a set *S* and itself as a collection of *nodes* and *edges*.

The elements of *S* are nodes, and an edge from node *m* to node *n* represents the existence of the pair ⟨*m*,*n*⟩ in the relation.

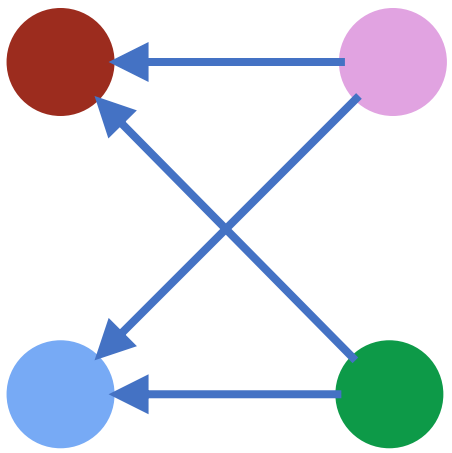
Such a visualization is known as a *directed graph*.

Example: **has-child**

{ ⟨Adam,Cain⟩, ⟨Adam,Abel⟩, ⟨Eve,Cain⟩, ⟨Eve, Abel⟩ }

Example: **has-parent**

{ ⟨Cain,Adam⟩, ⟨Abel,Adam⟩, ⟨Cain,Eve⟩, ⟨Abel,Eve⟩ }

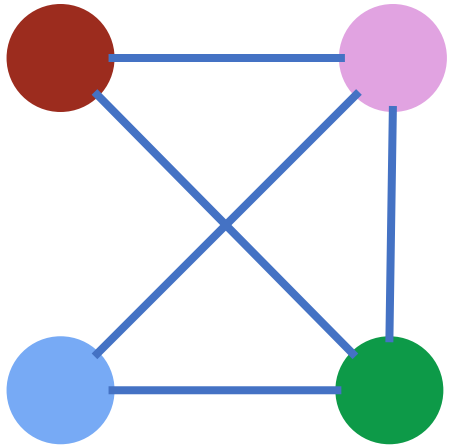


Undirected Graphs:

Some relations are *symmetric*, i.e., if ⟨*n*,*m*⟩ is in the relation, then ⟨*m*,*n*⟩ is also in the relation.

Example: **has-blood-relative**

{ ⟨Adam,Cain⟩, ⟨Adam,Abel⟩, ⟨Eve,Cain⟩, ⟨Eve, Abel⟩,
⟨Cain,Adam⟩, ⟨Abel,Adam⟩, ⟨Cain,Eve⟩, ⟨Abel,Eve⟩,
⟨Cain, Abel⟩, ⟨Abel, Cain⟩ }



In the visualization of a symmetric relation as a directed graph, edges would come in pairs that point in opposite directions. We render the pair as one edge with neither arrowhead, and call such a thing an *undirected graph*.

Reachability: Enumerate every node that can be reached from node n by following an edge.

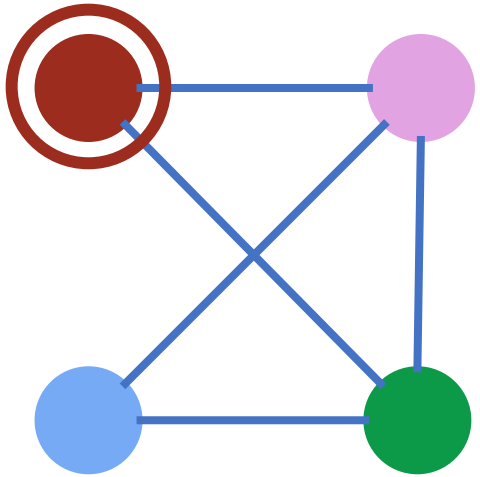
```
# If n was never visited, enumerate it and all its unvisited relatives.
def depth_first_search(n: node) -> None:
   if n-has-never-been-visited:
      #.Enumerate n.
      for each-edge-from-n-to-m:
         depth_first_search(m)
```

Although the definition is simple, its import is not necessarily readily apparent.
The following trace of its execution makes it clear

Adam

Reachability: Enumerate every node that can be reached from node n by following an edge.

```
def depth_first_search(n: node) -> None:
    """If n was never visited, enumerate it and all its unvisited relatives."""

    if n-has-never-been-visited:
        #.Enumerate n.
        for each-edge-from-n-to-m:
            depth_first_search(m)
```



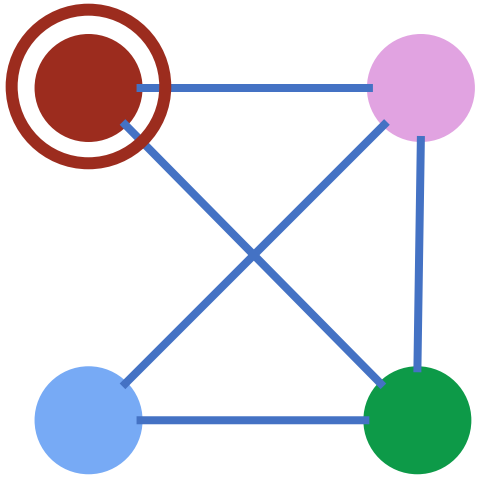
Depth-First Search

Adam

Reachability: Enumerate every node that can be reached from node n by following an edge.

```
def depth_first_search(n: node) -> None:
    """If n was never visited, enumerate it and all its unvisited relatives."""

    if n-has-never-been-visited:
        #.Enumerate n.
        for each-edge-from-n-to-m:
            depth_first_search(m)
```

True



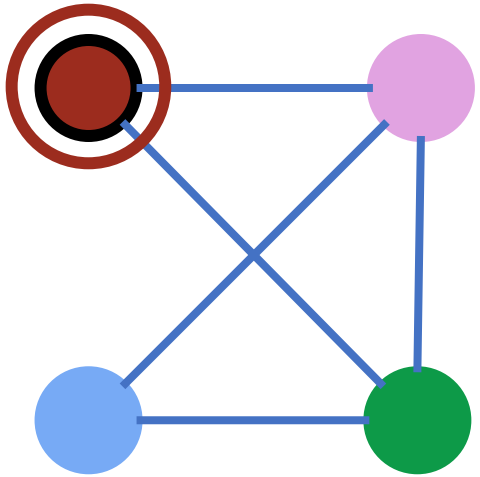
Adam

Reachability: Enumerate every node that can be reached from node n by following an edge.

```
def depth_first_search(n: node) -> None:
    """If n was never visited, enumerate it and all its unvisited relatives."""

    if n-has-never-been-visited:
        #.Enumerate n.
        for each-edge-from-n-to-m:
            depth_first_search(m)
```

enumeration

Adam



Means “marked as visited”

Adam

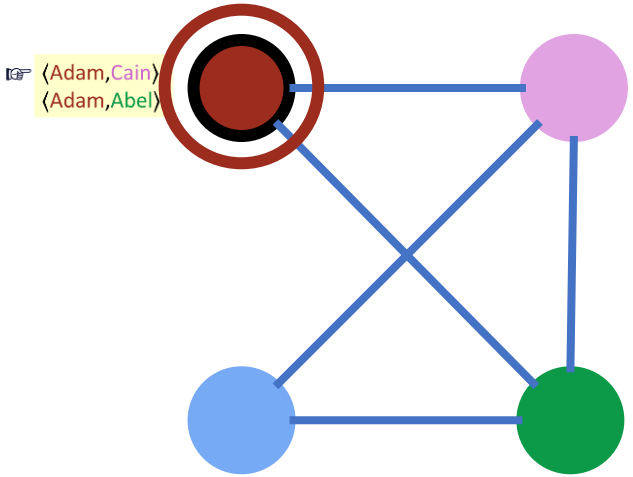
Reachability: Enumerate every node that can be reached from node n by following an edge.

```
def depth_first_search(n: node) -> None:
    """If n was never visited, enumerate it and all its unvisited relatives."""

    if n-has-never-been-visited:
        #.Enumerate n.
        for each-edge-from-n-to-m:
            depth_first_search(m)
```

enumeration

Adam

☞ ⟨Adam,Cain⟩
⟨Adam,Abel⟩



Depth-First Search

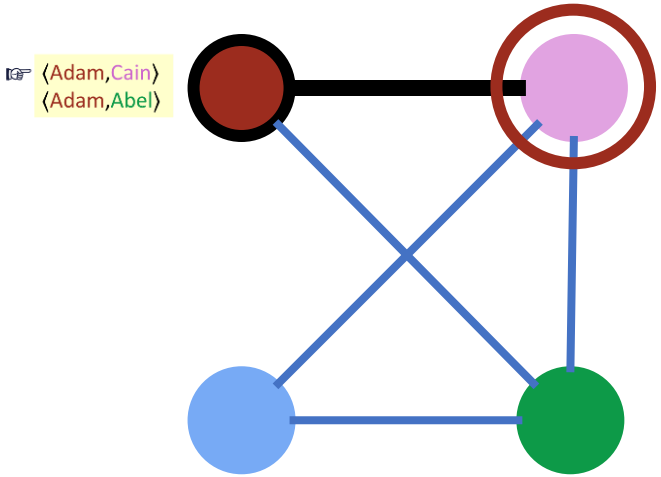
Cain

Reachability: Enumerate every node that can be reached from node n by following an edge.

```
def depth_first_search(n: node) -> None:
    """If n was never visited, enumerate it and all its unvisited relatives."""

    if n-has-never-been-visited:
        #.Enumerate n.
        for each-edge-from-n-to-m:
            depth_first_search(m)
```

enumeration

Adam



Means “first vist”

Cain

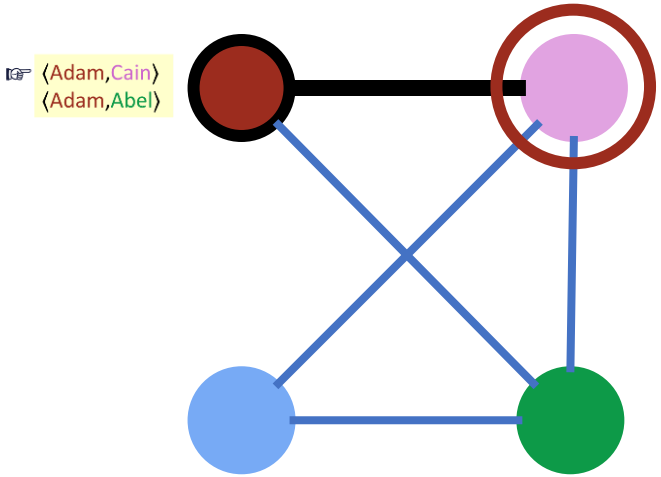
Reachability: Enumerate every node that can be reached from node n by following an edge.

```
def depth_first_search(n: node) -> None:
    """If n was never visited, enumerate it and all its unvisited relatives."""

    if n-has-never-been-visited:
        #.Enumerate n.
        for each-edge-from-n-to-m:
            depth_first_search(m)
```

True

enumeration

Adam



Cain

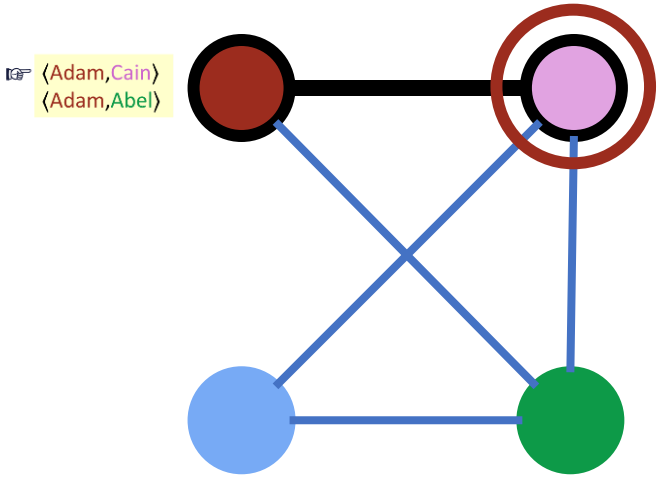
Reachability: Enumerate every node that can be reached from node n by following an edge.

```
def depth_first_search(n: node) -> None:
    """If n was never visited, enumerate it and all its unvisited relatives."""

    if n-has-never-been-visited:
        #.Enumerate n.
        for each-edge-from-n-to-m:
            depth_first_search(m)
```

enumeration

Adam

Cain



Cain

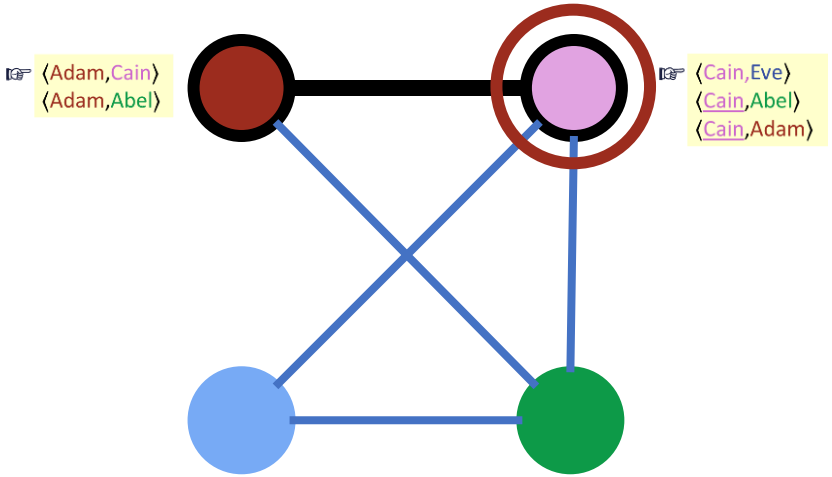
Reachability: Enumerate every node that can be reached from node n by following an edge.

```
def depth_first_search(n: node) -> None:
    """If n was never visited, enumerate it and all its unvisited relatives."""

    if n-has-never-been-visited:
        #.Enumerate n.
        for each-edge-from-n-to-m:
            depth_first_search(m)
```

☞ ⟨Cain,Eve⟩
⟨Cain,Abel⟩
⟨Cain,Adam⟩

enumeration

Adam
Cain



Eve

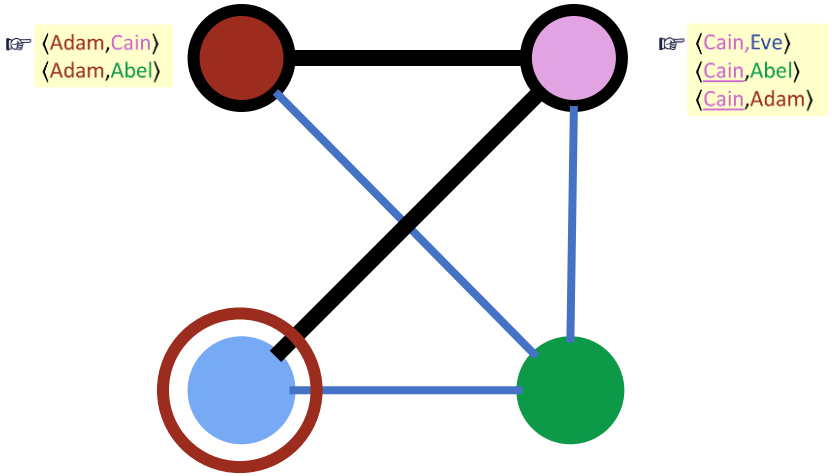
Reachability: Enumerate every node that can be reached from node n by following an edge.

```
def depth_first_search(n: node) -> None:
    """If n was never visited, enumerate it and all its unvisited relatives."""

    if n-has-never-been-visited:
        #.Enumerate n.
        for each-edge-from-n-to-m:
            depth_first_search(m)
```

enumeration

Adam

Cain



Eve

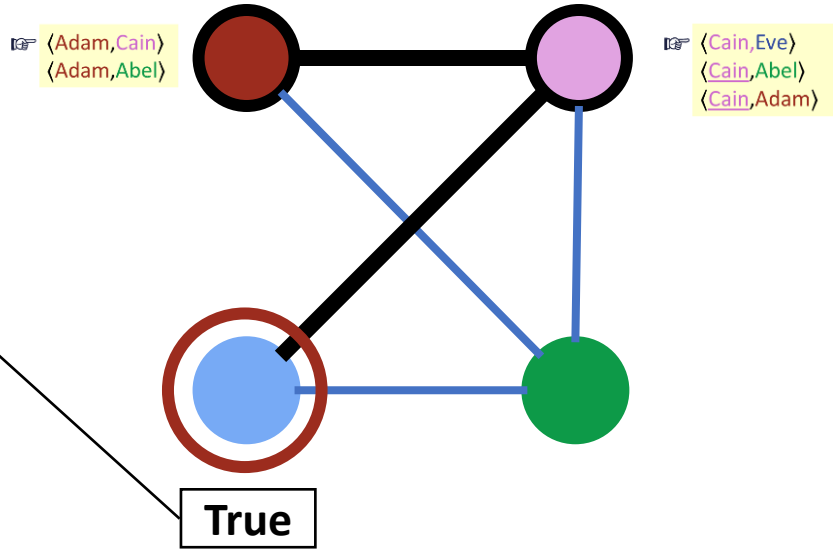
Reachability: Enumerate every node that can be reached from node n by following an edge.

```
def depth_first_search(n: node) -> None:
    """If n was never visited, enumerate it and all its unvisited relatives."""

    if n-has-never-been-visited:
        #.Enumerate n.
        for each-edge-from-n-to-m:
            depth_first_search(m)
```

enumeration

Adam

Cain



True

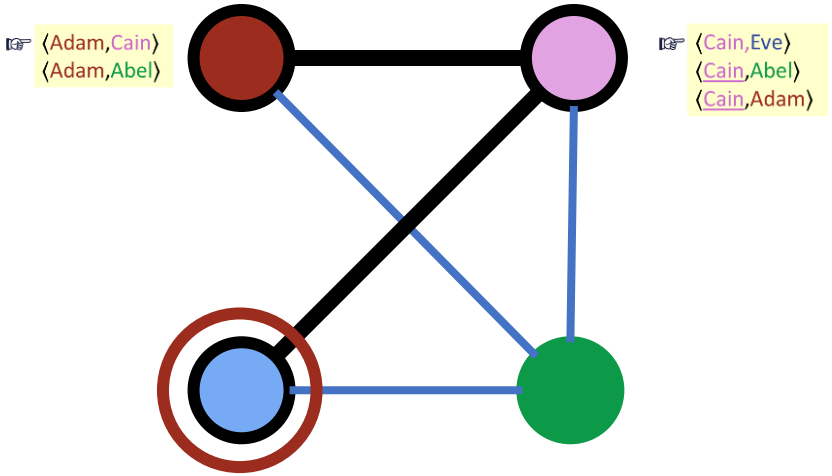
Eve

Reachability: Enumerate every node that can be reached from node n by following an edge.

```
def depth_first_search(n: node) -> None:
    """If n was never visited, enumerate it and all its unvisited relatives."""

    if n-has-never-been-visited:
        #.Enumerate n.
        for each-edge-from-n-to-m:
            depth_first_search(m)
```

enumeration

Adam
Cain
Eve



Eve

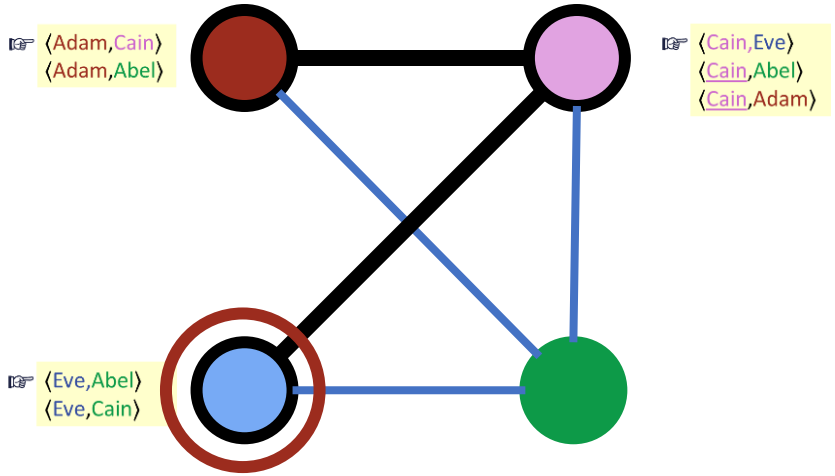
Reachability: Enumerate every node that can be reached from node n by following an edge.

```
def depth_first_search(n: node) -> None:
    """If n was never visited, enumerate it and all its unvisited relatives."""

    if n-has-never-been-visited:
        #.Enumerate n.
        for each-edge-from-n-to-m:
            depth_first_search(m)
```

☞ ⟨Eve,Abel⟩
⟨Eve,Cain⟩

enumeration

Adam
Cain
Eve



Depth-First Search

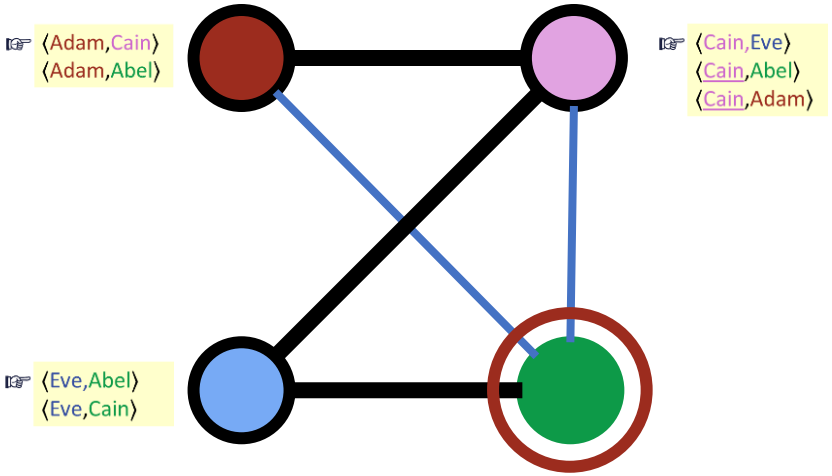
Able

Reachability: Enumerate every node that can be reached from node n by following an edge.

```
def depth_first_search(n: node) -> None:
    """If n was never visited, enumerate it and all its unvisited relatives."""

    if n-has-never-been-visited:
        #.Enumerate n.
        for each-edge-from-n-to-m:
            depth_first_search(m)
```

enumeration

Adam
Cain
Eve



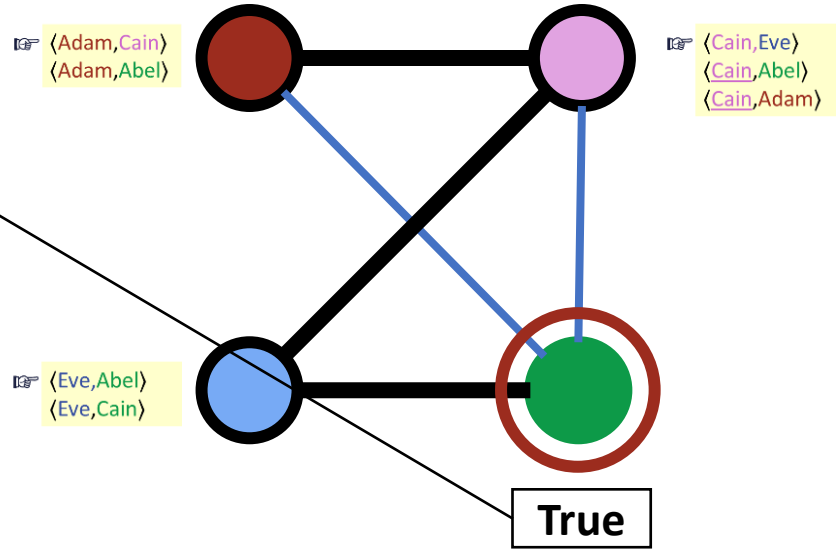
Able

Reachability: Enumerate every node that can be reached from node n by following an edge.

```
def depth_first_search(n: node) -> None:
    """If n was never visited, enumerate it and all its unvisited relatives."""

    if n-has-never-been-visited:
        #.Enumerate n.
        for each-edge-from-n-to-m:
            depth_first_search(m)
```

enumeration

Adam
Cain
Eve



True

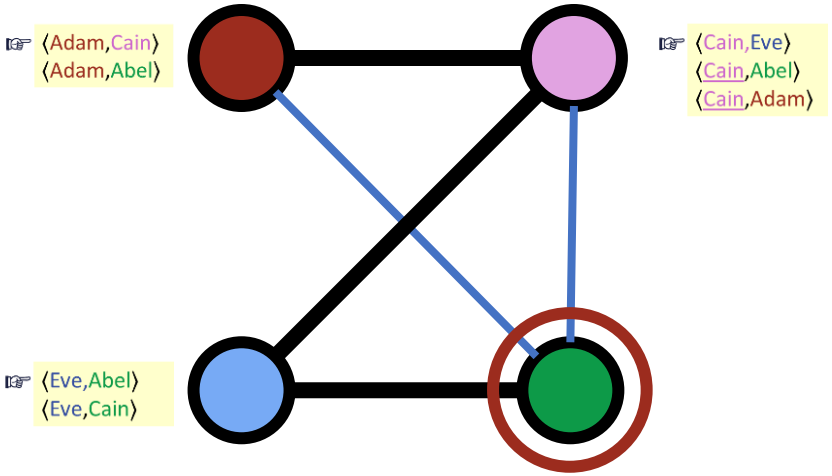
Able

Reachability: Enumerate every node that can be reached from node n by following an edge.

```
def depth_first_search(n: node) -> None:
    """If n was never visited, enumerate it and all its unvisited relatives."""

    if n-has-never-been-visited:
        #.Enumerate n.
        for each-edge-from-n-to-m:
            depth_first_search(m)
```

enumeration

Adam
Cain
Eve
Able



Able

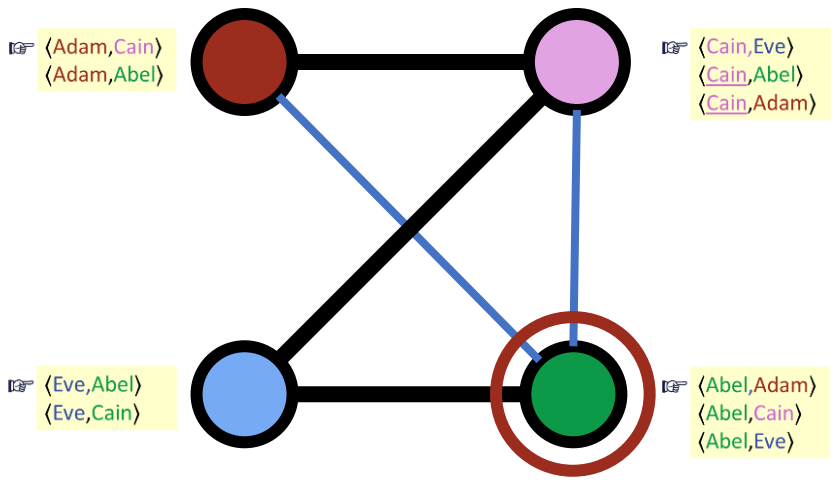
Reachability: Enumerate every node that can be reached from node n by following an edge.

```
def depth_first_search(n: node) -> None:
    """If n was never visited, enumerate it and all its unvisited relatives."""

    if n-has-never-been-visited:
        #.Enumerate n.
        for each-edge-from-n-to-m:
            depth_first_search(m)
```

☞ ⟨Abel,Adam⟩
⟨Abel,Cain⟩
⟨Abel,Eve⟩

enumeration

Adam
Cain
Eve
Able



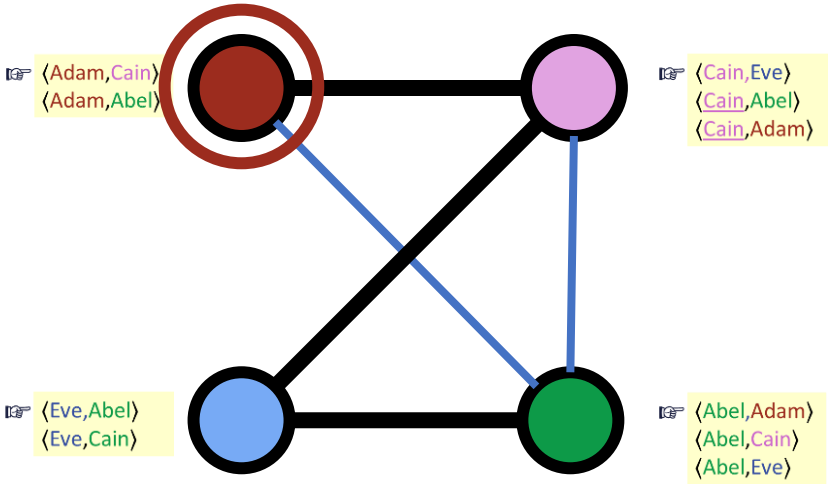
Adam

Reachability: Enumerate every node that can be reached from node n by following an edge.

```
def depth_first_search(n: node) -> None:
    """If n was never visited, enumerate it and all its unvisited relatives."""

    if n-has-never-been-visited:
        #.Enumerate n.
        for each-edge-from-n-to-m:
            depth_first_search(m)
```

enumeration

Adam
Cain
Eve
Able

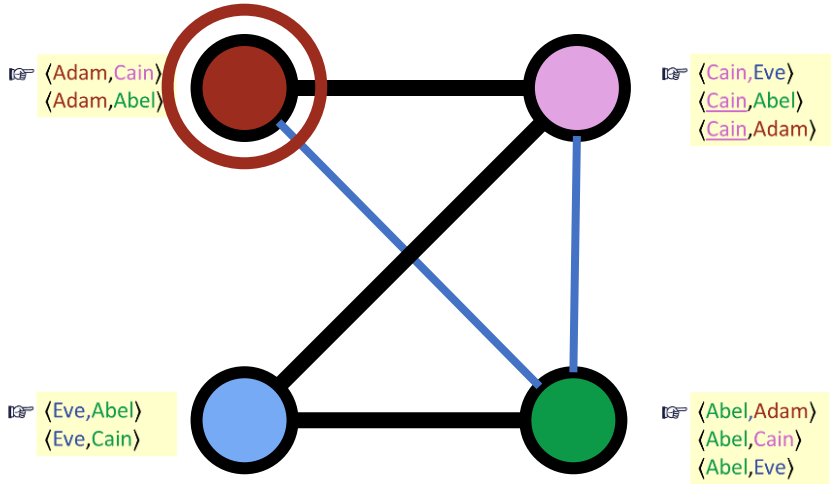


Adam

Reachability: Enumerate every node that can be reached from node n by following an edge.

```
def depth_first_search(n: node) -> None:
    """If n was never visited, enumerate it and all its unvisited relatives."""

    if n-has-never-been-visited:
        #.Enumerate n.
        for each-edge-from-n-to-m:
            depth_first_search(m)
```

False



enumeration

Adam
Cain
Eve
Able

Able

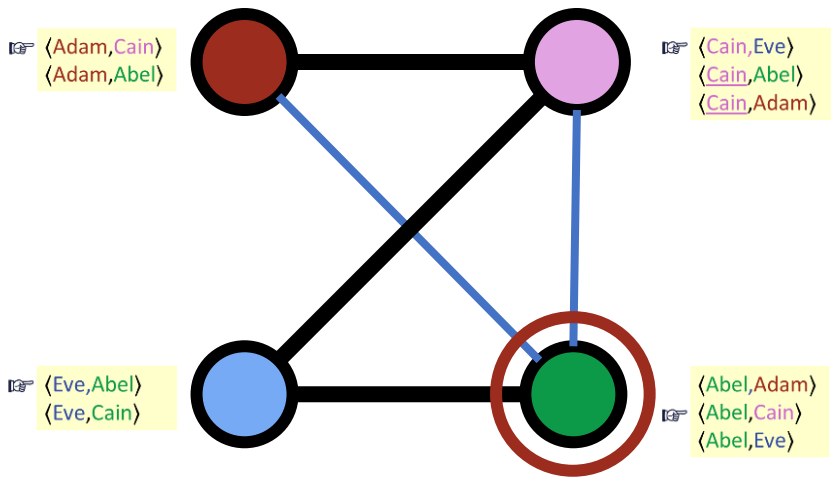
Reachability: Enumerate every node that can be reached from node n by following an edge.

```
def depth_first_search(n: node) -> None:
    """If n was never visited, enumerate it and all its unvisited relatives."""

    if n-has-never-been-visited:
        #.Enumerate n.
        for each-edge-from-n-to-m:
            depth_first_search(m)
```

☞ ⟨Abel,Adam⟩
⟨Abel,Cain⟩
⟨Abel,Eve⟩

enumeration

Adam
Cain
Eve
Able



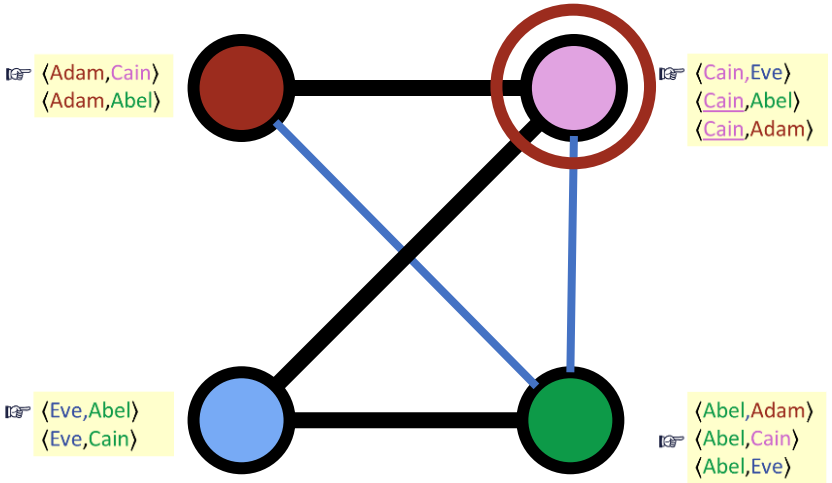
Cain

Reachability: Enumerate every node that can be reached from node n by following an edge.

```
def depth_first_search(n: node) -> None:
    """If n was never visited, enumerate it and all its unvisited relatives."""

    if n-has-never-been-visited:
        #.Enumerate n.
        for each-edge-from-n-to-m:
            depth_first_search(m)
```

enumeration

Adam
Cain
Eve
Able



Cain

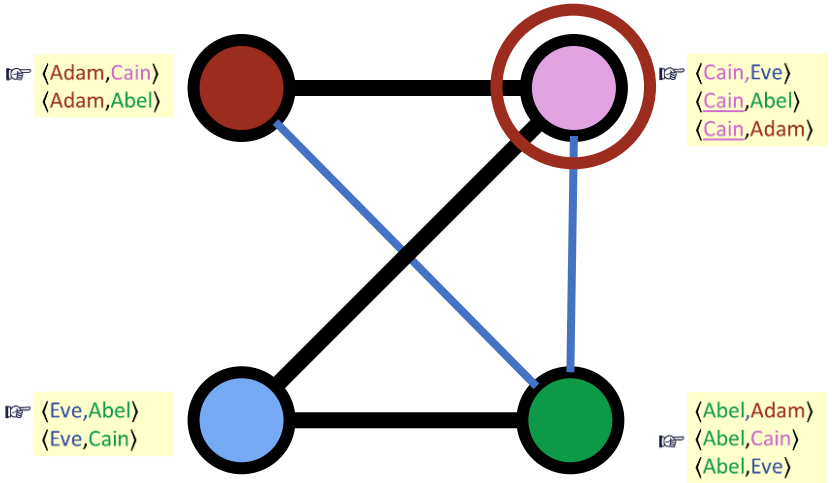
Reachability: Enumerate every node that can be reached from node n by following an edge.

```
def depth_first_search(n: node) -> None:
    """If n was never visited, enumerate it and all its unvisited relatives."""

    if n-has-never-been-visited:
        #.Enumerate n.
        for each-edge-from-n-to-m:
            depth_first_search(m)
```

False

enumeration

Adam
Cain
Eve
Able



Able

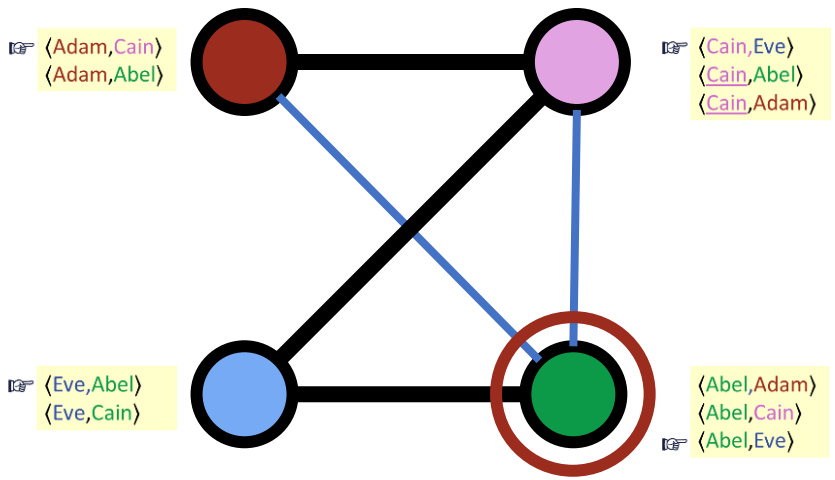
Reachability: Enumerate every node that can be reached from node n by following an edge.

```
def depth_first_search(n: node) -> None:
    """If n was never visited, enumerate it and all its unvisited relatives."""

    if n-has-never-been-visited:
        #.Enumerate n.
        for each-edge-from-n-to-m:
            depth_first_search(m)
```

⟨Abel,Adam⟩
⟨Abel,Cain⟩
☞ ⟨Abel,Eve⟩

enumeration

Adam
Cain
Eve
Able



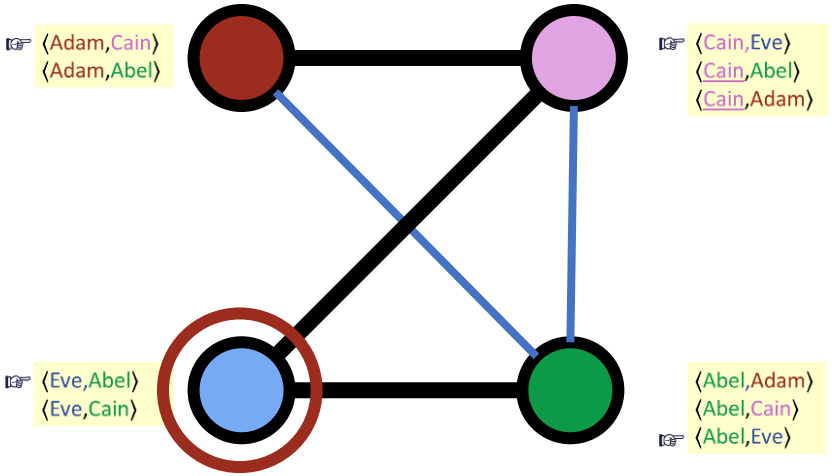
Eve

Reachability: Enumerate every node that can be reached from node n by following an edge.

```
def depth_first_search(n: node) -> None:
    """If n was never visited, enumerate it and all its unvisited relatives."""

    if n-has-never-been-visited:
        #.Enumerate n.
        for each-edge-from-n-to-m:
            depth_first_search(m)
```

enumeration

Adam
Cain
Eve
Able



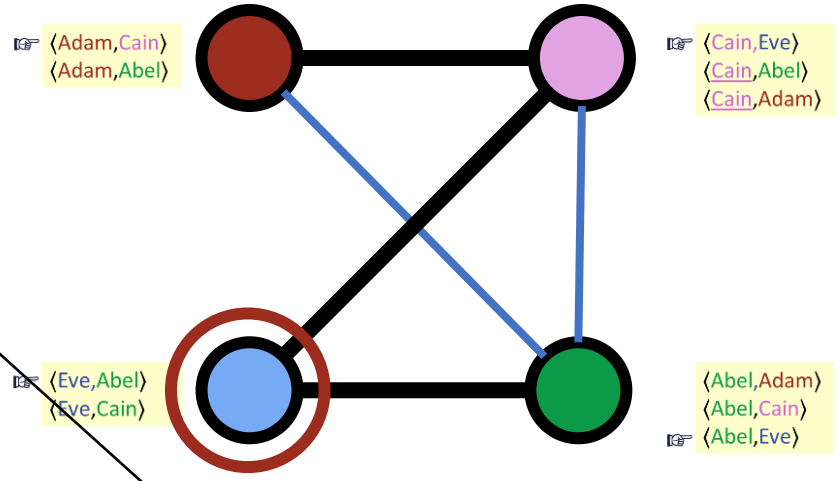
Eve

Reachability: Enumerate every node that can be reached from node n by following an edge.

```
def depth_first_search(n: node) -> None:
    """If n was never visited, enumerate it and all its unvisited relatives."""

    if n-has-never-been-visited:
        #.Enumerate n.
        for each-edge-from-n-to-m:
            depth_first_search(m)
```

enumeration

Adam
Cain
Eve
Able

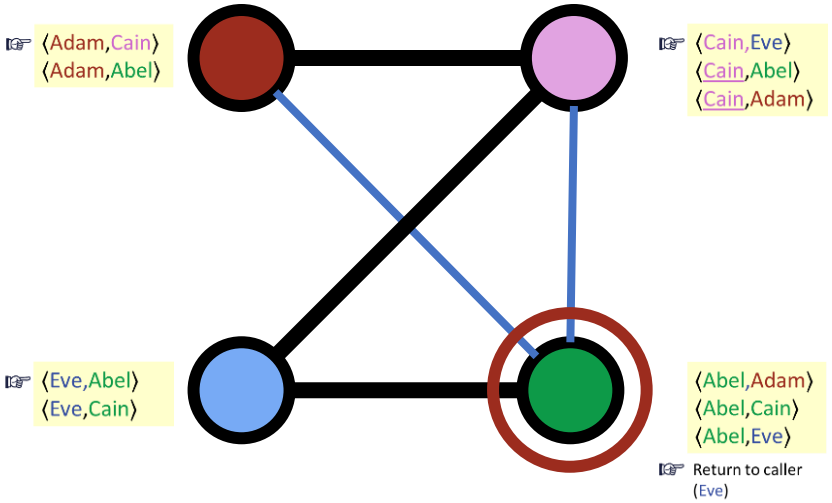


False

Able

Reachability: Enumerate every node that can be reached from node n by following an edge.

```
def depth_first_search(n: node) -> None:
    """If n was never visited, enumerate it and all its unvisited relatives."""

    if n-has-never-been-visited:
        #.Enumerate n.
        for each-edge-from-n-to-m:
            depth_first_search(m)
```



enumeration

Adam
Cain
Eve
Able

⟨Abel,Adam⟩
⟨Abel,Cain⟩
⟨Abel,Eve⟩

☞ Return to caller
(Eve)

Eve

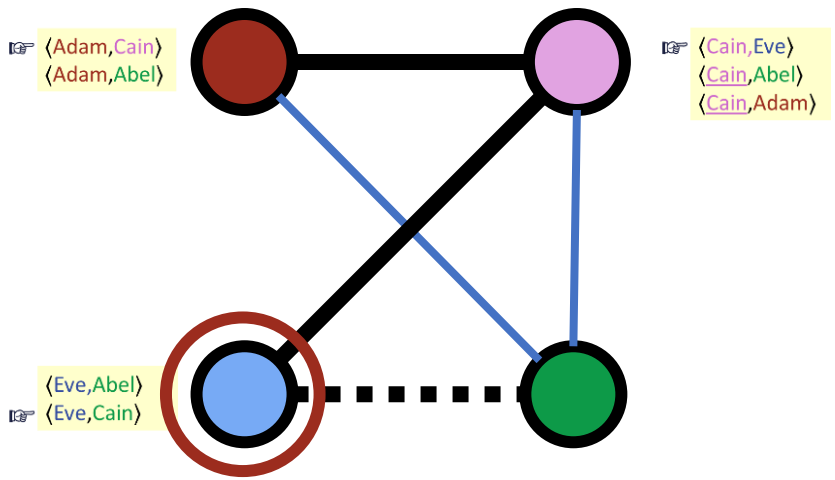
Reachability: Enumerate every node that can be reached from node n by following an edge.

```
def depth_first_search(n: node) -> None:
    """If n was never visited, enumerate it and all its unvisited relatives."""

    if n-has-never-been-visited:
        #.Enumerate n.
        for each-edge-from-n-to-m:
            depth_first_search(m)
```

⟨Eve,Abel⟩
☞ ⟨Eve,Cain⟩

enumeration

Adam
Cain
Eve
Able



▪▪▪▪▪▪▪▪▪▪ Means “first visitor finished”

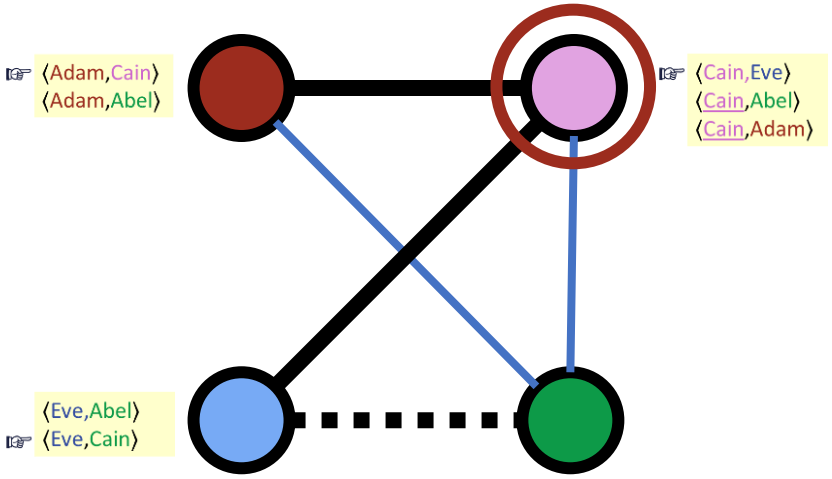
Cain

Reachability: Enumerate every node that can be reached from node n by following an edge.

```
def depth_first_search(n: node) -> None:
    """If n was never visited, enumerate it and all its unvisited relatives."""

    if n-has-never-been-visited:
        #.Enumerate n.
        for each-edge-from-n-to-m:
            depth_first_search(m)
```

enumeration

Adam
Cain
Eve
Able



Depth-First Search

Cain

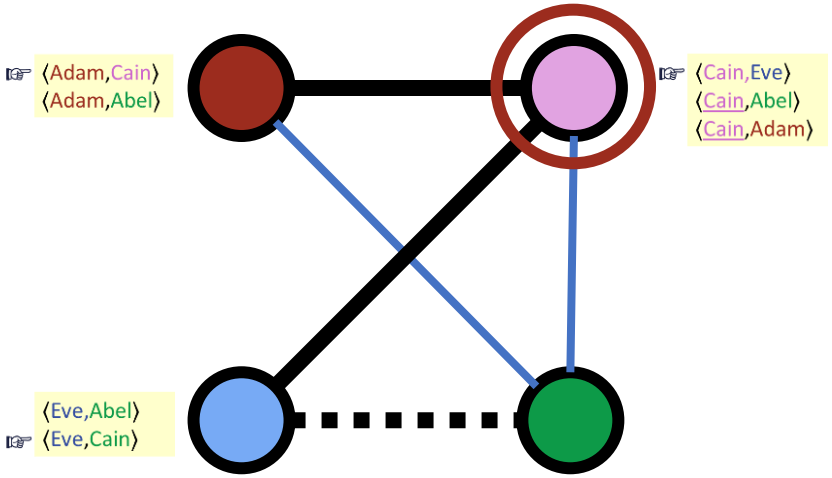
Reachability: Enumerate every node that can be reached from node n by following an edge.

```
def depth_first_search(n: node) -> None:
    """If n was never visited, enumerate it and all its unvisited relatives."""

    if n-has-never-been-visited:
        #.Enumerate n.
        for each-edge-from-n-to-m:
            depth_first_search(m)
```

False

enumeration

Adam
Cain
Eve
Able



Eve

Reachability: Enumerate every node that can be reached from node n by following an edge.

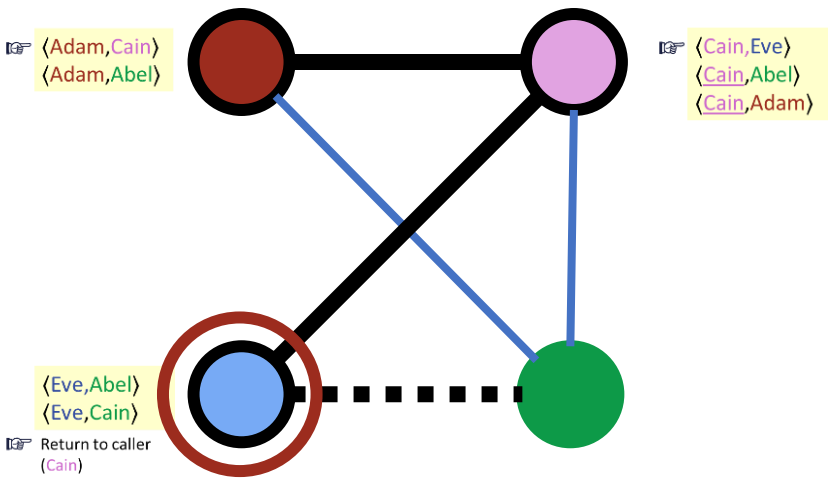
```
def depth_first_search(n: node) -> None:
    """If n was never visited, enumerate it and all its unvisited relatives."""

    if n-has-never-been-visited:
        #.Enumerate n.
        for each-edge-from-n-to-m:
            depth_first_search(m)
```

⟨Eve,Abel⟩
⟨Eve,Cain⟩

☞ Return to caller
(Cain)

enumeration

Adam
Cain
Eve
Able



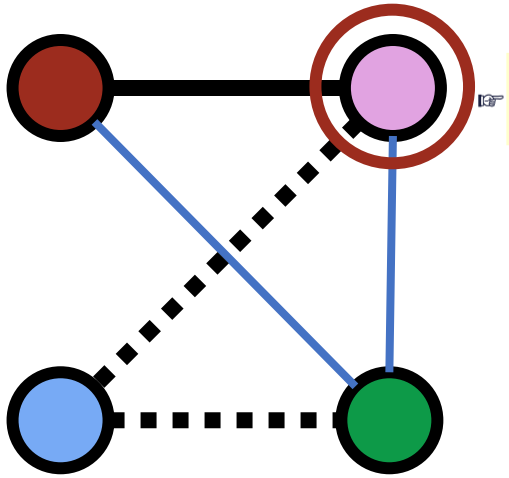
Depth-First Search

Cain

Reachability: Enumerate every node that can be reached from node n by following an edge.

```
def depth_first_search(n: node) -> None:
    """If n was never visited, enumerate it and all its unvisited relatives."""

    if n-has-never-been-visited:
        #.Enumerate n.
        for each-edge-from-n-to-m:
            depth_first_search(m)
```

☞ ⟨Adam,Cain⟩
⟨Adam,Abel⟩



☞ ⟨Cain,Eve⟩
⟨Cain,Abel⟩
⟨Cain,Adam⟩

⟨Cain,Eve⟩
☞ ⟨Cain,Abel⟩
⟨Cain,Adam⟩

enumeration

Adam
Cain
Eve
Able

Depth-First Search

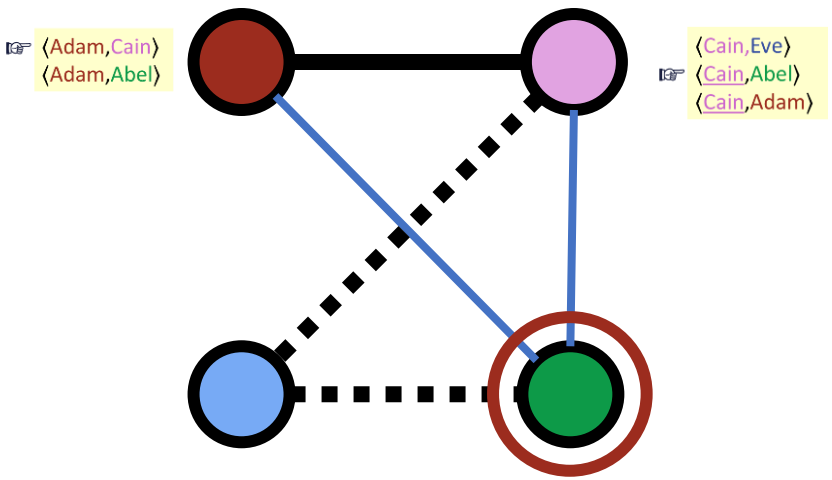
Able

Reachability: Enumerate every node that can be reached from node n by following an edge.

```
def depth_first_search(n: node) -> None:
    """If n was never visited, enumerate it and all its unvisited relatives."""

    if n-has-never-been-visited:
        #.Enumerate n.
        for each-edge-from-n-to-m:
            depth_first_search(m)
```

enumeration

Adam
Cain
Eve
Able



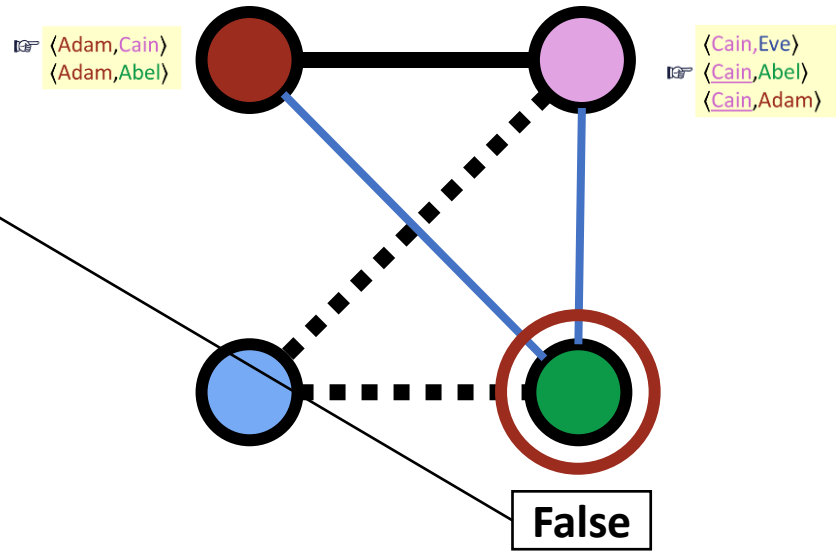
Able

Reachability: Enumerate every node that can be reached from node n by following an edge.

```
def depth_first_search(n: node) -> None:
    """If n was never visited, enumerate it and all its unvisited relatives."""

    if n-has-never-been-visited:
        #.Enumerate n.
        for each-edge-from-n-to-m:
            depth_first_search(m)
```

enumeration

Adam
Cain
Eve
Able



Cain

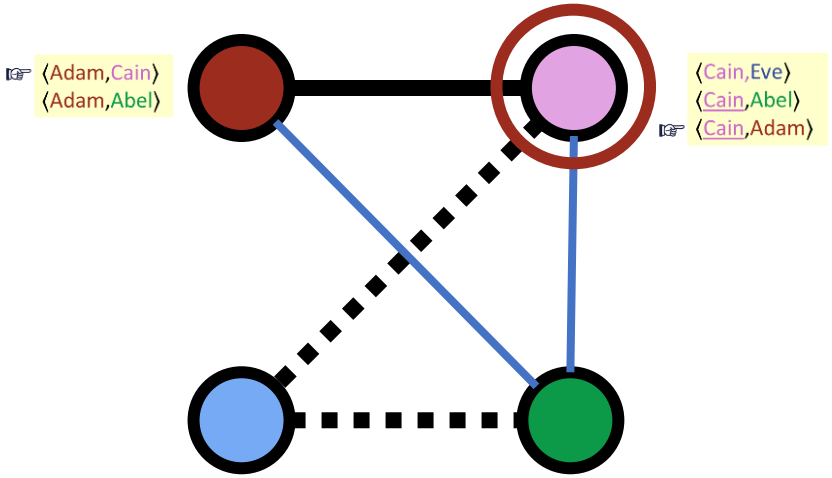
Reachability: Enumerate every node that can be reached from node n by following an edge.

```
def depth_first_search(n: node) -> None:
    """If n was never visited, enumerate it and all its unvisited relatives."""

    if n-has-never-been-visited:
        #.Enumerate n.
        for each-edge-from-n-to-m:
            depth_first_search(m)
```

⟨Cain,Eve⟩
⟨Cain,Abel⟩
☞ ⟨Cain,Adam⟩

enumeration

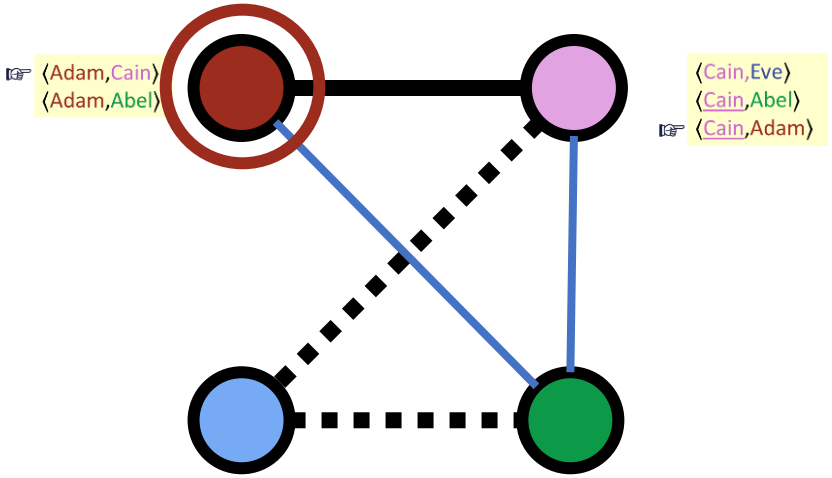
Adam
Cain
Eve
Able



Adam

Reachability: Enumerate every node that can be reached from node n by following an edge.

```
def depth_first_search(n: node) -> None:
    """If n was never visited, enumerate it and all its unvisited relatives."""

    if n-has-never-been-visited:
        #.Enumerate n.
        for each-edge-from-n-to-m:
            depth_first_search(m)
```



enumeration

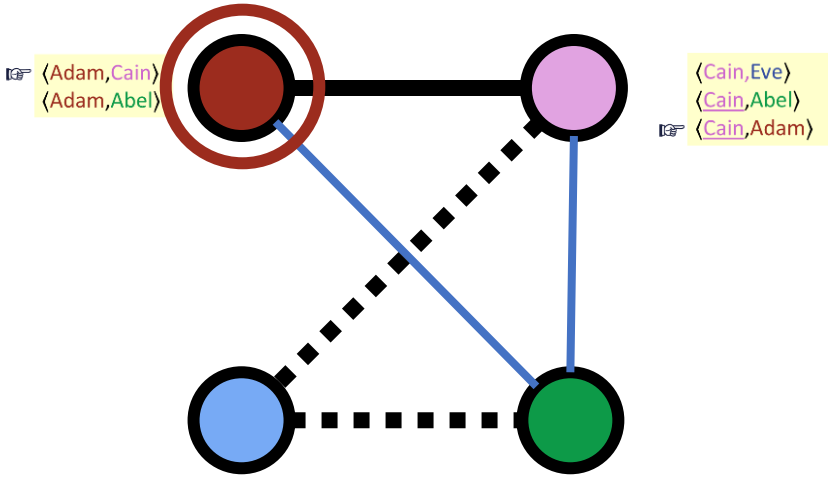
Adam
Cain
Eve
Able

Adam

Reachability: Enumerate every node that can be reached from node n by following an edge.

```
def depth_first_search(n: node) -> None:
    """If n was never visited, enumerate it and all its unvisited relatives."""

    if n-has-never-been-visited:
        #.Enumerate n.
        for each-edge-from-n-to-m:
            depth_first_search(m)
```

False



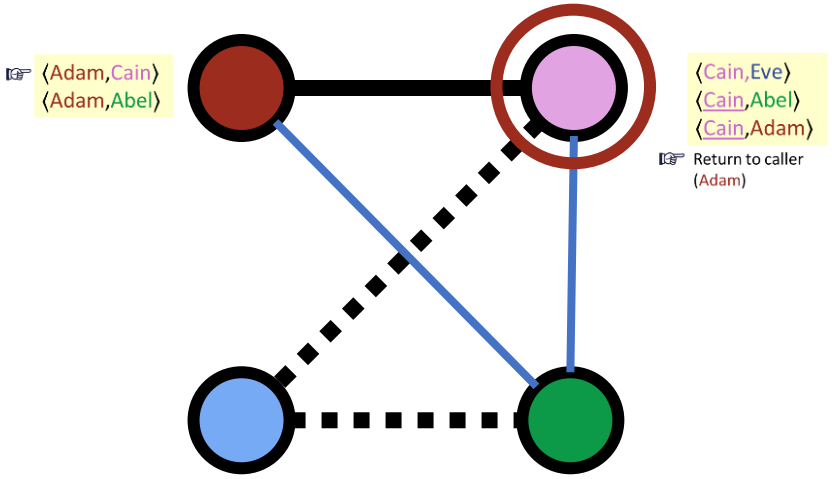
enumeration

Adam
Cain
Eve
Able

Cain

Reachability: Enumerate every node that can be reached from node n by following an edge.

```
def depth_first_search(n: node) -> None:
    """If n was never visited, enumerate it and all its unvisited relatives."""

    if n-has-never-been-visited:
        #.Enumerate n.
        for each-edge-from-n-to-m:
            depth_first_search(m)
```



⟨Cain,Eve⟩
⟨Cain,Abel⟩
⟨Cain,Adam⟩

☞ Return to caller (Adam)

enumeration

Adam
Cain
Eve
Able

Adam

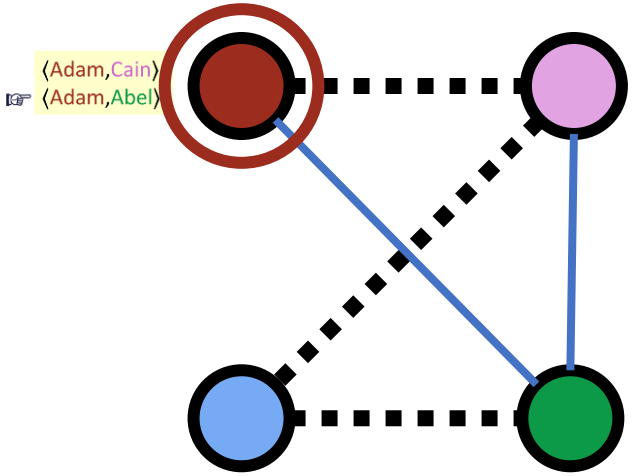
Reachability: Enumerate every node that can be reached from node n by following an edge.

```
def depth_first_search(n: node) -> None:
    """If n was never visited, enumerate it and all its unvisited relatives."""

    if n-has-never-been-visited:
        #.Enumerate n.
        for each-edge-from-n-to-m:
            depth_first_search(m)
```

⟨Adam,Cain⟩
☞ ⟨Adam,Abel⟩

enumeration

Adam
Cain
Eve
Able



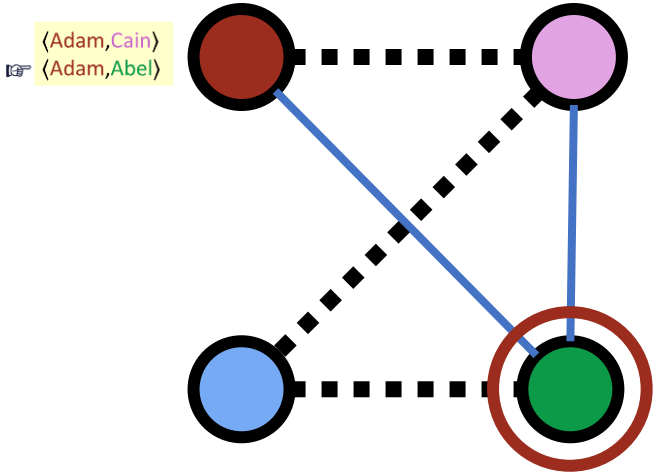
Able

Reachability: Enumerate every node that can be reached from node n by following an edge.

```
def depth_first_search(n: node) -> None:
    """If n was never visited, enumerate it and all its unvisited relatives."""

    if n-has-never-been-visited:
        #.Enumerate n.
        for each-edge-from-n-to-m:
            depth_first_search(m)
```

enumeration

Adam
Cain
Eve
Able



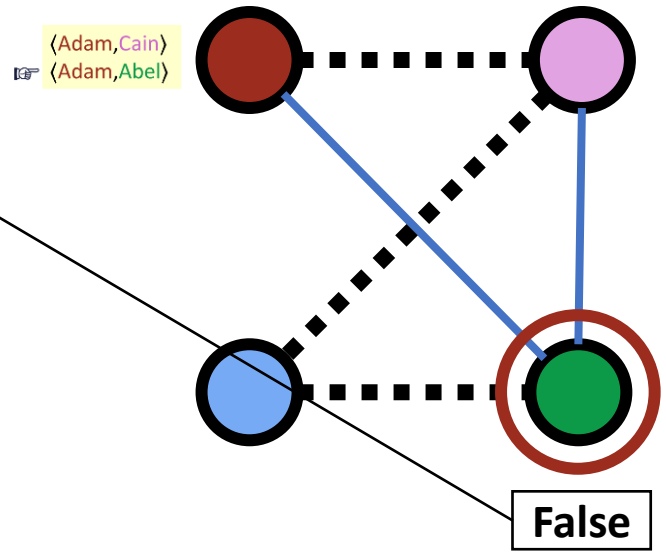
Able

Reachability: Enumerate every node that can be reached from node n by following an edge.

```
def depth_first_search(n: node) -> None:
    """If n was never visited, enumerate it and all its unvisited relatives."""

    if n-has-never-been-visited:
        #.Enumerate n.
        for each-edge-from-n-to-m:
            depth_first_search(m)
```

enumeration

Adam
Cain
Eve
Able



Adam

Reachability: Enumerate every node that can be reached from node n by following an edge.

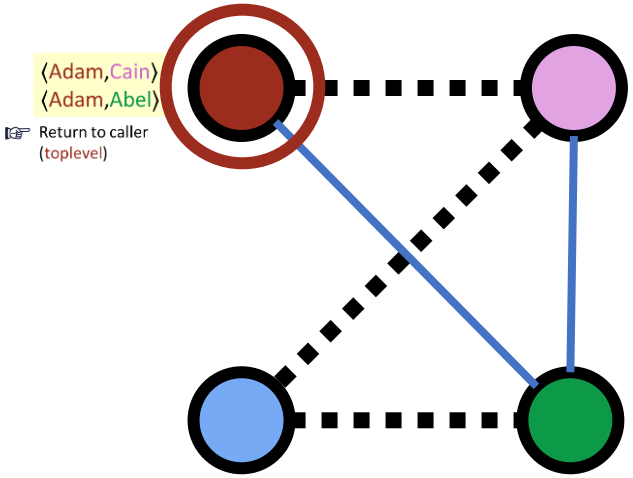
```
def depth_first_search(n: node) -> None:
    """If n was never visited, enumerate it and all its unvisited relatives."""

    if n-has-never-been-visited:
        #.Enumerate n.
        for each-edge-from-n-to-m:
            depth_first_search(m)
```

⟨Adam,Cain⟩
⟨Adam,Abel⟩

☞ Return to caller (toplevel)

enumeration

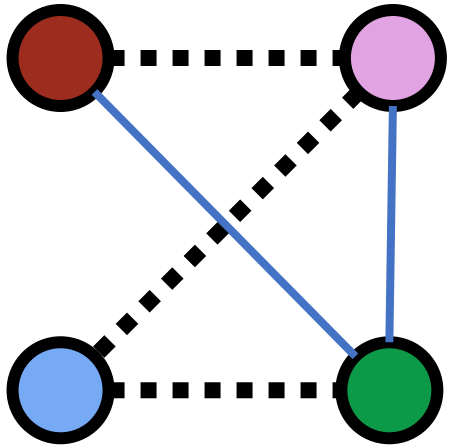
Adam
Cain
Eve
Able



Depth-First Search

Reachability: Enumerate every node that can be reached from node n by following an edge.

```
def depth_first_search(n: node) -> None:
    """If n was never visited, enumerate it and all its unvisited relatives."""

    if n-has-never-been-visited:
        #.Enumerate n.
        for each-edge-from-n-to-m:
            depth_first_search(m)
```



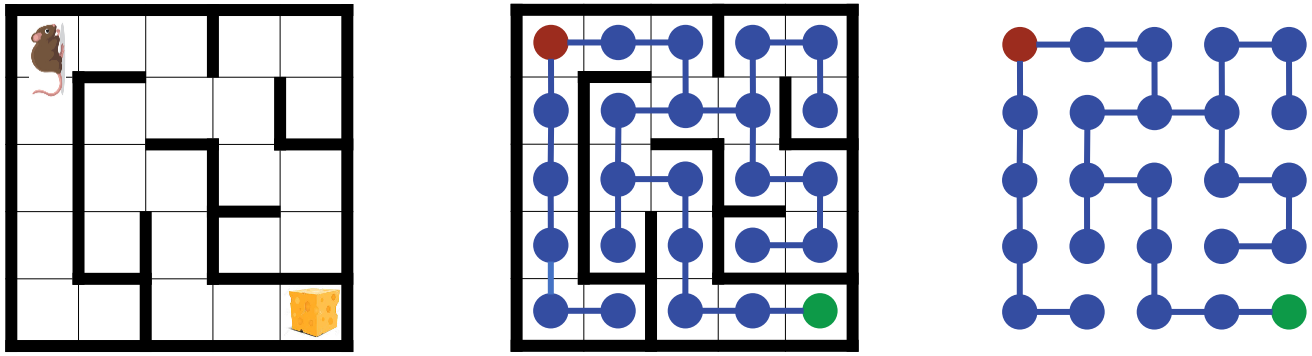
DONE

Q. What is Depth-First Search searching for?

A. It is just a way to visit all reachable nodes from n.
You can do anything you want when you get there.

Running a Maze, Revisited

Maze as Undirected Graph: cells are nodes, and open doorways are edges.

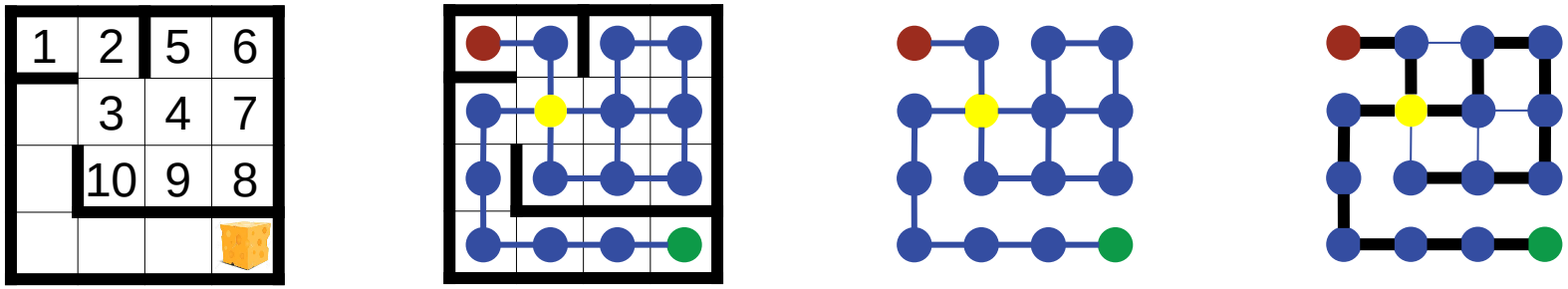


To solve the maze, perform `depth-first-search(upper-left-cell)`.
Stop if you encounter the `lower-right-cell`.

Reachability between two cells of a maze is reachability between two nodes of a graph.

Running a Maze, Revisited

Domain-Specific Subtleties: Gone.

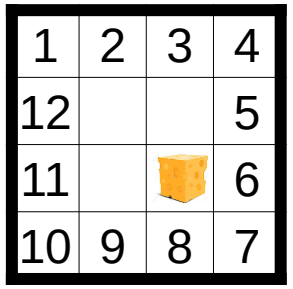


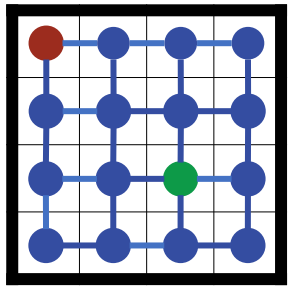
Recall the distinction been corridor-like cul-de-sacs and room-like cul-de-sacs. Gone.

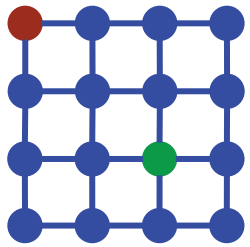
Recall the question of how to back out of a cul-de-sac, and when to stop. Gone

Running a Maze, Revisited

Finding Centrally-Located Cheese : No problem.

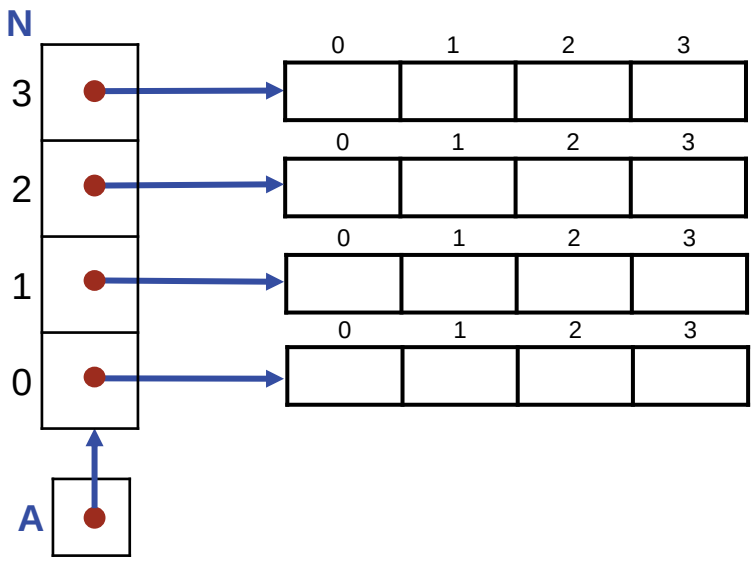






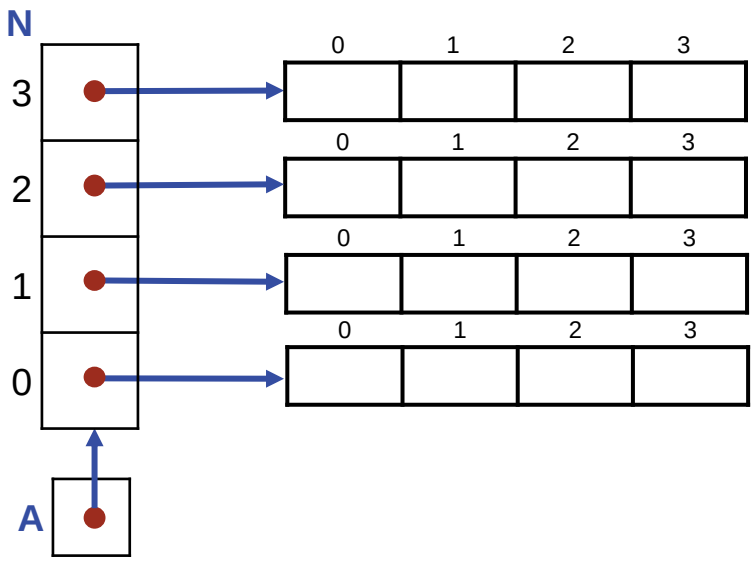
Regardless of the cheese's location, the problem is just graph reachability, and can be solved by Depth-First Search.

Representation: Recall that a 2-D array is really a 1-D array of 1-D arrays.



For example, the N-by-N square array A, for N=4, would be as shown.

Representation: Recall that a 2-D array is really a 1-D array of 1-D arrays.*

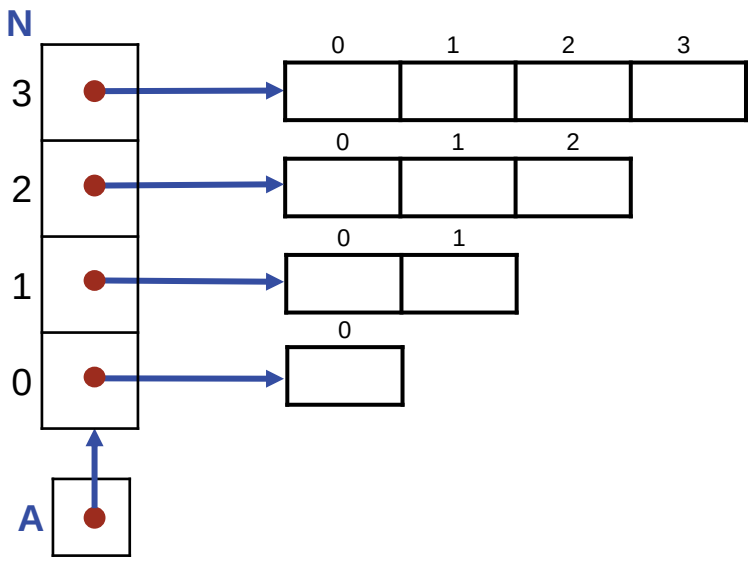


For example, the N-by-N square array A, for N=4, would be as shown.

***C/C++**

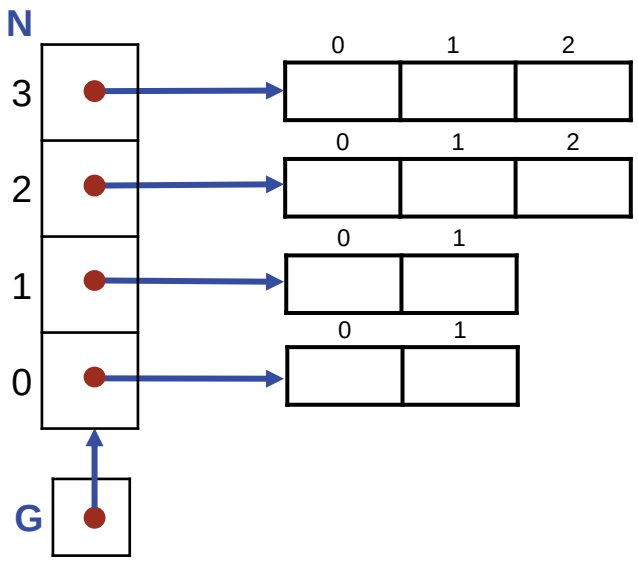
This section is not valid for C-style arrays in C/C++. Rather, it can be read as describing one of the alternatives to C-style arrays that are available in C++.

Representation: Recall, also, that each row can have a different number of columns.



For example, the closed triangular array inscribed in a 4-by-4 square would be as shown.

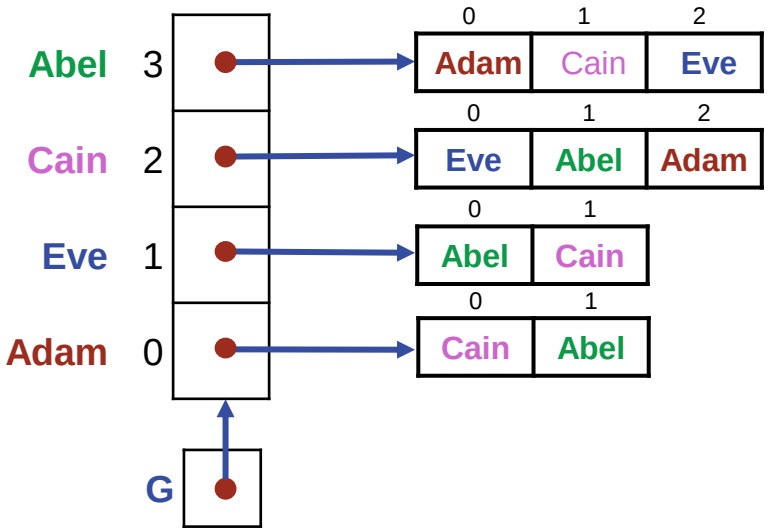
Representation: A 2-D array can be used to represent a graph G with N nodes.

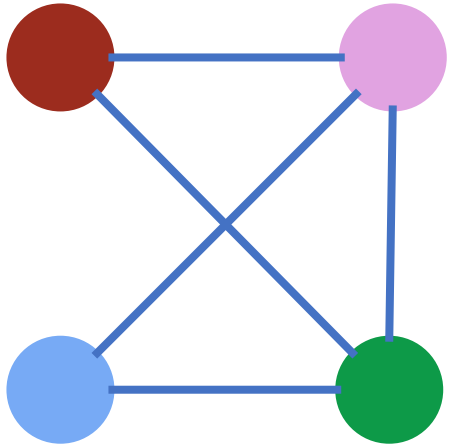


Number the nodes 0 through N-1.

Let `G[0..N-1]` be *edge lists*, i.e., `G[`*n*`]` is a `1-D` **`int`** array that contain the target nodes of edges emanating from node *n*.

Representation: A 2-D array can be used to represent a graph with N nodes. For example:

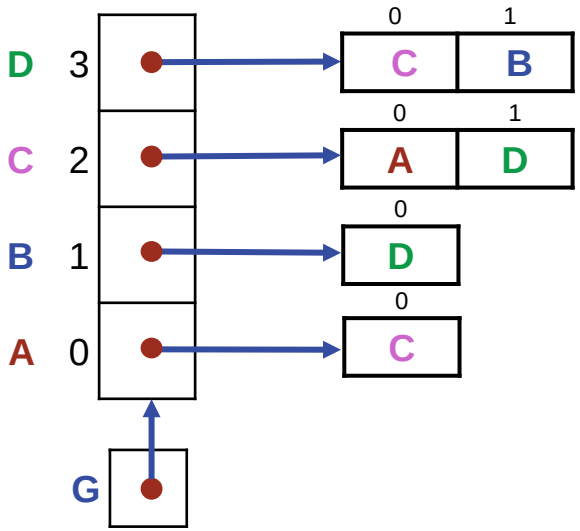




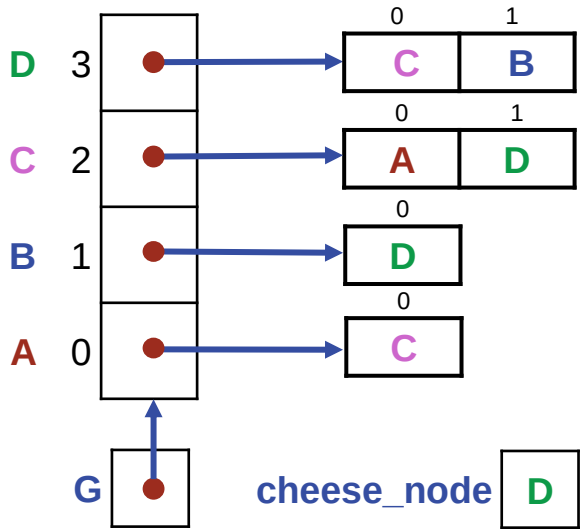
Number the nodes 0 through N-1.

Let `G[0..N-1]` be *edge lists*, i.e., `G[`*n*`]` is a **`1-D int`** array that contain the target nodes of edges emanating from node *n*. The order of nodes in an edge list is irrelevant.

Representation: and here is the representation of the 2-by-2 maze shown:



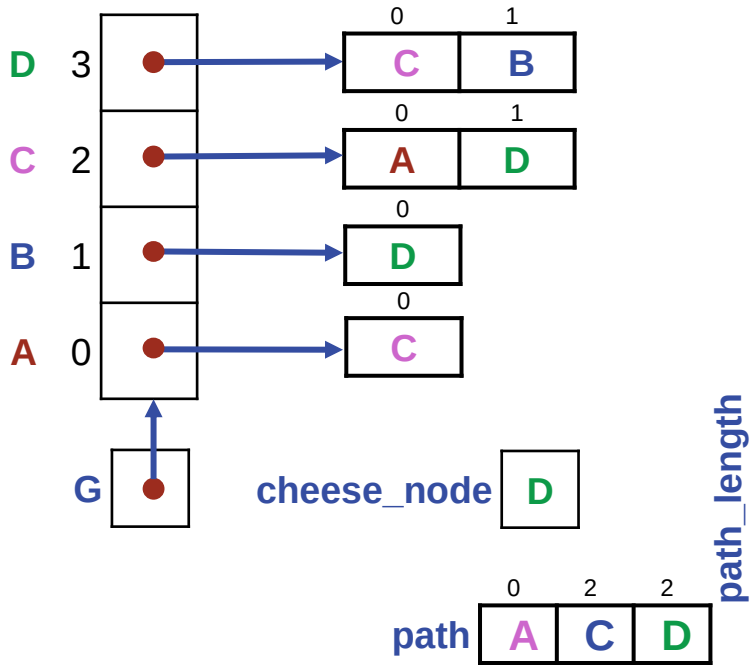
Representation: invariant.




```
# Maze, Rat, and Path (MRP) Representations.
class MRP:
    # Maze. Maze cells are represented by N*N nodes
    #   of graph G, where G[n] is an edge list for node
    #   n, i.e., for 0≤e<len(G[n]), G[n][e] is an
    #   adjacent node m, i.e., a cell m adjacent to n
    #   with intervening Wall. The upper-left cell is
    #   node 0. Cheese is at cheese_node.
    _G: list[list[int]]                # Edge lists.
    _cheese_node: int                  # Cheese.
    ...
```

Running a Maze, Revisited

Representation: invariant.




```
# Maze, Rat, and Path (MRP) Representations.
class MRP:
    # Maze. Maze cells are represented by N*N nodes
    #   of graph G, where G[n] is an edge list for node
    #   n, i.e., for 0≤e<len(G[n]), G[n][e] is an
    #   adjacent node m, i.e., a cell m adjacent to n
    #   with intervening Wall. The upper-left cell is
    #   node 0. Cheese is at cheese_node.
    _G: list[list[int]]                # Edge lists.
    _cheese_node: int                  # Cheese.
    # Path. Array path[0..path_length-1] is a list of
    #   adjacent nodes in G reaching from node 0 to some
    #   node path[path_length-1].
    _path: list[int]
    _path_length: int
    def isAtCheese() -> bool:
        return MRP._path[MRP._path_length - 1
                         ] == MRP._cheese_node
    ...
```

Running a Maze, Revisited

Representation: Depth-First Search.

```
# Maze, Rat, and Path (MRP) Representations.
class MRP:
    _mark: list[bool]                          # mark[n] iff DFS reached node n.
    # Depth First Search (DFS) of node n for cheese_node.
    def DFS(n: int) -> None:
        if not MRP._mark[n]:                   # Node n has not been visited before.
            MRP._mark[n] = True                # Mark that n has been visited.
            for e in range(0, len(MRP._G[n])): DFS(MRP._G[n][e])
   ...
```

Running a Maze, Revisited

Representation: Depth-First Search, with path.

```
# Maze, Rat, and Path (MRP) Representations.
class MRP:
    _mark: list[bool]                          # mark[n] iff DFS reached node n.
    # Depth First Search (DFS) of node n for cheese_node at depth p.
    def DFS(n: int, p: int) -> None:
        if not MRP._mark[n]:                   # Node n has not been visited before.
            MRP._mark[n] = True                # Mark that n has been visited.
            MRP._path[p] = n                   # Extend the path to include n.
            for e in range(0, len(MRP._G[n])): DFS(MRP._G[n][e], p + 1)
    ...
```

Running a Maze, Revisited

Representation: Depth-First Search, with path, and early termination if cheese is found.

```
# Maze, Rat, and Path (MRP) Representations.
class MRP:
    _mark: list[bool]                          # mark[n] iff DFS reached node n.
    # Depth First Search (DFS) of node n for cheese_node at depth p.
    def DFS(n: int, p: int) -> None:
        if not MRP._mark[n]:                   # Node n has not been visited before.
            MRP._mark[n] = True                # Mark that n has been visited.
            MRP._path[p] = n                   # Extend the path to include n.
            if n == MRP._cheese_node:          # Terminate search if cheese found.
                MRP._path_length = p + 1       # Length of path is one longer than p.
                raise Exception("found cheese")
            for e in range(0, len(MRP._G[n])): DFS(MRP._G[n][e], p + 1)
    ...
```

_DFS is a class method, and _G, _cheese_node, _mark, _path, and _path_length are protected class variables.

Representation: Depth-First Search, with path, and early termination if cheese is found.

```
# Maze, Rat, and Path (MRP) Representations.
class MRP:
    ...
    _mark: list[bool]                          # mark[n] iff DFS reached node n.
    # Depth First Search (DFS) of node n for cheese_node at depth p.
    @classmethod
    def _DFS(cls, n: int, p: int) -> None:
        if not MRP._mark[n]:                   # Node n has not been visited before.
            MRP._mark[n] = True                # Mark that n has been visited.
            MRP._path[p] = n                   # Extend the path to include n.
            if n == MRP._cheese_node:          # Terminate search if cheese found.
                MRP._path_length = p + 1        # Length of path is one longer than p.
                raise Exception("found cheese")
            for e in range(0, len(MRP._G[n])): MRP._DFS(MRP._G[n][e], p + 1)
    ...
```

If cheese is found, the **raise** in DFS is executed, which terminates all DFS invocations and is then caught by this **except**. If cheese is not found, DFS will return to the **try** normally.

Representation: The top-level call to DFS.

```
# Maze, Rat, and Path (MRP) Representations.
class MRP:
    ...
    # Convert representation M[N][N] to graph G, then perform DFS from upper-left,
    #   then convert computed path to representation M[N][N].
    def search(cls) -> None:
        MRP.make_graph_from_input()
        try:
            MRP._DFS(0,0)
        except Exception:
            pass
        MRP.make_output_from_path()
    ...
```

Methods `make_graph_from_input` and `make_output_from_path` must mediate between the geometric layout of an N-by-N Maze and the arbitrary ordering of graph nodes numbered 0..N*N-1. It can do so by using a row-major ordering of the maze cells. (See text.)

Reflection:

The simplicity of Depth-First Search compared with the subtleties of the domain-specific analyses in which we engaged is dramatic, and should inspire your study of graph algorithms.