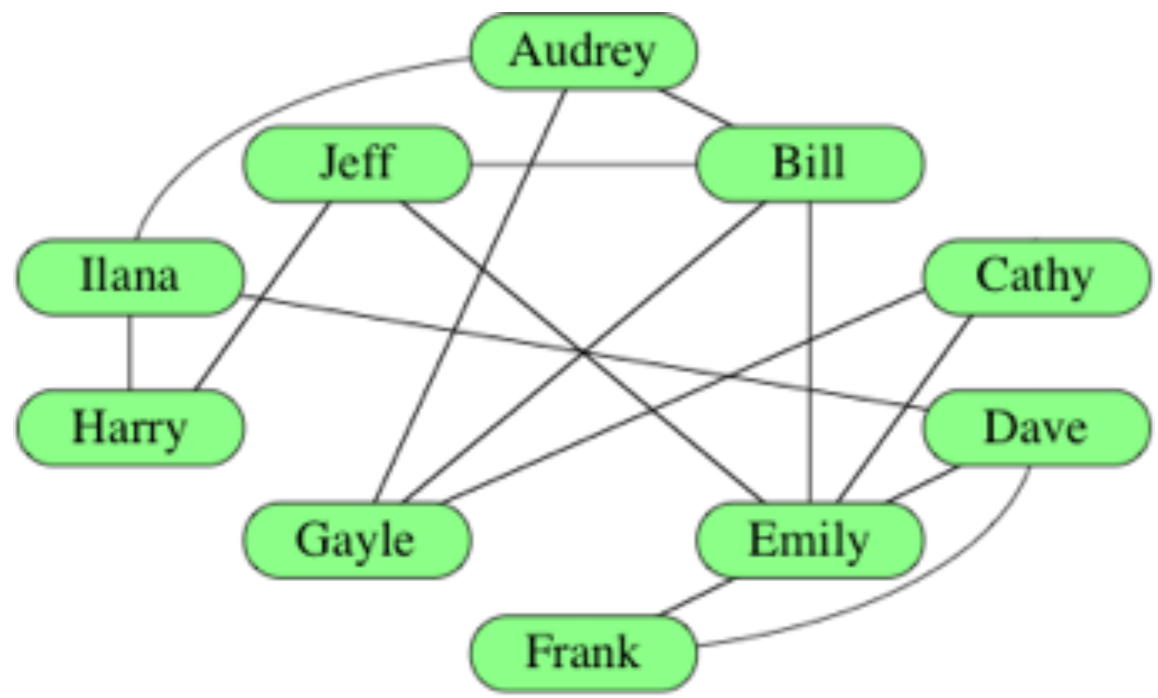
uninformed graph search

graphs express connections between data

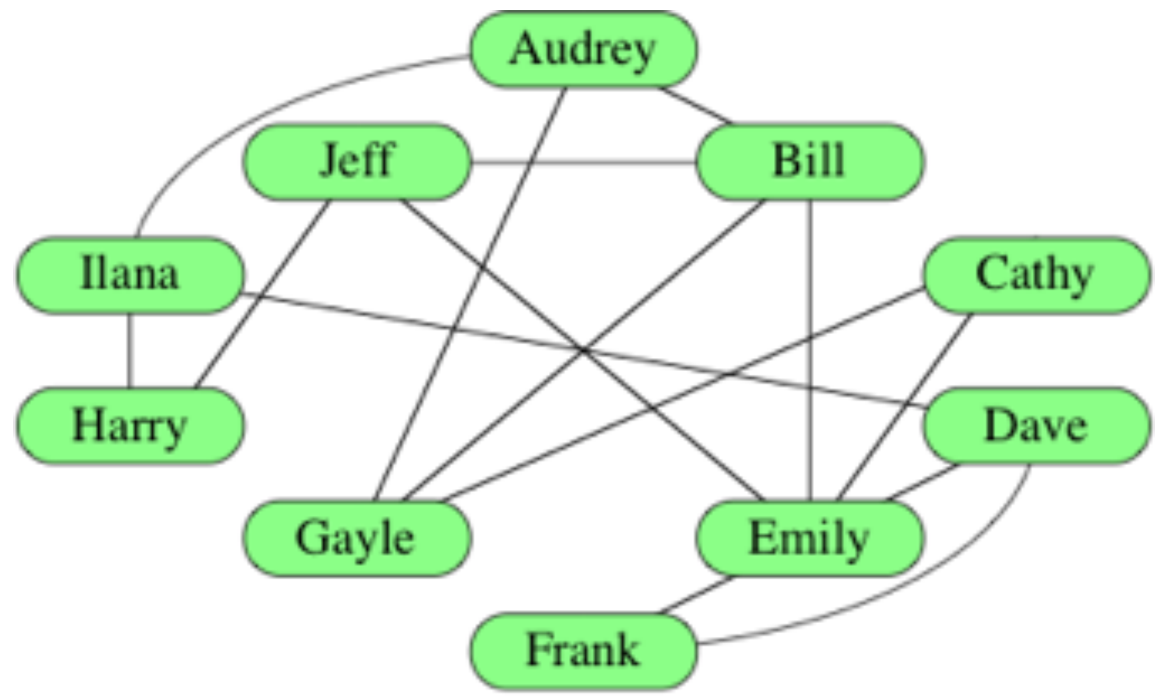
A social network:



Each **vertex** stores some data. Each **edge** connects a pair of vertices. (The words **node** and **vertex** are used interchangeably.)

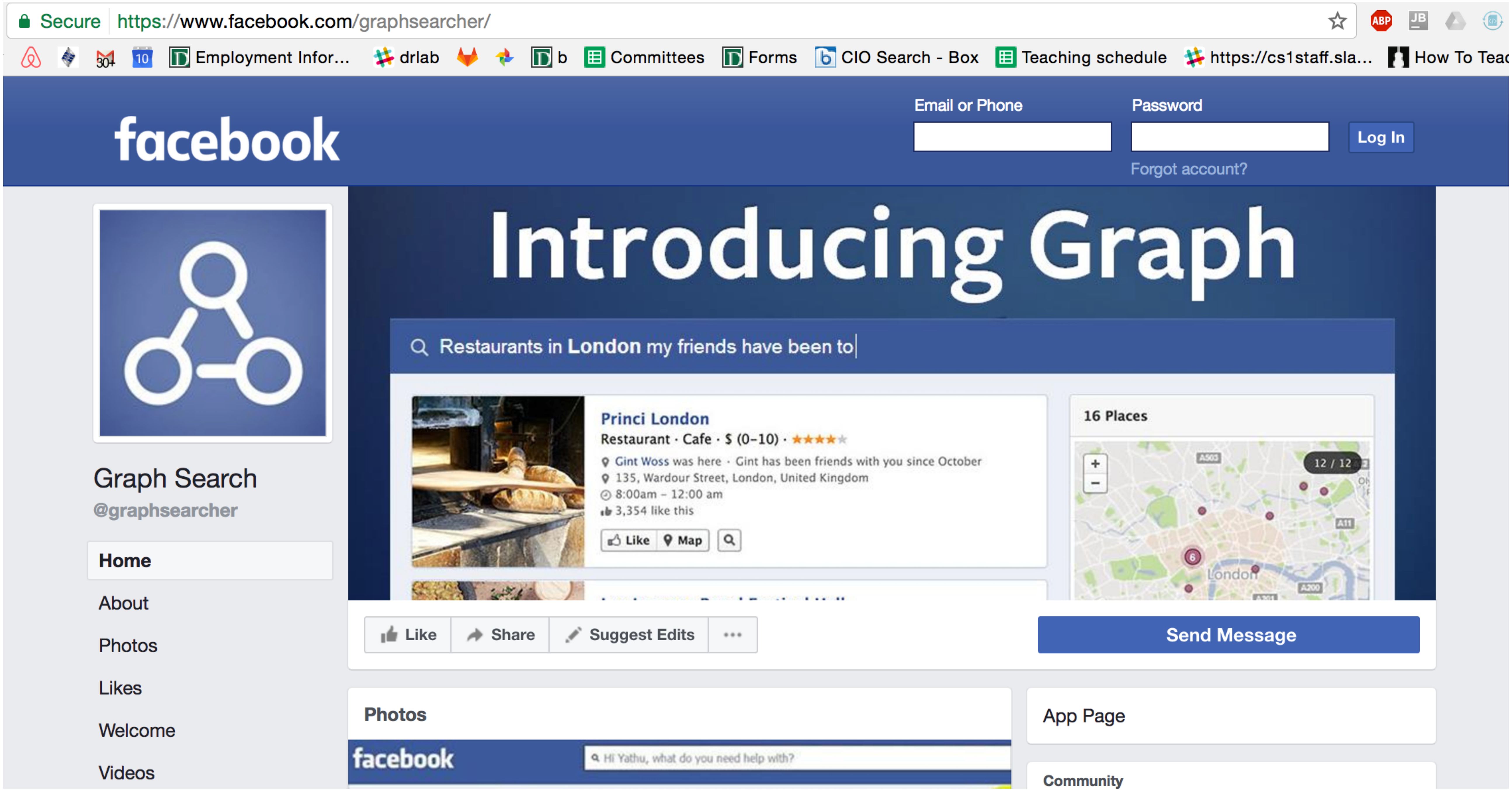
If there are *n* vertices, there may be up to *n (n - 1)* edges.

questions we could ask



- Does Cathy know Gayle? (Yes, there is an edge.)
- Is there a pathway between Harry and Emily? (Same component.)
- What is the shortest path between Harry and Emily? (H to J to E)
- Who is the most well-connected person? (Emily, vertex degree 5.)
- Largest group in which each knows everyone else (clique)?

graph models: social networks



graph models: social networks

Secure https://www.cs.cmu.edu/~./enron/

Enron Email Dataset

This dataset was collected and prepared by the CALO Project (A Cognitive Assistant that Learns and Organizes). It contains data from about 150 users, mostly senior management of Enron, organized into folders. The corpus contains a total of about 0.5M messages. This data was originally made public, and posted to the web, by the Federal Energy Regulatory Commission during its investigation.

The email dataset was later purchased by Leslie Kaelbling at MIT, and turned out to have a number of integrity problems. A number of folks at SRI, notably Melinda Gervasio, worked hard to correct these problems, and it is thanks to them (not me) that the dataset is available. The dataset here does not include attachments, and some messages have been deleted "as part of a redaction effort due to requests from affected employees". Invalid email addresses were converted to something of the form user@enron.com whenever possible (i.e., recipient is specified in some parse-able format like "Doe, John" or "Mary K. Smith") and to no_address@enron.com when no recipient was specified.

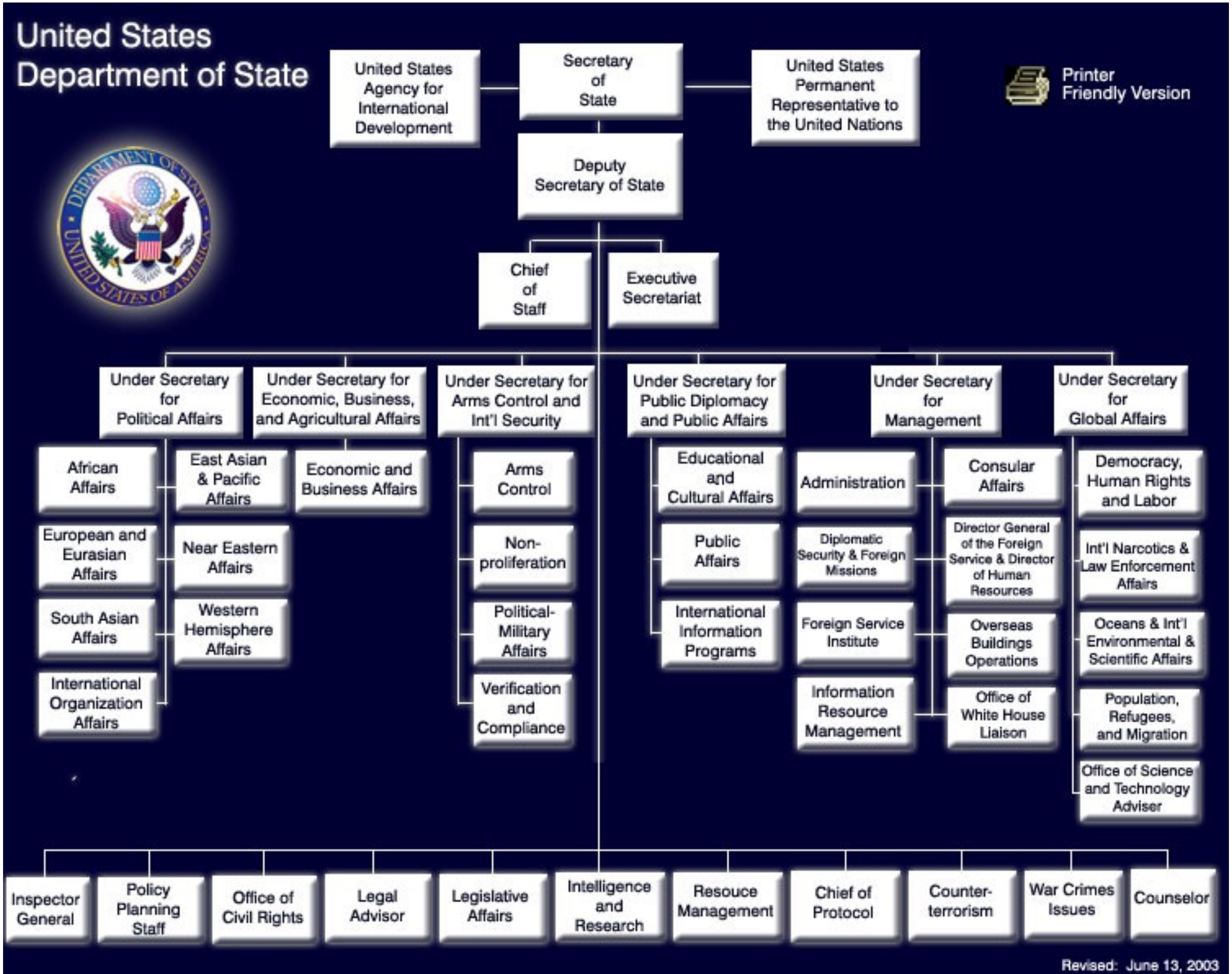
I get a number of questions about this corpus each week, which I am unable to answer, mostly because they deal with preparation issues and such that I just don't know about. If you ask me a question and I don't answer, please don't feel slighted.

I am distributing this dataset as a resource for researchers who are interested in improving current email tools, or understanding how email is currently used. This data is valuable; to my knowledge it is the only substantial collection of "real" email that is public. The reason other datasets are not public is because of privacy concerns. In using this dataset, please be sensitive to the privacy of the people involved (and remember that many of these people were certainly not involved in any of the actions which precipitated the investigation.)

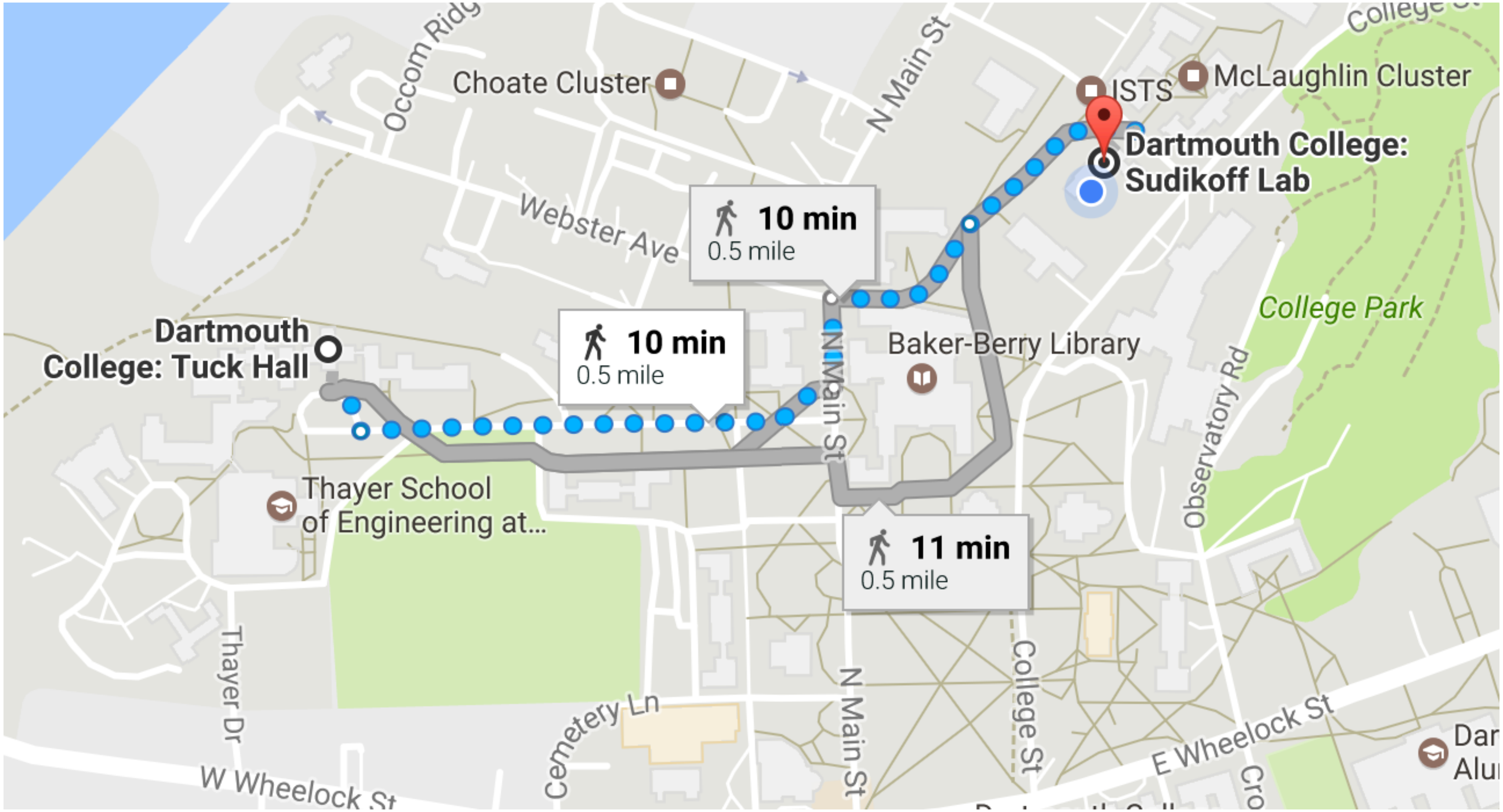
- Prior versions of the dataset are **no longer being distributed.** If you are using the March 2, 2004 Version; the August 21, 2009 Version; or the April 2, 2011 Version this dataset for your work, you are requested to replace it with the newer version of the dataset below, or make the the appropriate changes to your local copy.
- May 7, 2015 Version of dataset (about 423Mb, tarred and gzipped).

There are also several on-line databases that allow you to search the data, at Enronemail.com, UCB, and www.enron-mail.com

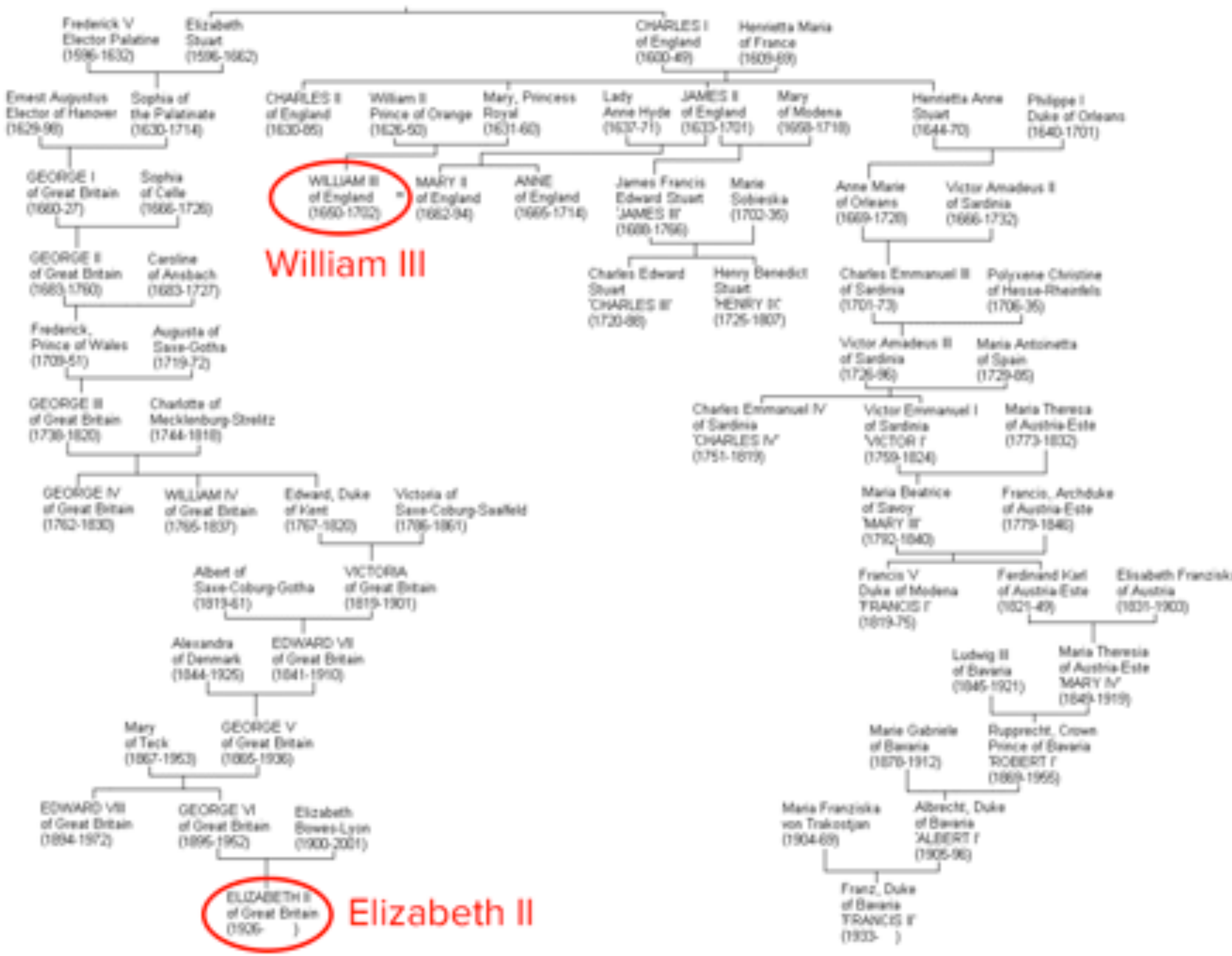
graph models: hierarchies



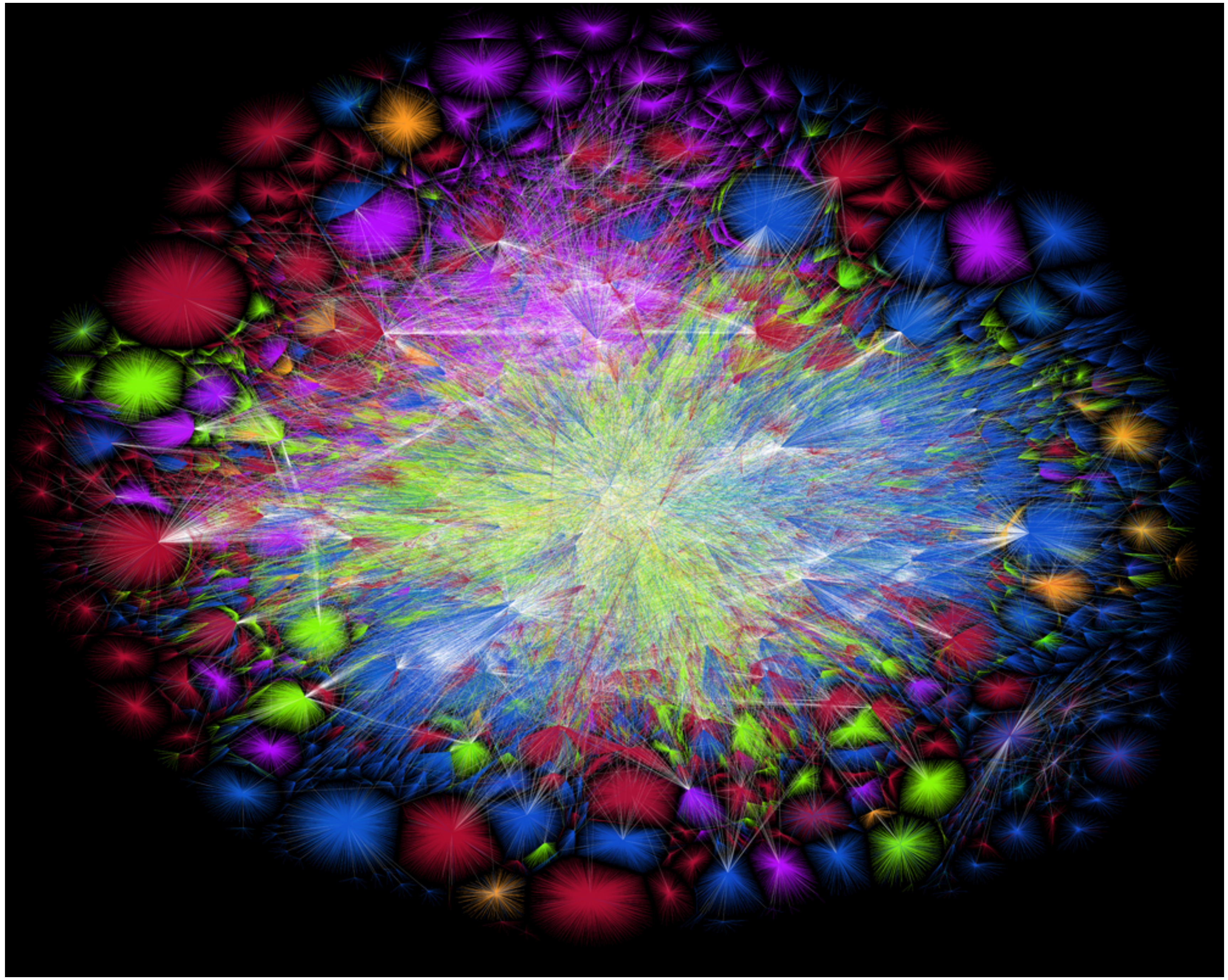
graph models: physical maps



graph models: genealogy



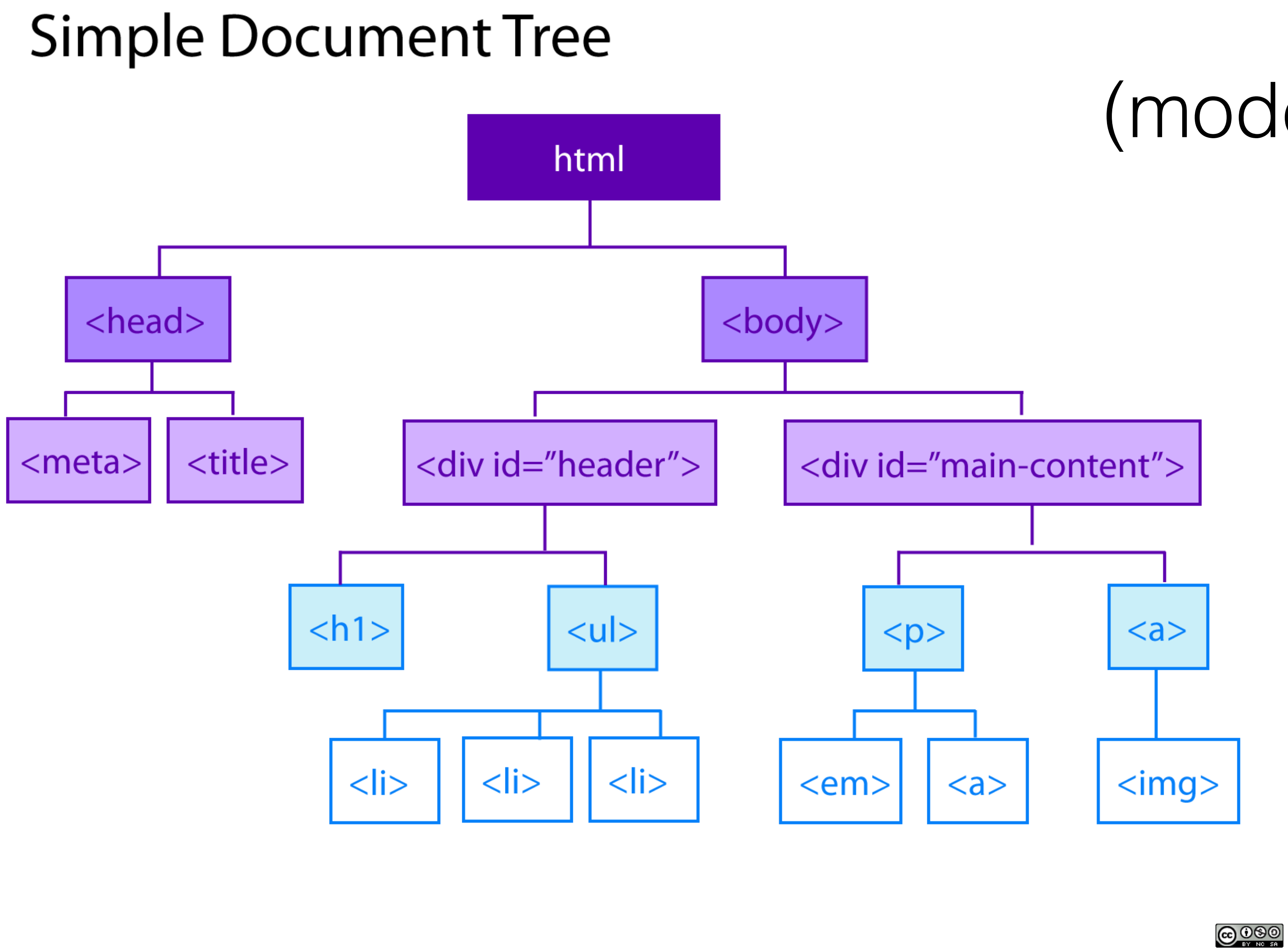
graph models: the web



(Image from the Opte project.)

graph models: document structure

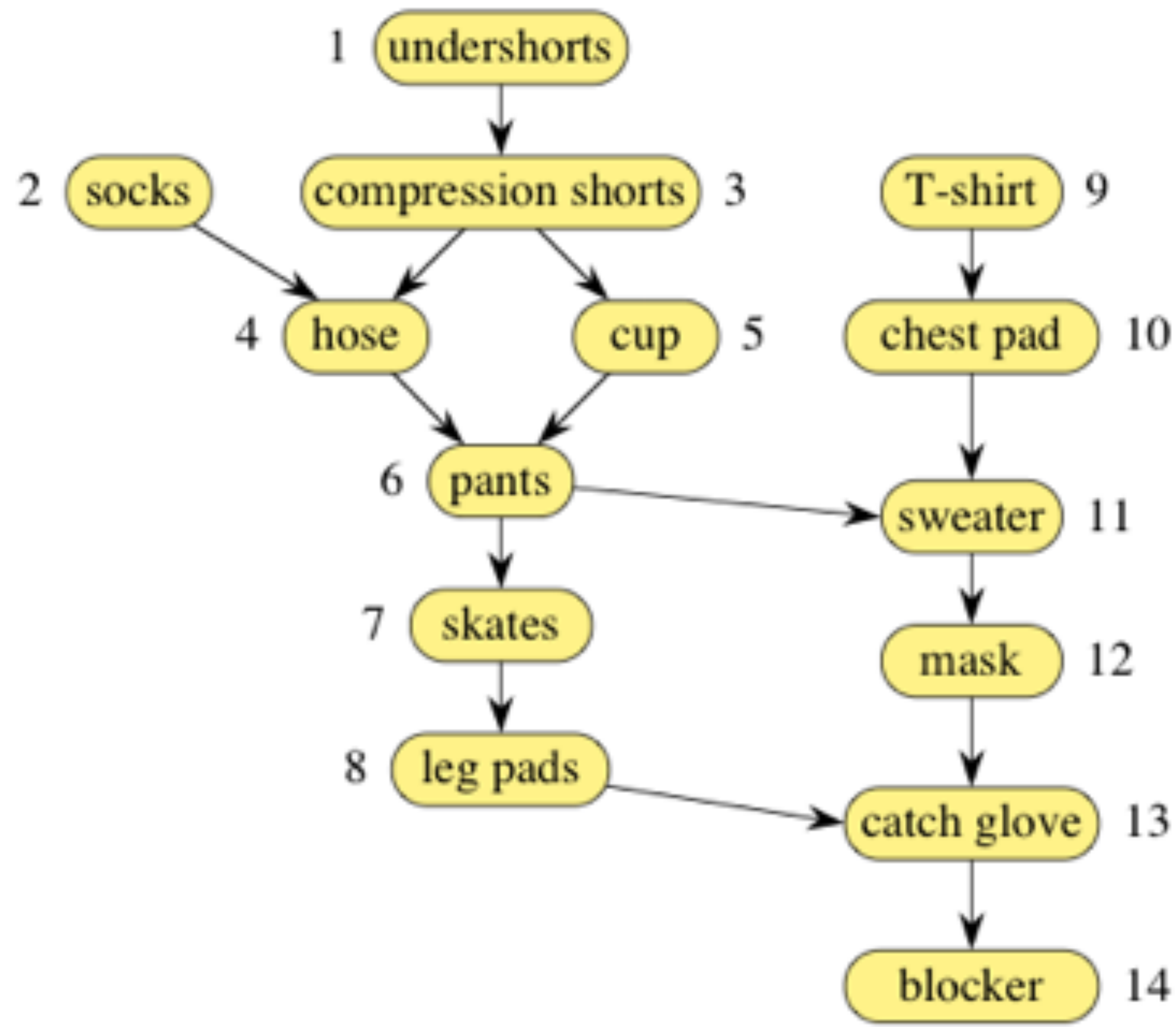
(models containership)



(Image from dabrook.org.)

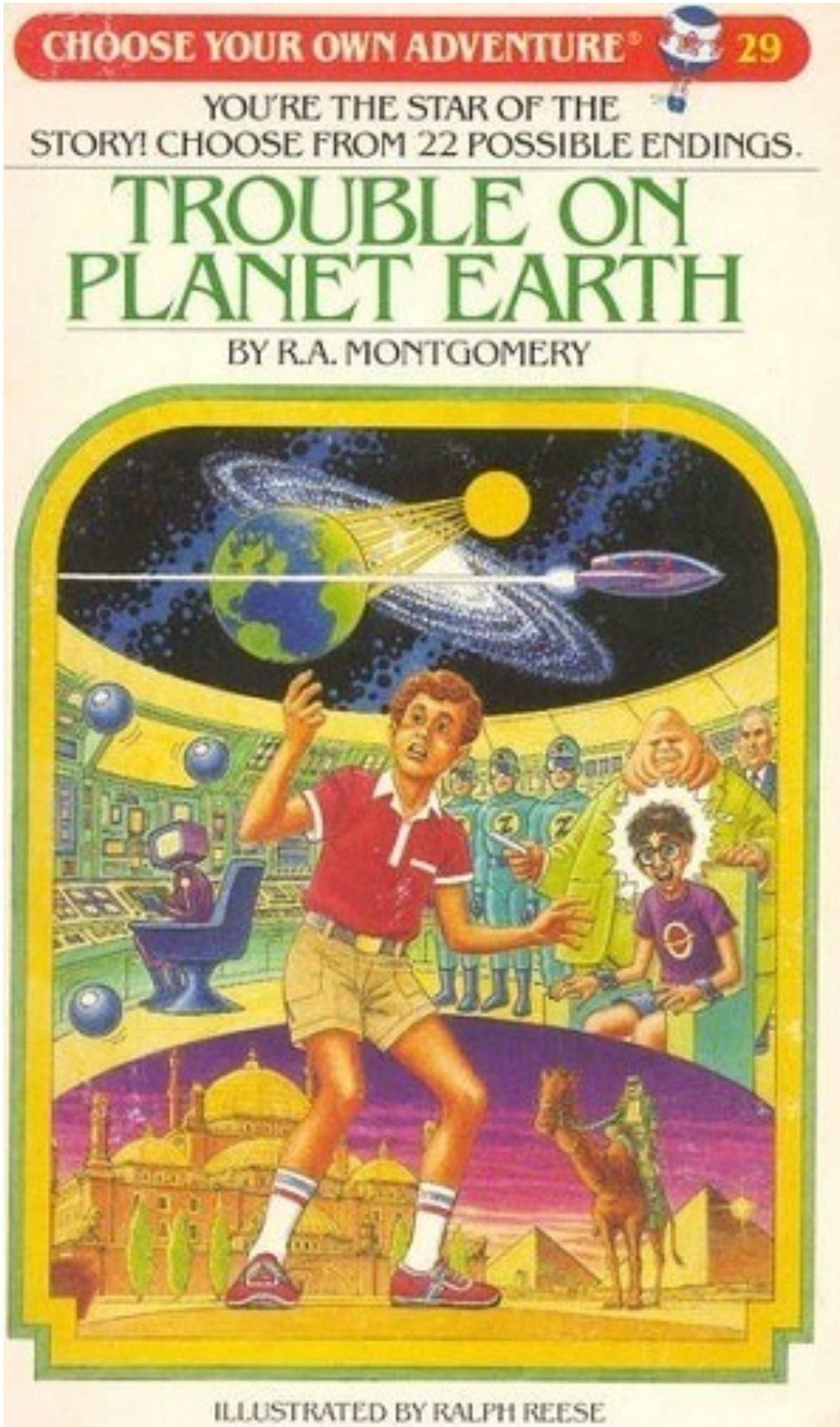
graph models: ordering constraints

Restrictions on the order in which a hockey goalie can get dressed:

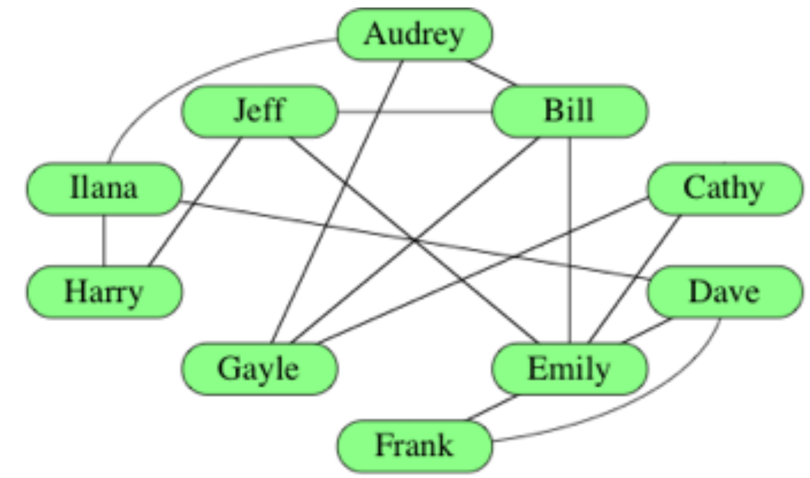


(Note: **directed** graph. Example by Tom Cormen.)

graph models: decisions and AI



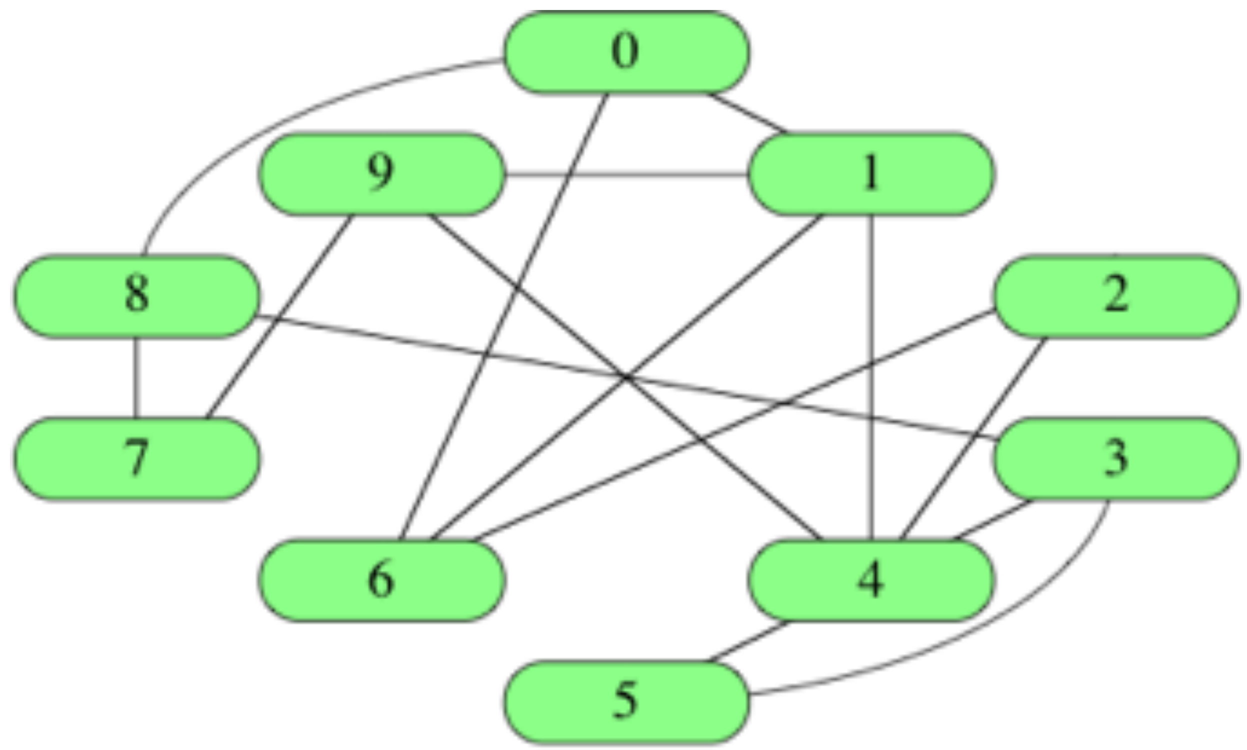
questions we could ask



- Does Cathy know Gayle? (Yes, there is an edge.)
- Is there a pathway between Harry and Emily? (Same component.)
- What is the shortest path between Harry and Emily? (H to J to E)
- Who is the most well-connected person? (Emily, vertex degree 5.)
- Largest group in which each knows everyone else (clique)?

Food for thought: what are analogous questions for each of the previous applications?

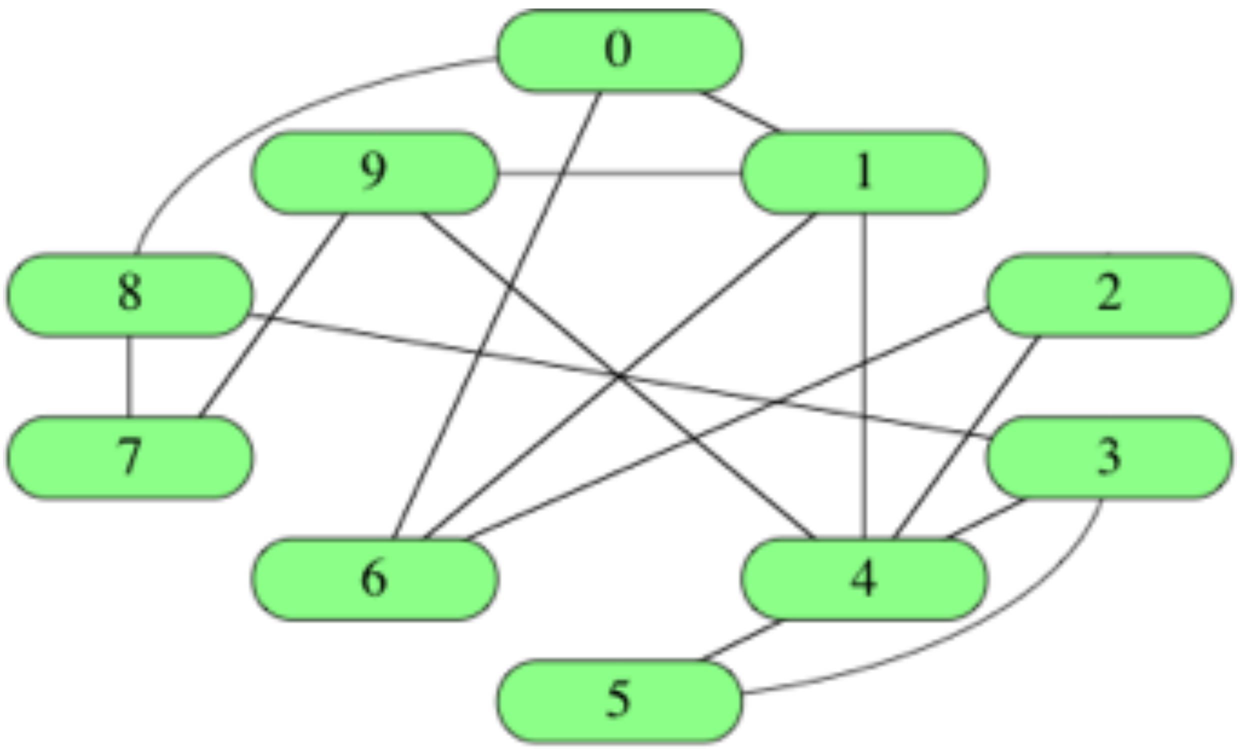
representing a graph: edge list



[[0,1], [0,6], [0,8], [1,4], [1,6], [1,9], [2,4], [2,6], [3,4], [3,5], [3,8], [4,5], [4,9], [7,8], [7,9]]

- How long does it take to answer whether two vertices are connected?
- How much memory is required?

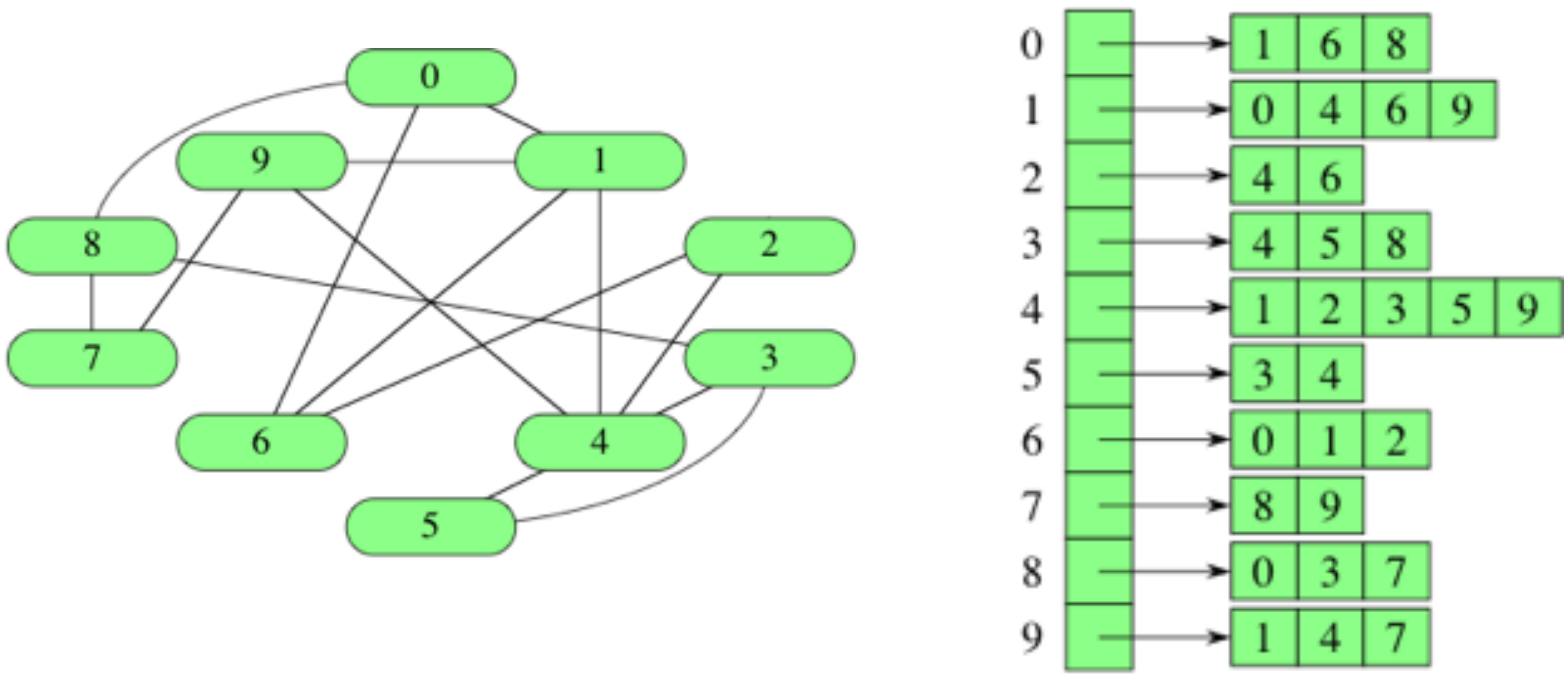
representing a graph: adjacency matrix



	0	1	2	3	4	5	6	7	8	9
0	0	1	0	0	0	0	1	0	1	0
1	1	0	0	0	1	0	1	0	0	1
2	0	0	0	0	1	0	1	0	0	0
3	0	0	0	0	1	1	0	0	1	0
4	0	1	1	1	0	1	0	0	0	1
5	0	0	0	1	1	0	0	0	0	0
6	1	1	1	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0	1	1
8	1	0	0	1	0	0	0	1	0	0
9	0	1	0	0	1	0	0	1	0	0

- How long does it take to answer whether two vertices are connected?
- How much memory is required?

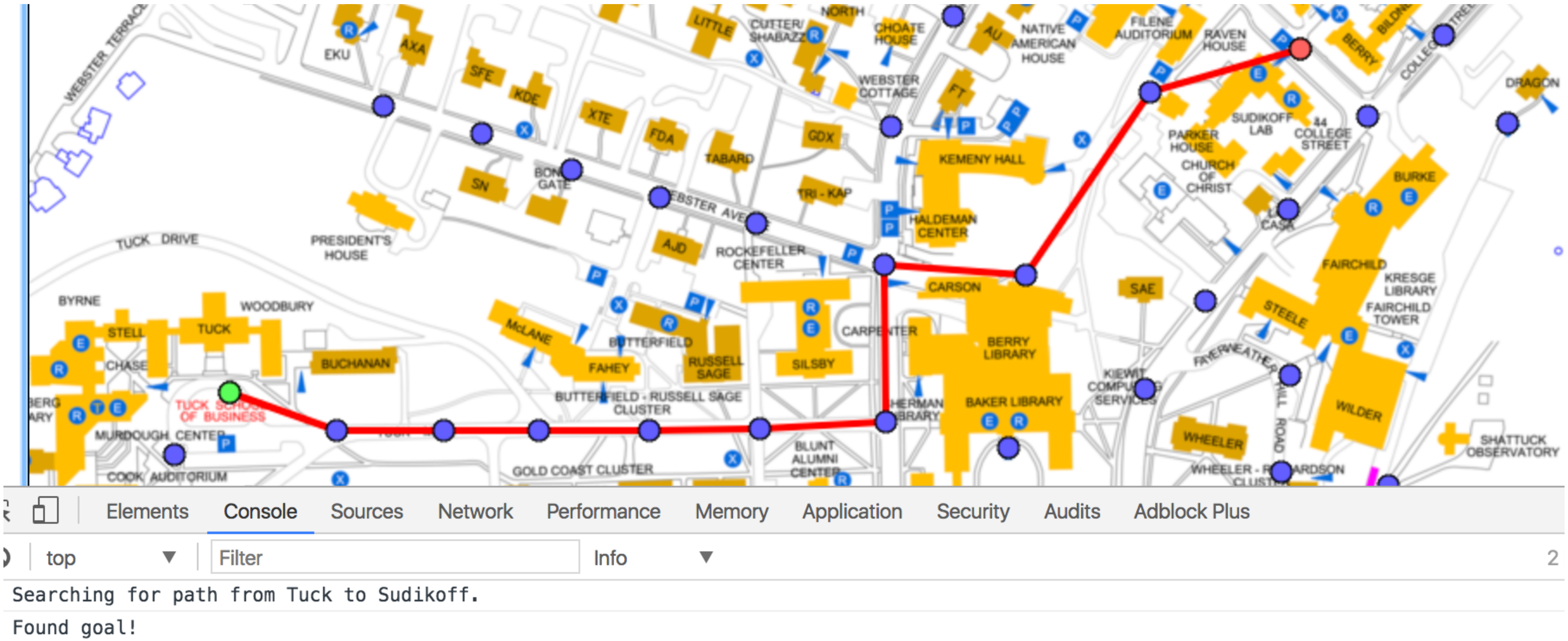
representing a graph: adjacency lists



- How long does it take to answer whether two vertices are connected?
- How much memory is required?

(Our preferred method)

representing a graph: example



what's in a node?

dartmouth_graph.txt

```
Baker West; Sanborn, Blunt, Carson; 496, 504
Sanborn; Baker West, Green Northwest, Baker, North Mass; 498, 560
Butterfield; Gold Coast, Blunt, FDA; 359, 509
Gold Coast; Tuck Dr, Butterfield; 295, 509
Tuck Dr; Gold Coast, Buchanan; 240, 509
Buchanan; Tuck Dr, Thayer, Tuck, Murdough; 178, 509
Tuck; Murdough, Buchanan; 116, 487
Thayer; Murdough, Cummings, Buchanan; 127, 548
```

- some data: name, pixel coordinates:

```
tuckNode.name = "Tuck";
tuckNode.x = 116;
tuckNode.y = 487;
```

- an adjacency list:

```
tuckNode.adjacent =
  ["Murdough", "Buchanan"];
```

given the name of a node, how do you get the node?

`graphDict` dictionary indexes nodes using names (strings):

```
var myNode = graphDict["Tuck"];
console.log(myNode.name);
console.log(myNode.x);
console.log(myNode.y);
```

graph.js

```
var graphSearch = function(graphDict, startNodeName, goalNodeName) {
  // returns an array of strings listing the nodes in the path
  // starting with startNodeName and ending with goalNodeName

  console.log("Searching for path from " + startNodeName + " to " +
  goalNodeName + ".");
```

start by experimenting with fetching nodes from graphDict.

given node name, how do you get names of adjacent nodes?

`graphDict` dictionary indexes nodes using names (strings):

```
var currentNodeName = “Tuck”;

// Grab the node from the dictionary
var currentNode = graphDict[currentNodeName];

// The node contains the adjacency list:
console.log(currentNode.adjacent);
```

In this example, `currentNode.adjacent` contains an array of strings.

breadth-first search on a graph

given two strings representing the start and goal locations, what is the shortest connecting sequence of node names?

dartmouth_graph.txt

```
Baker West; Sanborn, Blunt, Carson; 496, 504
Sanborn; Baker West, Green Northwest, Baker, North Mass; 498, 560
Butterfield; Gold Coast, Blunt, FDA; 359, 509
Gold Coast; Tuck Dr, Butterfield; 295, 509
Tuck Dr; Gold Coast, Buchanan; 240, 509
Buchanan; Tuck Dr, Thayer, Tuck, Murdough; 178, 509
Tuck; Murdough, Buchanan; 116, 487
Thayer; Murdough, Cummings, Buchanan; 127, 548
```

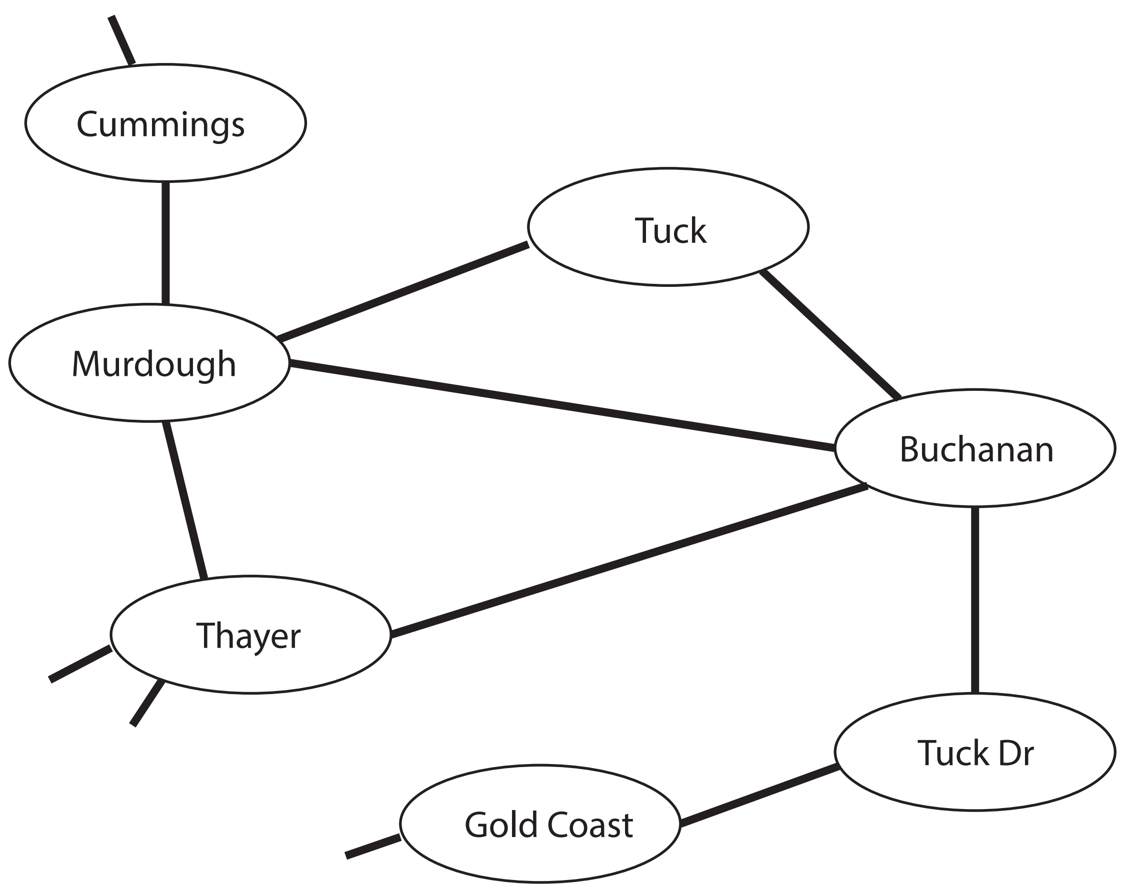
Example:

```
// test out the graph search code. Once you have written the graphSearch
//   function, this should print out "testPath: Tuck,Buchanan,Tuck Dr"

var testPath = graphSearch(mapGraph, "Tuck", "Tuck Dr");
console.log("testPath: " + testPath);
```

a ‘harder’ problem that is easier to solve

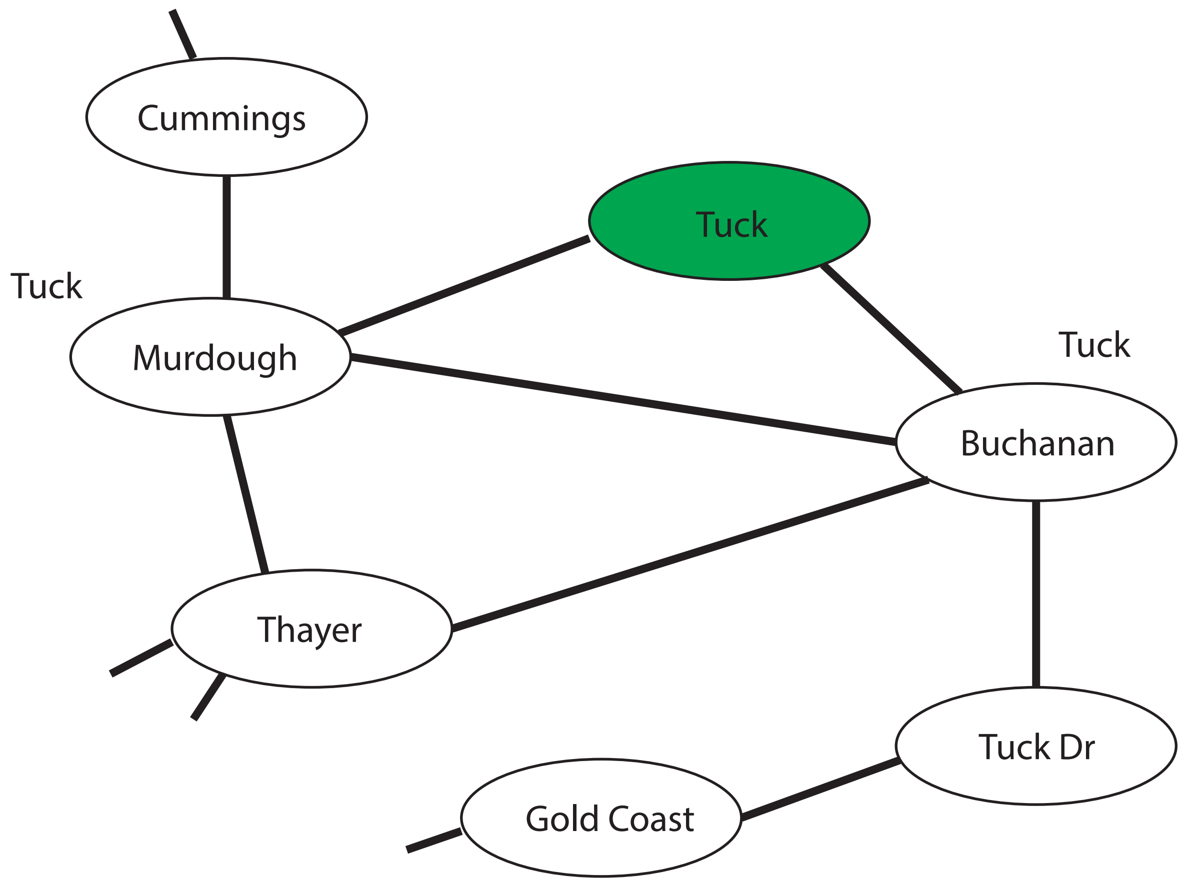
given a string for the start, what is the shortest connecting sequence to every other node?



(Note — geometry does not matter.)

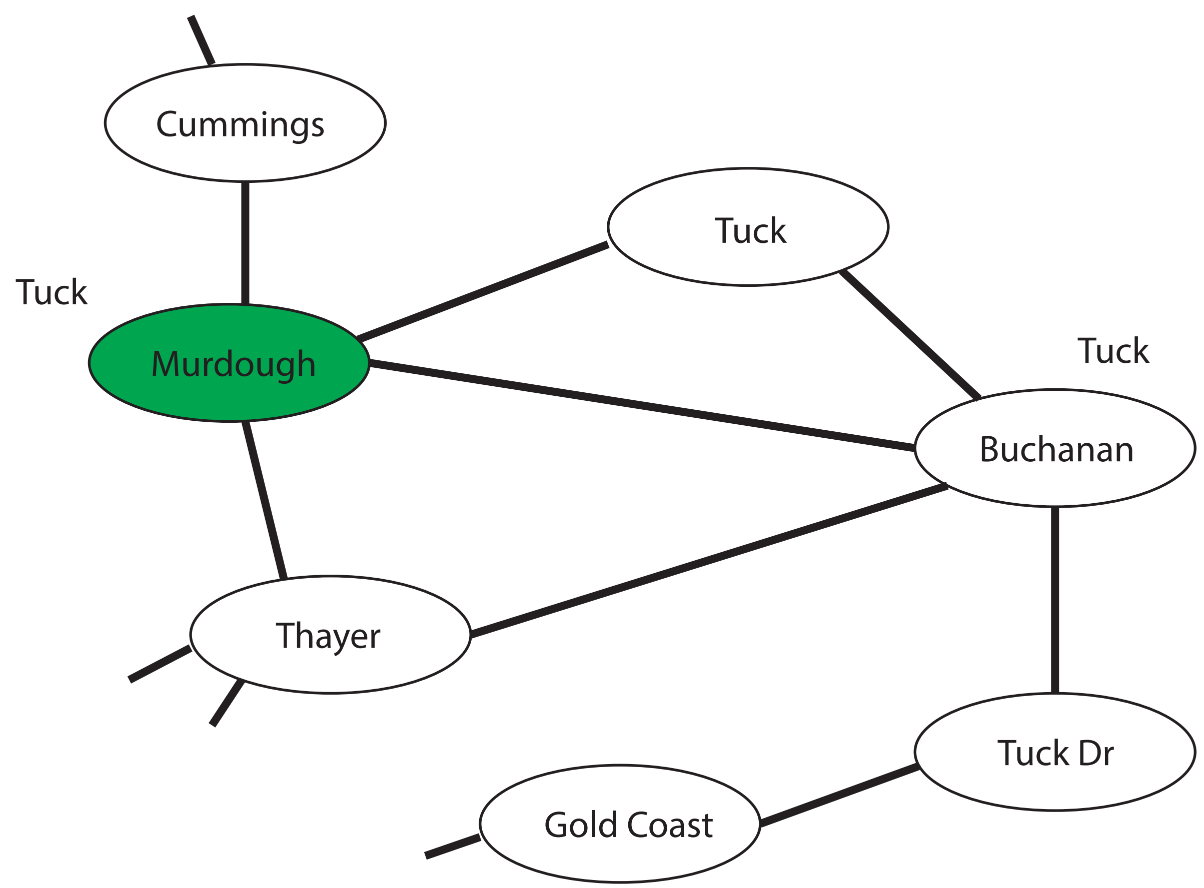
breadth-first exploration from Tuck

Start at Tuck. Send minions to claim adjacent nodes.



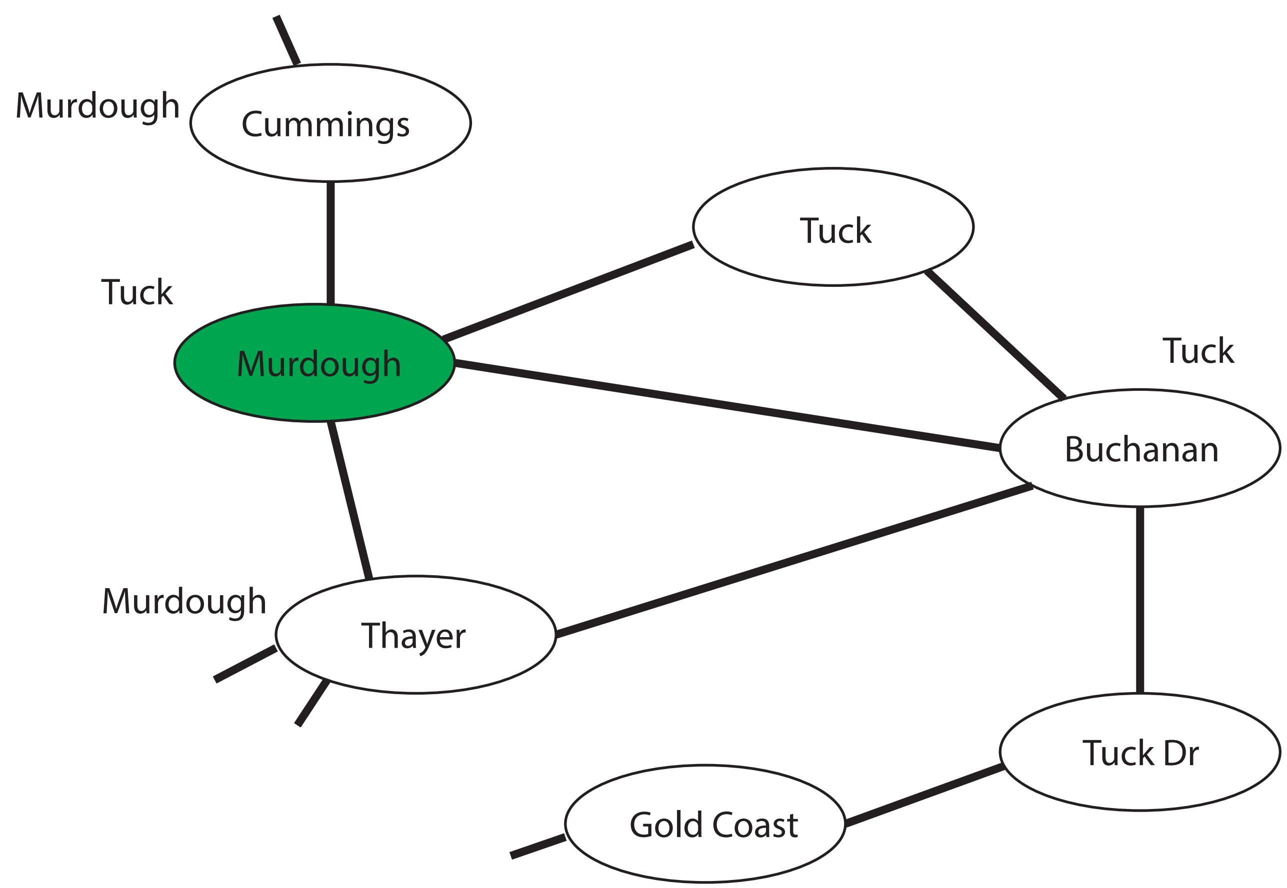
breadth-first exploration from Tuck

Now that Murdough has been claimed, it starts producing minions of its own:



breadth-first exploration from Tuck

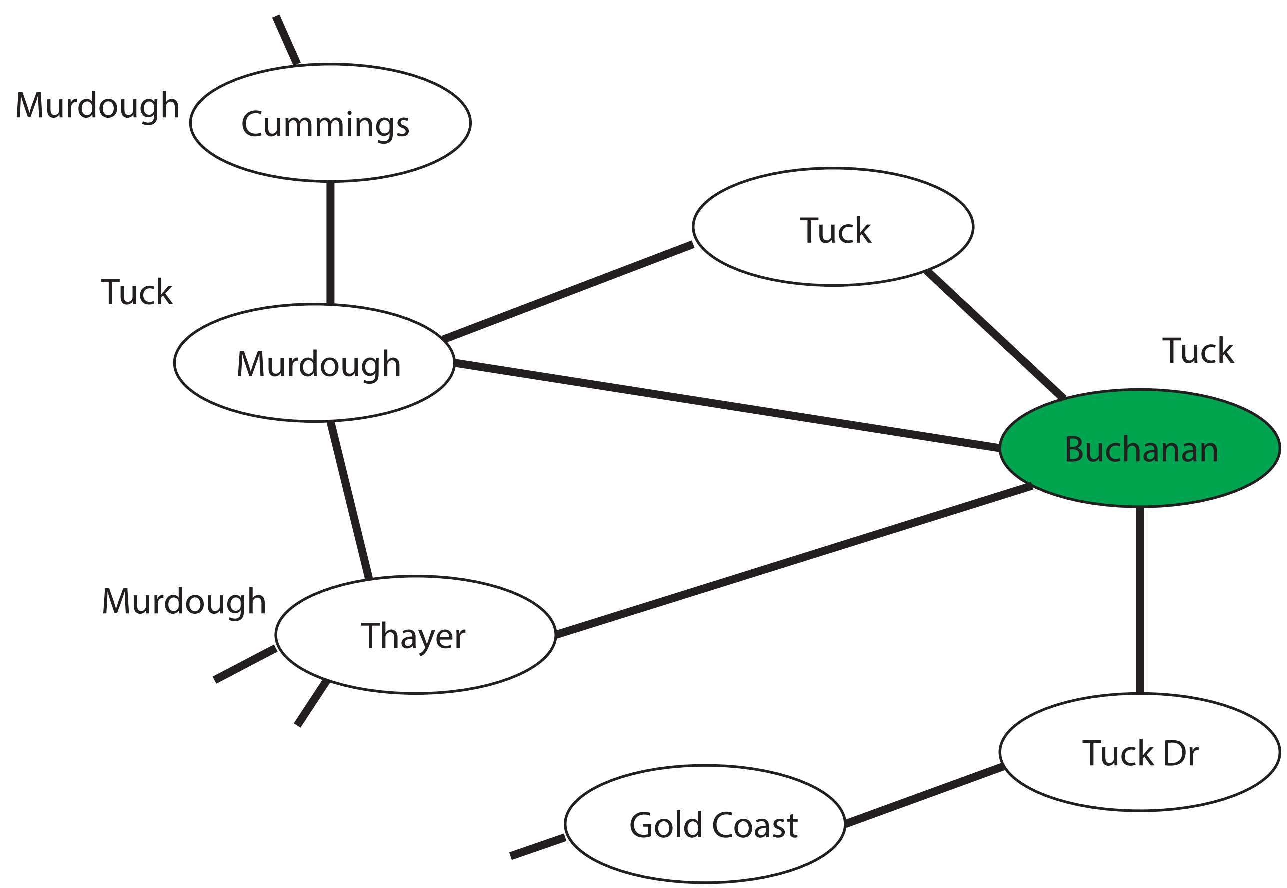
Now that Murdough has been claimed, it starts producing minions of its own:



Notice: Murdough-ians do not reclaim Tuck.

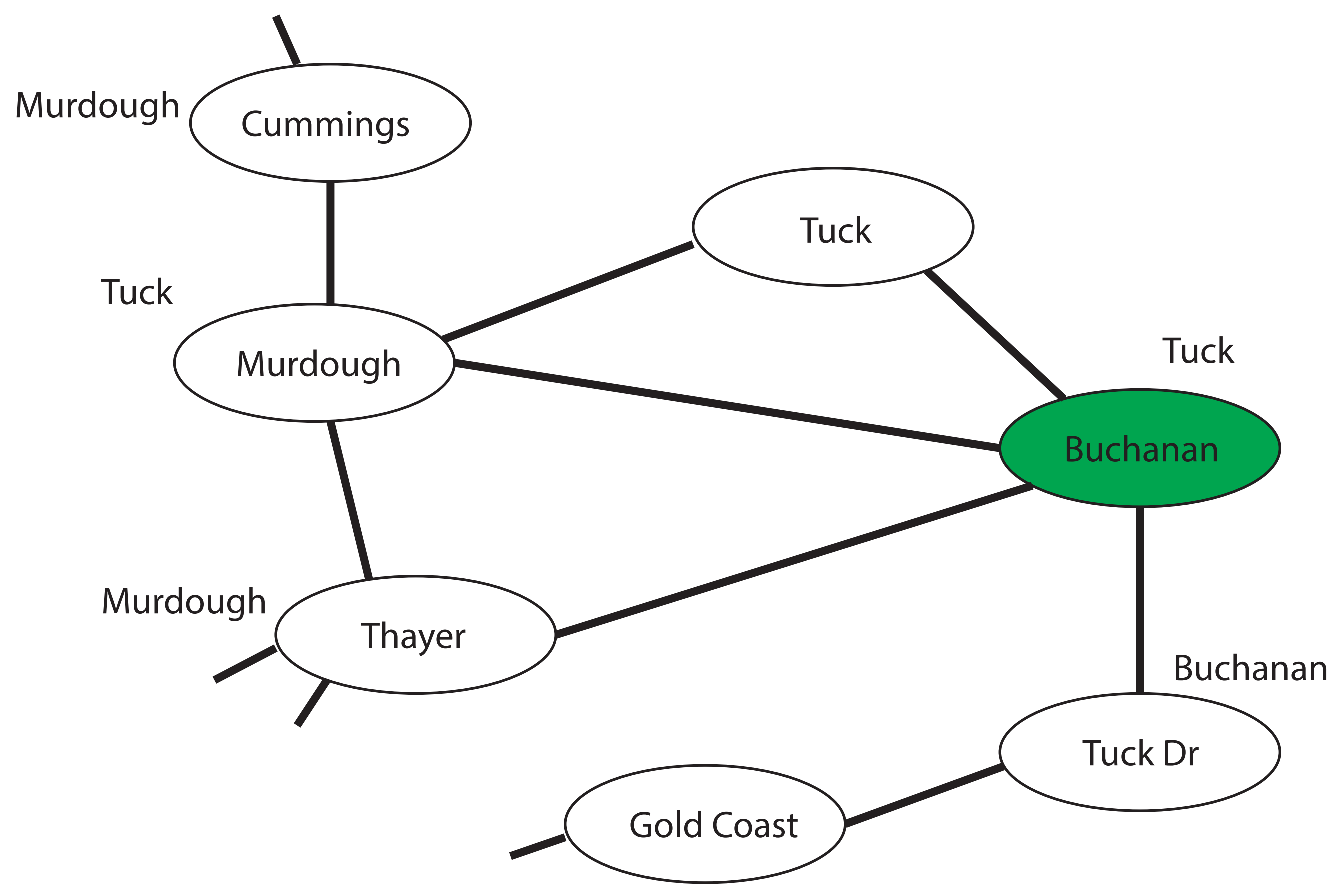
breadth-first exploration from Tuck

Buchanan was also claimed by Tuck, and starts producing minions of its own:



breadth-first exploration from Tuck

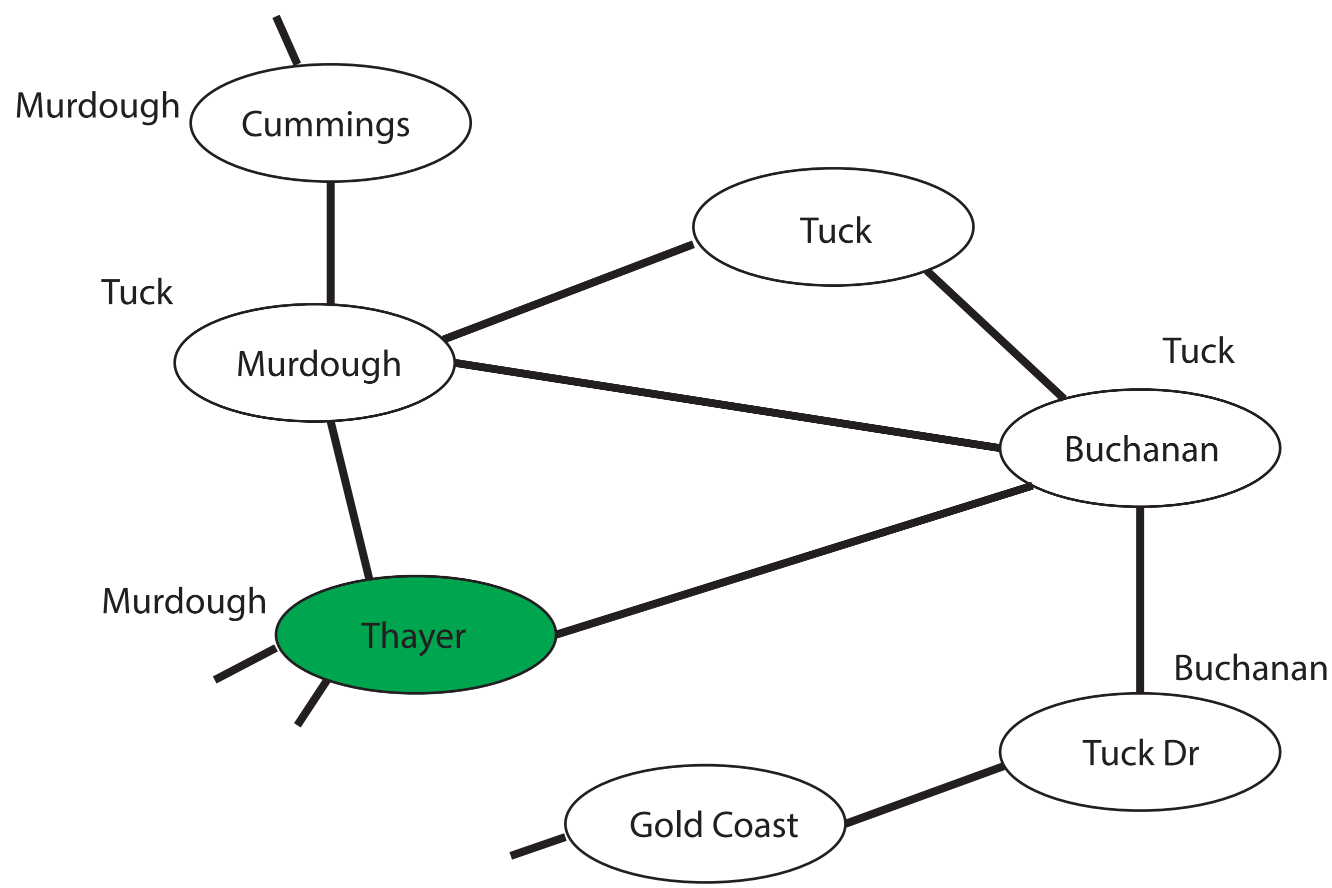
Buchanan was also claimed by Tuck, and starts producing minions of its own:



Notice: Buchanites do not claim Tuck, Murdough, or Thayer (already claimed). They do claim Tuck Dr.

breadth-first exploration from Tuck

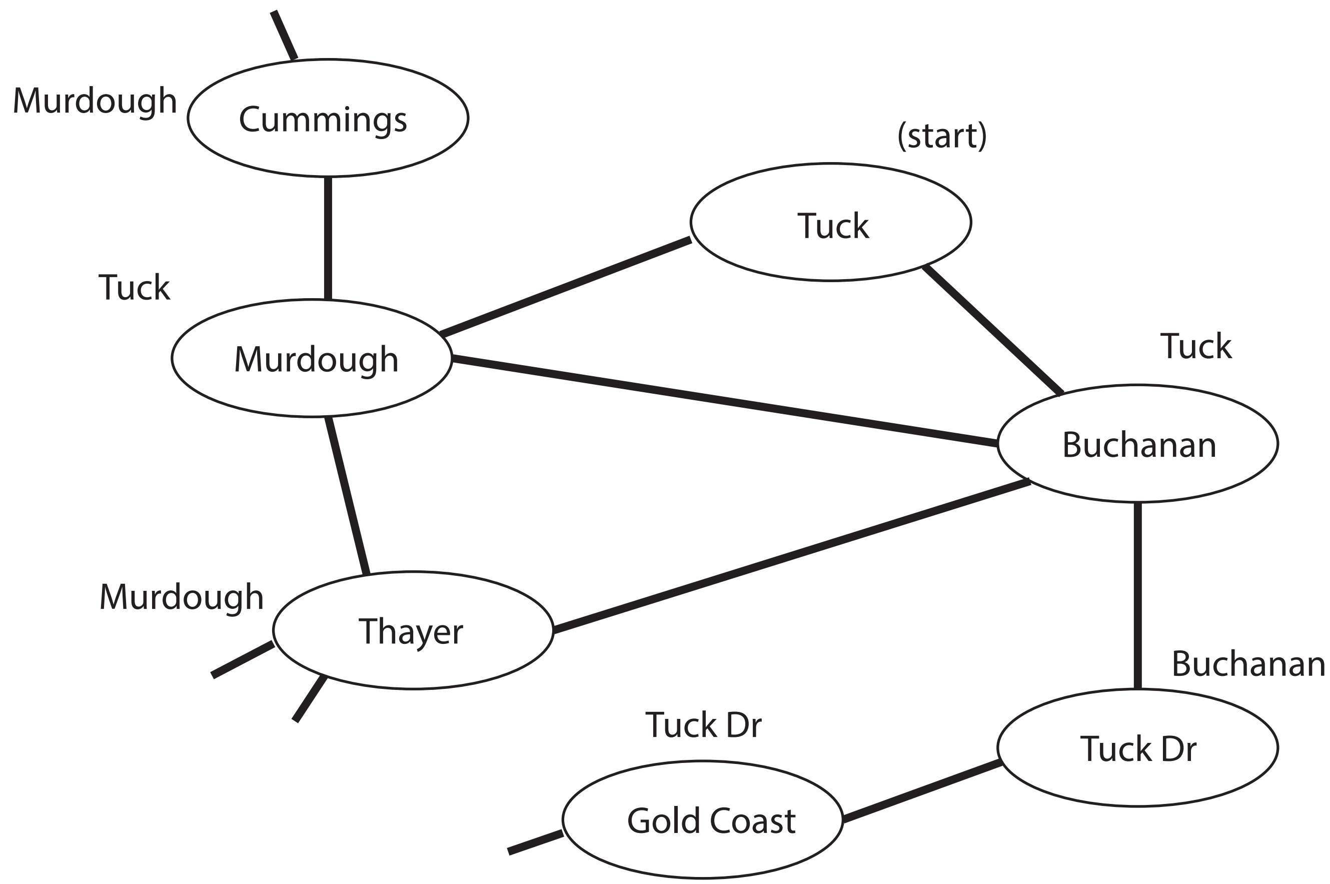
Thayer starts producing minions:



Continue this process until all nodes have been claimed.

finding the path with backchaining

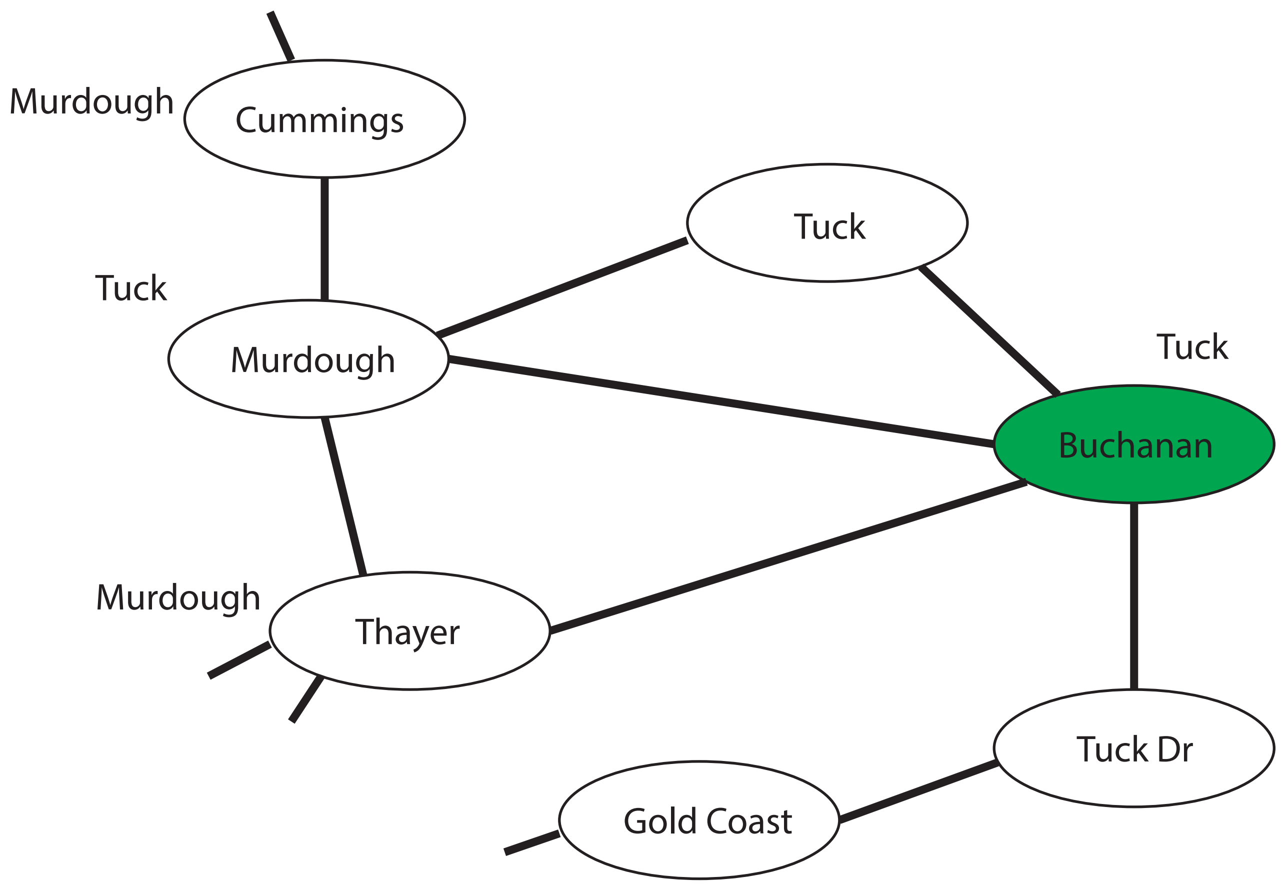
What is a fastest way from Goal Coast to Tuck?



Gold Coast was first claimed from Tuck Dr. Tuck Dr was first claimed by Buchanan. Buchanan was first claimed from Tuck.

Reverse this sequence: Tuck, Buchanan, Tuck Dr, Gold Coast.

breadth-first search: data structures



- Which node should produce minions next? We keep a **queue**.
- Which nodes have been reached first (claimed) from where? We keep a **dictionary**, `visitedFrom`.

`visitedFrom["Thayer"]` is `"Murdough"`

breadth-first search: pseudo-code

add starting node name to new queue (e.g., (`"Tuck"`))
create dictionary `visitedFrom` and add entry for starting name

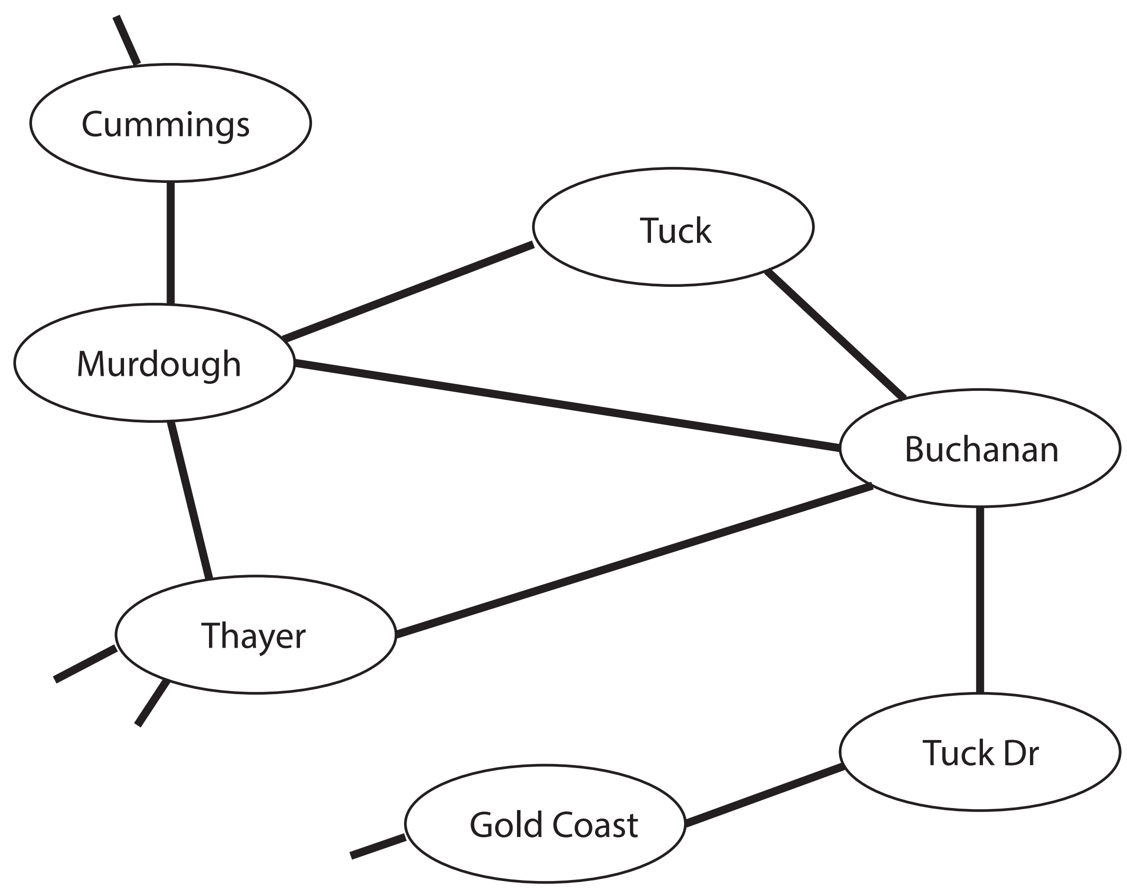
while queue is not empty:
    dequeue current node name from the queue
    get the corresponding node from `graphDict`

    if the current node is the goal, success,
        backchain

    for each adjacent node name that is not in `visitedFrom`:
        add node name to queue for future exploration
        mark where node name was reached from in `visitedFrom`

bfs: data structures example

add start to queue and visitedFrom

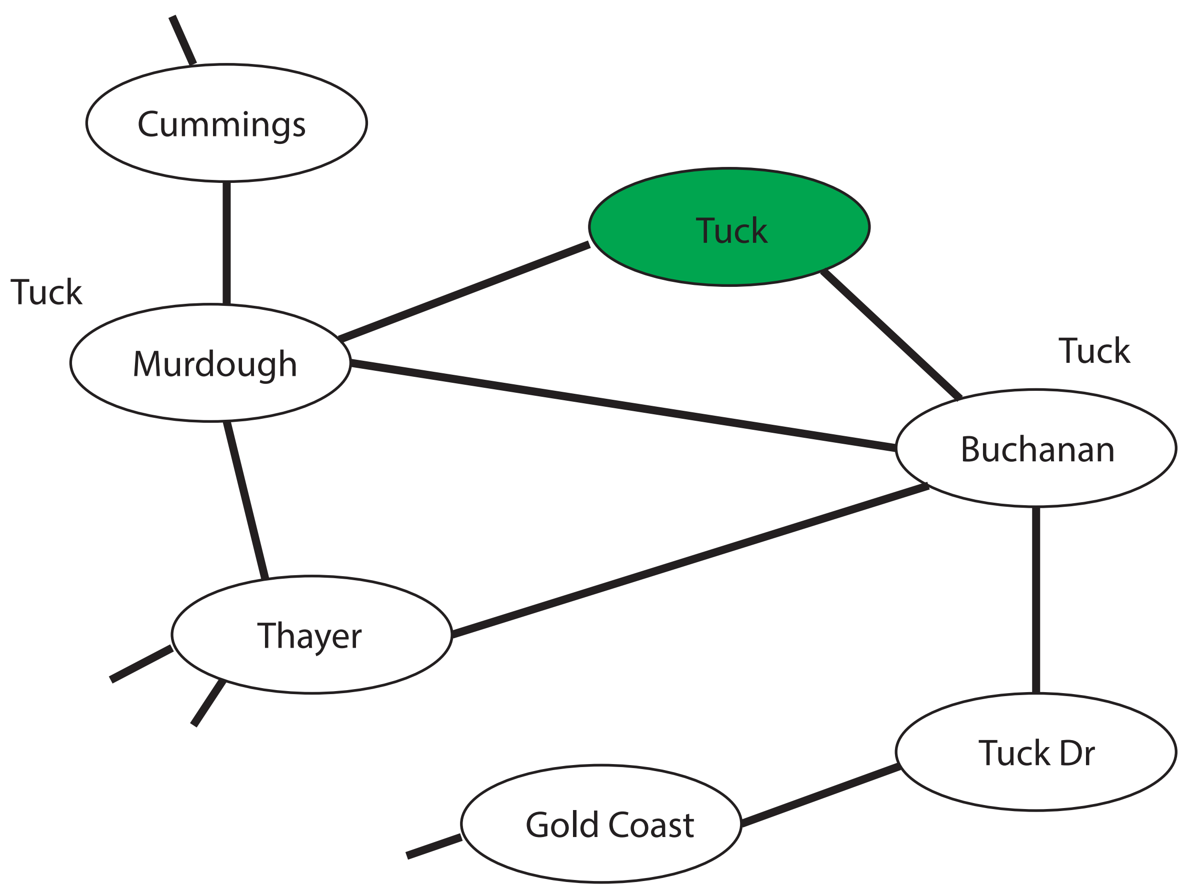


queue: “Tuck”

visitedFrom: {“Tuck”: “start”}

breadth-first exploration from Tuck

Start at Tuck. Send minions to claim adjacent nodes.

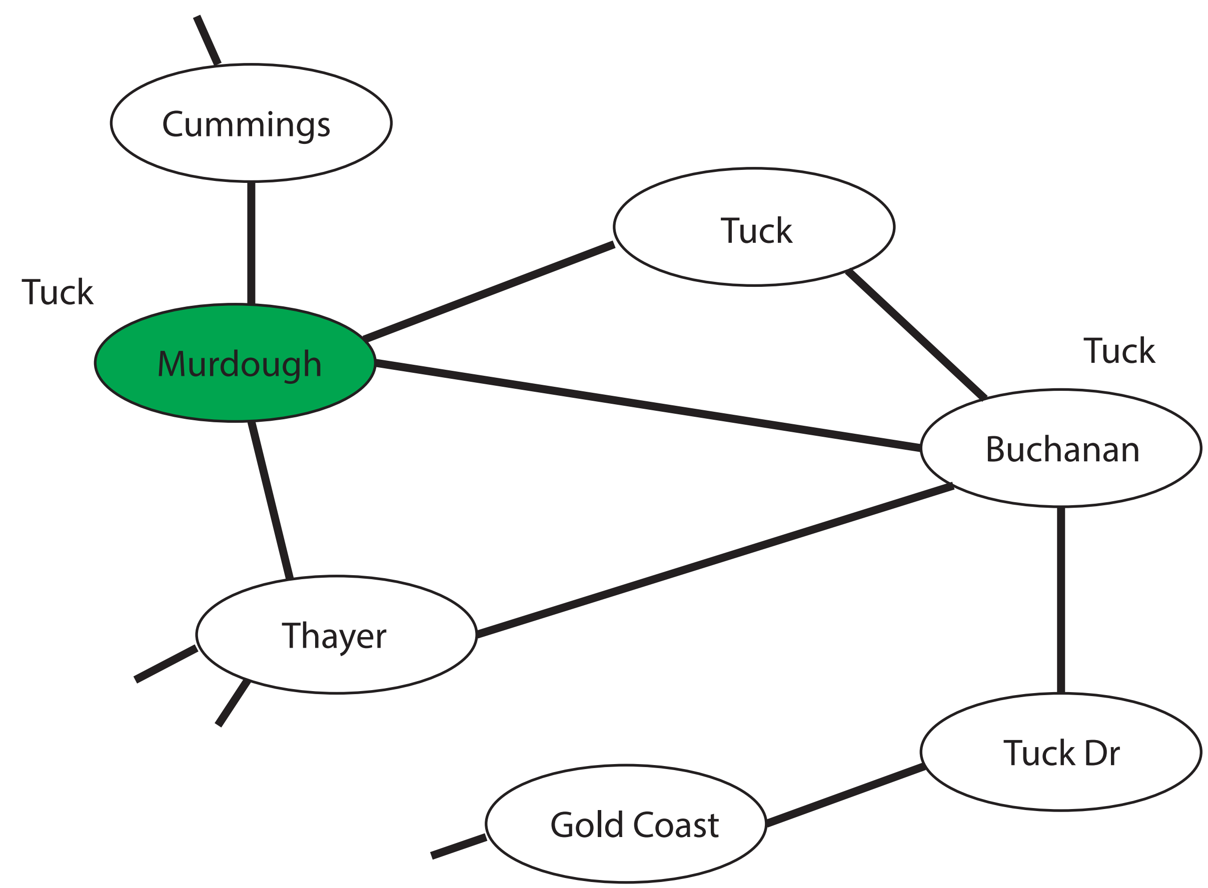


queue: “Murdough”, “Buchanan”

visitedFrom: {“Tuck”: “start”, “Murdough”: “Tuck”, “Buchanan”: “Tuck”}

breadth-first exploration from Tuck

Next, dequeue “Murdough”. Its adjacent node names are “Tuck”, “Cummings”, and “Thayer”. Since “Tuck” is already in visitedFrom, just add “Cummings” and “Thayer” to queue and visitedFrom.

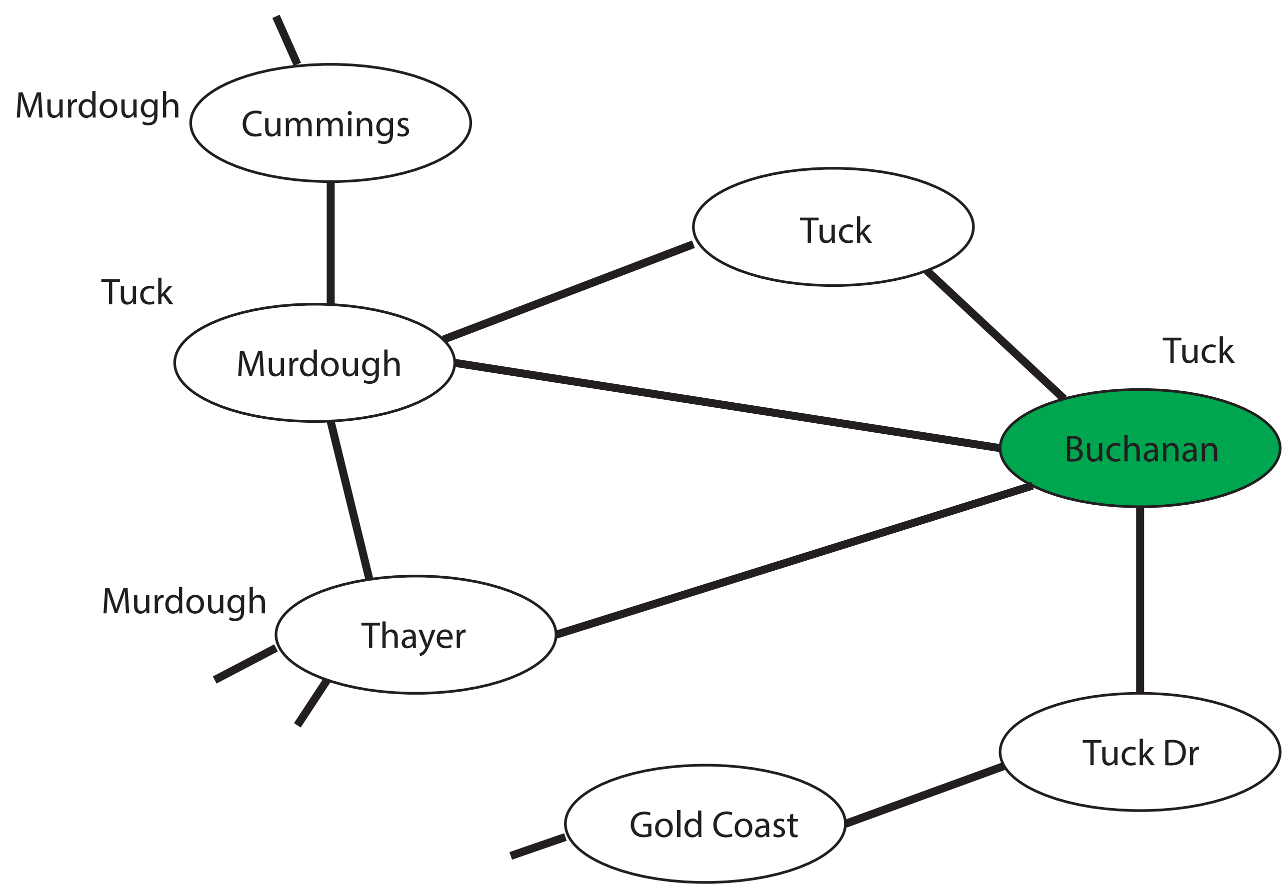


queue: “Buchanan”, “Cummings”, “Thayer”

visitedFrom: {“Tuck”: “start”, “Murdough”: “Tuck”, “Buchanan”: “Tuck”, “Cummings”: “Murdough”, “Thayer”: “Murdough”}

breadth-first exploration from Tuck

Buchanan is next in the queue. It will add Tuck Dr. to queue and visitedFrom.



queue: “Cummings”, “Thayer”, “Buchanan”

visitedFrom: {“Tuck”: “start”, “Murdough”: “Tuck”, “Buchanan”: “Tuck”, “Cummings”: “Murdough”, “Thayer”: “Murdough”, “Tuck Dr.”: “Buchanan”}

Foxes and chickens

(Drop the course if the first homework is crushing.)

- 3 chickens, 3 foxes, 1 boat. Boat can carry 2.
- If at any point, there are more foxes than chickens on either side, the game ends.
- Give a sequence of actions that takes all across safely.

agents and search

- An agent begins in some state, the initial or start state.
- The agent would like to get to some goal state.
- The agent has certain actions available
- The agent knows the results of each action
- The agent might have some preference for "better" paths

States and nodes are often created “on the fly.”

formal search problems

- A `goal_test` function that checks if a state is a goal state
- A `get_actions` function that finds the legal **actions** from some state and a `transition` function that accepts a state, an action, and returns a new state, or alternatively, a `get_successors` function that returns a list of states given a starting states (wrapping get_actions and transition)
- A path_cost function that gives the cost of a path between a pair of states.