Principled Programming

Introduction to Coding in Any Imperative Language

Tim Teitelbaum

Emeritus Professor
Department of Computer Science
Cornell University

Graphs and Depth-First Search

Introduction

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**

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

Example: **has-parent**

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

Directed Graphs:

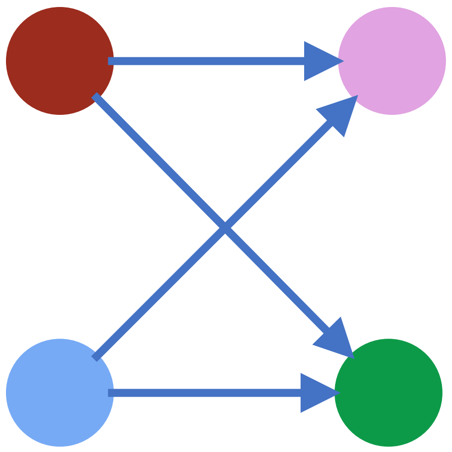
It is convenient to visualize a relation between a set *S* and itself as a collection of *nodes* and *edges*.

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.

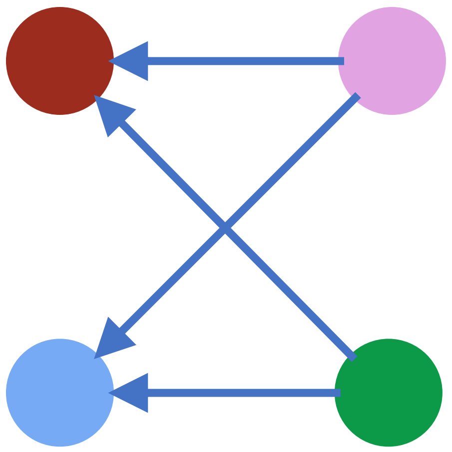
Such a visualization is known as a *directed graph*.

Example: **has-child**

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

Example: **has-parent**

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

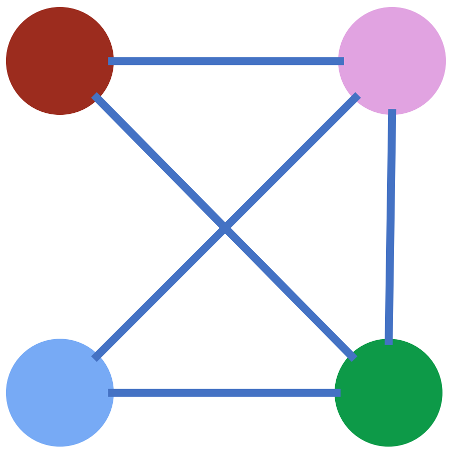


Undirected Graphs:

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. */
void DepthFirstSearch(node n) {
   if ( /* n has never been visited */ ) {
      /* Enumerate n. */
      for ( /* each edge ⟨n,m⟩ */ )
         DepthFirstSearch(m);
      }
   } /* DepthFirstSearch */
```

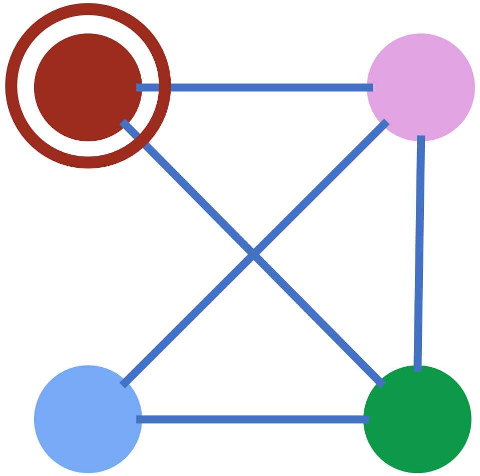
Although the definition is simple, its import is not necessarily readily apparent.
The following trace of its execution makes it clear.

Depth-First Search

Adam

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. */
void DepthFirstSearch(node n) {
   if ( /* n has never been visited */ ) {
      /* Enumerate n. */
      for ( /* each edge ⟨n,m⟩ */ )
         DepthFirstSearch(m);
      }
   } /* DepthFirstSearch */
```



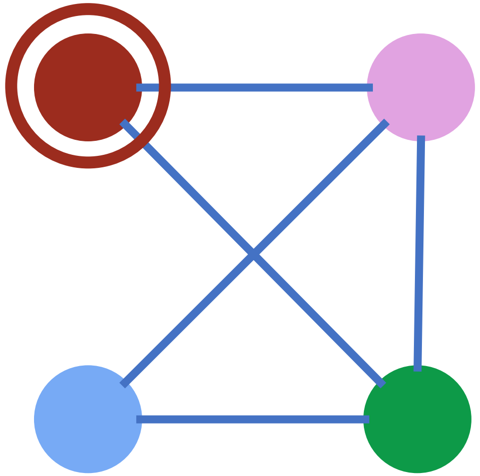
Depth-First Search

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

Adam

```
/* If n was never visited, enumerate it and all its unvisited relatives. */
void DepthFirstSearch(node n) {
   if ( /* n has never been visited */ ) {
      /* Enumerate n. */
      for ( /* each edge ⟨n,m⟩ */ )
         DepthFirstSearch(m);
      }
   } /* DepthFirstSearch */
```

true



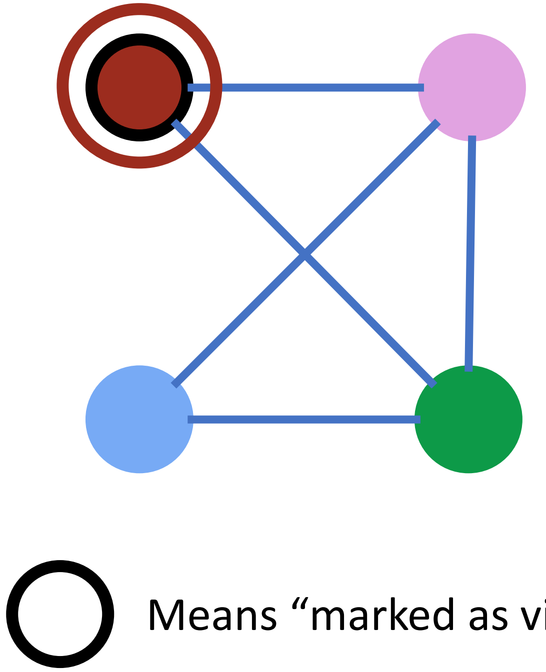
Adam

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. */
void DepthFirstSearch(node n) {
   if ( /* n has never been visited */ ) {
      /* Enumerate n. */
      for ( /* each edge ⟨n,m⟩ */ )
         DepthFirstSearch(m);
      }
   } /* DepthFirstSearch */
```

enumeration

Adam



Means “marked as visited”

Depth-First Search

Adam

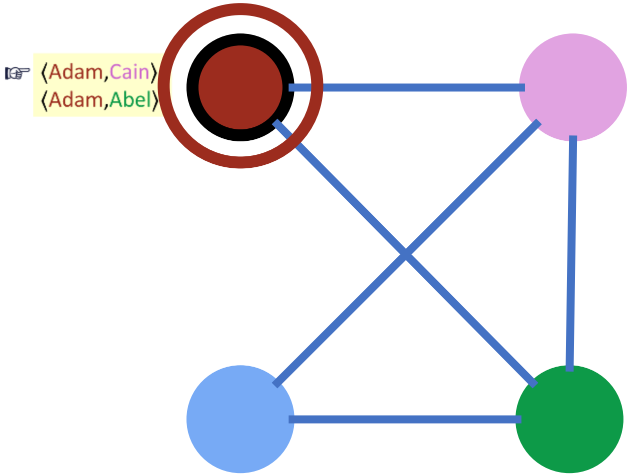
Reachability: Enumerate every node that can be reached from node n by following an edge.

```
/* If n was never visited, enumerate it and all its unvisited relatives. */
void DepthFirstSearch(node n) {
   if ( /* n has never been visited */ ) {
      /* Enumerate n. */
      for ( /* each edge ⟨n,m⟩ */ )
         DepthFirstSearch(m);
      }
   } /* DepthFirstSearch */
```

enumeration

Adam

☞ ⟨Adam,Cain⟩
⟨Adam,Abel⟩



Depth-First Search

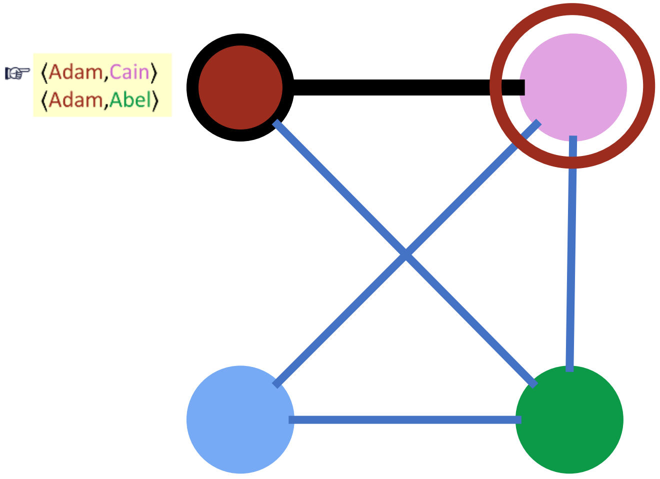
Cain

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

```
/* If n was never visited, enumerate it and all its unvisited relatives. */
void DepthFirstSearch(node n) {
   if ( /* n has never been visited */ ) {
      /* Enumerate n. */
      for ( /* each edge ⟨n,m⟩ */ )
         DepthFirstSearch(m);
      }
   } /* DepthFirstSearch */
```

enumeration

Adam



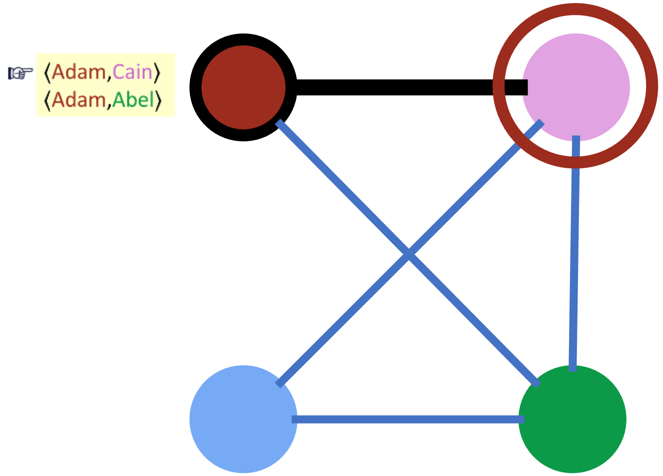
Means “first vist”

Cain

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. */
void DepthFirstSearch(node n) {
   if ( /* n has never been visited */ ) {
      /* Enumerate n. */
      for ( /* each edge ⟨n,m⟩ */ )
         DepthFirstSearch(m);
      }
   } /* DepthFirstSearch */
```

true



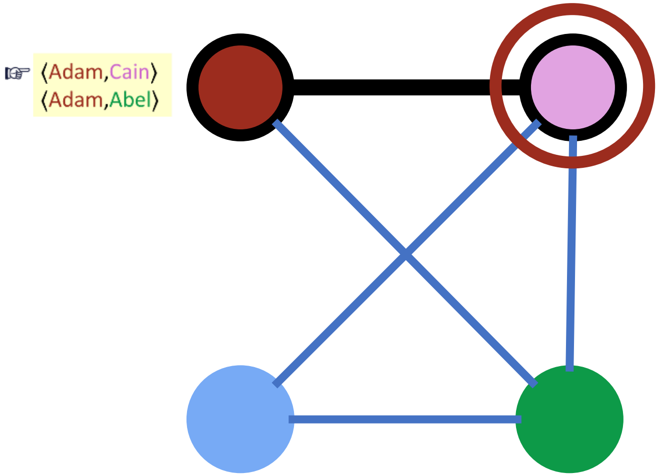
enumeration

Adam

Cain

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. */
void DepthFirstSearch(node n) {
   if ( /* n has never been visited */ ) {
      /* Enumerate n. */
      for ( /* each edge ⟨n,m⟩ */ )
         DepthFirstSearch(m);
      }
   } /* DepthFirstSearch */
```



enumeration

Adam
Cain

Cain

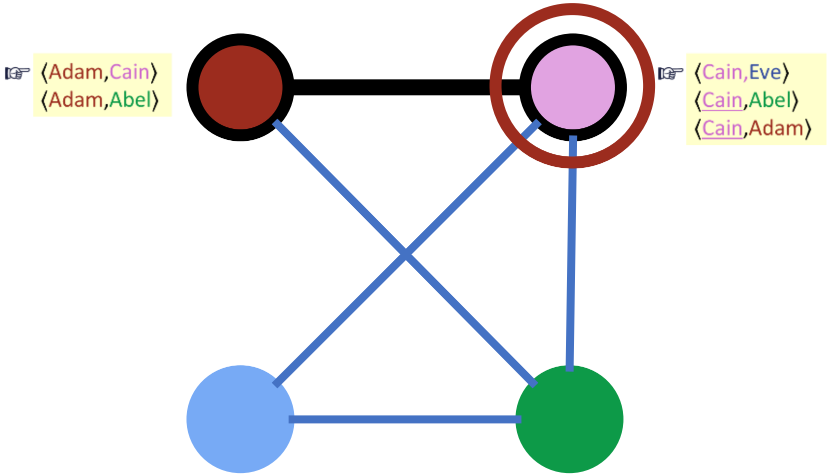
Reachability: Enumerate every node that can be reached from node n by following an edge.

```
/* If n was never visited, enumerate it and all its unvisited relatives. */
void DepthFirstSearch(node n) {
   if ( /* n has never been visited */ ) {
      /* Enumerate n. */
      for ( /* each edge ⟨n,m⟩ */ )
         DepthFirstSearch(m);
      }
   } /* DepthFirstSearch */
```

☞ ⟨Cain,Eve⟩
⟨Cain,Abel⟩
⟨Cain,Adam⟩

enumeration

Adam
Cain



Depth-First Search

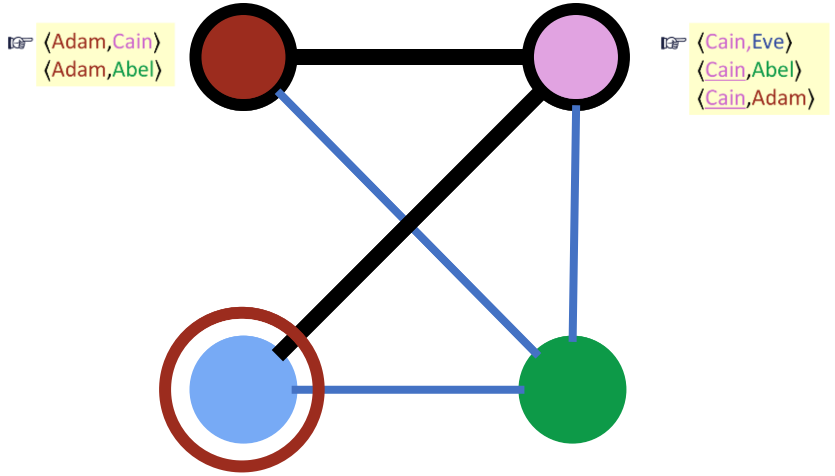
Eve

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. */
void DepthFirstSearch(node n) {
   if ( /* n has never been visited */ ) {
      /* Enumerate n. */
      for ( /* each edge ⟨n,m⟩ */ )
         DepthFirstSearch(m);
      }
   } /* DepthFirstSearch */
```

enumeration

Adam
Cain



Depth-First Search

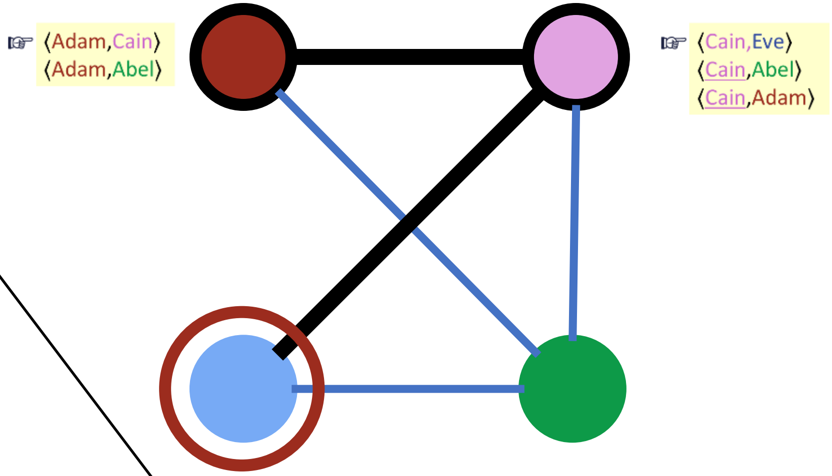
Eve

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. */
void DepthFirstSearch(node n) {
   if ( /* n has never been visited */ ) {
      /* Enumerate n. */
      for ( /* each edge ⟨n,m⟩ */ )
         DepthFirstSearch(m);
      }
   } /* DepthFirstSearch */
```

enumeration

Adam
Cain



true

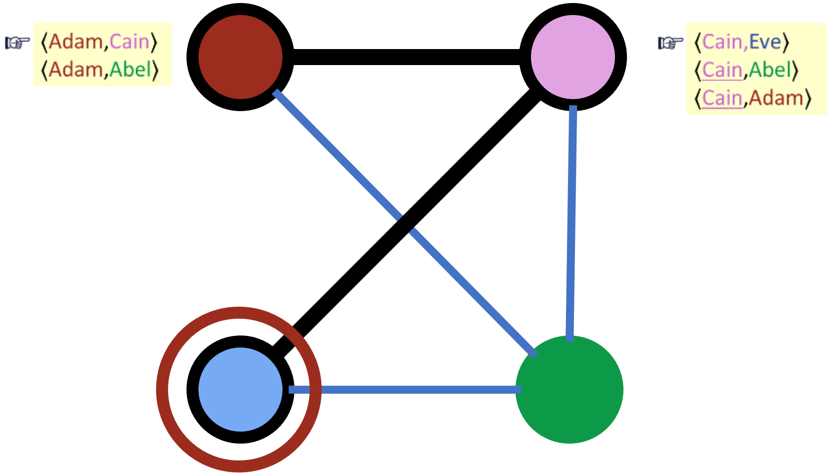
Eve

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. */
void DepthFirstSearch(node n) {
   if ( /* n has never been visited */ ) {
      /* Enumerate n. */
      for ( /* each edge ⟨n,m⟩ */ )
         DepthFirstSearch(m);
      }
   } /* DepthFirstSearch */
```

enumeration

Adam
Cain
Eve



Eve

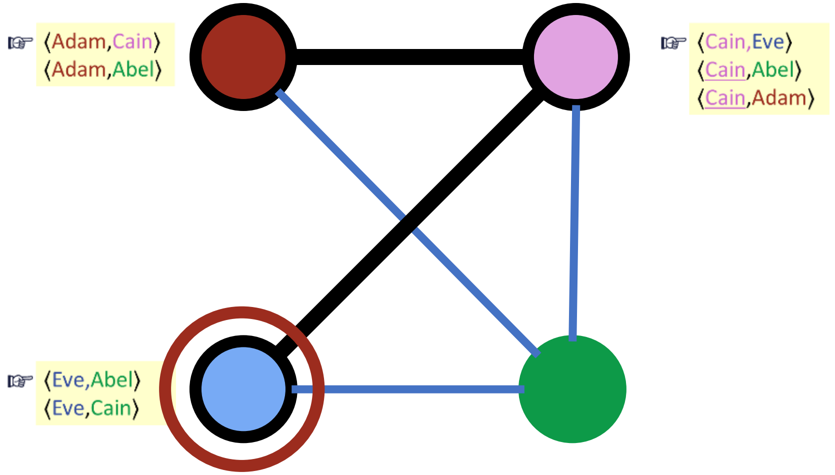
Reachability: Enumerate every node that can be reached from node n by following an edge.

```
/* If n was never visited, enumerate it and all its unvisited relatives. */
void DepthFirstSearch(node n) {
   if ( /* n has never been visited */ ) {
      /* Enumerate n. */
      for ( /* each edge ⟨n,m⟩ */ )
         DepthFirstSearch(m);
      }
   } /* DepthFirstSearch */
```

☞ ⟨Eve,Abel⟩
⟨Eve,Cain⟩

enumeration

Adam
Cain
Eve



Depth-First Search

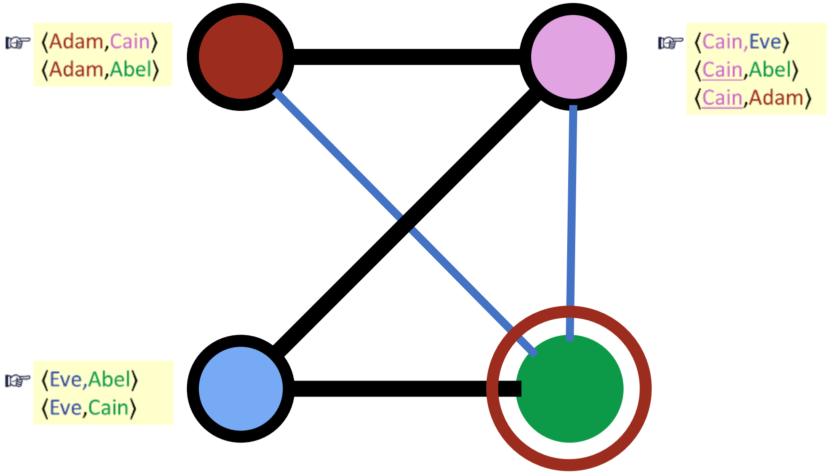
Able

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

```
/* If n was never visited, enumerate it and all its unvisited relatives. */
void DepthFirstSearch(node n) {
   if ( /* n has never been visited */ ) {
      /* Enumerate n. */
      for ( /* each edge ⟨n,m⟩ */ )
         DepthFirstSearch(m);
      }
   } /* DepthFirstSearch */
```

enumeration

Adam
Cain
Eve



Depth-First Search

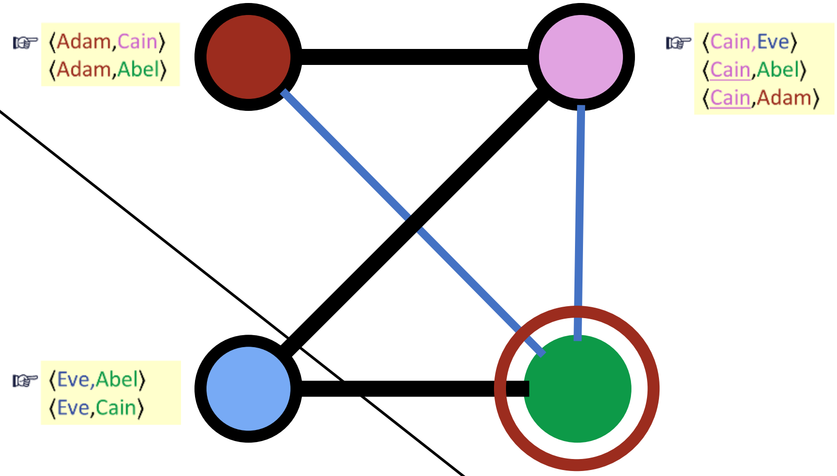
Able

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. */
void DepthFirstSearch(node n) {
   if ( /* n has never been visited */ ) {
      /* Enumerate n. */
      for ( /* each edge ⟨n,m⟩ */ )
         DepthFirstSearch(m);
      }
   } /* DepthFirstSearch */
```

enumeration

Adam
Cain
Eve



true

Depth-First Search

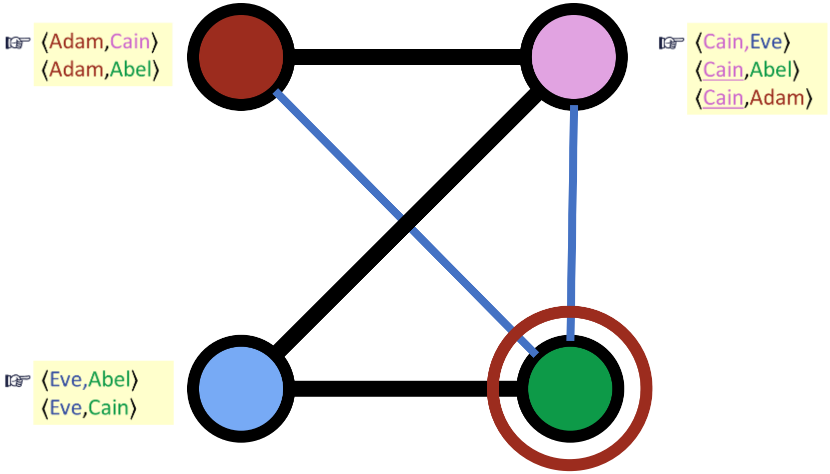
Able

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. */
void DepthFirstSearch(node n) {
   if ( /* n has never been visited */ ) {
      /* Enumerate n. */
      for ( /* each edge ⟨n,m⟩ */ )
         DepthFirstSearch(m);
      }
   } /* DepthFirstSearch */
```

enumeration

Adam
Cain
Eve
Able



Depth-First Search

Able

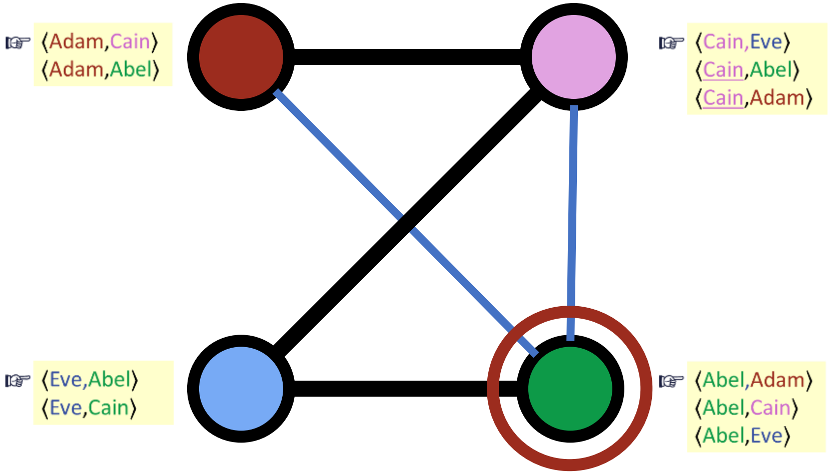
Reachability: Enumerate every node that can be reached from node n by following an edge.

```
/* If n was never visited, enumerate it and all its unvisited relatives. */
void DepthFirstSearch(node n) {
   if ( /* n has never been visited */ ) {
      /* Enumerate n. */
      for ( /* each edge ⟨n,m⟩ */ )
         DepthFirstSearch(m);
      }
   } /* DepthFirstSearch */
```

☞ ⟨Abel,Adam⟩
⟨Abel,Cain⟩
⟨Abel,Eve⟩

enumeration

Adam
Cain
Eve
Able



Depth-First Search

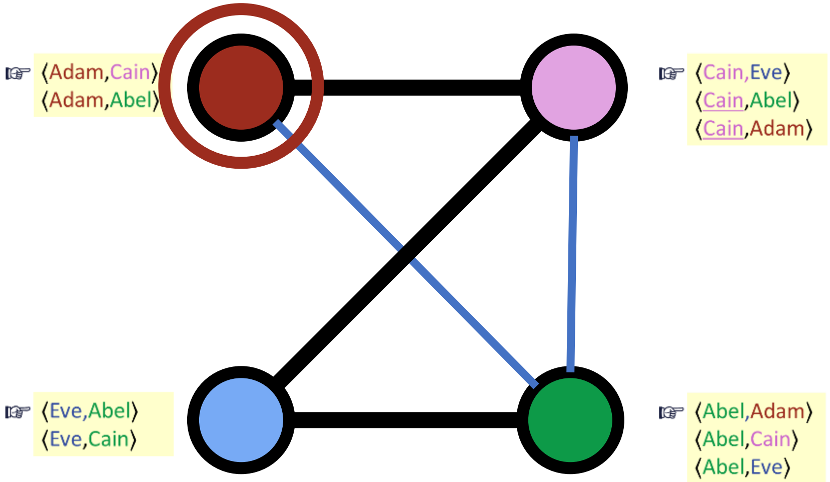
Adam

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. */
void DepthFirstSearch(node n) {
   if ( /* n has never been visited */ ) {
      /* Enumerate n. */
      for ( /* each edge ⟨n,m⟩ */ )
         DepthFirstSearch(m);
      }
   } /* DepthFirstSearch */
```

enumeration

Adam
Cain
Eve
Able

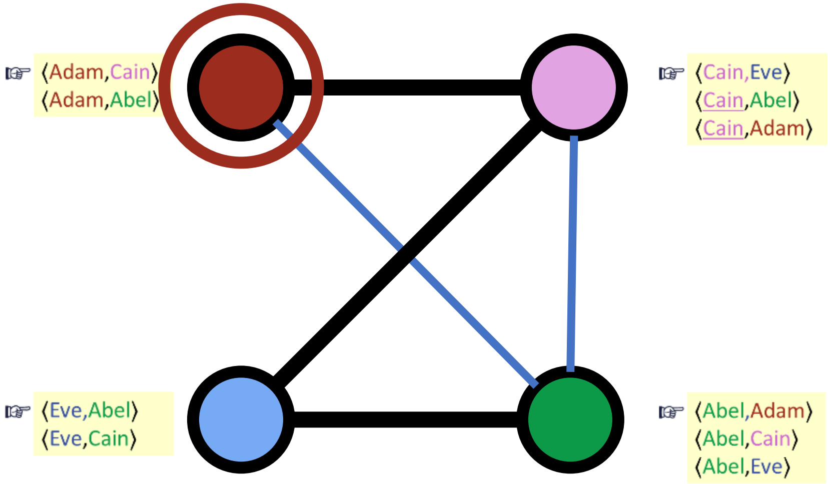


Adam

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. */
void DepthFirstSearch(node n) {
   if ( /* n has never been visited */ ) {
      /* Enumerate n. */
      for ( /* each edge ⟨n,m⟩ */ )
         DepthFirstSearch(m);
      }
   } /* DepthFirstSearch */
```

false



enumeration

Adam
Cain
Eve
Able

Depth-First Search

Able

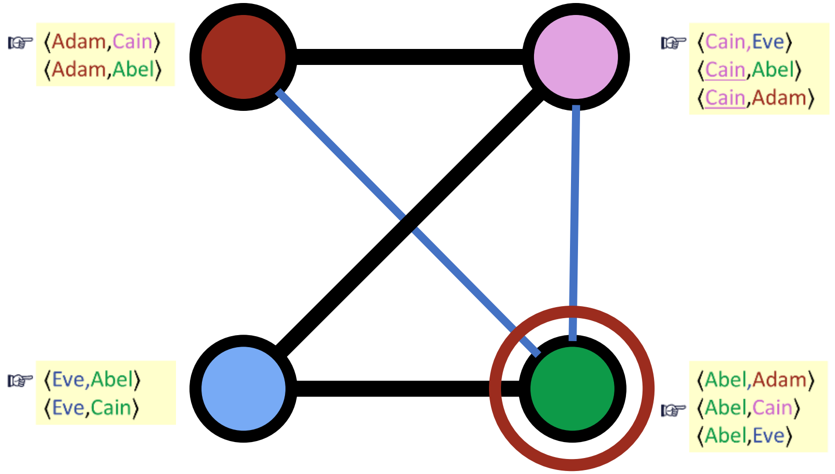
Reachability: Enumerate every node that can be reached from node n by following an edge.

```
/* If n was never visited, enumerate it and all its unvisited relatives. */
void DepthFirstSearch(node n) {
   if ( /* n has never been visited */ ) {
      /* Enumerate n. */
      for ( /* each edge ⟨n,m⟩ */ )
         DepthFirstSearch(m);
      }
   } /* DepthFirstSearch */
```

☞ ⟨Abel,Adam⟩
⟨Abel,Cain⟩
⟨Abel,Eve⟩

enumeration

Adam
Cain
Eve
Able



Depth-First Search

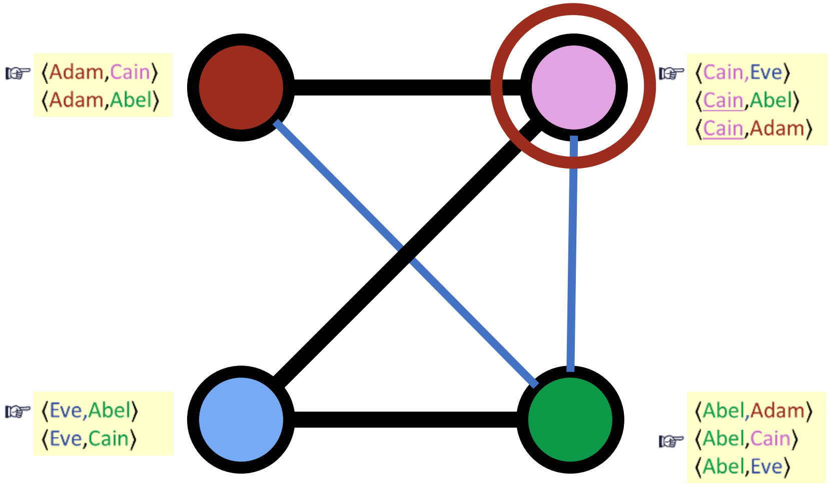
Cain

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. */
void DepthFirstSearch(node n) {
   if ( /* n has never been visited */ ) {
      /* Enumerate n. */
      for ( /* each edge ⟨n,m⟩ */ )
         DepthFirstSearch(m);
      }
   } /* DepthFirstSearch */
```

enumeration

Adam
Cain
Eve
Able



Cain

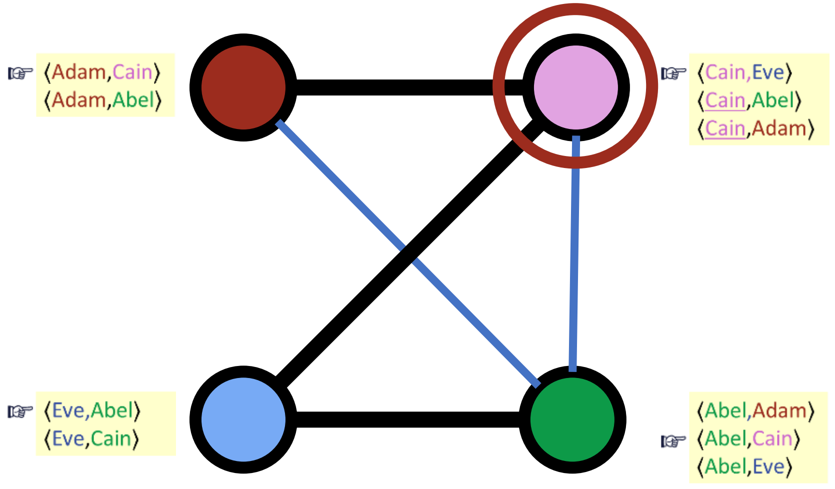
Reachability: Enumerate every node that can be reached from node n by following an edge.

```
/* If n was never visited, enumerate it and all its unvisited relatives. */
void DepthFirstSearch(node n) {
   if ( /* n has never been visited */ ) {
      /* Enumerate n. */
      for ( /* each edge ⟨n,m⟩ */ )
         DepthFirstSearch(m);
      }
   } /* DepthFirstSearch */
```

false

enumeration

Adam
Cain
Eve
Able



Able

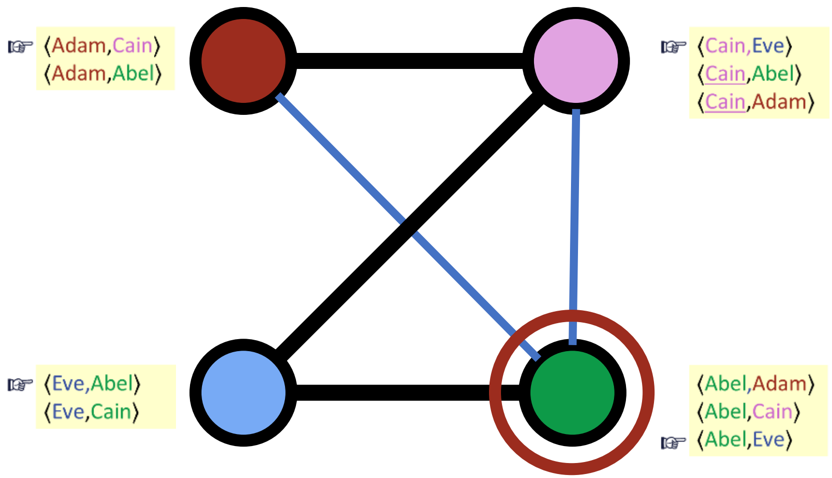
Reachability: Enumerate every node that can be reached from node n by following an edge.

```
/* If n was never visited, enumerate it and all its unvisited relatives. */
void DepthFirstSearch(node n) {
   if ( /* n has never been visited */ ) {
      /* Enumerate n. */
      for ( /* each edge ⟨n,m⟩ */ )
         DepthFirstSearch(m);
      }
   } /* DepthFirstSearch */
```

⟨Abel,Adam⟩
⟨Abel,Cain⟩
☞ ⟨Abel,Eve⟩

enumeration

Adam
Cain
Eve
Able



Depth-First Search

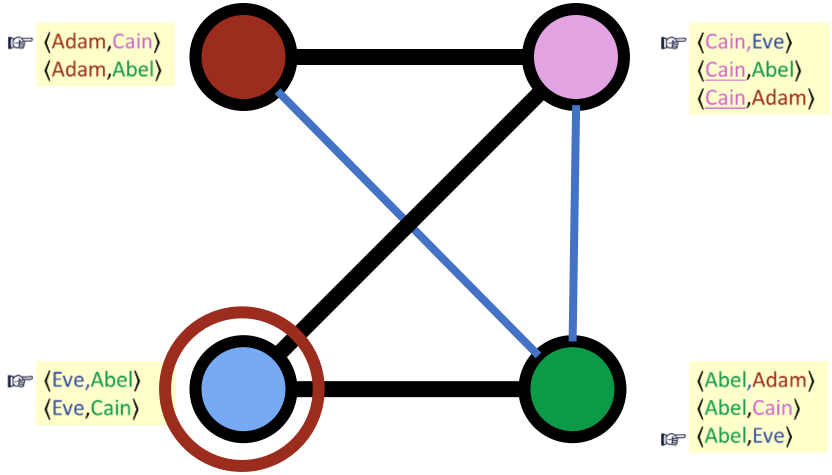
Eve

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. */
void DepthFirstSearch(node n) {
   if ( /* n has never been visited */ ) {
      /* Enumerate n. */
      for ( /* each edge ⟨n,m⟩ */ )
         DepthFirstSearch(m);
      }
   } /* DepthFirstSearch */
```

enumeration

Adam
Cain
Eve
Able



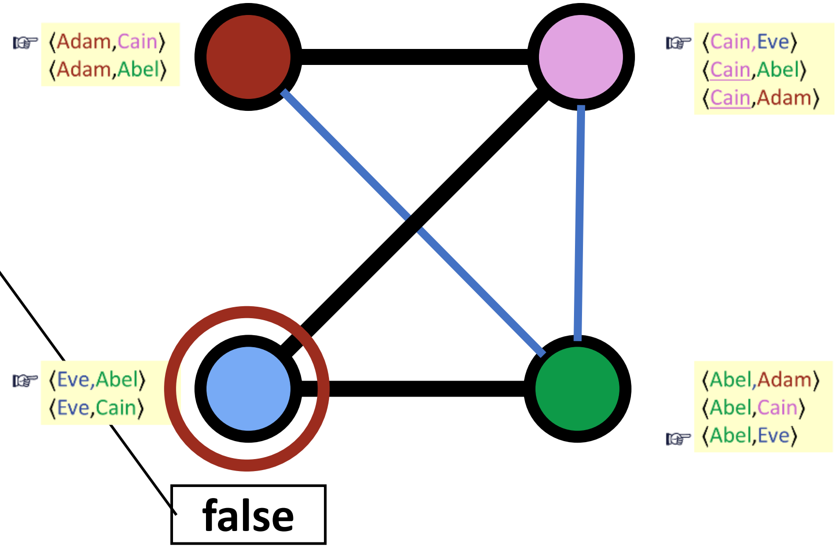
Eve

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. */
void DepthFirstSearch(node n) {
   if ( /* n has never been visited */ ) {
      /* Enumerate n. */
      for ( /* each edge ⟨n,m⟩ */ )
         DepthFirstSearch(m);
      }
   } /* DepthFirstSearch */
```

enumeration

Adam
Cain
Eve
Able



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

Able

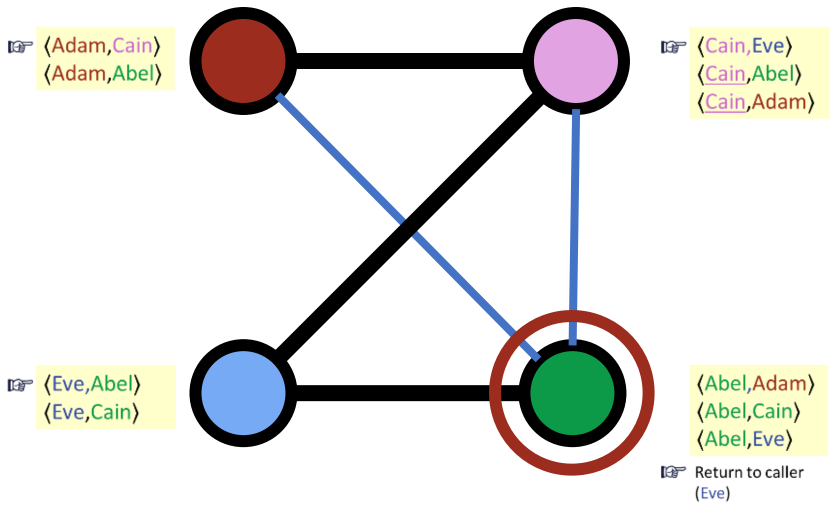
```
/* If n was never visited, enumerate it and all its unvisited relatives. */
void DepthFirstSearch(node n) {
   if ( /* n has never been visited */ ) {
      /* Enumerate n. */
      for ( /* each edge ⟨n,m⟩ */ )
         DepthFirstSearch(m);
      }
   } /* DepthFirstSearch */
```

enumeration

Adam
Cain
Eve
Able

⟨Abel,Adam⟩
⟨Abel,Cain⟩
⟨Abel,Eve⟩

☞ Return to caller (Eve)



Depth-First Search

Eve

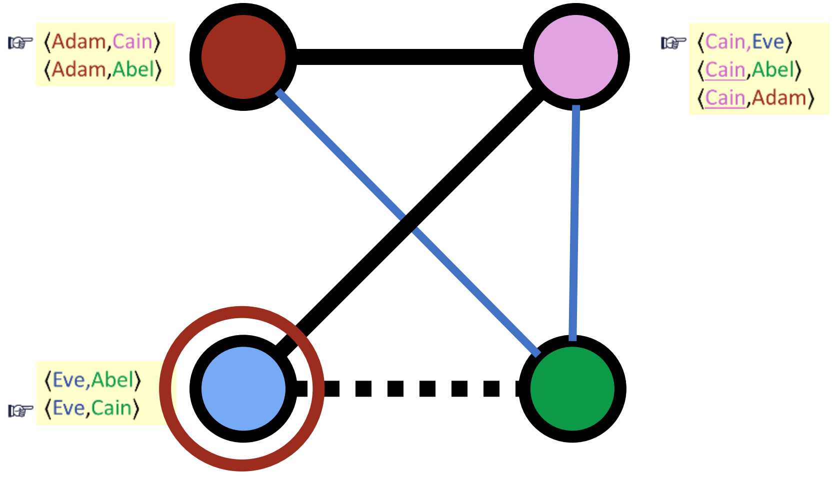
Reachability: Enumerate every node that can be reached from node n by following an edge.

```
/* If n was never visited, enumerate it and all its unvisited relatives. */
void DepthFirstSearch(node n) {
   if ( /* n has never been visited */ ) {
      /* Enumerate n. */
      for ( /* each edge ⟨n,m⟩ */ )
         DepthFirstSearch(m);
      }
   } /* DepthFirstSearch */
```

enumeration

Adam
Cain
Eve
Able

⟨Eve,Abel⟩
☞ ⟨Eve,Cain⟩



Means “first visitor finished”

Depth-First Search

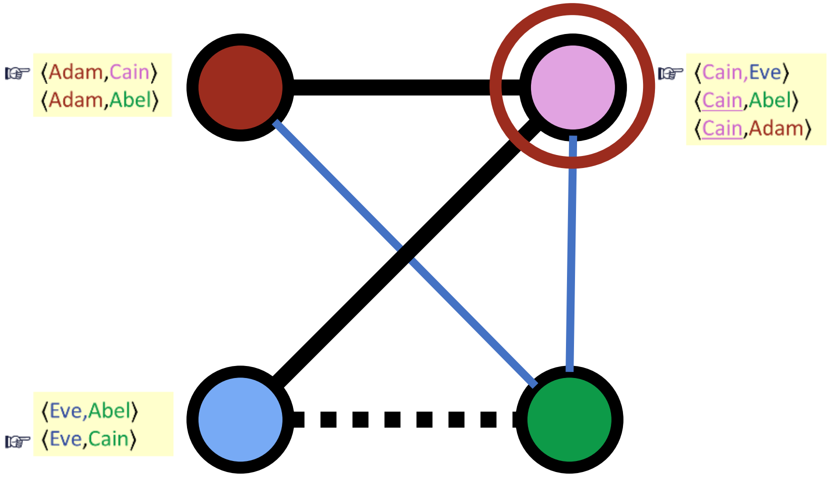
Cain

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. */
void DepthFirstSearch(node n) {
   if ( /* n has never been visited */ ) {
      /* Enumerate n. */
      for ( /* each edge ⟨n,m⟩ */ )
         DepthFirstSearch(m);
      }
   } /* DepthFirstSearch */
```

enumeration

Adam
Cain
Eve
Able



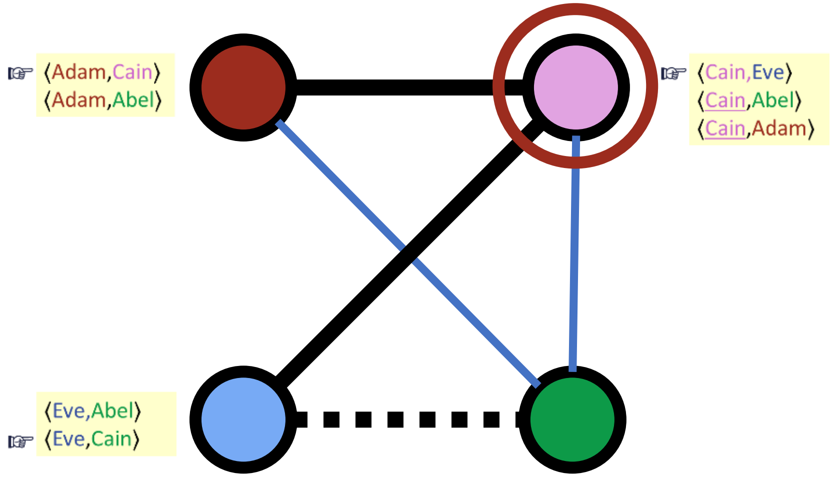
Depth-First Search

Cain

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. */
void DepthFirstSearch(node n) {
   if ( /* n has never been visited */ ) {
      /* Enumerate n. */
      for ( /* each edge ⟨n,m⟩ */ )
         DepthFirstSearch(m);
      }
   } /* DepthFirstSearch */
```

false



enumeration

Adam
Cain
Eve
Able

Depth-First Search

Eve

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

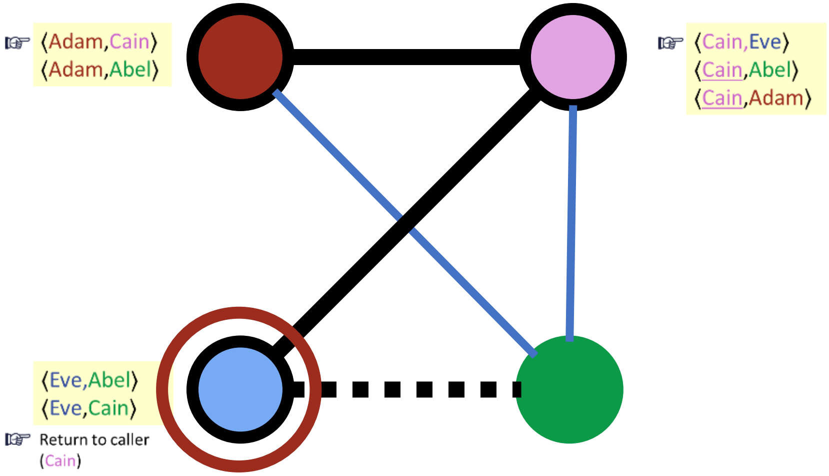
```
/* If n was never visited, enumerate it and all its unvisited relatives. */
void DepthFirstSearch(node n) {
   if ( /* n has never been visited */ ) {
      /* Enumerate n. */
      for ( /* each edge ⟨n,m⟩ */ )
         DepthFirstSearch(m);
      }
   } /* DepthFirstSearch */
```

⟨Eve,Abel⟩
⟨Eve,Cain⟩

☞ Return to caller (Cain)

enumeration

Adam
Cain
Eve
Able



Depth-First Search

Cain

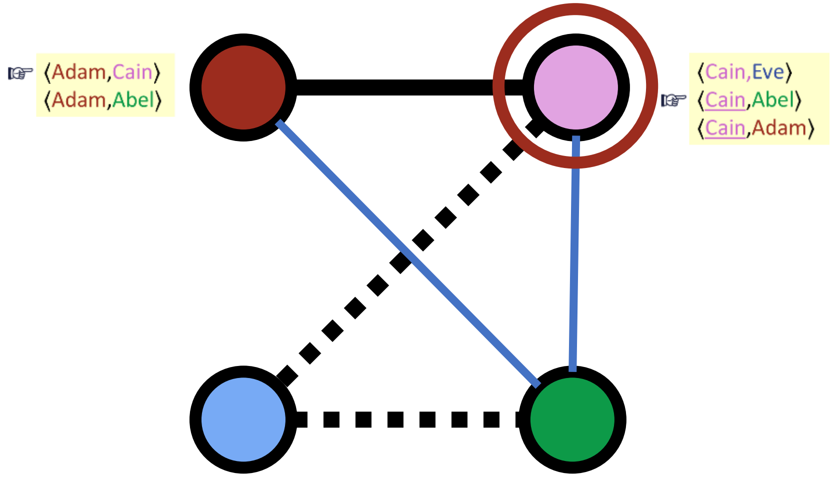
Reachability: Enumerate every node that can be reached from node n by following an edge.

```
/* If n was never visited, enumerate it and all its unvisited relatives. */
void DepthFirstSearch(node n) {
   if ( /* n has never been visited */ ) {
      /* Enumerate n. */
      for ( /* each edge ⟨n,m⟩ */ )
         DepthFirstSearch(m);
      }
   } /* DepthFirstSearch */
```

⟨Cain,Eve⟩
☞ ⟨Cain,Abel⟩
⟨Cain,Adam⟩

enumeration

Adam
Cain
Eve
Able



Depth-First Search

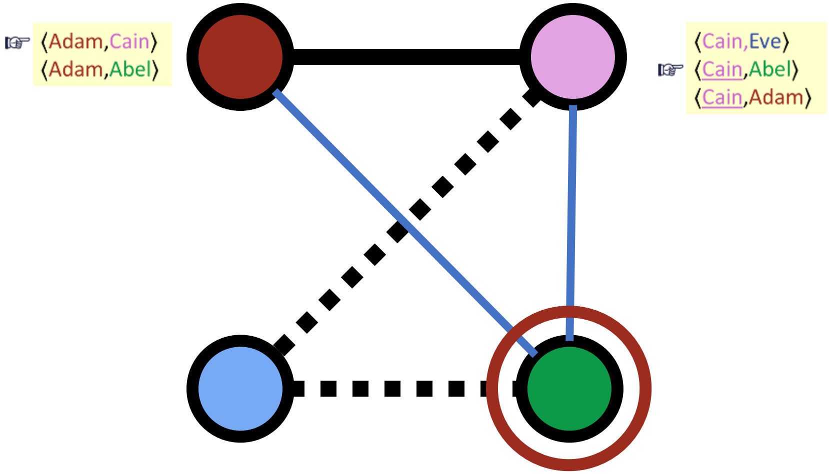
Abel

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. */
void DepthFirstSearch(node n) {
   if ( /* n has never been visited */ ) {
      /* Enumerate n. */
      for ( /* each edge ⟨n,m⟩ */ )
         DepthFirstSearch(m);
      }
   } /* DepthFirstSearch */
```

enumeration

Adam
Cain
Eve
Able



Depth-First Search

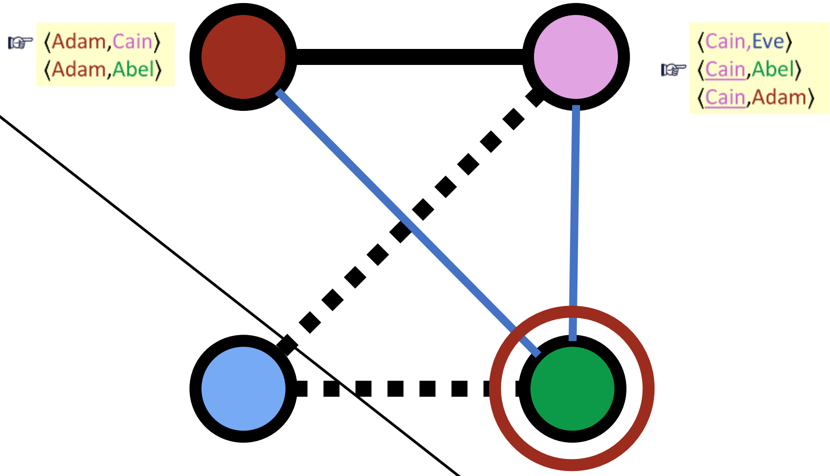
Abel

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. */
void DepthFirstSearch(node n) {
   if ( /* n has never been visited */ ) {
      /* Enumerate n. */
      for ( /* each edge ⟨n,m⟩ */ )
         DepthFirstSearch(m);
      }
   } /* DepthFirstSearch */
```

enumeration

Adam
Cain
Eve
Able



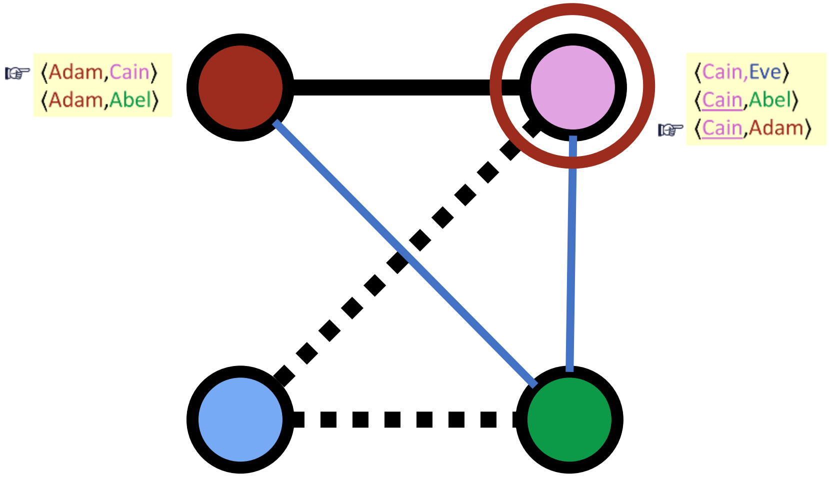
false

Depth-First Search

Cain

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. */
void DepthFirstSearch(node n) {
   if ( /* n has never been visited */ ) {
      /* Enumerate n. */
      for ( /* each edge ⟨n,m⟩ */ )
         DepthFirstSearch(m);
      }
   } /* DepthFirstSearch */
```



⟨Cain,Eve⟩
⟨Cain,Abel⟩
☞ ⟨Cain,Adam⟩

enumeration

Adam
Cain
Eve
Able

Depth-First Search

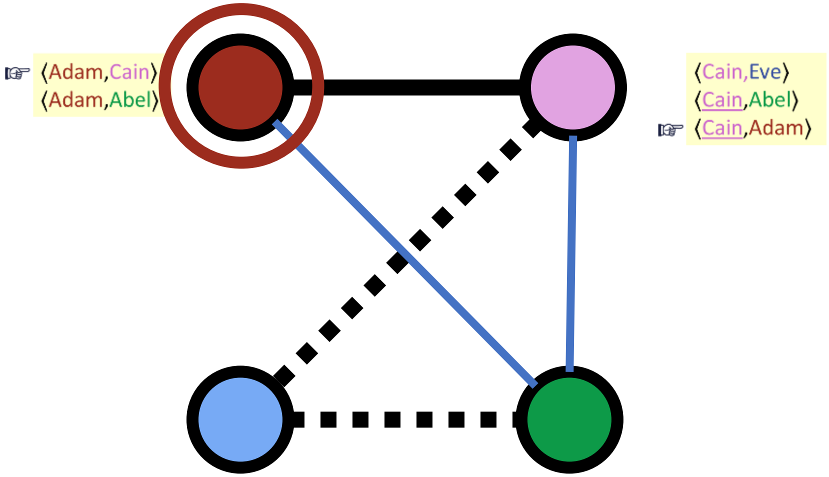
Adam

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. */
void DepthFirstSearch(node n) {
   if ( /* n has never been visited */ ) {
      /* Enumerate n. */
      for ( /* each edge ⟨n,m⟩ */ )
         DepthFirstSearch(m);
      }
   } /* DepthFirstSearch */
```

enumeration

Adam
Cain
Eve
Able

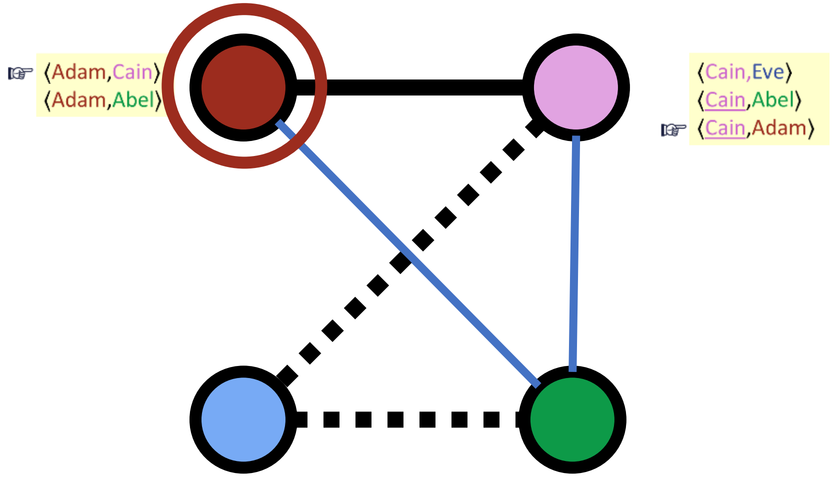


Adam

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. */
void DepthFirstSearch(node n) {
   if ( /* n has never been visited */ ) {
      /* Enumerate n. */
      for ( /* each edge ⟨n,m⟩ */ )
         DepthFirstSearch(m);
      }
   } /* DepthFirstSearch */
```

false



enumeration

Adam
Cain
Eve
Able

Cain

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

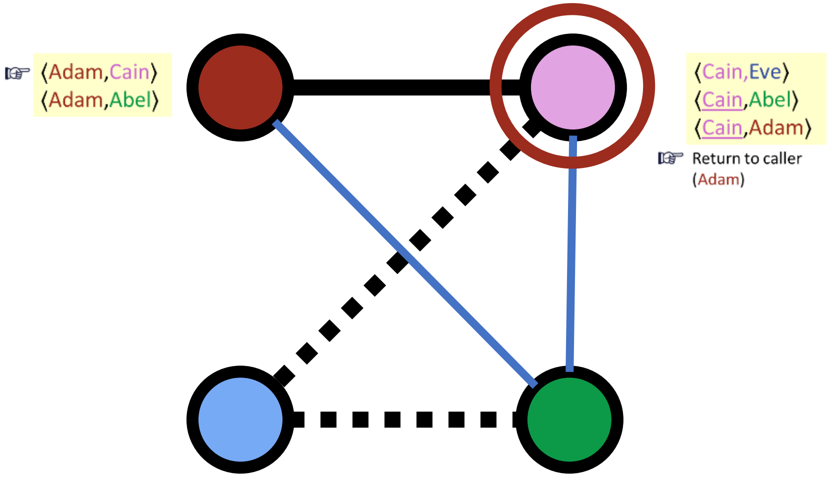
```
/* If n was never visited, enumerate it and all its unvisited relatives. */
void DepthFirstSearch(node n) {
   if ( /* n has never been visited */ ) {
      /* Enumerate n. */
      for ( /* each edge ⟨n,m⟩ */ )
         DepthFirstSearch(m);
      }
   } /* DepthFirstSearch */
```

⟨Cain,Eve⟩
⟨Cain,Abel⟩
⟨Cain,Adam⟩

☞ Return to caller (Adam)

enumeration

Adam
Cain
Eve
Able



Depth-First Search

Adam

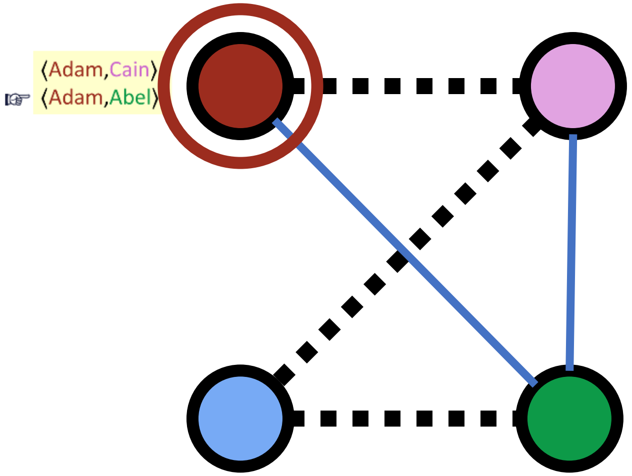
Reachability: Enumerate every node that can be reached from node n by following an edge.

```
/* If n was never visited, enumerate it and all its unvisited relatives. */
void DepthFirstSearch(node n) {
   if ( /* n has never been visited */ ) {
      /* Enumerate n. */
      for ( /* each edge ⟨n,m⟩ */ )
         DepthFirstSearch(m);
      }
   } /* DepthFirstSearch */
```

enumeration

Adam
Cain
Eve
Able

⟨Adam,Cain⟩
☞ ⟨Adam,Abel⟩



Depth-First Search

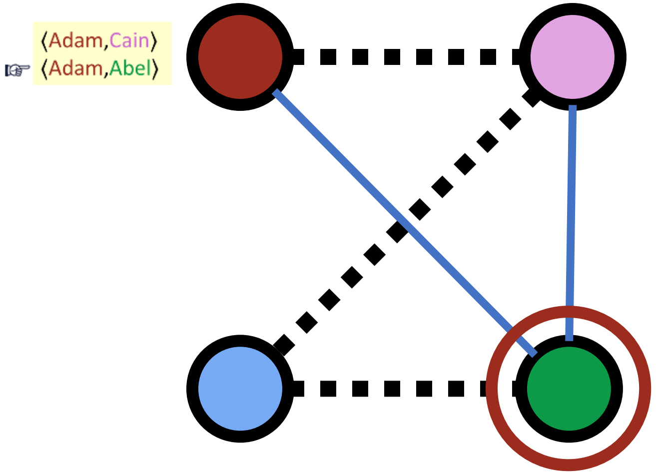
Abel

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. */
void DepthFirstSearch(node n) {
   if ( /* n has never been visited */ ) {
      /* Enumerate n. */
      for ( /* each edge ⟨n,m⟩ */ )
         DepthFirstSearch(m);
      }
   } /* DepthFirstSearch */
```

enumeration

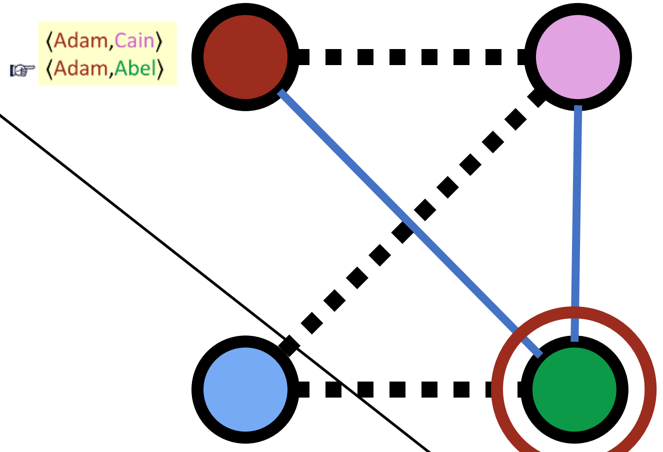
Adam
Cain
Eve
Able



Abel

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. */
void DepthFirstSearch(node n) {
   if ( /* n has never been visited */ ) {
      /* Enumerate n. */
      for ( /* each edge ⟨n,m⟩ */ )
         DepthFirstSearch(m);
      }
   } /* DepthFirstSearch */
```



false

enumeration

Adam
Cain
Eve
Able

Depth-First Search

Adam

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

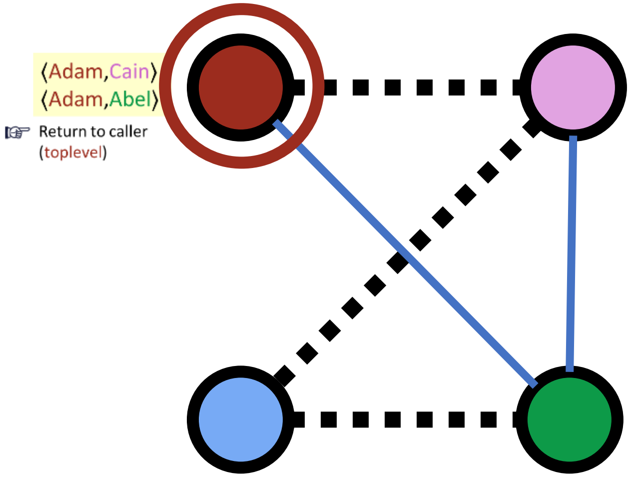
```
/* If n was never visited, enumerate it and all its unvisited relatives. */
void DepthFirstSearch(node n) {
   if ( /* n has never been visited */ ) {
      /* Enumerate n. */
      for ( /* each edge ⟨n,m⟩ */ )
         DepthFirstSearch(m);
      }
   } /* DepthFirstSearch */
```

⟨Adam,Cain⟩
⟨Adam,Abel⟩

☞ Return to caller (toplevel)

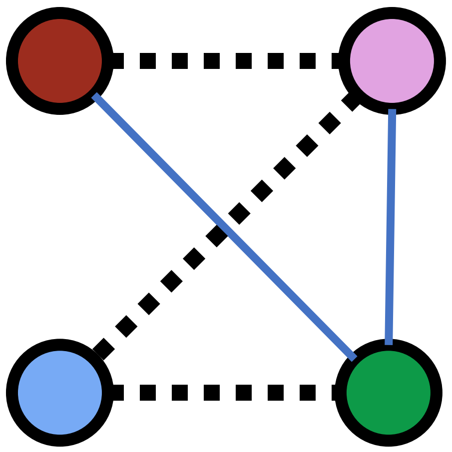
enumeration

Adam
Cain
Eve
Able



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. */
void DepthFirstSearch(node n) {
   if ( /* n has never been visited */ ) {
      /* Enumerate n. */
      for ( /* each edge ⟨n,m⟩ */ )
         DepthFirstSearch(m);
      }
   } /* DepthFirstSearch */
```



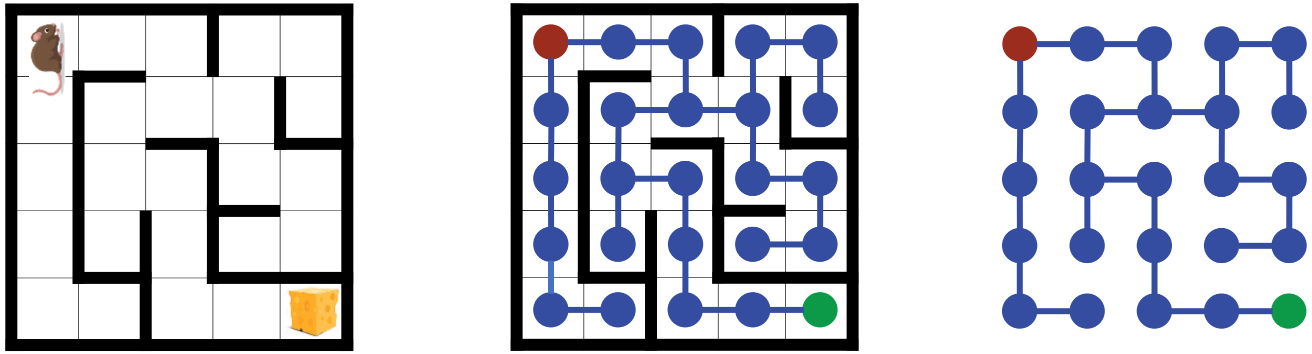
DONE

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.

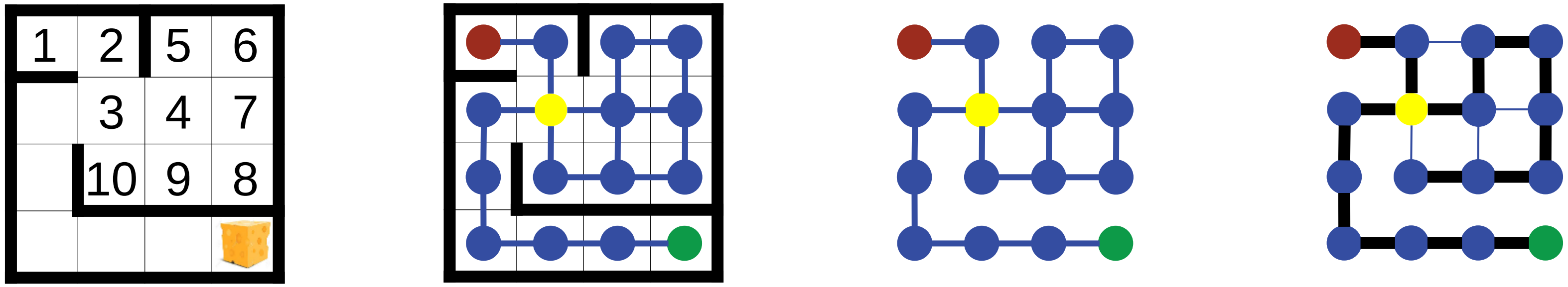


To solve the maze, perform `DepthFirstSearch(upper-left-cell)`.
Stop if you encounter the `lower-right-cell`.

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

Running a Maze, Revisited

Domain-Specific Subtleties: Gone.

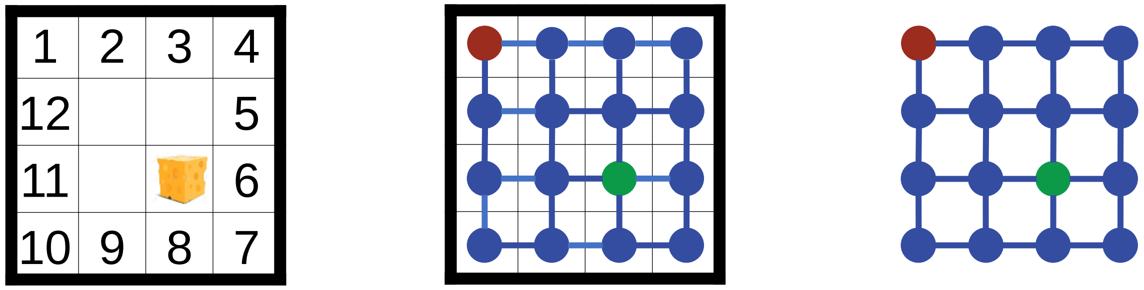


Recall the distinction been corridor-like cul-de-sacs and room-like cul-de-sacs. Gone.

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

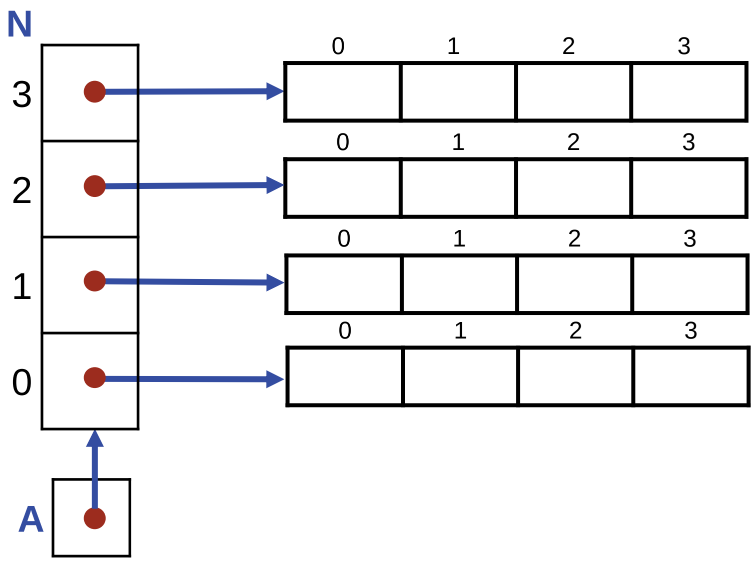
Running a Maze, Revisited

Finding Centrally-Located Cheese : No problem.



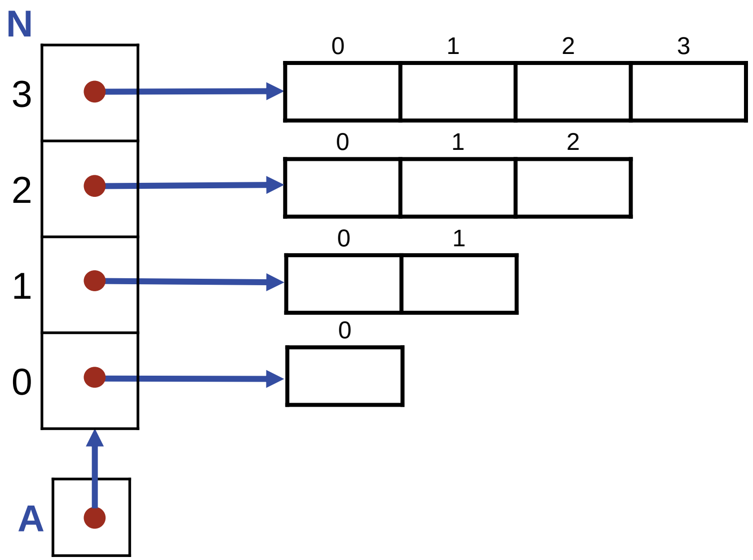
Regardless of the cheese’s location, the problem is just graph reachability, and can be solved by Depth-First Search.

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



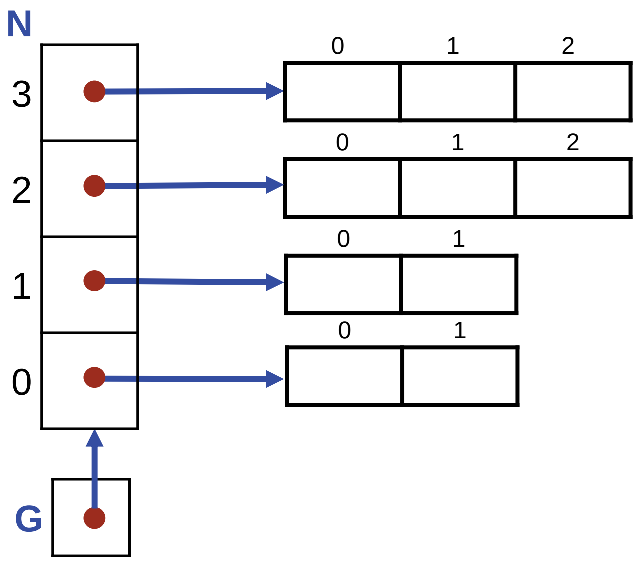
For example, the N-by-N square array A, for N=4, would be as shown.

Representation: Recall, 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.

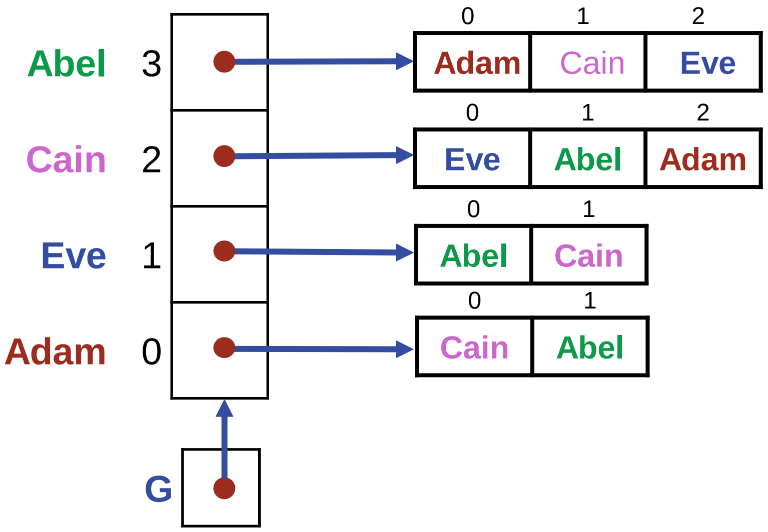
Representation: A 2-D array can be used to represent a graph G with N nodes.

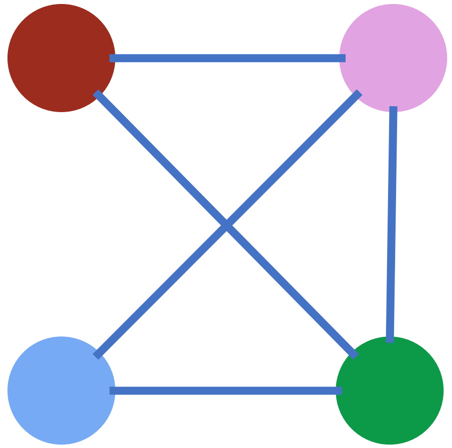


Number the nodes 0 through N-1.

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:



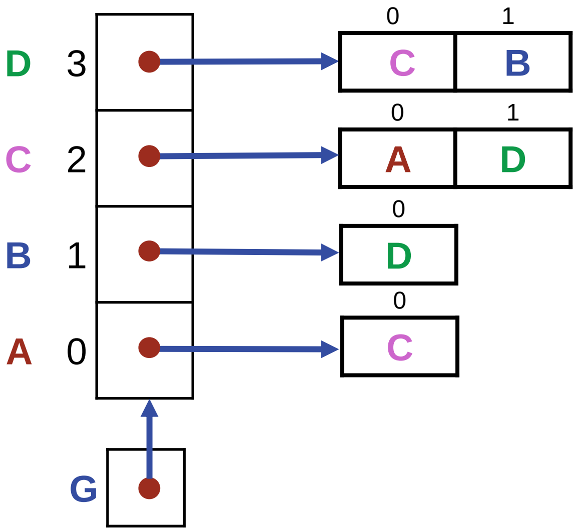


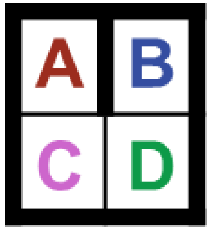
Number the nodes 0 through N-1.

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.

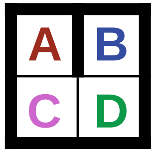
Running a Maze, Revisited

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

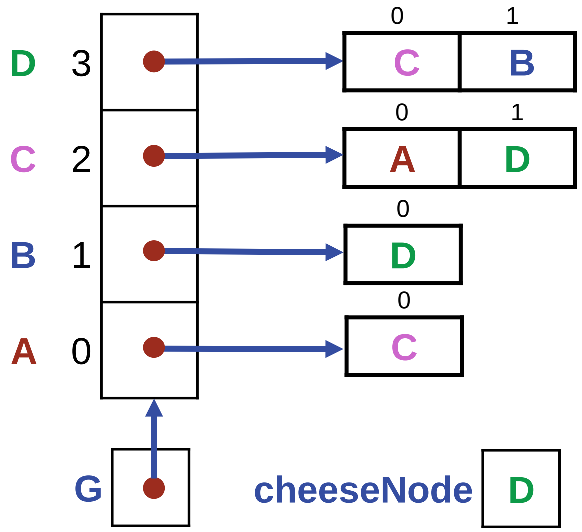




Running a Maze, Revisited

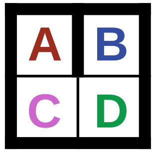


Representation: invariant.

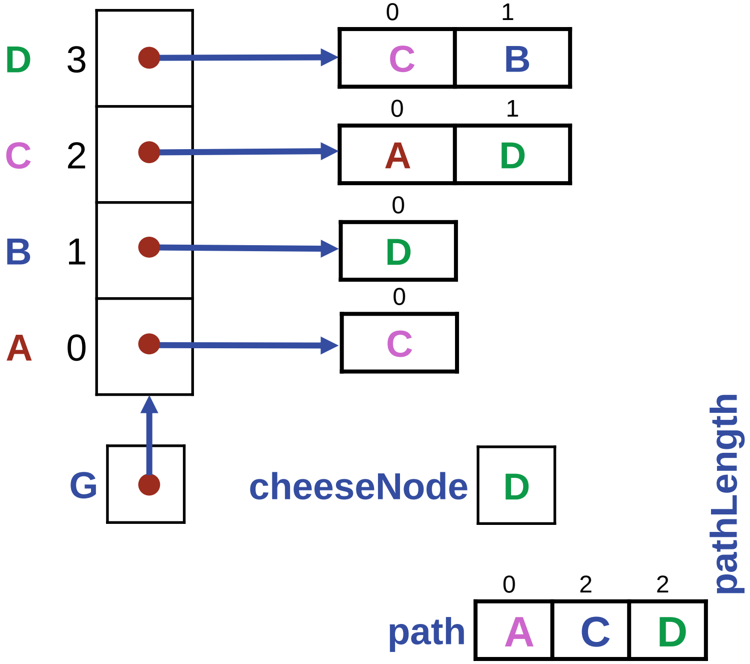



```
/* Maze, Rat, and Path (MRP) Representations. */
class MRP {
   /* Maze. Maze cells are represented by N*N nodes
      of graph G, where G[n] is an edge list for node
      n, i.e., for 0≤e<G[n].length, G[n][e] is an
      adjacent node m, i.e., a cell m adjacent to n
      with intervening Wall. The upper-left cell is
      node 0. Cheese is at cheeseNode. */
      private static int G[][];      // Edge lists.
      private static int cheeseNode; // Cheese.
...
   } /* MRP */
```

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<G[n].length, 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 cheeseNode. */
      private static int G[][];      // Edge lists.
      private static int cheeseNode; // Cheese.
   /* Path. Array path[0..pathLength-1] is a list of
      adjacent nodes in G reaching from node 0 to some
      node path[pathlength-1]. */
      private static int path[];
      private static int pathLength;
      public static boolean isAtCheese() {
         return path[pathLength-1]==cheeseNode;
         }
   ...
   } /* MRP */
```

Running a Maze, Revisited

Representation: Depth-First Search.

```
/* Maze, Rat, and Path (MRP) Representations. */
   class MRP {
   private static boolean mark[];          // mark[n] iff DFS reached node n.
   /* Depth First Search (DFS) of node n for cheeseNode at depth p. */
   private static void DFS(int n) {
      if ( !mark[n] ) {                    // Node n has not been visited before.
         mark[n] = true;                   // Mark that n has been visited.
         for (int e=0; e<G[n].length; e++) DFS(G[n][e]);
         }
      } /* DFS */
   ...
   } /* MRP */
```

Representation: Depth-First Search, with path.

```
/* Maze, Rat, and Path (MRP) Representations. */
   class MRP {
   private static boolean mark[];          // mark[n] iff DFS reached node n.
   /* Depth First Search (DFS) of node n for cheeseNode at depth p. */
   private static void DFS(int n, int p) {
      if ( !mark[n] ) {                    // Node n has not been visited before.
         mark[n] = true;                   // Mark that n has been visited.
         path[p] = n;                      // Extend the path to include n.
         for (int e=0; e<G[n].length; e++) DFS(G[n][e], p+1);
         }
      } /* DFS */
   ...
   } /* MRP */
```

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 {
   private static boolean mark[];           // mark[n] iff DFS reached node n.
   /* Depth First Search (DFS) of node n for cheeseNode at depth p. */
   private static void DFS(int n, int p) {
      if ( !mark[n] ) {                     // Node n has not been visited before.
         mark[n] = true;                    // Mark that n has been visited.
         path[p] = n;                       // Extend the path to include n.
         if ( n==cheeseNode ) {             // Terminate search if cheese found.
           pathLength = p+1;                // Length of path is one longer than p.
           throw new RuntimeException("found cheese");
           }
         for (int e=0; e<G[n].length; e++) DFS(G[n][e], p+1);
         }
      } /* DFS */
   ...
   } /* MRP */
```

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

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]. */
      public static void Search() {
         MakeGraphFromInput();
         try { DFS(0,0); } catch ( RuntimeException e ) { }
         MakeOutputFromPath();
         } /* Search */
   ...
   } /* MRP */
```

`MakeGraphFromInput` and `MakeOutputFromPath` 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.