2 – Graphs and Coding

Intro To Code
Let’s write our first programs

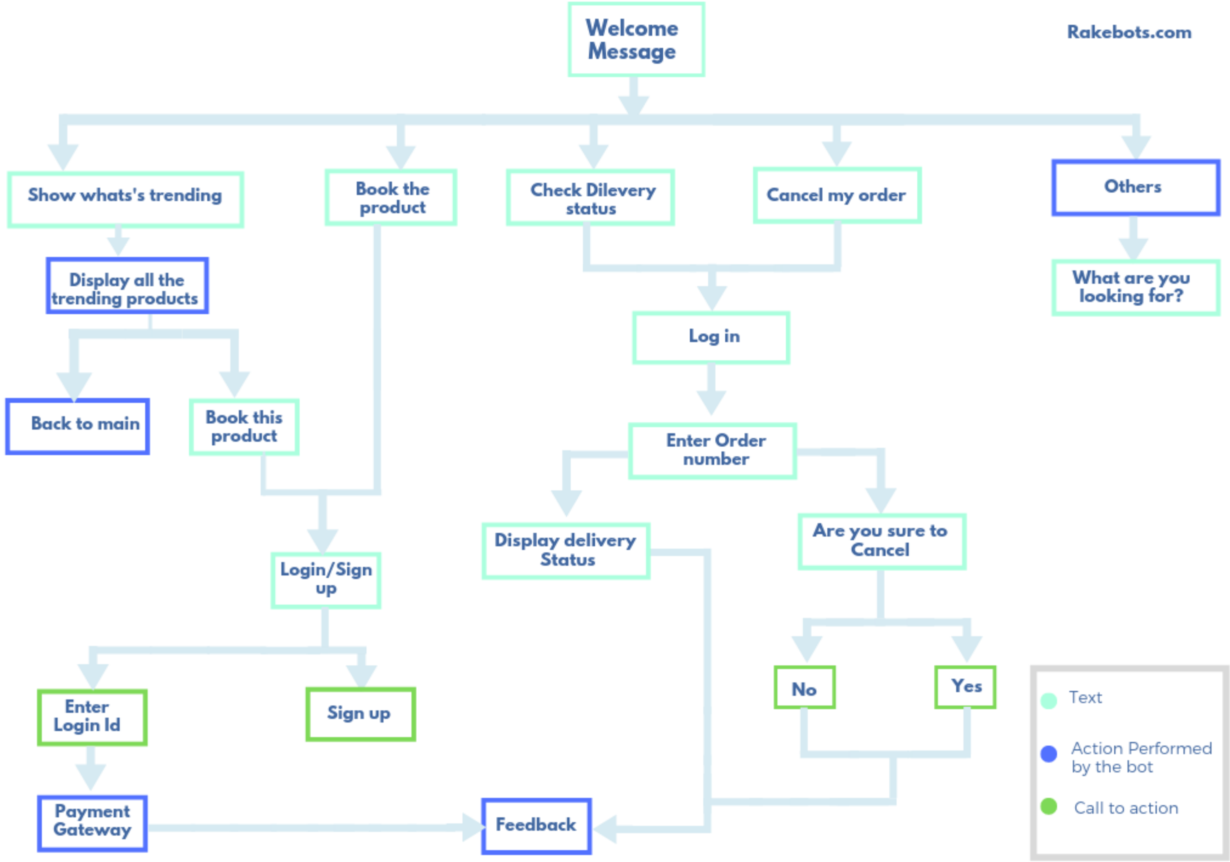
The Graph Data Structure
Incredibly Rich and Powerful

Build and Tinker
Explore

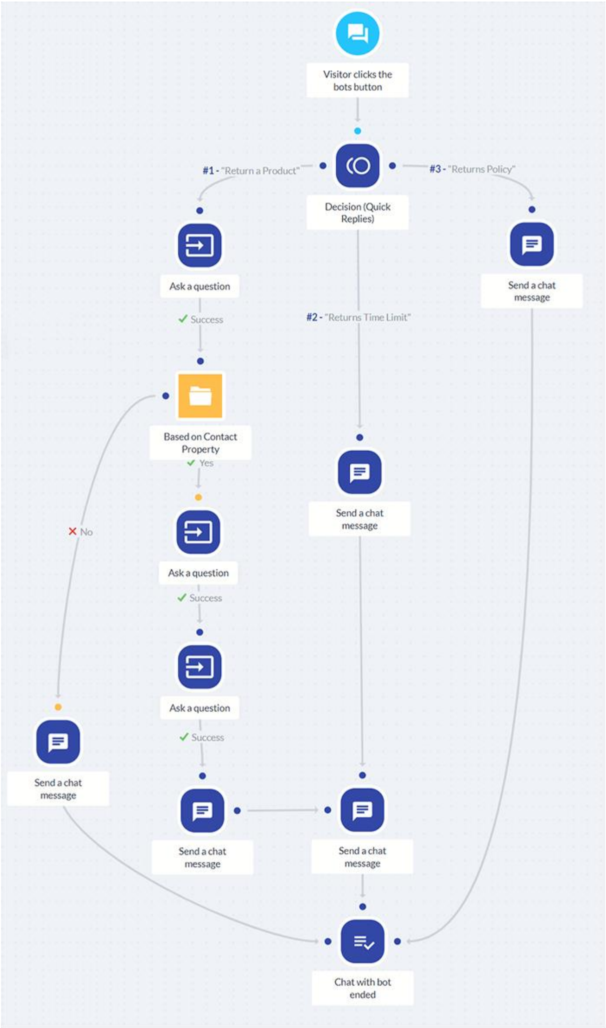
Graphs

"The important graphs are the ones where some things are not connected to some other things. When the unenlightened ones [make connections between everything] until their graph is fully connected and also totally useless." – Eliezer Yudkowsky

Why Graphs?

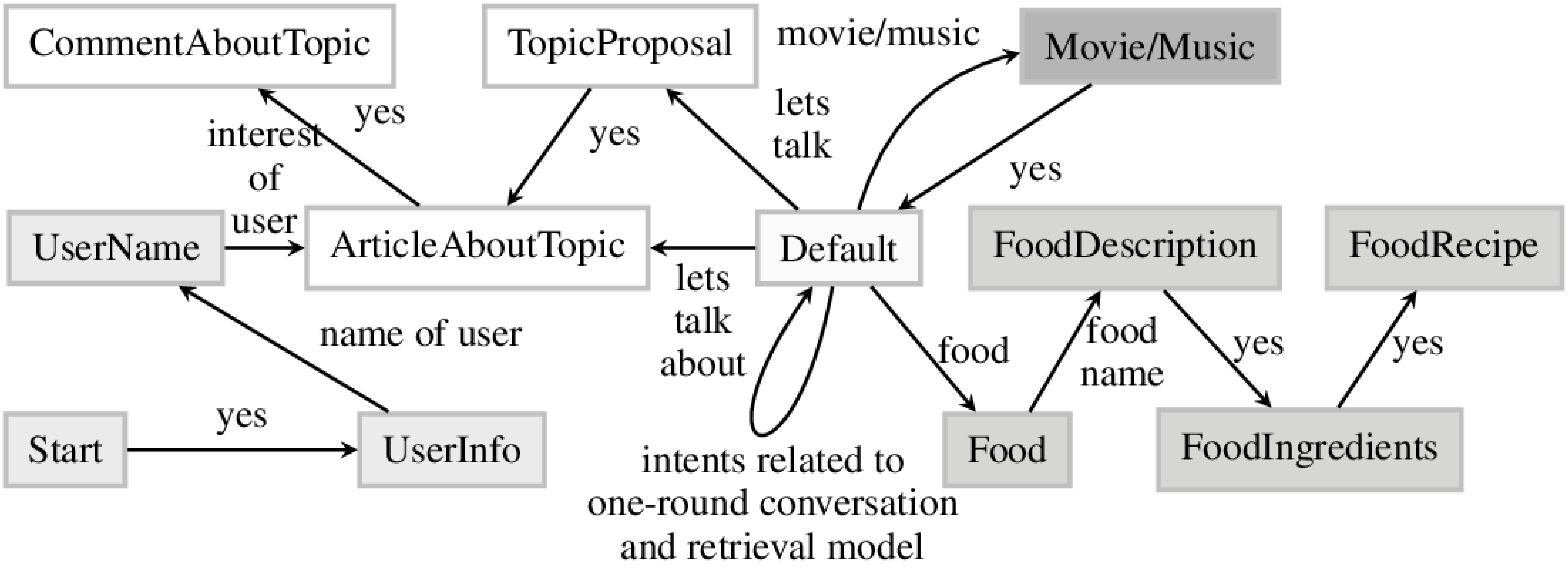


Welcome Message
Rakebots.com
Show whats's trending
Book the product
Check Dilevery status
Cancel my order
Others
Display all the trending products
What are you looking for?
Log in
Back to main
Book this product
Enter Order number
Display delivery Status
Are you sure to Cancel
Login/Sign up
No
Yes
Enter Login Id
Sign up
Payment Gateway
Feedback
Text
Action Performed by the bot
Call to action



Visitor clicks the bots button
#1 - "Return a Product"
#3 - "Returns Policy"
Decision (Quick Replies)
Ask a question
Send a chat message
Success
#2 - "Returns Time Limit"
Based on Contact Property
Yes
No
Send a chat message
Ask a question
Success
Ask a question
Success
Send a chat message
Send a chat message
Send a chat message
Chat with bot ended

Why Graphs?

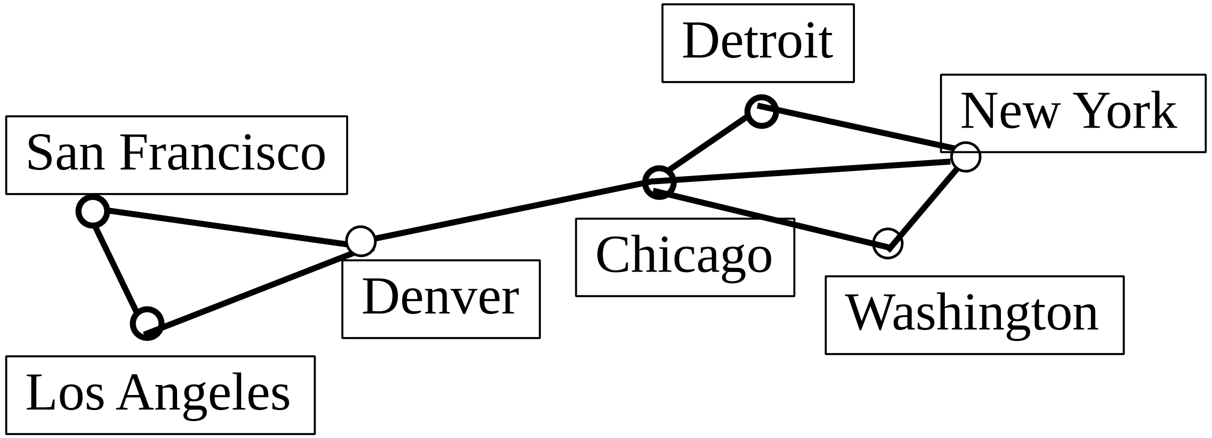


CommentAboutTopic
TopicProposal
movie/music
Movie/Music
yes
interest of user
yes
lets talk
yes
UserName
ArticleAboutTopic
Default
FoodDescription
FoodRecipe
lets talk about
name of user
food
food name
yes
yes
yes
Start
UserInfo
intents related to one-round conversation and retrieval model
Food
FoodIngredients

Graph Theory Preview

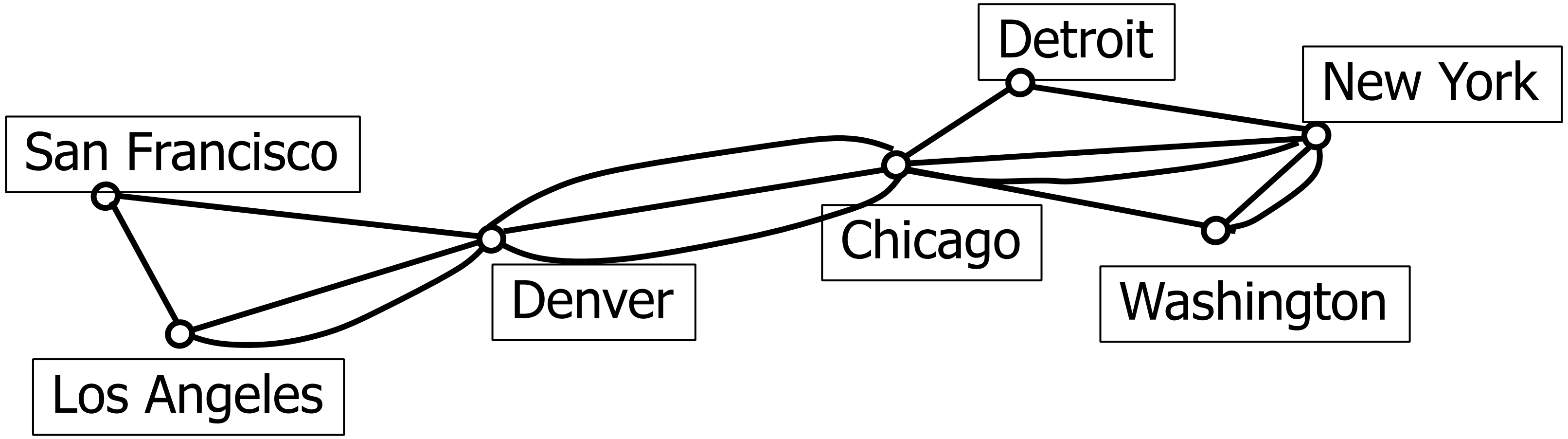
- A Graphs is simply a non-sequential data structure type that consist of **nodes (aka vertices)** and **edges**.
- A simple graph consists of:
  - A nonempty set of vertices called V
  - A set of edges (unordered pairs of distinct elements of V) called E
- The notation for describing a graph would be:
  - G = (V,E)

Simple Graph



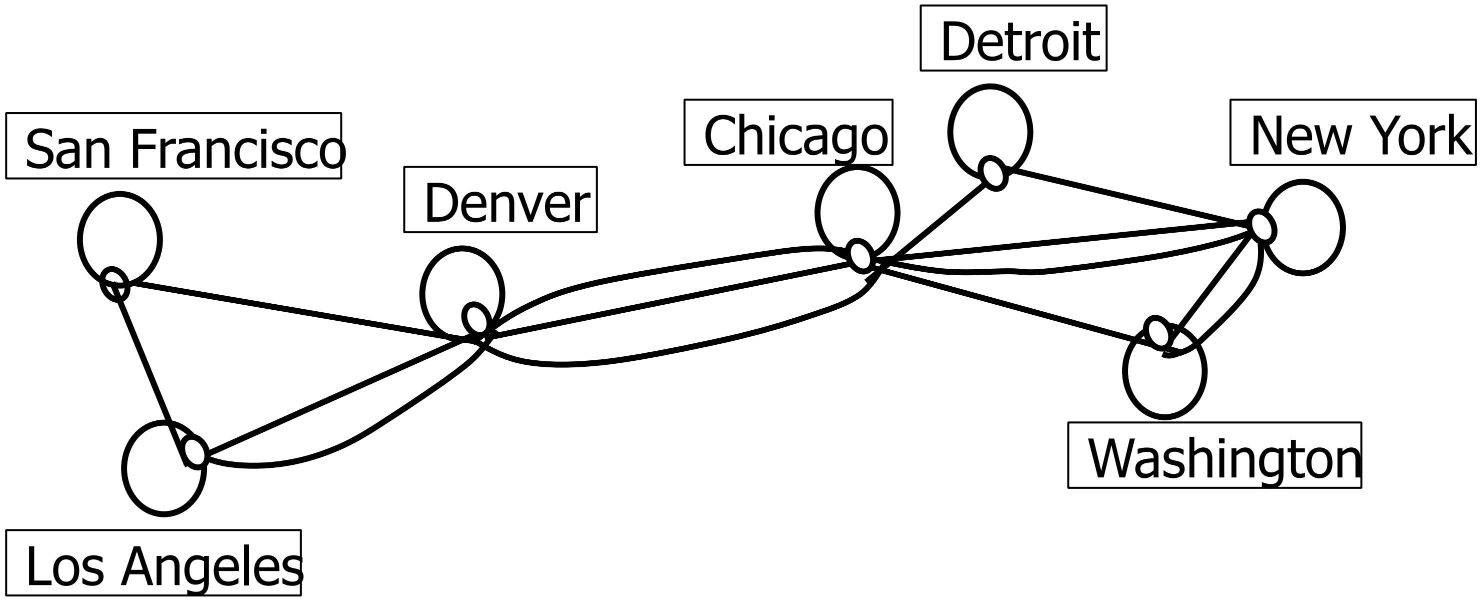
- This simple graph represents a network.
- The network is made up of computers and telephone links between computers

Multigraph



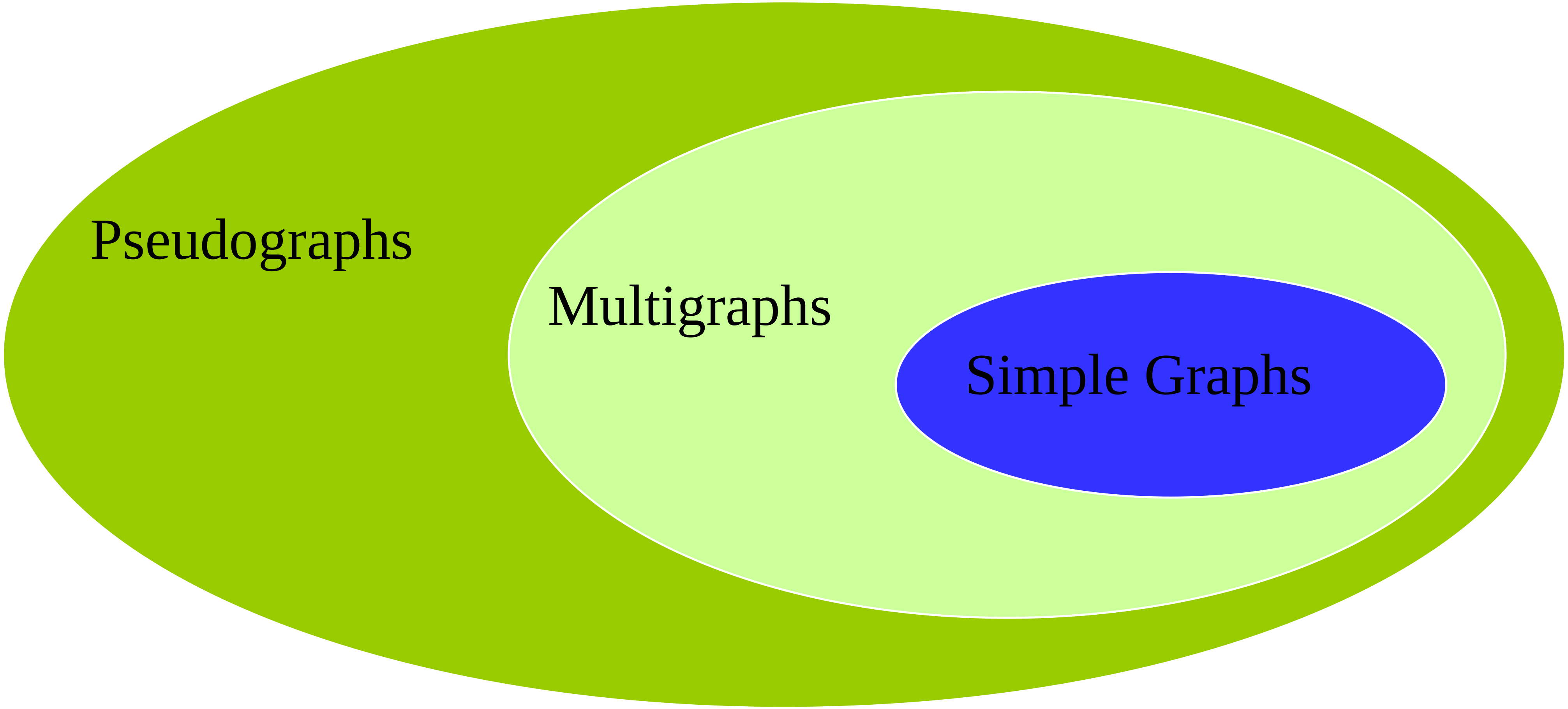
- A multigraph can have multiple edges (two or more edges connecting the same pair of vertices).
- There can be multiple telephone lines between two computers in the network.

Pseudograph

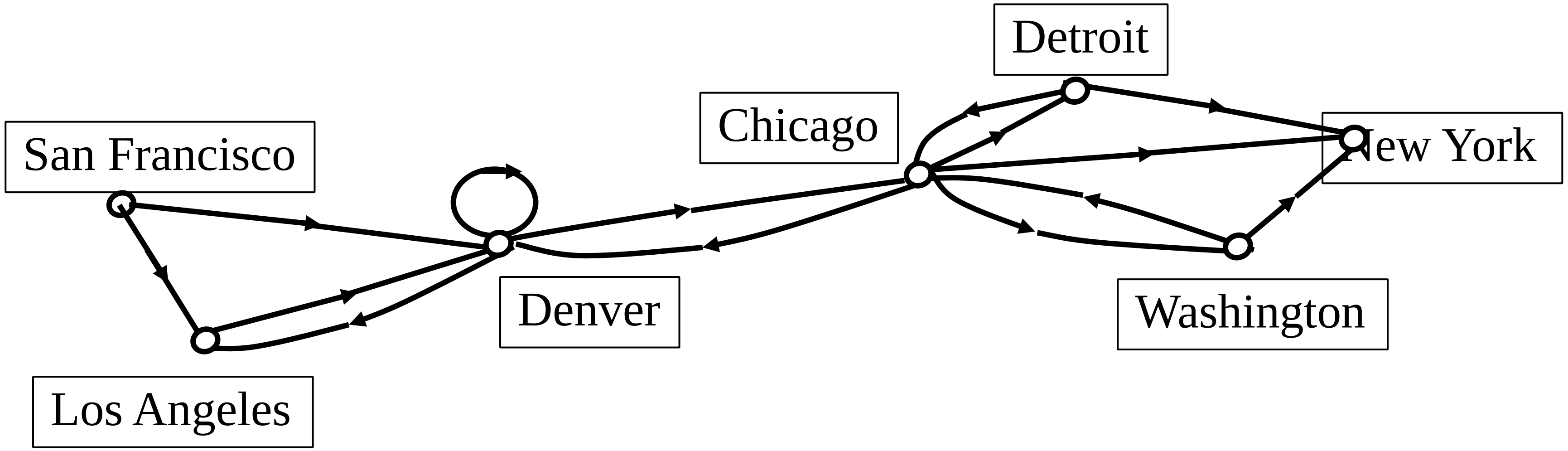


- A Pseudograph can have multiple edges and loops (an edge connecting a vertex to itself).
- There can be telephone lines in the network from a computer to itself.

Types of Undirected Graphs

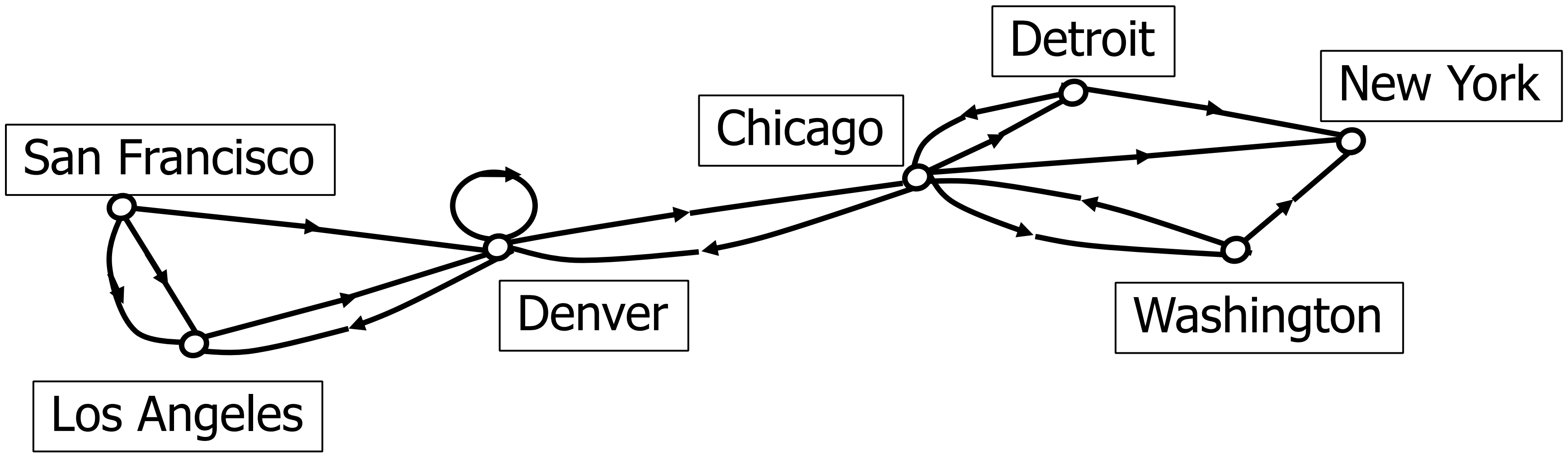


Directed Graph



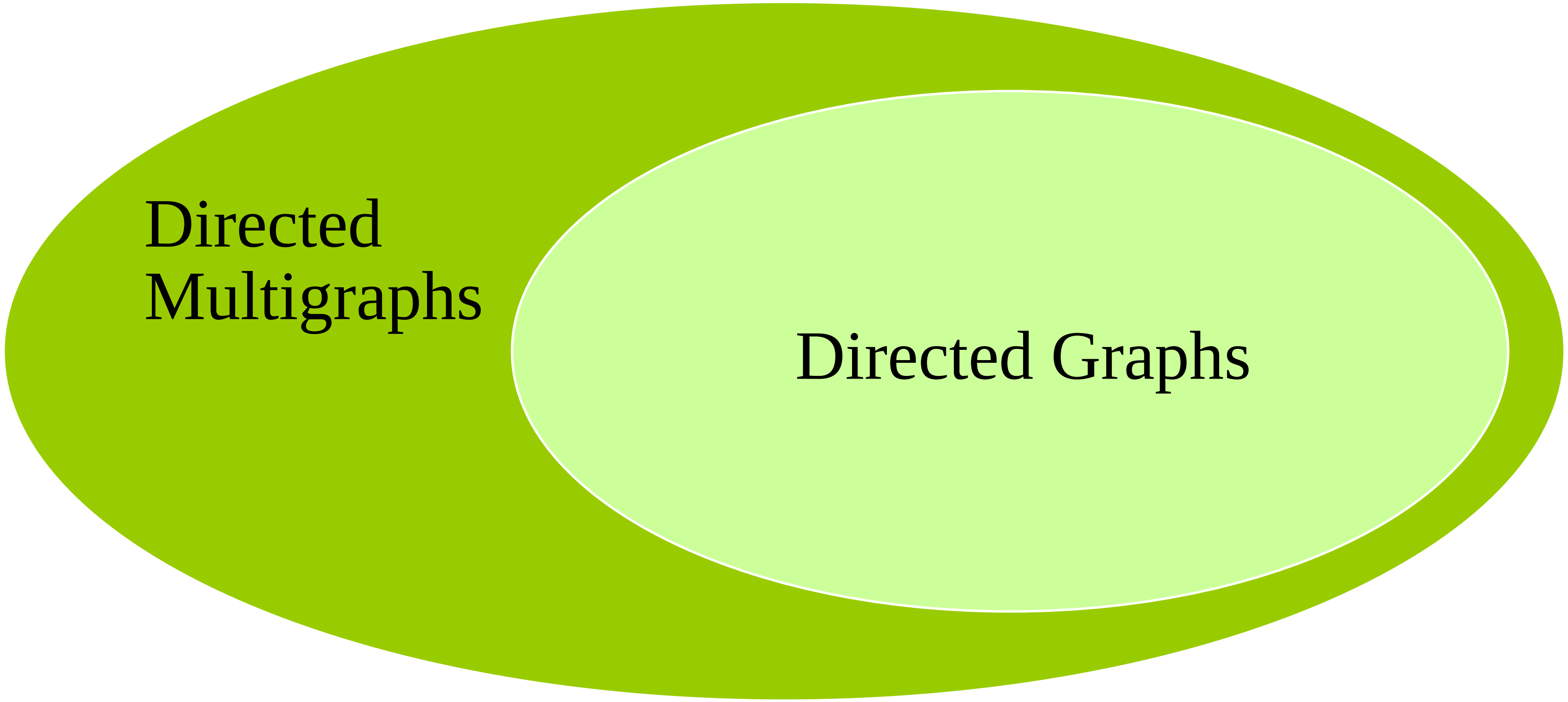
- The edges are ordered pairs of (not necessarily distinct) vertices.
- Some telephone lines in the network may operate in only one direction. Those that operate in two directions are represented by pairs of edges in opposite directions.

Directed Multigraph



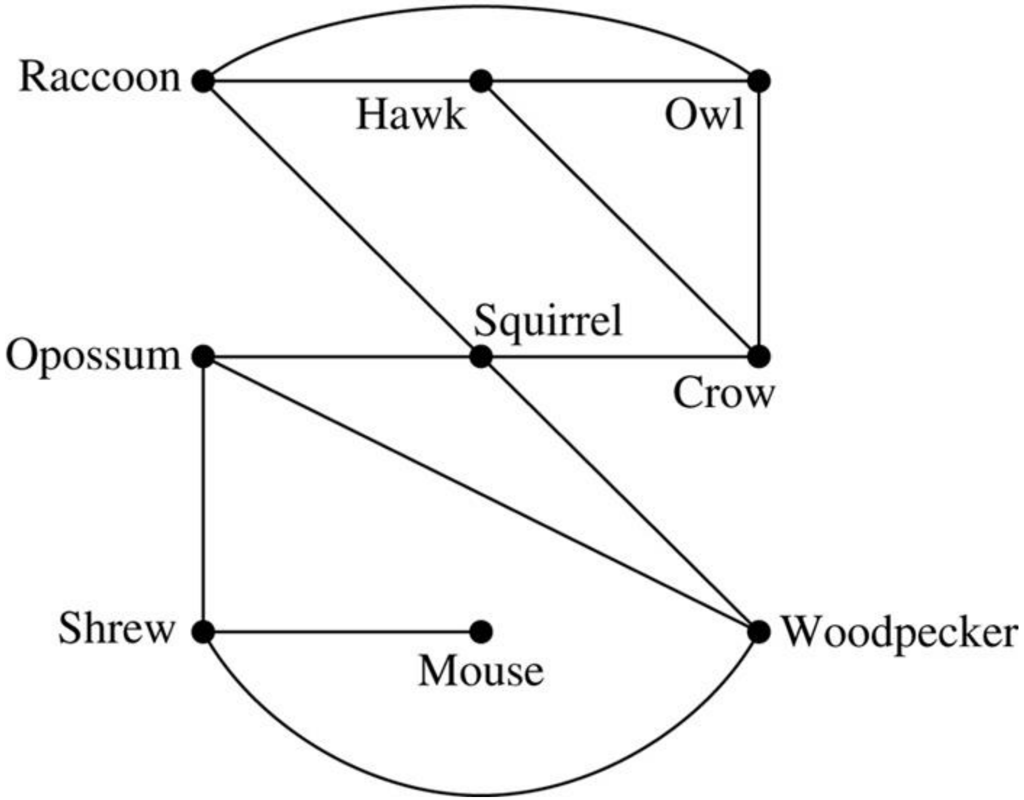
- A directed multigraph is a directed graph with multiple edges between the same two distinct vertices. Some telephone lines in the network may operate in only one direction. Those that operate in two directions are represented by pairs of edges in opposite directions.
- There may be several one-way lines in the same direction from one computer to another in the network.

Types of Undirected Graphs

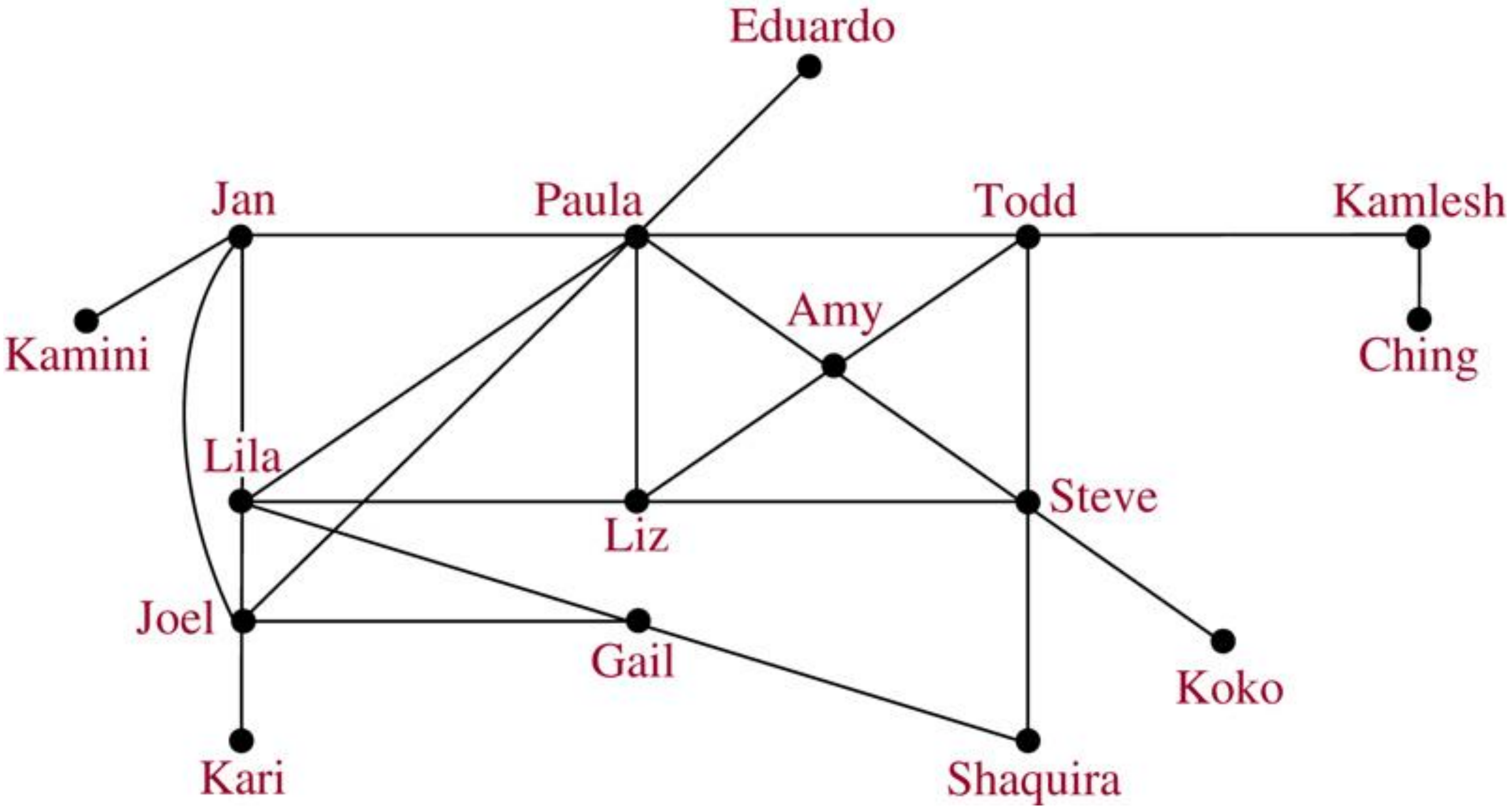


Modeling Problems with Graphs

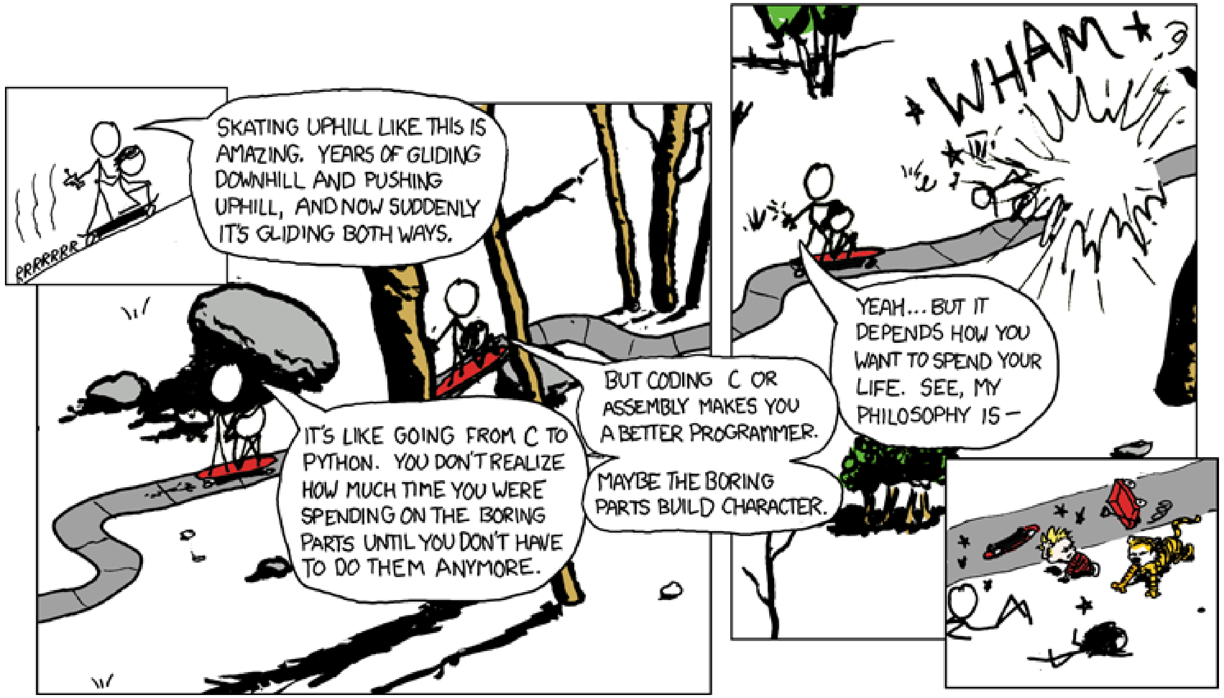
- Graphs can be used to model structures, sequences, and other relationships.
- Example: ecological niche overlay graph
  - Species are represented by vertices
  - If two species compete for food, they are connected by a vertex



Is Facebook a graph?



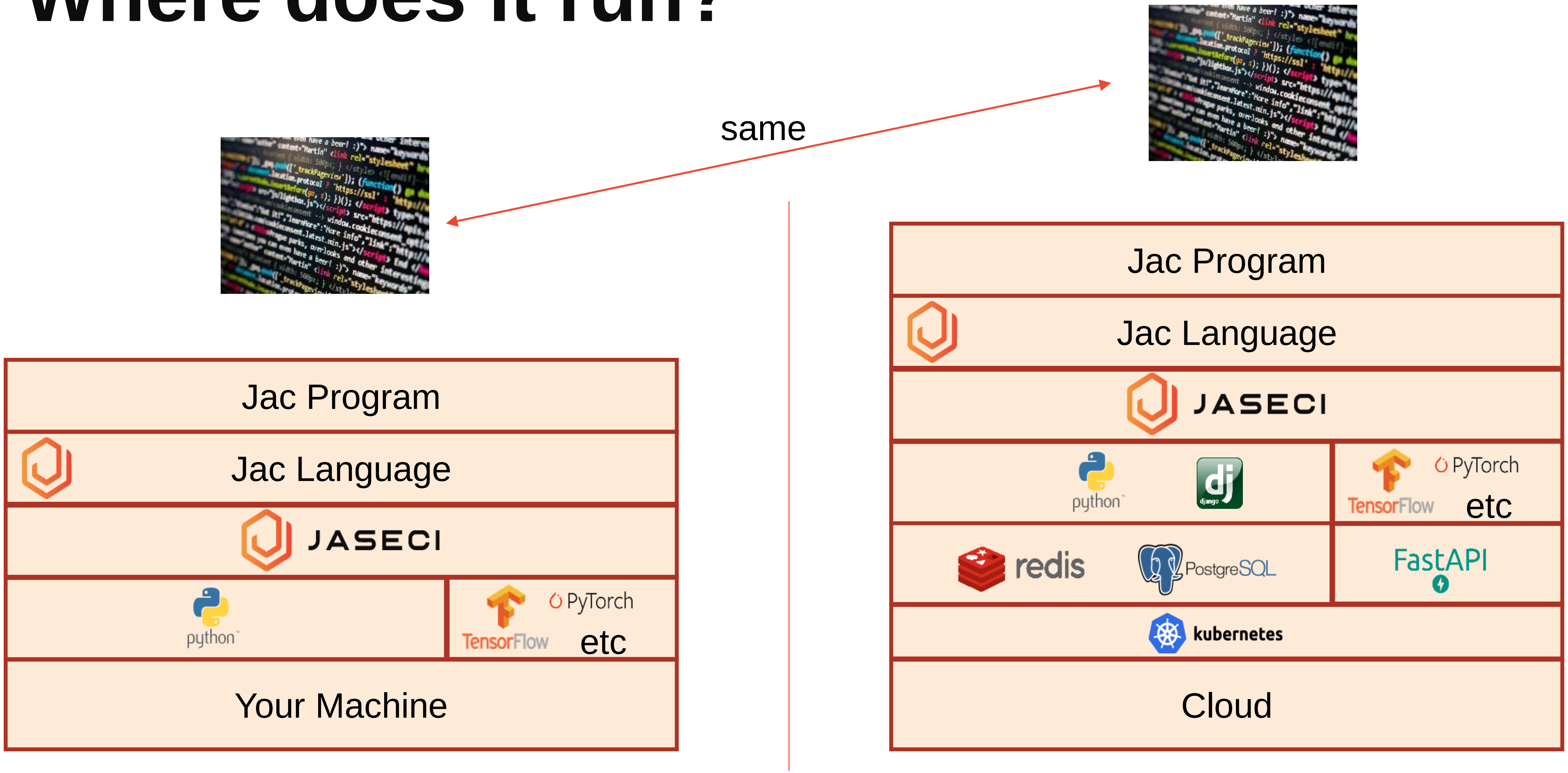
Coding Intro



A Primer on Jac

- Developed in early 2020s
- Named after **Ja**seci **C**ode
- Design inspired by Javascript and Python
- Can be used standalone or as glue
- Interpreted
- Dynamically Typed
- All types based on JSON
- Concise

Where does it run?



Basic Coding

- Code Block
  - Region of Execution
  - Defined with { } for multiple statements
  - or single statement blocks :; (for succinctness)
- First assignment to a variable creates it
  - No types needed (but types exist internally)
- Python and JS style comments
  - #, //, /* */

```
walker init {
    x = 34 - 30;  # This is a comment
    y = "Hello";
    z = 3.45;

    if(z==3.45 or y=="Bye"){  # if statement
        x=x-1;
        y=y+" World";  # the + concatenates
    }

    std.out(x);
    for i=0 to i<3 by i+=1:  # single line block
        std.out(x-i,'-', y);  # prints to screen
    report [x, y+'s'];  # adds data to payload
}
```

Basic Coding

- Assignment uses = and comparison uses ==
- For numbers +, -, *, /, %, are expected
  - Special use of + for strings is concatenation
  - Also +=, -=, *=, /=, etc
    - `a += 1` same as `a = a + 1`
- Logical operators can be symbols or words (&&, and, ||, or, !, not)
- std.out("string"); represents printing to screen
- report "string"; represents adding to return payload

```
walker init {
    x = 34 - 30;  # This is a comment
    y = "Hello";
    z = 3.45;

    if(z==3.45 or y=="Bye"){  # if statement
        x=x-1;
        y=y+" World";  # the + concatenates
    }

    std.out(x);
    for i=0 to i<3 by i+=1:  # single line block
        std.out(x-i,'-', y);  # prints to screen
    report [x, y+'s'];  # adds data to payload
}
```

Basic Coding Output

```
walker init {
    x = 34 - 30;  # This is a comment
    y = "Hello";
    z = 3.45;

    if(z==3.45 or y=="Bye"){  # if statement
        x=x-1;
        y=y+" World";  # the + concatenates
    }

    std.out(x);
    for i=0 to i<3 by i+=1:  # single line block
        std.out(x-i,'-', y);  # prints to screen
    report [x, y+'s'];  # adds data to payload
}
```

```
3
3 - Hello World
2 - Hello World
1 - Hello World
{
  "success": true,
  "report": [
    [
      3,
      "Hello Worlds"
    ]
  ]
}
```

Jac Types

- Types in Jac are 1 tro 1 mapped to JSON types

Jac Type	Json Type	Example
String	String	“Hello”, ‘world’, “Joe’s World”
Int, float	Number	4, 3.14
Dict, node, edge	Json object	{“five”: 5}, node, edge
List	Array	[4, 3.14, 5]
Bool	Bool	True
Null	Null	null

Jac Types Output

```
walker init {
    a=5;
    b=5.0;
    c=true;
    d='5';
    e=[a, b, c, d, 5];
    f={'num': 5};

    summary = {'int': a, 'float': b, 'bool': c,
               'string': d, 'list': e, 'dict': f};

    report summary;
}
```


```
{
  "success": true,
  "report": [
    {
      "int": 5,
      "float": 5.0,
      "bool": true,
      "string": "5",
      "list": [
        5,
        5.0,
        true,
        "5",
        5
      ],
      "dict": {
        "num": 5
      }
    }
  ]
}
```

Naming Variables

- Names are case sensitive and cannot start with a number. They can contain letters, numbers, and underscores.
  - `bob   Bob   _bob   _2_bob_   bob_2   BoB`
- There are some reserved words:
  - `import, node, ignore, take, entry, activity, exit, spawn, with, edge, walker, and, or, if, elif, else, for, with, by, while, continue, break, disengage, report, anchor, has, can, true, false, context, info, details, try, strict, length, test, type, str, int, float, list, dict, bool, digraph, subgraph, test, by, in, to, skip, assert, etc`

Note: Jac Piggy Backs on Python

- Jac takes a piggy-back approach
  - Interpreter translate to python execution
- When in doubt, python rules apply
- My notice some odd error outputs from the python layer

Working with Lists and Strings

- Lists
  - `li = [“abc”, 34, 4.34, 23]`
  - `report li[0];`
  - `li = li[1:3]; report li[-1];`
  - `li.l::sort; report li[-1];`
  - More on lists: https://towardsdatascience.com/python-basics-6-lists-and-list-manipulation-a56be62b1f95
- Strings
  - `st = “Hello” + ‘ World’`
  - `report st.s::split;`
  - `report st[3:].s::upper.s::split(‘r’);`

Working with Dictionaries

- Dictionary
  - `dt = {`
  - `'one': 1,`
  - `'two': 2,`
  - `'three': 3,`
  - `'four': 4`
  - `};`

- `report dt['one']=6;`
- `report dt.d::keys;`
- `report dt.d::values;`
- `report dt.d::items;`
- `report dt;`

Control Flow

- For loops
  - Loops that specify start, and range to increment
- While loops
  - Loops that test a condition each iteration
- If Statement
  - Decides whether to execute a block based on condition
- Break
  - Quit the loop immediately
- Continue
  - Start next iteration immediately

```
walker init {
    fav_nums=[];

    for i=0 to i<10 by i+=1:
        fav_nums.l::append(i*2);
    report fav_nums;

    fancy_str = "";
    for i in fav_nums {
        fancy_str = fancy_str + "two * " + i.str +
                    " = " + (i*2).str + " # ";
    }
    report fancy_str;
    count_down = fav_nums[-1];
    while (count_down > 0) {
        count_down -= 1;
        if (count_down == 14):
            continue;
        std.out("I'm at countdown "+count_down.str);
        if (count_down == 10):
            break;
    }
}
```

Control Flow

- Casting with .type notation
- (5).str = “5”
- (“4”).int = 4
- “4”.int * 3 = 12
- “4” * 3 = Crash!

```
walker init {
    fav_nums=[];

    for i=0 to i<10 by i+=1:
        fav_nums.l::append(i*2);
    report fav_nums;

    fancy_str = "";
    for i in fav_nums {
        fancy_str = fancy_str + "two * " + i.str +
                    " = " + (i*2).str + " # ";
    }
    report fancy_str;
    count_down = fav_nums[-1];
    while (count_down > 0) {
        count_down -= 1;
        if (count_down == 14):
            continue;
        std.out("I'm at countdown "+count_down.str);
        if (count_down == 10):
            break;
    }
}
```

Control Flow Output

```
walker init {
    fav_nums=[];

    for i=0 to i<10 by i+=1:
        fav_nums.l::append(i*2);
    report fav_nums;

    fancy_str = "";
    for i in fav_nums {
        fancy_str = fancy_str + "two * " + i.str +
                  " = " + (i*2).str + " # ";
    }
    report fancy_str;
    count_down = fav_nums[-1];
    while (count_down > 0) {
        count_down -= 1;
        if (count_down == 14):
            continue;
        std.out("I'm at countdown "+count_down.str);
        if (count_down == 10):
            break;
    }
}
```

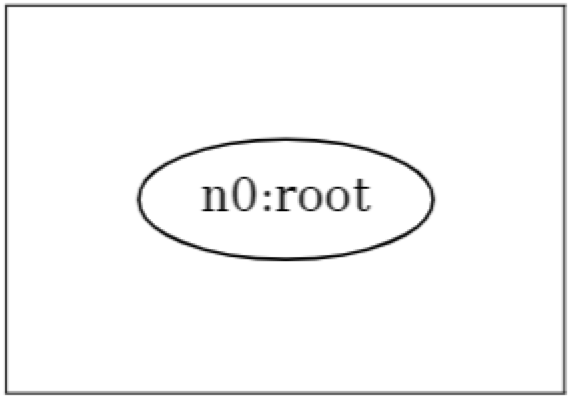
```
I'm at countdown 17
I'm at countdown 16
I'm at countdown 15
I'm at countdown 13
I'm at countdown 12
I'm at countdown 11
I'm at countdown 10
{
  "success": true,
  "report": [
    [
      0,
      2,
      4,
      6,
      8,
      10,
      12,
      14,
      16,
      18
    ],
    "two * 0 = 0 # two * 2 = 4 # two * 4 = 8 # two * 6 = 12 # two * 8 = 16 # two *
10 = 20 # two * 12 = 24 # two * 14 = 28 # two * 16 = 32 # two * 18 = 36 # "
  ]
}
```

Graphs in Jac

Nodes and Edges: Where Memory Starts

```
walker init {
}
```

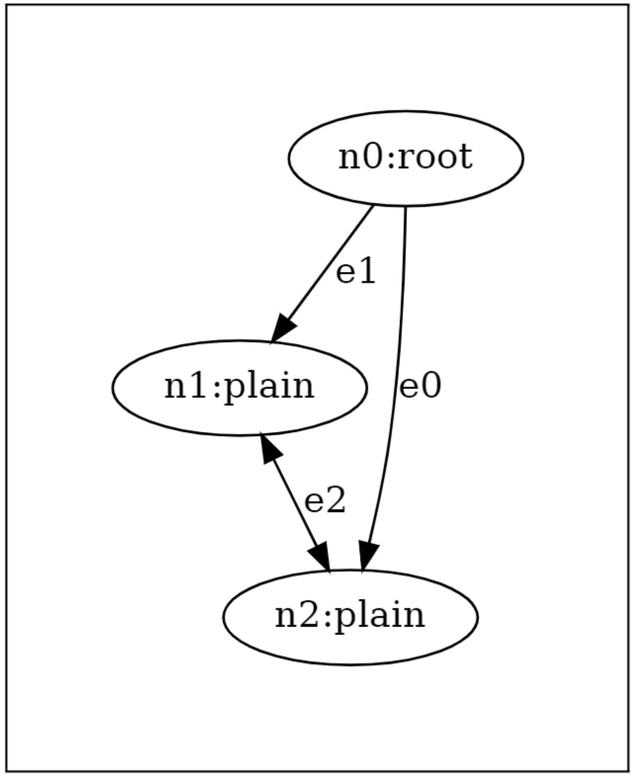
Memory



Nodes and Edges: Basic

```
node plain;

walker init {
    node1 = spawn node::plain;
    node2 = spawn node::plain;
    node1 <--> node2;
    here --> node1;
    node2 <-- here;
}
```

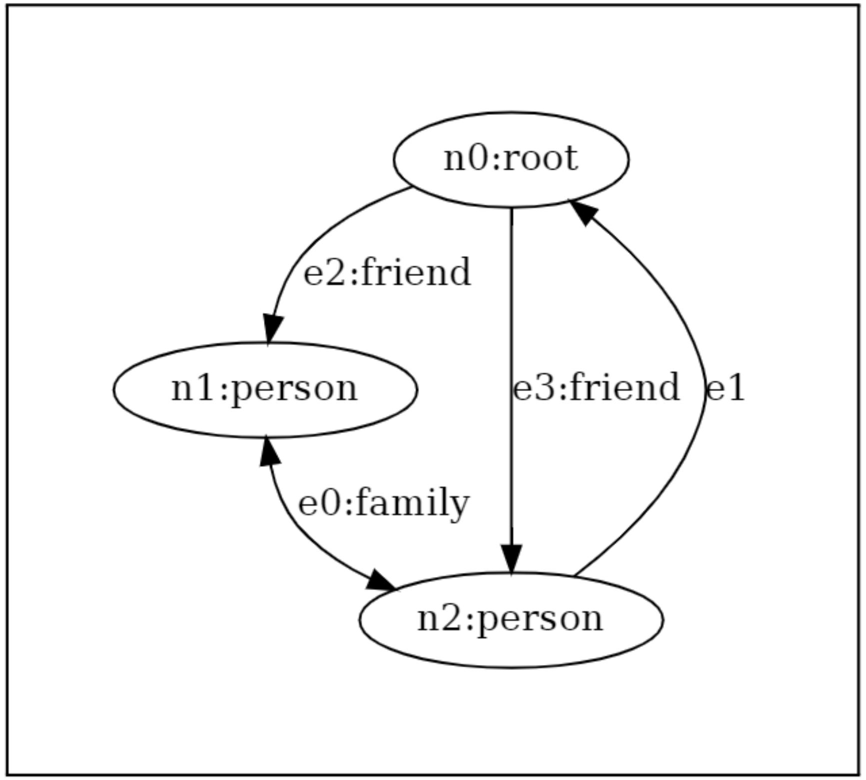
Memory



Nodes and Edges: Named Edges

```
node person;
edge family;
edge friend;

walker init {
    node1 = spawn node::person;
    node2 = spawn node::person;
    node1 <-[family]-> node2;
    here -[friend]-> node1;
    node2 <-[friend]- here;

    # named and unnamed edges and nodes
can be mixed
    node2 --> here;
}
```

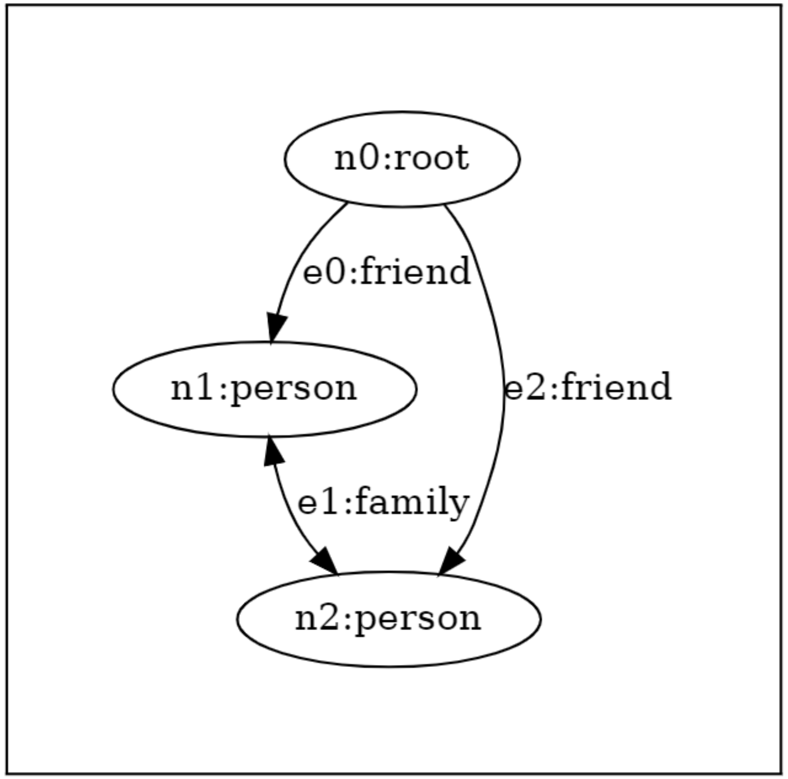
Memory



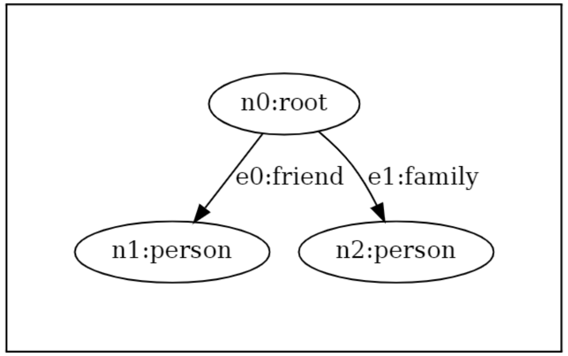
Nodes and Edges: Spawn Connects

```
node person;
edge friend;
edge family;

walker init {
    node1 = spawn here -[friend]->
                node::person;
    node2 = spawn node1 <-[family]->
               node::person;
    here -[friend]-> node2;
}
```

Memory



Node / Edge Contexts




```
node person {
    has name;
    has age;
    has birthday, profession;
}

edge friend: has meeting_place;
edge family: has kind;

walker init {
    person1 = spawn here -[friend]-> node::person;
    person2 = spawn here -[family]-> node::person;
    person1.name = "Josh"; person1.age = 32;
    person2.name = "Jane"; person2.age = 30;
    e1 = -[friend]->.edge[0];
    e1.meeting_place = "college";
    e2 = -[family]->.edge[0];
    e2.kind = "sister";

    std.out("Context for our people nodes:");
    for i in -->: std.out(i.context);
    # or, for i in -->.node: std.out(i.context);
    std.out("\nContext for our edges to those people:");
    for i in -->.edge: std.out(i.context);
}
```

```
Context for our people nodes:
{"name": "Josh", "age": 32, "birthday": null,
"profession": null}
{"name": "Jane", "age": 30, "birthday": null,
"profession": null}

Context for our edges to those people:
{"meeting_place": "college"}
{"kind": "sister"}
{
  "success": true,
  "report": []
}
```

More Concise Contexts

```
node person {
    has name;
    has age;
    has birthday, profession;
}

edge friend: has meeting_place;
edge family: has kind;

walker init {
    person1 = spawn here -[friend]-> node::person;
    person2 = spawn here -[family]-> node::person;
    person1.name = "Josh"; person1.age = 32;
    person2.name = "Jane"; person2.age = 30;
    e1 = -[friend]->.edge[0];
    e1.meeting_place = "college";
    e2 = -[family]->.edge[0];
    e2.kind = "sister";

    std.out("Context for our people nodes:");
    for i in -->: std.out(i.context);
    # or, for i in -->.node: std.out(i.context);
    std.out("\nContext for our edges to those people:");
    for i in -->.edge: std.out(i.context);
}
```

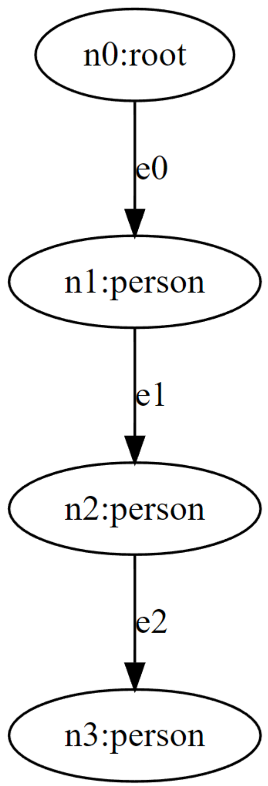
```
node person: has name, age, birthday, profession;
edge friend: has meeting_place;
edge family: has kind;

walker init {
    person1 = spawn here -[friend(meeting_place =
"college")] ->
        node::person(name = "Josh", age = 32);
    person2 = spawn here -[family(kind = "sister")] ->
        node::person(name = "Jane", age = 30);

    std.out("Context for our people nodes and edges:");
    for i in -->:
        std.out(i.context, '\n', i.edge[0].context);
}
```

Walking Graphs

```
node person: has name;

walker get_names {
    std.out(here.name)
    take -->;
}

walker build_example {
    node1 = spawn here --> node::person(name="Joe");
    node2 = spawn node1 --> node::person(name="Susan");
    spawn node2 --> node::person(name="Matt");
}

walker init {
    root {
        spawn here walker::build_example;
        take -->;
    }
    person {
        spawn here walker::get_names;
        disengage;
    }
}
```

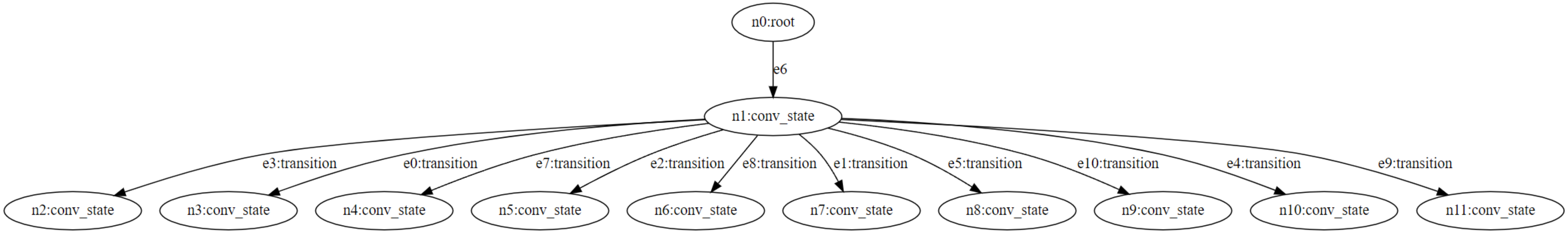
```
Joe
Susan
Matt
```

Memory



Assignment 2 Deep Dive

Graph Example of Assignment 1



Deep Dive on Thursday But Now

- Enter every example in this slide deck into your VSCode and
- Run each one
- Make a few changes to see what happens
- Do it by hand
    - "Wax on / Wax off"