

# Fundamental Graph Algorithms

## Part II

# Outline for Today

- **Dijkstra's Algorithm**
  - An algorithm for finding shortest paths in more realistic settings
- **Depth-First Search**
  - A different graph search algorithm.
- **Directed Acyclic Graphs**
  - Graphs for representing prerequisites.
- **(ITA) Topological Sorting**
  - Algorithms for ordering dependencies.

Recap from Last Time

# Breadth-First Search

- Given an arbitrary graph  $G = (V, E)$  and a starting node  $s \in V$ , **breadth-first search** finds shortest paths from  $s$  to each reachable node  $v$ .
- When implemented using an adjacency list, runs in  $O(m + n)$  time, which we defined to be linear time on a graph.
- One correctness proof worked in terms of “layers:” the algorithm finds all nodes at distance 0, 1, 2, ... in order.

# A Second Intuition for BFS

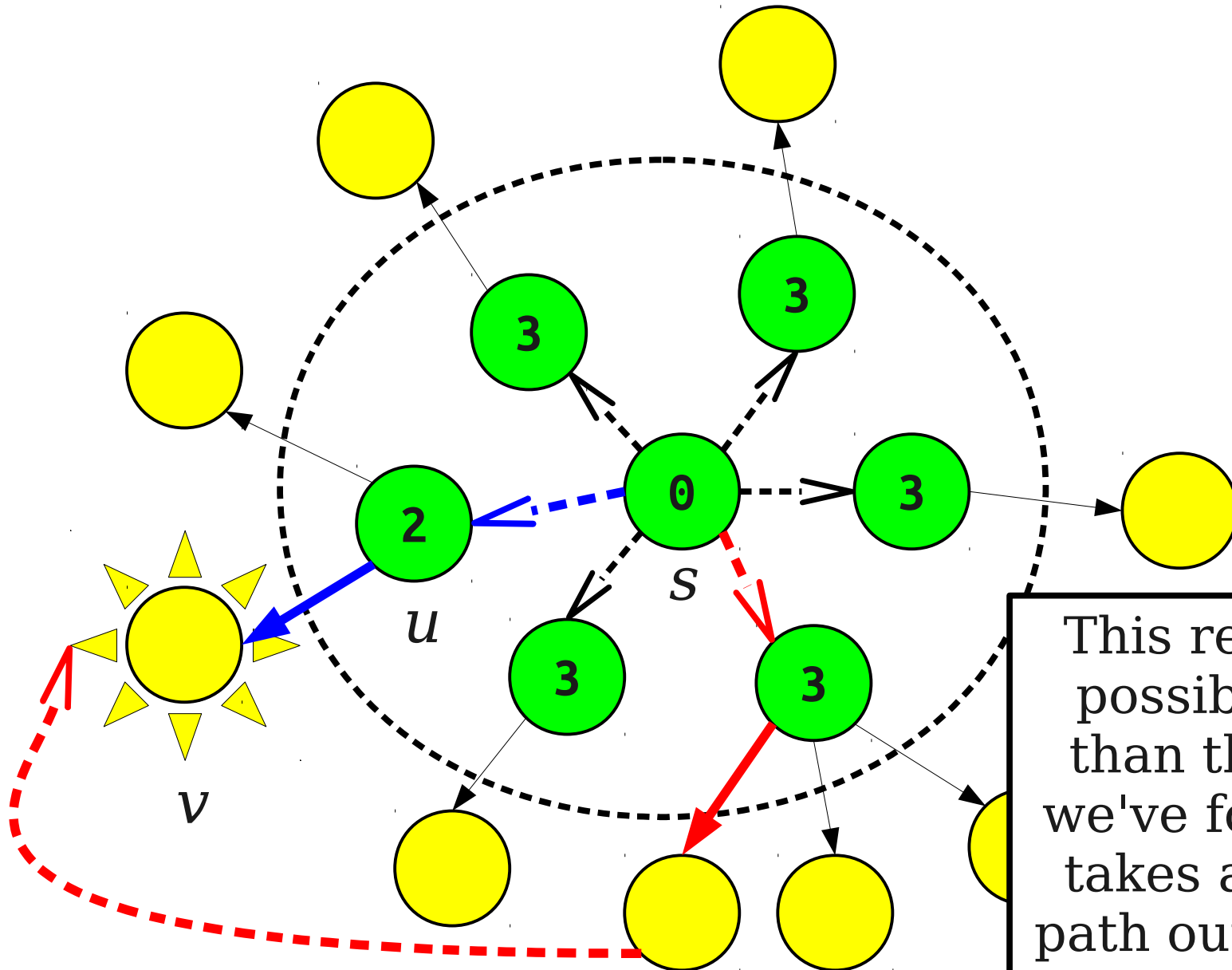
# A Second Intuition

- At each point in the execution of BFS, a node  $v$  is either
  - green, and we have the shortest path to  $v$ ;
  - yellow, and it is connected to some green node; or
  - gray, and  $v$  is undiscovered.
- Each iteration, we pick a yellow node with minimal distance from the start node and color it green. So what is the cost of the lowest-cost yellow node?
- If  $v$  is yellow, it is connected to a green node  $u$  by an edge.
- The cost of getting from  $s$  to  $v$  is then  $d(s, u) + 1$ .
- BFS works by picking the yellow node  $v$  minimizing

$$\mathbf{d(s, u) + 1}$$

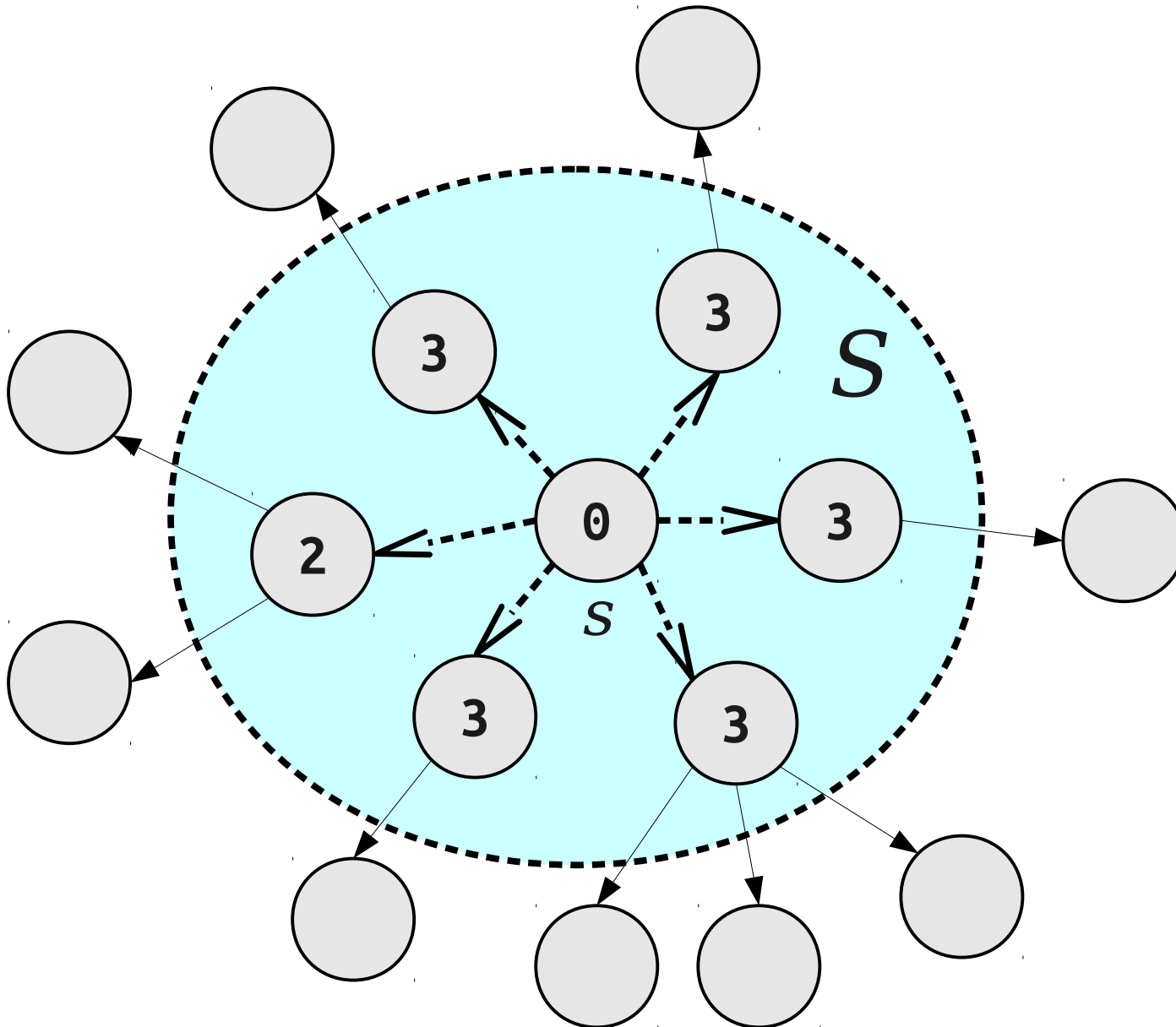
where  $(u, v)$  is an edge and  $u$  is green.

Pick yellow node  $v$  minimizing  $d(s, u) + 1$ , where  $(u, v)$  is an edge and  $u$  is green.



This red path can't possibly be better than the blue path we've found, since it takes a suboptimal path out of the circle!

Pick node  $v \notin S$  minimizing  $d(s, u) + 1$ ,  
where  $(u, v)$  is an edge and  $u \in S$





*Lemma:* Suppose we have shortest paths computed for nodes  $S \subseteq V$ , where  $s \in S$ . Consider a node  $v$  where  $(u, v) \in E$ ,  $u \in S$ , and the quantity  $d(s, u) + 1$  is minimized. Then  $d(s, v) = d(s, u) + 1$ .

*Proof:* There is a path to  $v$  of cost  $d(s, u) + 1$ : follow the shortest path to  $u$  (which has cost  $d(s, u)$ ), then follow one more edge to  $v$  for total cost  $d(s, u) + 1$ .

Now suppose for the sake of contradiction that there is a shorter path  $P$  to  $v$ . This path must start in  $S$  (since  $s \in S$ ) and leave  $S$  (since  $v \notin S$ ). So consider when  $P$  leaves  $S$ . When this happens,  $P$  must go from  $s$  to some node  $x \in S$ , cross an edge  $(x, y)$  to some node  $y$ , then continue from  $y$  to  $v$ . This means that  $|P|$  is at least  $d(s, x) + 1$ , since the path goes from  $s$  to  $x$  and then follows at least one more edge.

Since  $v$  was picked to minimize  $d(s, u) + 1$  for any choice of  $u \in S$  adjacent to an edge  $(u, v)$ , we know

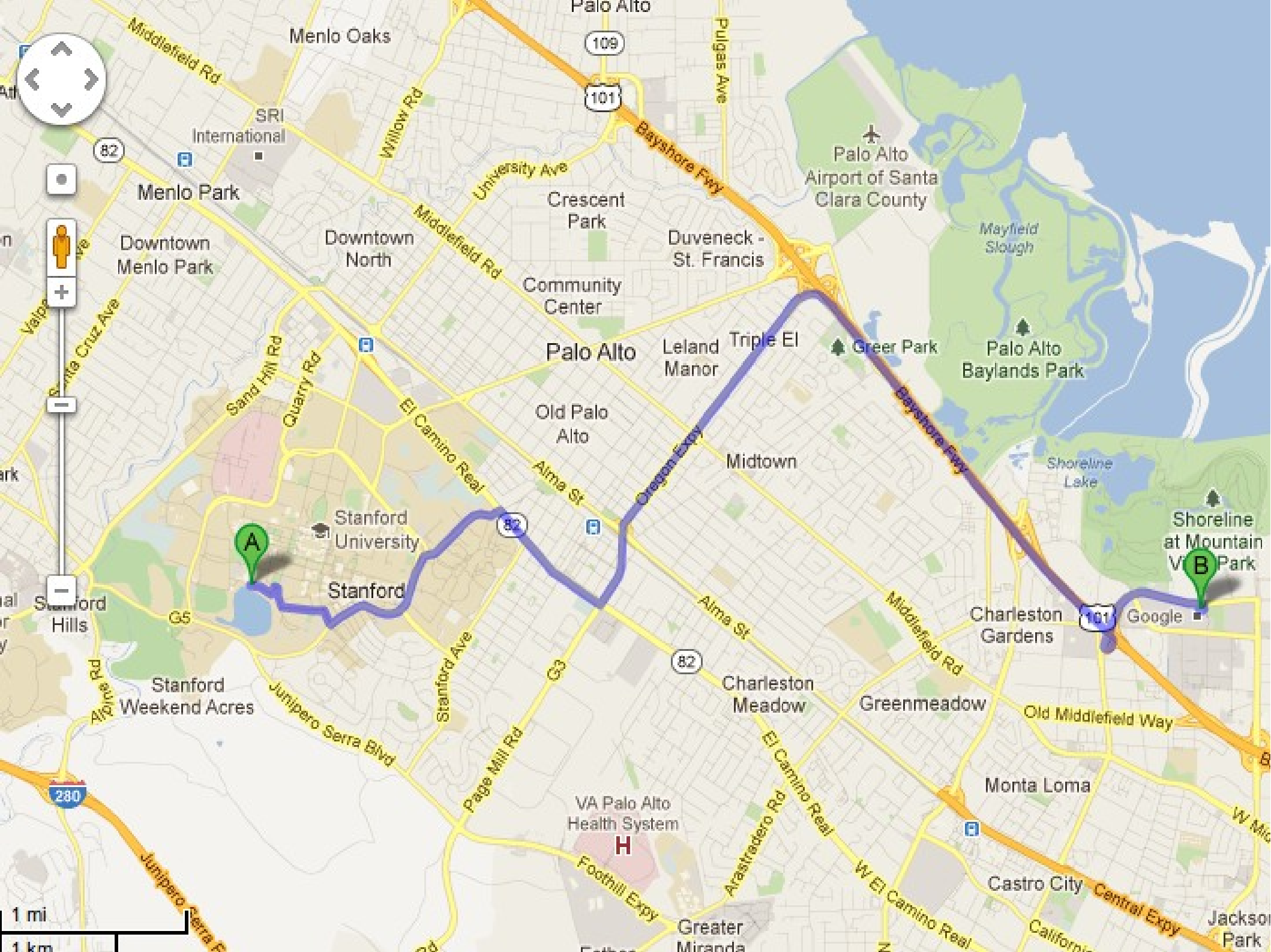
$$d(s, u) + 1 \leq d(s, x) + 1 \leq |P|$$

contradicting the fact that  $|P| < d(s, u) + 1$ . We have reached a contradiction, so our assumption was wrong and no shorter path exists.

Since there is a path of length  $d(s, u) + 1$  from  $s$  to  $v$  and no shorter path, this means that  $d(s, v) = d(s, u) + 1$ . ■

# Why These Two Proofs Matter

- The first proof of correctness (based on layers) is based on our first observation: the nodes visited in BFS radiate outward from the start node in ascending order of distance.
- The second proof of correctness (based on picking the lowest yellow node) is based on our second observation: picking the lowest-cost yellow node correctly computes a shortest path.
- Interestingly, this second correctness proof can be generalized to a larger setting...

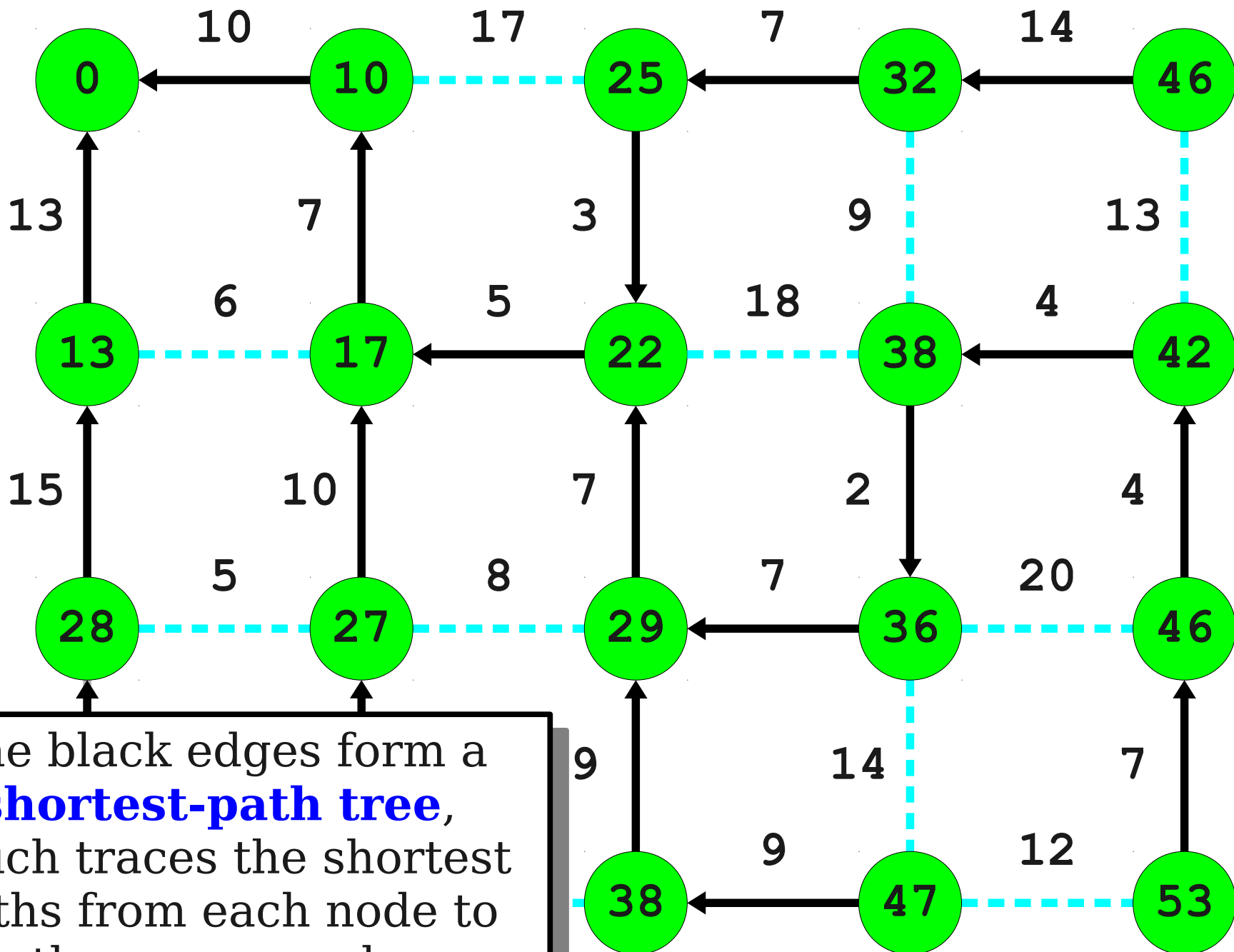


1 mi

1 km

# Edges with Costs

- In many applications, edges have an associated length (or cost, weight, etc.), denoted  $l(u, v)$ .
- **Assumption:** Lengths are nonnegative. (We'll revisit this later in the quarter.)
- Let's say that the length of a path  $P$  (denoted  $l(P)$ ) is the sum of all the edge lengths in the path  $P$ .
- Goal: find the shortest path from  $s$  to every node in  $V$ , taking costs into account.



The black edges form a **shortest-path tree**, which traces the shortest paths from each node to the source node.

```
procedure dijkstrasAlgorithm(s, G):
```

```
  let q be a new queue
```

```
  for each v in V:
```

```
    dist[v] =  $\infty$ 
```

```
  dist[s] = 0
```

```
  enqueue(s, q)
```

```
  while q is not empty:
```

```
    let v be a node in q minimizing dist[v]
```

```
    remove(v, q)
```

```
    for each node u connected to v:
```

```
      if dist[u] > dist[v] + l(u, v):
```

```
        dist[u] = dist[v] + l(u, v)
```

```
        if u is not enqueued into q:
```

```
          enqueue(u, q)
```

# Dijkstra's Algorithm

- Assuming nonnegative edge lengths, finds the shortest path from  $s$  to each node in  $G$ .
- Correctness proof sketch is based on the second argument for breadth-first search:
  - Always picks the node  $v$  minimizing  $d(s, u) + l(u, v)$  for yellow  $v$  and green  $u$ .
  - If a shorter path  $P$  exists to  $v$ , it must leave the set of green nodes through some edge  $(x, y)$ .
  - But then  $l(P)$  is at least  $d(s, x) + l(x, y)$ , which is at least  $d(s, u) + l(u, v)$ .
  - So the “shorter” path costs at least as much as the path we found.

**procedure** dijkstrasAlgorithm( $s, G$ ):

**let**  $q$  be a new queue

**for** each  $v$  in  $V$ :

$\text{dist}[v] = \infty$

$\text{dist}[s] = 0$

enqueue( $s, q$ )

**$O(m + n)$**

**while**  $q$  is not empty:

**let**  $v$  be a node in  $q$  minimizing  $\text{dist}[v]$

remove( $v, q$ )

**for** each node  $u$  connected to  $v$ :

**if**  $\text{dist}[u] > \text{dist}[v] + l(u, v)$ :

$\text{dist}[u] = \text{dist}[v] + l(u, v)$

**if**  $u$  is not enqueued into  $q$ :

enqueue( $u, q$ )



```
procedure dijkstrasAlgorithm(s, G):
```

```
  let q be a new queue
```

```
  for each v in V:
```

```
    dist[v] =  $\infty$ 
```

```
  dist[s] = 0
```

```
  enqueue(s, q)
```

**$O(n^2)$**

```
  while q is not empty:
```

```
    let v be a node in q minimizing dist[v]
```

```
    remove(v, q)
```

```
    for each node u connected to v:
```

```
      if dist[u] > dist[v] + l(u, v):
```

```
        dist[u] = dist[v] + l(u, v)
```

```
        if u is not enqueued into q:
```

```
          enqueue(u, q)
```

# Dijkstra Runtime

- Using a standard implementation of a queue, Dijkstra's algorithm runs in time  **$O(n^2)$** .
  - $O(n + m)$  time processing nodes and edges, plus  $O(n^2)$  time finding the lowest-cost node.
  - Since  $m = O(n^2)$ ,  $O(n + m + n^2) = O(n^2)$ .
- Using a slightly fancier data structure (a binary heap), can be made to run in time  **$O(m \log n)$** .
  - Is this necessarily more efficient?
  - More on how to do this later this quarter.
- Using a *much* fancier data structure (the *Fibonacci heap*), can be made to run in time  **$O(m + n \log n)$** .
  - Take CS166 for details!

# Shortest Path Algorithms

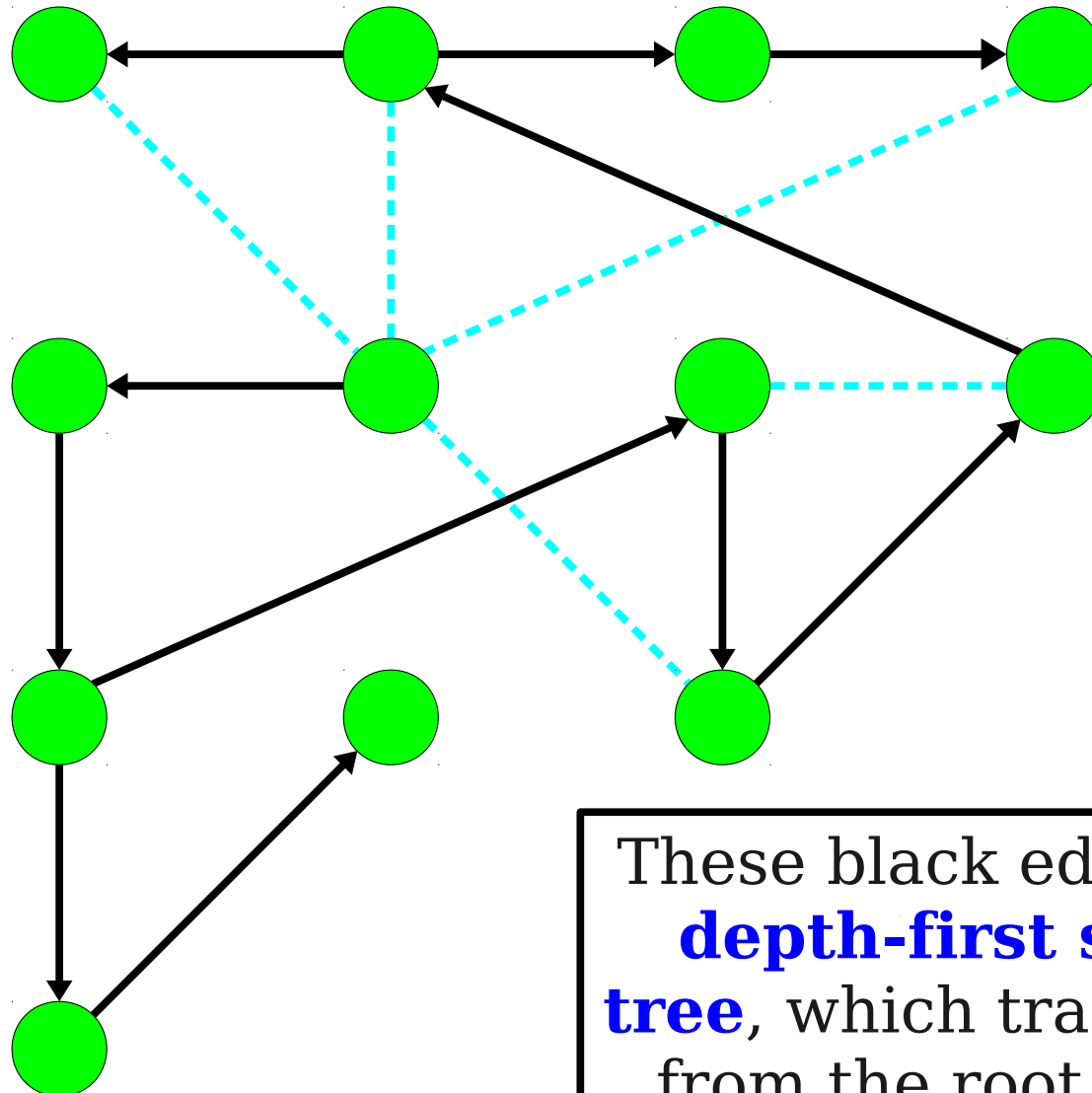
- If all edges have the same weight, can use breadth-first search to find shortest paths.
  - Takes time  $O(m + n)$ .
- If edges have nonnegative weight, can use Dijkstra's algorithm.
  - Takes time  $O(n^2)$ , or less using more complex data structures.
- What about the case where edges can have negative weight?
  - More on that later in the quarter...

# Depth-First Search

# BFS and DFS

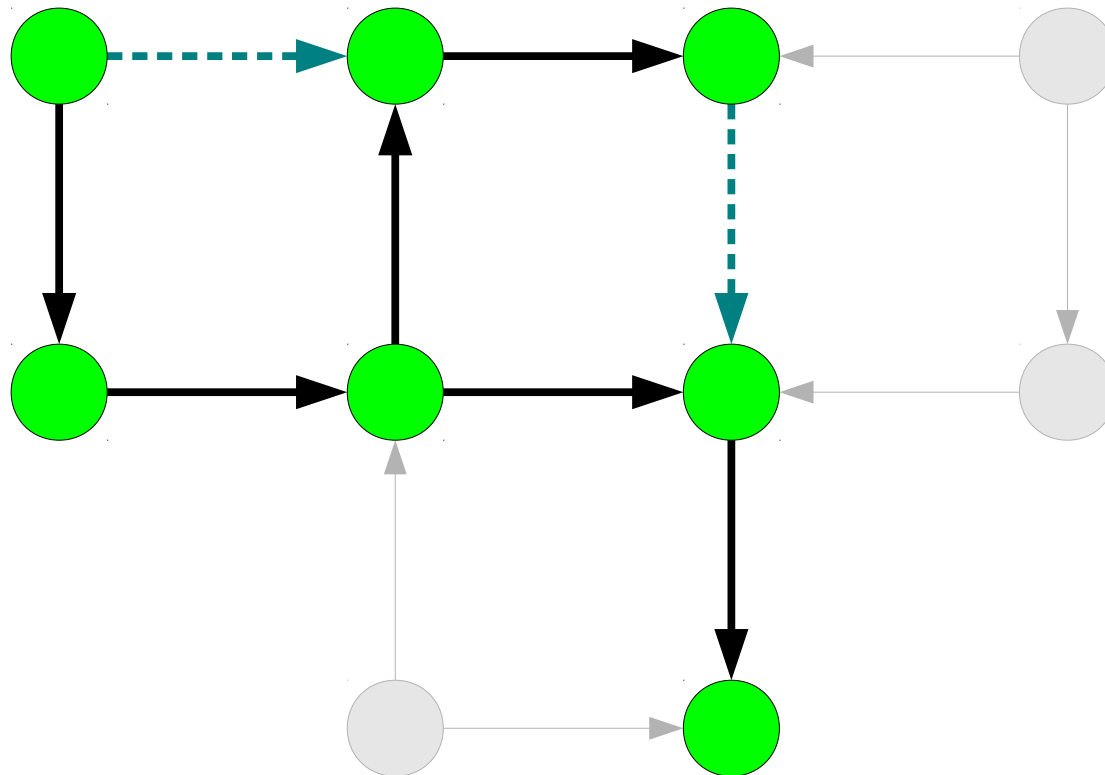
- Last time, we saw the **breadth-first search (BFS)** algorithm, which explored a graph and found shortest paths.
- The algorithm explored outward in all directions uniformly.
- We will now see **depth-first search (DFS)**, an algorithm that explores out in one direction, backing up when necessary.

# Depth-First Search



These black edges form a **depth-first search tree**, which traces paths from the root to each node in the graph.

# Depth-First Search, Again



```
procedure DFS(node v):  
    color v yellow.  
  
    for each neighbor u of v:  
        if u is gray:  
            DFS(u)  
  
    color v green
```

```
procedure doDFS(graph G, node s):  
    for each node v in G:  
        color v gray  
    DFS(s)
```



*Question 1:* What nodes will DFS reach?

*Question 2:* How *efficiently* will DFS reach those nodes?

*Theorem:* When DFS( $s$ ) is called on node  $s$ , no nodes reachable from  $s$  will be gray when DFS( $s$ ) and all ancestor calls return.

*Proof:* By induction on the distance of nodes from  $s$ . As a base case, consider all nodes at distance 0 from  $s$ . This is just  $s$  itself. When DFS( $s$ ) is called, DFS( $s$ ) will color  $s$  yellow, then green.

Suppose the claim holds for all nodes at distance  $n$  from  $s$ ; we'll prove it holds for all nodes at distance  $n + 1$  from  $s$ . Take any node  $v$  at distance  $n + 1$  from  $s$ ;  $v$  is adjacent to some node  $u$  at distance  $n$  from  $s$ . By our IH,  $u$  will not be gray when DFS( $s$ ) and its ancestor calls return, so DFS( $u$ ) must have been called at some point. This call must have called DFS on each of  $u$ 's gray neighbors. If  $v$  was gray at this time, DFS( $v$ ) must have been called on  $v$ , coloring  $v$  yellow and then green. Otherwise,  $v$  was already not colored gray.

Since our choice of  $v$  was arbitrary, no nodes at distance  $n + 1$  will be gray when DFS( $s$ ) and its ancestor calls return, completing the induction. ■

*Theorem:* When  $\text{DFS}(s)$  is called on a node  $s$ , no recursive calls will be made on nodes not reachable from  $s$ .

*Proof:* By contradiction; assume a recursive call is made on at least one node not reachable from  $s$ . There must be a first node visited this way; call it  $v$ .  $v$  can't be  $s$ , since  $s$  is trivially reachable from itself. Thus  $\text{DFS}(v)$  must have been recursively invoked by  $\text{DFS}(u)$  for some node  $u \neq v$ , which in turn called  $\text{DFS}(v)$ . This means edge  $(u, v)$  must exist. Now, we consider two cases:

- *Case 1:*  $u$  is reachable from  $s$ . But then  $v$  is reachable from  $s$ , because we can take the path from  $s$  to  $u$  and follow edge  $(u, v)$ .
- *Case 2:*  $u$  is not reachable from  $s$ . But then  $v$  was not the first node not reachable from  $s$  to have  $\text{DFS}$  called on it.

In either case, we reach a contradiction, so our assumption was wrong. Thus  $\text{DFS}(s)$  never makes recursive calls on nodes not reachable from  $s$ . ■

# What DFS Visits

- Taken together, the two theorems we have proven show the following:
  - When  $\text{DFS}(s)$  terminates, every node reachable from  $s$  will have had DFS called on it, though the call to  $\text{DFS}(s)$  might not have initiated those other calls.
  - When  $\text{DFS}(s)$  terminates, it will never have called DFS on a node not reachable from  $s$ .
- Thus when  $\text{DFS}(s)$  terminates, the only nodes DFS will have been called on are nodes on which DFS had already been called, plus the nodes reachable from  $s$ .

*Question 1:* What nodes will DFS reach?

*Question 2:* How *efficiently* will DFS reach those nodes?

```
procedure DFS(node v):  
    color v yellow.  
  
    for each neighbor u of v:  
        if u is gray:  
            DFS(u)  
  
    color v green
```

```
procedure doDFS(graph G, node s):  
    for each node v in G:  
        color v gray  
    DFS(s)
```

# Analyzing Recursive Functions

- In general, it can be very difficult to analyze the runtime of a recursive function.
  - We'll see some techniques for special cases later in the quarter.
- One general technique is to look at the total number of calls made and the work done at each call.

# Analyzing DFS

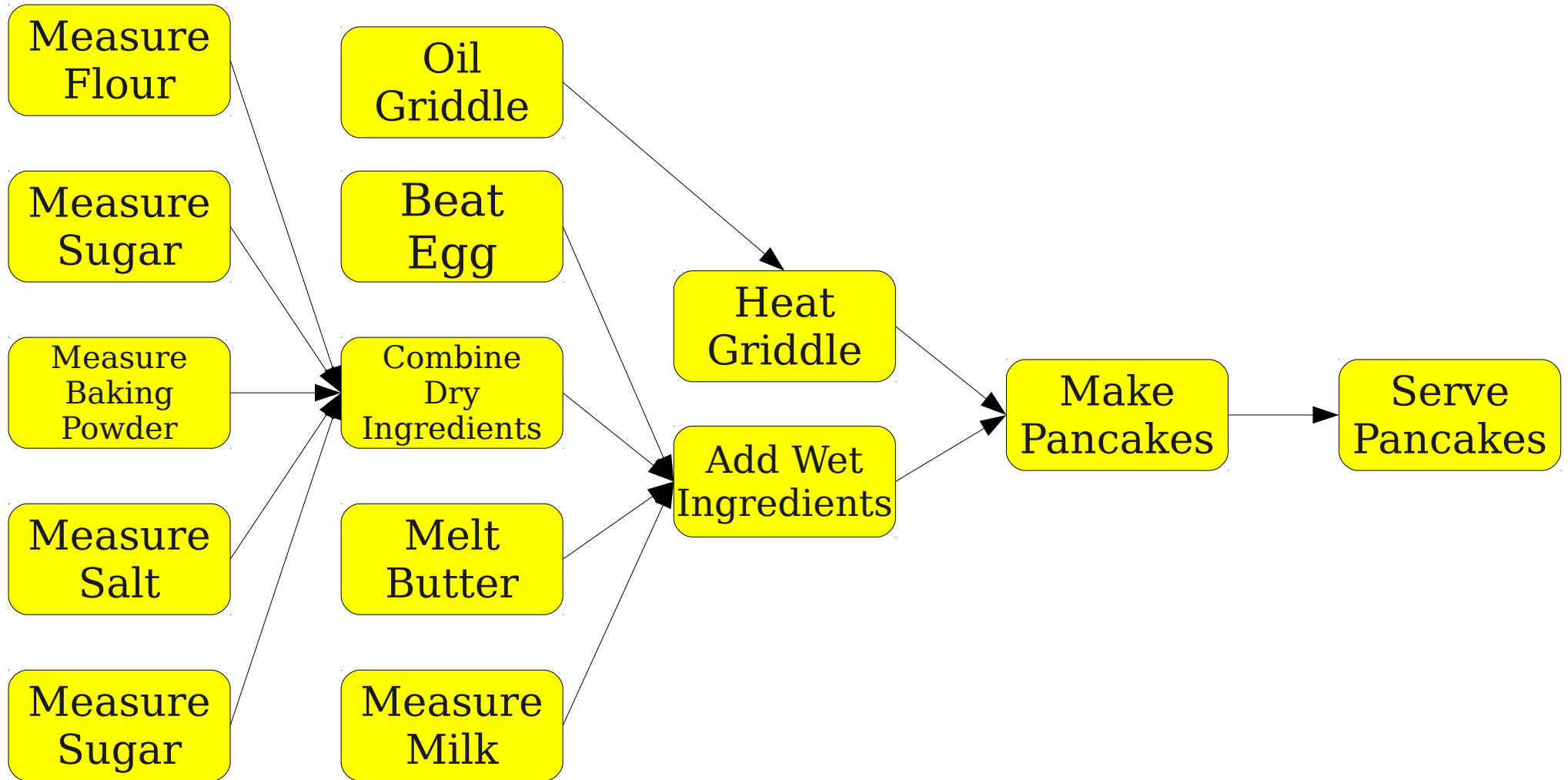
- The maximum number of function calls made is  $O(n)$ , since we can't call DFS on a node twice.
- Each call to DFS on node  $v$  does  $\Theta(\deg^+(v))$  work, since it visits each outgoing edge from  $v$  exactly once.
- Summing across all recursive calls:
  - $O(n)$  work done initially coloring nodes.
  - $O(n)$  work done coloring nodes yellow / green.
  - $O(m)$  work visiting edges.
  - Total work done:  **$O(m + n)$** .
- When might this not do  $\Theta(m + n)$  work?

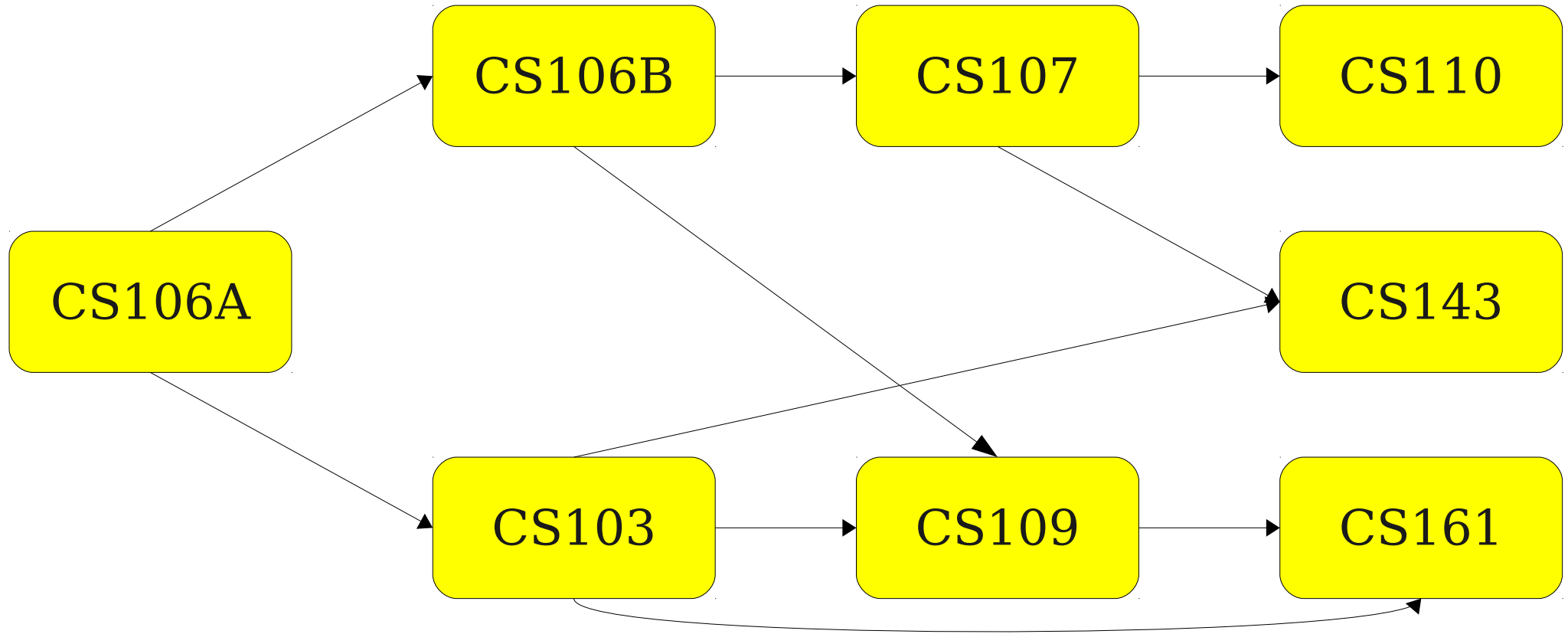


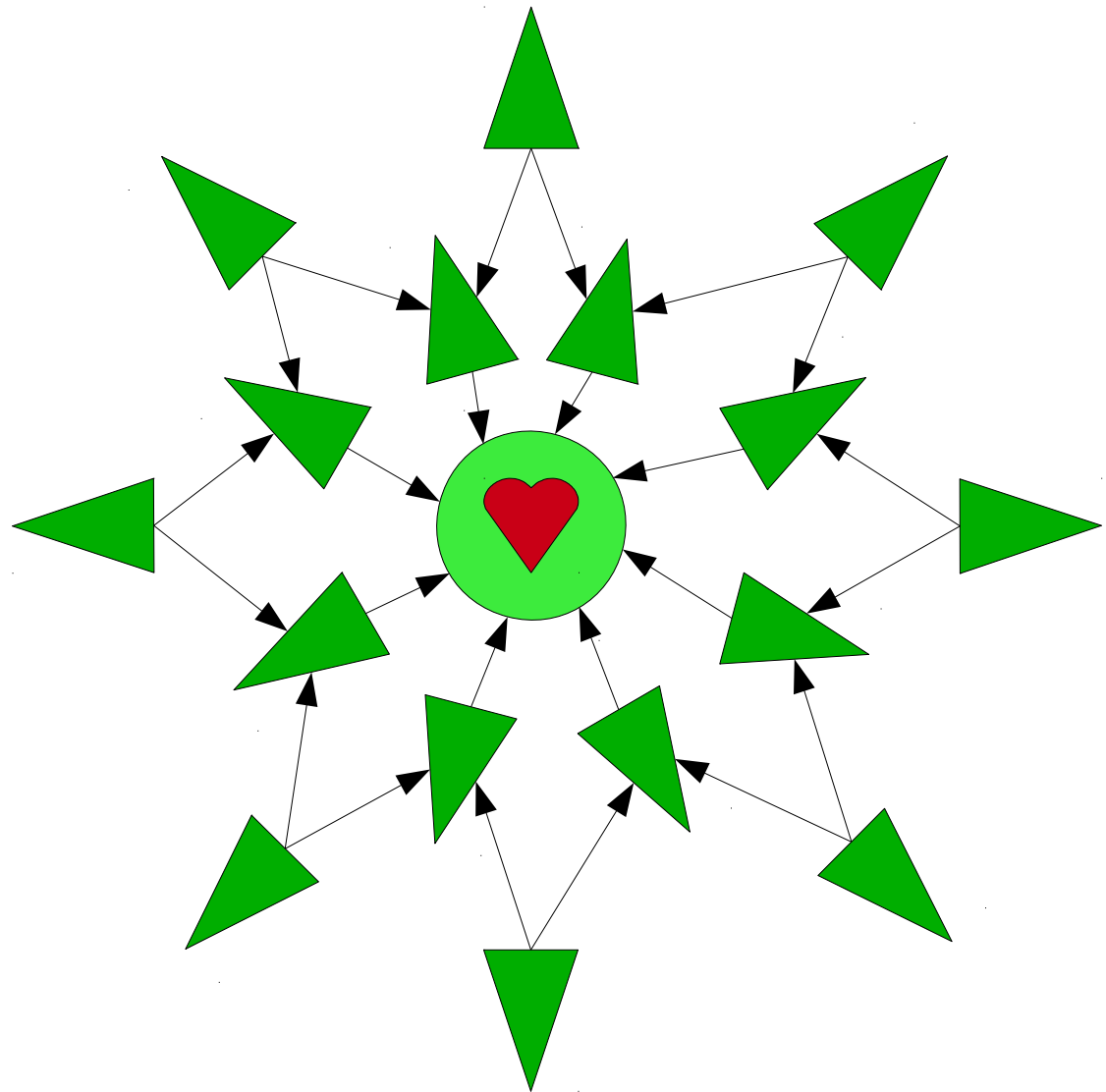
# BFS and DFS

- BFS and DFS always visit the same set of nodes.
- However, BFS always finds the shortest path from the source node to each other node in the graph, while DFS might not.
- That said: the order in which DFS visits nodes is pretty important and has lots of applications. We'll see some of them soon...

# Ordering Prerequisites





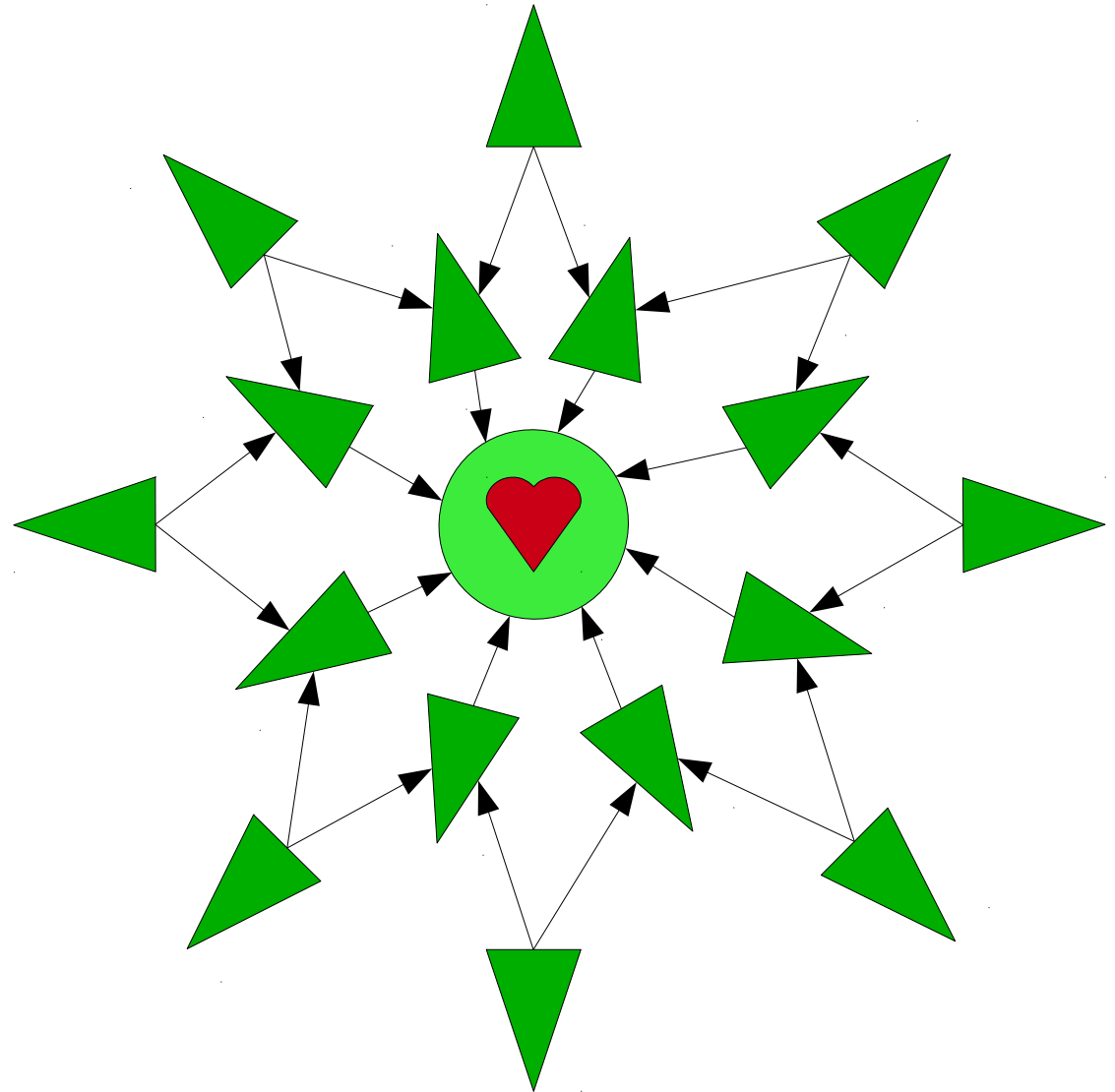


# Modeling Prerequisites

- We can model prerequisites as a graph with the following properties:
  - The graph has to be **directed**, since we have to be able to distinguish “A depends on B” from “B depends on A.”
  - The graph has to be **acyclic** (containing no cycles), since otherwise there is no way to accomplish all the tasks.
- A graph with this property is called a **directed acyclic graph**, or **DAG**.

# Some DAG Terminology

- A **source** node in a DAG is a node with no incoming edges.
- A **sink** node in a DAG is a node with no outgoing edges.
- DAGs can have many sources and sinks.



*Theorem:* Every nonempty DAG has at least one source node.

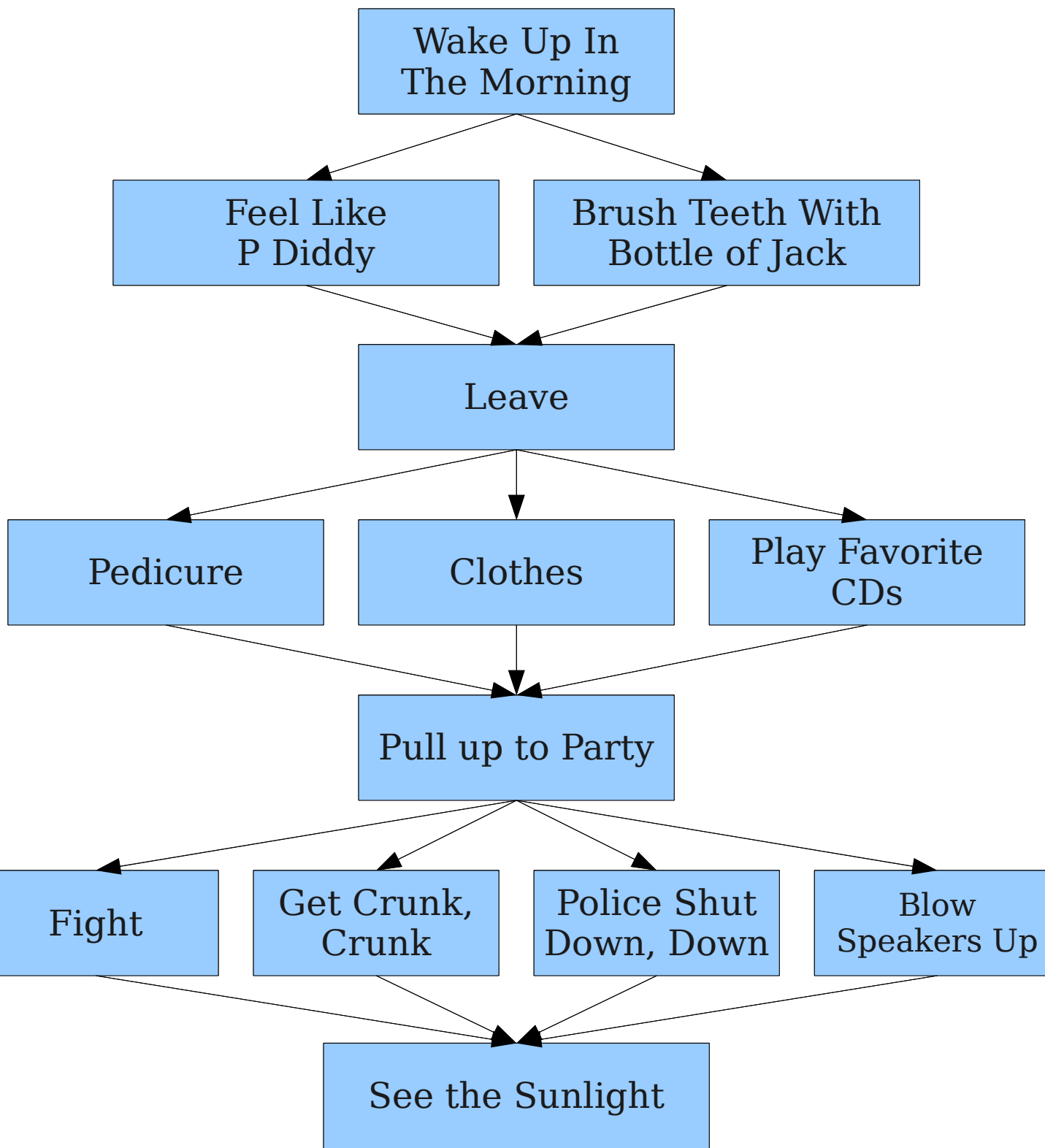
*Proof:* Suppose for the sake of contradiction that there is a nonempty DAG  $G$  where each node has at least one incoming edge. Start at any node  $v_1 \in G$  and repeatedly follow an edge entering  $v_1$  in reverse. This gives a sequence of nodes  $v_1, v_2, v_3, \dots$

Since there are only finitely many nodes in the DAG, this process eventually must revisit a node  $v_i$ . But then we have that  $v_i, v_{i+1}, v_{i+2}, \dots, v_i$  is a cycle in  $G$  traced in reverse order, contradicting the fact that  $G$  is a DAG. We have reached a contradiction, so our assumption was wrong and every DAG must contain at least one node with no incoming edges. ■



# Ordering Prerequisites

- When ordering prerequisites, we want to order the tasks such that no task is placed before tasks it depends on.
- In graph-theoretic terms: given a DAG  $G = (V, E)$ , we want to order the nodes so that if  $(u, v) \in E$ , then  $v$  appears after  $u$ .
- Such an ordering is called a **topological ordering**. An algorithm for finding a topological ordering is called a **topological sort**.



- Wake Up In The Morning
- Feel Like P Diddy
- Brush Teeth With Bottle of Jack
- Leave
- Clothes
- Play Favorite CDs
- Pedicure
- Pull up to Party
- Fight
- Blow Speakers Up
- Police Shut Down, Down
- Get Crunk, Crunk
- See the Sunlight

```
procedure topologicalSort(DAG G):  
  let result be an empty list.  
  while G is not empty:  
    let v be a node in G with indegree 0  
    add v to result  
    remove v from G  
  return result
```

# Correctness Proof Sketch

- Whenever a node  $v$  is added to the **result**, it has no incoming edges.
- Therefore, either
  - $v$  never had any incoming edges, in which case adding  $v$  to **result** cannot place  $v$  out of order, or
  - All of  $v$ 's predecessors have already been placed into **result**, and  $v$  comes after all of them.
- Can't get stuck, since every nonempty DAG has at least one source.

# Next Time

- Topological Sorting, Part II
- Connected Components
- Strongly-Connected Components
- Kosaraju's Algorithm I