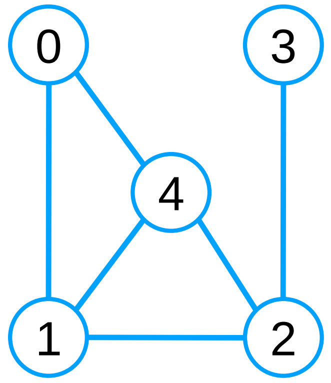
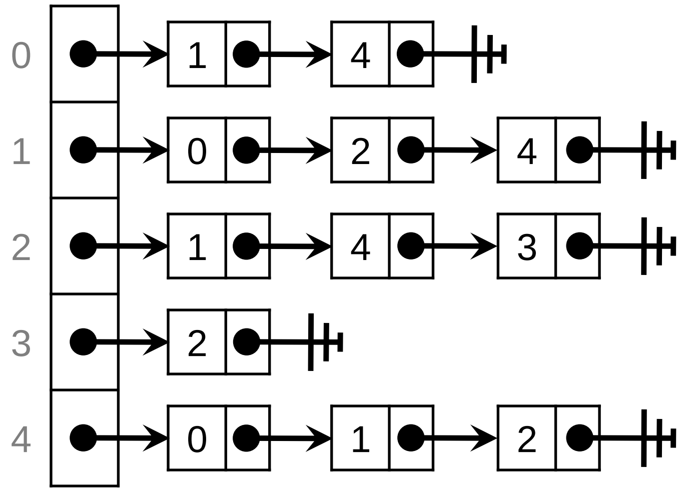
Graph Search

Review

- Graphs
  - Vertices, edges, neighbors, …
  - Dense, sparse
- Adjacency matrix implementation
- Adjacency list implementation



	0	1	2	3	4
0		✓			✓
1	✓		✓		✓
2		✓		✓	✓
3			✓		
4	✓	✓	✓		



graph.h

```
typedef unsigned int vertex;
typedef struct graph_header *graph_t;

graph_t graph_new(unsigned int numvert);
//@ensures \result != NULL;

void graph_free(graph_t G);
//@requires G != NULL;

unsigned int graph_size(graph_t G);
//@requires G != NULL;

bool graph_hasedge(graph_t G, vertex v, vertex w);
//@requires G != NULL;
//@requires v < graph_size(G) && w < graph_size(G);

void graph_addedge(graph_t G, vertex v, vertex w);
//@requires G != NULL;
//@requires v < graph_size(G) && w < graph_size(G);
//@requires v != w && !graph_hasedge(G, v, w);
-----------------------------------------------------------
typedef struct neighbor_header *neighbors_t;

neighbors_t graph_get_neighbors(graph_t G, vertex v);
//@requires G != NULL && v < graph_size(G);
//@ensures \result != NULL;

bool graph_hasmore_neighbors(neighbors_t nbors);
//@requires nbors  != NULL;

vertex graph_next_neighbor(neighbors_t nbors);
//@requires nbors  != NULL;
//@requires graph_hasmore_neighbors(nbors);
//@ensures is_vertex(\result);

void graph_free_neighbors(neighbors_t nbors);
//@requires nbors  != NULL;
```

Review

- Costs are similar for dense graphs
- AL is **more space-efficient** for sparse graphs
  - very common graphs
    - e ∊ O(v) is typical

	Adjacency list	Adjacency matrix
Space	O(v + e)	$O(v^2)$
graph_new	O(v)	$O(v^2)$
graph_free	O(v + e)	O(1)
graph_size	O(1)	O(1)
graph_hasedge	O(min(v,e))	O(1)
graph_addedge	O(1)	O(1)
graph_get_neighbors	O(1)	O(v)
graph_hasmore_neighbors	O(1)	O(1)
graph_next_neighbor	O(1)	O(1)
graph_free_neighbors	O(1)	O(min(v,e))

Assuming the neighbors are represented as a linked list

Review

- Typical function that **traverses** a graph
  - go over most vertices and edges

```
                                                         Cost                   Tally
void graph_print(graph_t G) {
  for (vertex v = 0; v < graph_size(G); v++) {          v times
    printf("Vertices connected to %u: ", v);               O(1)                 O(v)
    neighbors_t nbors = graph_get_neighbors(G, v);         O(1)                 O(v)
    while (graph_hasmore_neighbors(nbors)) {            ┐
      vertex w = graph_next_neighbor(nbors);            ├ O(e) altogether       O(v + e)
      printf(" %u,", w);                                │
    }                                                   ┘
    graph_free_neighbors(nbors);                           O(1)                 O(v + e)
    printf("\n");                                          O(1)                 O(v + e)
  }
}
```

  - Adjacency list: O(v + e)
  - Adjacency matrix: O(v^2)

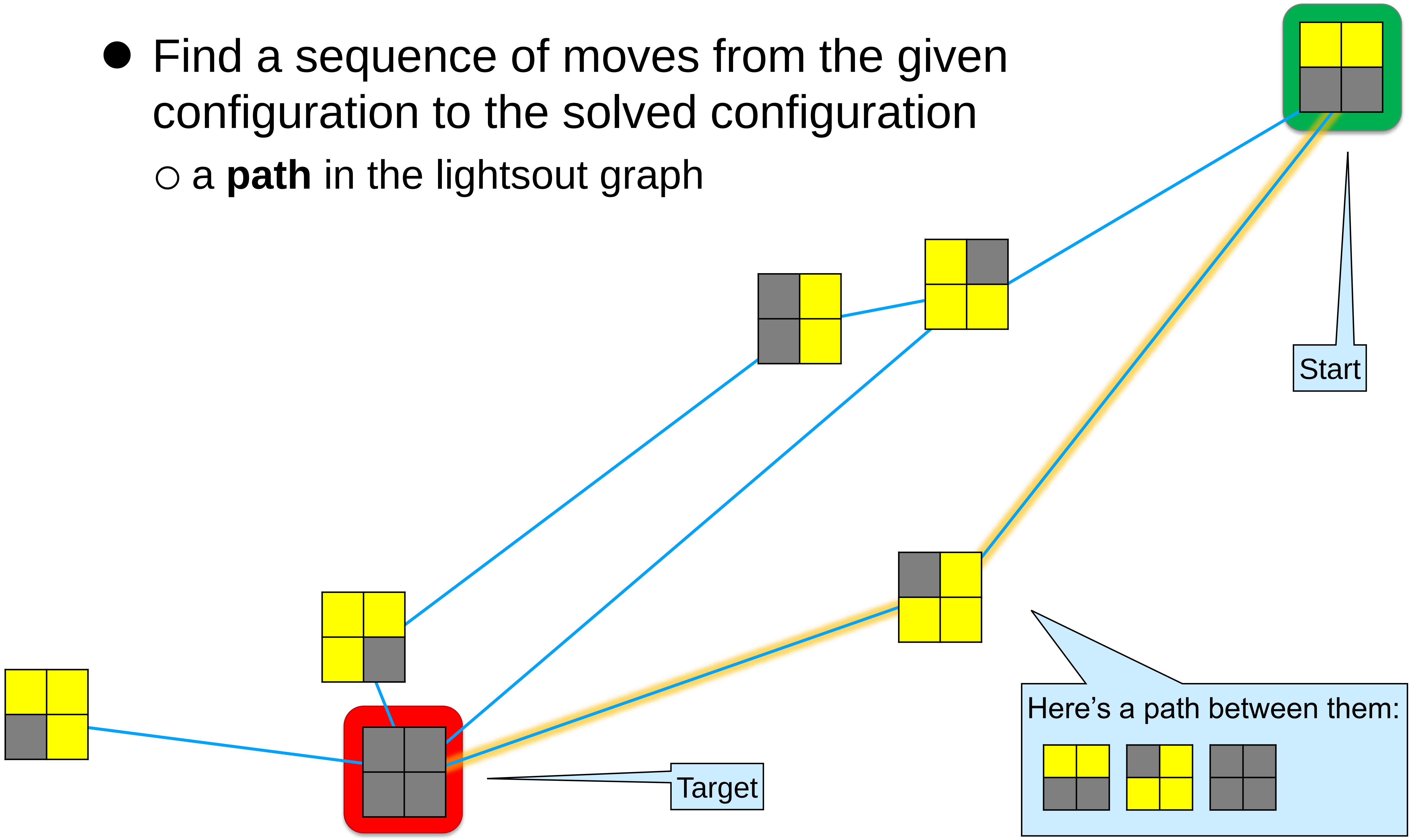
AL is much better for sparse graphs

Same as space bounds

Graph Connectivity

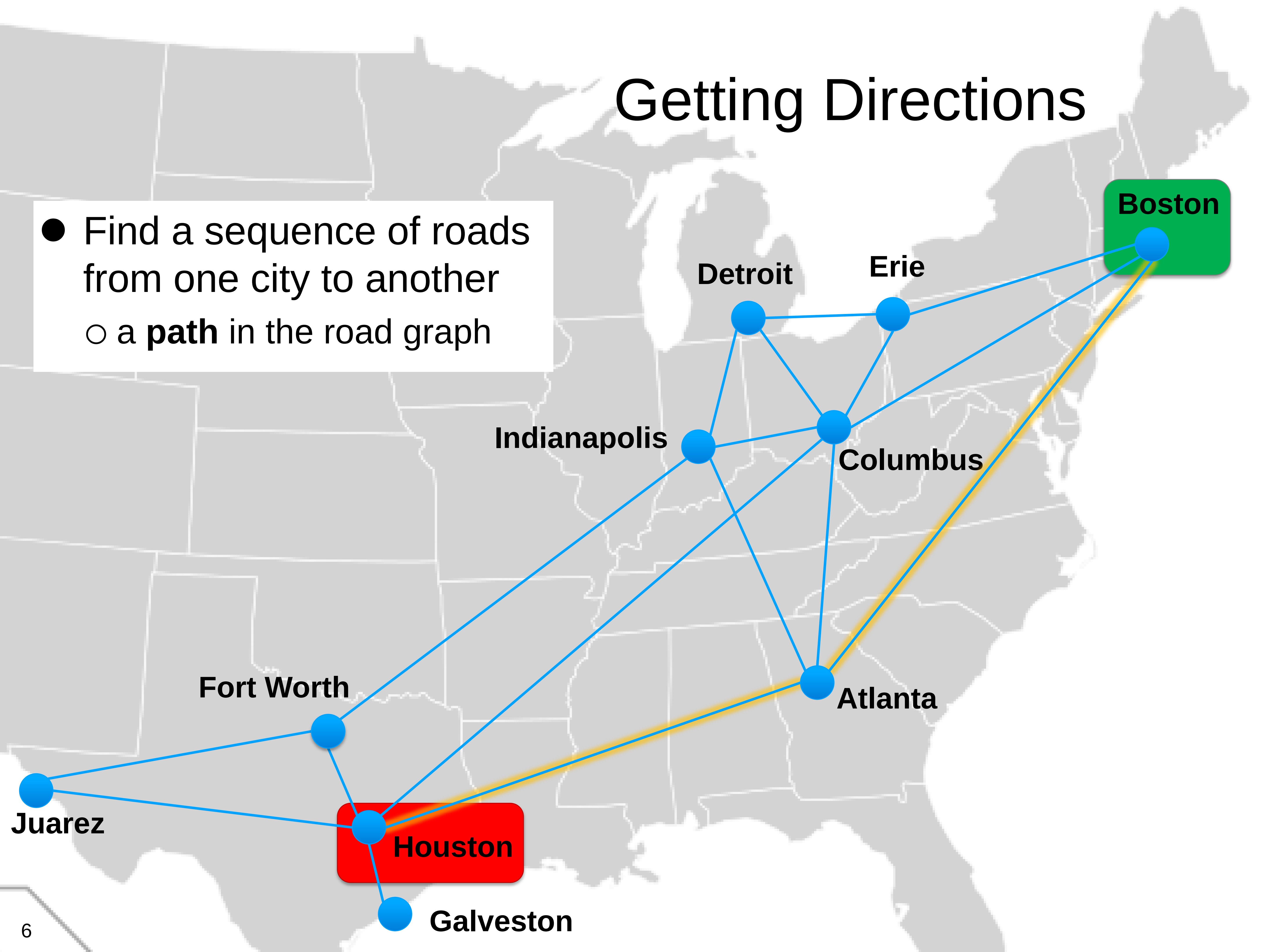
Solving Lightsout

- Find a sequence of moves from the given configuration to the solved configuration
  - a **path** in the lightsout graph



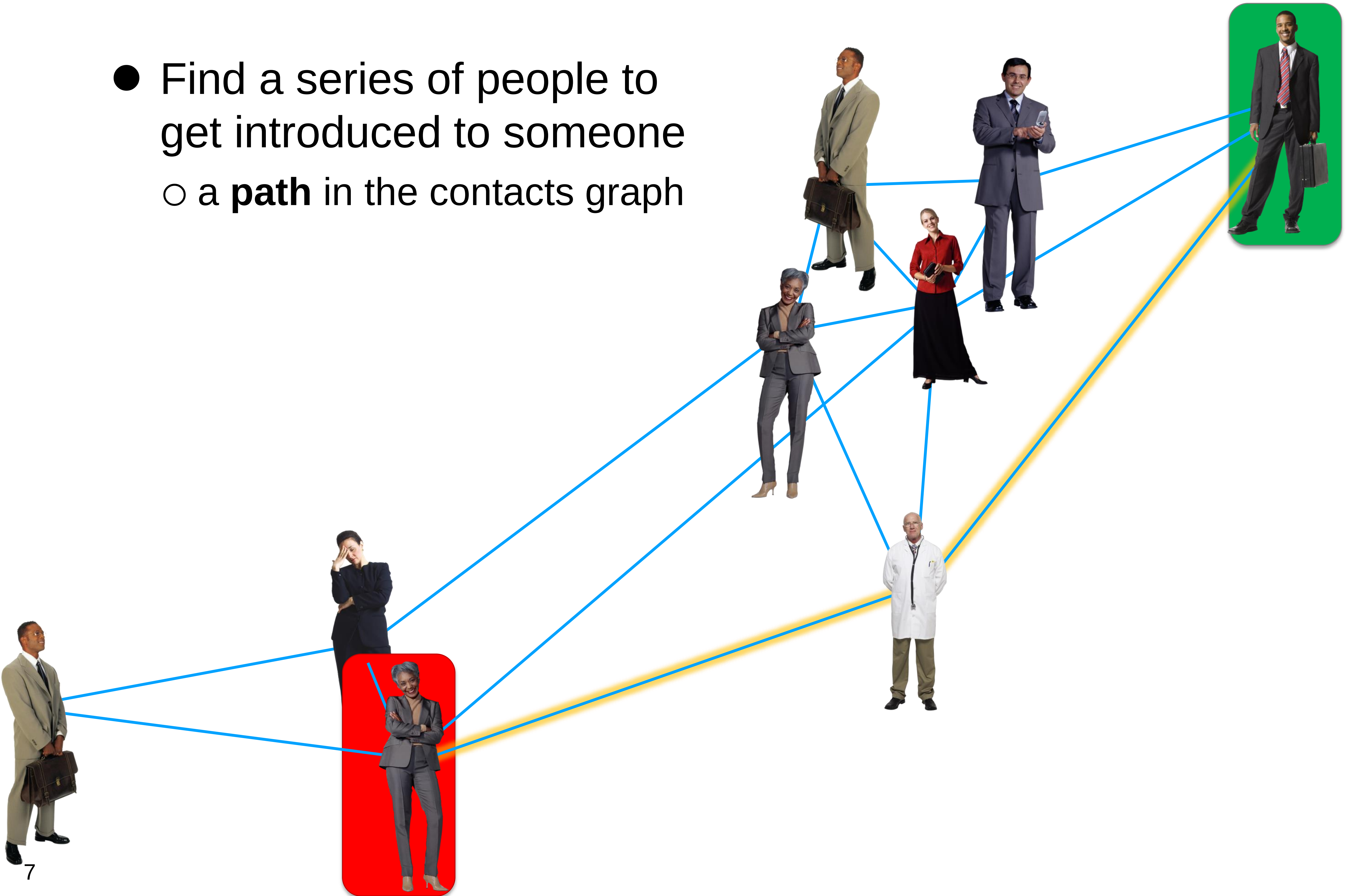
Getting Directions

- Find a sequence of roads from one city to another
  - a **path** in the road graph



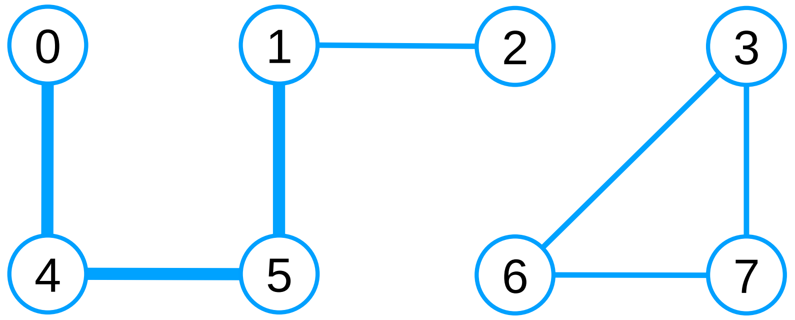
Getting Introduced

- Find a series of people to get introduced to someone
  - a **path** in the contacts graph



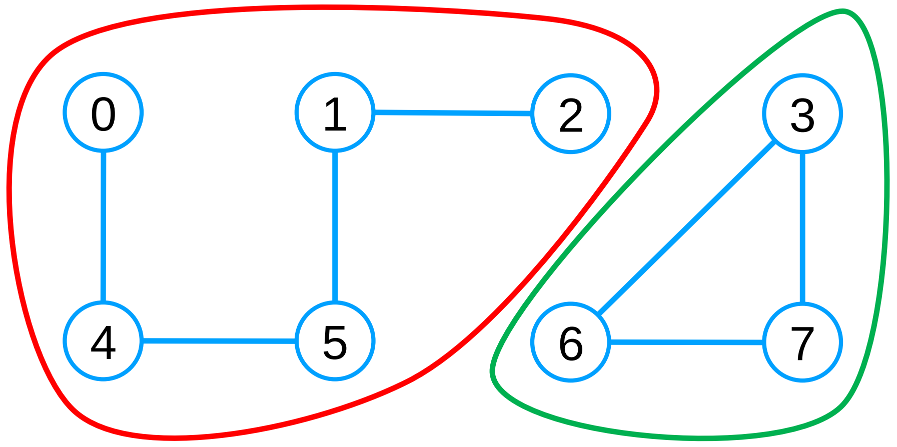
Connected Vertices

- A **path** is a sequence of vertices linked by edges
  - 0-4-5-1 is a path between 0 and 1



- Two vertices are **connected** if there is a path between them
  - 0 and 1 are connected
  - 0 and 7 are not connected
- If v_1 and v_2 are connected, then v_2 is **reachable** from v_1

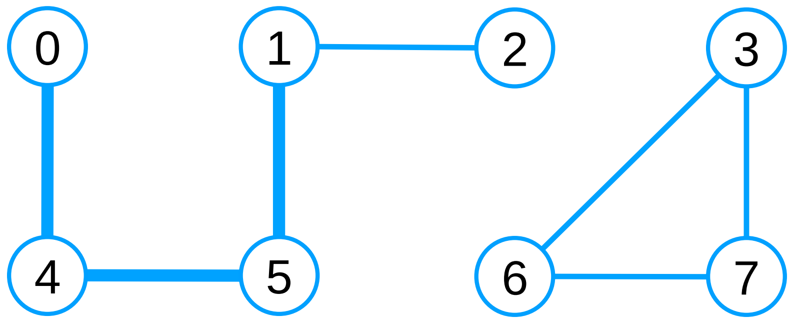
- A **connected component** is a maximal set of vertices that are connected
  - this graph has two connected components



Checking Reachability

- How do we check if two vertices are connected?
  - graph_hasedge only tells us if they are *directly* connected
    - by an edge
  - We want to develop a general algorithm to check reachability
    - then we can use it to check reachability in any domain
      - to check if lightsout is solvable from a given board
      - to figure out if there are roads between two cities
      - to know if there is any social connection between two people

The rest of this lecture



Finding Paths

- How do we find a path between two vertices?
    - ❑ what is a solution to lightsout from a given board?
    - ❑ what roads are there between two cities?
    - ❑ what series of people can get me introduced to person X?
  - ○ an algorithm that checks reachability can be instrumented to report a path between the two vertices

We will limit ourselves to reachability

- A path is a **witness** that two vertices are connected
  - ○ Finding a witness is called a **search problem**
  - ○ Checking a witness is called a **verification problem**
    - ➢ checking that a witness is valid is often a lot easier than finding a witness

This is the basic principle underlying **cryptography**

Checking Reachability

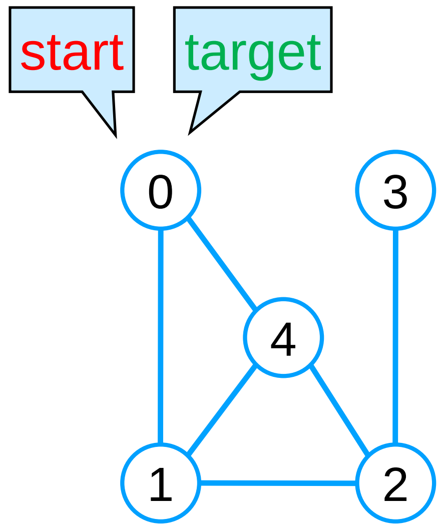
- Let's define reachability mathematically

This is an inductive definition

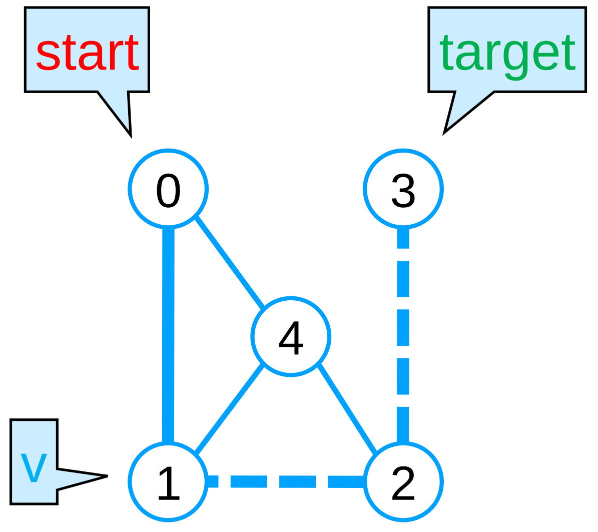
There is a path from start to target if
- ○ start == target, or
- ○ there is an edge from start to some vertex v and there is a path from v to target

base case

inductive case



There is a path from 0 to 0



There is a path from 0 to 3

Recursive Depth-first Search – I

Implementing the Definition

- We can immediately transcribe this inductive definition into a recursive **client-side** function

There is a path from start to target if
- start == target, or
- there is an edge from start to some vertex v
  and there is a path from v to target

```
bool naive_dfs(graph_t G, vertex start, vertex target) {
  REQUIRES(G != NULL);
  REQUIRES(start < graph_size(G) && target < graph_size(G));

  // there is a path from start to target if
  // target == start, or

  // there is an edge from start to ...

    // ... some vertex v …

    // ... and there is  a path from v to target

}
```

Contracts

Implementing the Definition

```
bool naive_dfs(graph_t G, vertex start, vertex target) {
  REQUIRES(G != NULL);
  REQUIRES(start < graph_size(G) && target < graph_size(G));
  printf("    Visiting %u\n", start);

  // there is a path from start to target if
  // target == start, or
  if (target == start) return true;
  // there is an edge from start to ...
  neighbors_t nbors = graph_get_neighbors(G, start);
  while (graph_hasmore_neighbors(nbors)) {
    // ... some vertex v …
    vertex v = graph_next_neighbor(nbors);
    // ... and there is  a path from v to target
    if (naive_dfs(G, v, target)) {
      graph_free_neighbors(nbors);
      return true;
    }
  }
  graph_free_neighbors(nbors);
  return false;
}
```

graph.h

```
typedef unsigned int vertex;
typedef struct graph_header *graph_t;

graph_t graph_new(unsigned int numvert);
//@ensures \result != NULL;

void graph_free(graph_t G);
//@requires G != NULL;

unsigned int graph_size(graph_t G);
//@requires G != NULL;

bool graph_hasedge(graph_t G, vertex v, vertex w);
//@requires G != NULL;
//@requires v < graph_size(G) && w < graph_size(G);

void graph_addedge(graph_t G, vertex v, vertex w);
//@requires G != NULL;
//@requires v < graph_size(G) && w < graph_size(G);
//@requires v != w && !graph_hasedge(G, v, w);
-----------------------------------------------------------
typedef struct neighbor_header *neighbors_t;

neighbors_t graph_get_neighbors(graph_t G, vertex v);
//@requires G != NULL && v < graph_size(G);
//@ensures \result != NULL;

bool graph_hasmore_neighbors(neighbors_t nbors);
//@requires nbors  != NULL;

vertex graph_next_neighbor(neighbors_t nbors);
//@requires nbors  != NULL;
//@requires graph_hasmore_neighbors(nbors);
//@ensures is_vertex(\result);

void graph_free_neighbors(neighbors_t nbors);
//@requires nbors  != NULL;
```

Implementing the Definition

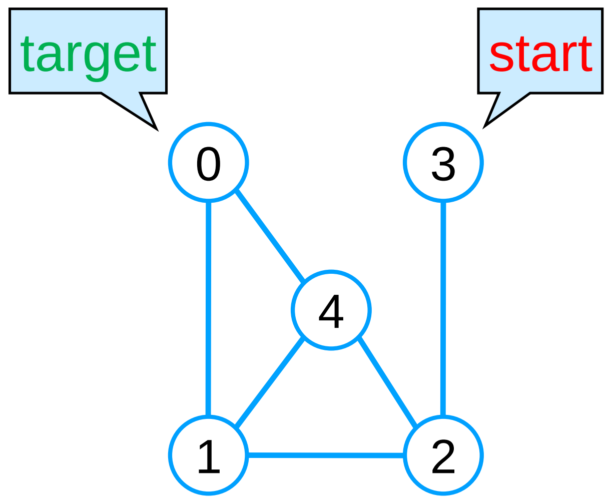
```
bool naive_dfs(graph_t G, vertex start, vertex target) {
  REQUIRES(G != NULL);
  REQUIRES(start < graph_size(G) && target < graph_size(G));
  printf("    Visiting %u\n", start);

  // there is a path from start to target if
  // target == start, or
  if (target == start) return true;
  // there is an edge from start to ...
  neighbors_t nbors = graph_get_neighbors(G, start);
  while (graph_hasmore_neighbors(nbors)) {
    // ... some vertex v …
    vertex v = graph_next_neighbor(nbors);
    // ... and there is  a path from v to target
    if (naive_dfs(G, v, target)) {
      graph_free_neighbors(nbors);
      return true;
    }
  }
  graph_free_neighbors(nbors);
  return false;
}
```

- It has the same structure as graph_print

```
void graph_print(graph_t G) {
  for (vertex v = 0; v < graph_size(G); v++) {
    printf("Vertices connected to %u: ", v);
    neighbors_t nbors = graph_get_neighbors(G, v);
    while (graph_hasmore_neighbors(nbors)) {
      vertex w = graph_next_neighbor(nbors);
      printf(" %u,", w);
    }
    graph_free_neighbors(nbors);
    printf("\n");
  }
}
```

  - the outer loop is replaced with recursion

Does it Work?



- Let's check there is a path from 3 to 0

start	target	nbors
3	0	2
2	0	1, 3, 4
1	0	0, 2, 4
0	0	✓

Assume the neighbors are returned from smallest to biggest

- Let's run it

Linux Terminal

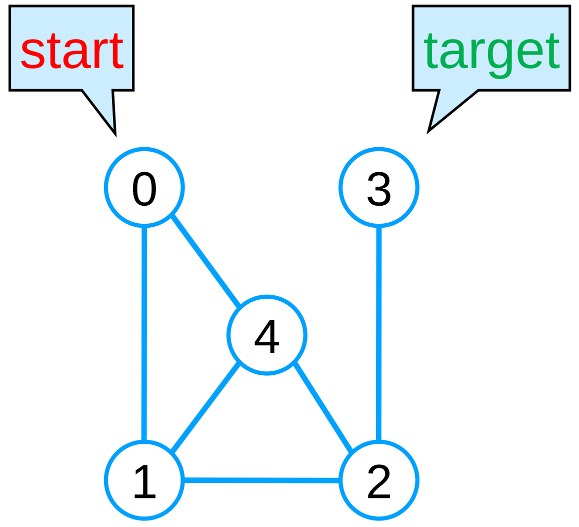
```
# gcc … lib/*.c connected.c main.c
# ./a.out 3 0
    Visiting 3
    Visiting 2
    Visiting 1
    Visiting 0
Reachable
```

… from to

Looks good

```
bool naive_dfs(graph_t G, vertex start, vertex target) {
  REQUIRES(G != NULL);
  REQUIRES(start < graph_size(G) && target < graph_size(G));
  printf("    Visiting %u\n", start);

  // there is a path from start to target if
  // target == start, or
  if (target == start) return true;
  // there is an edge from start to ...
  neighbors_t nbors = graph_get_neighbors(G, start);
  while (graph_hasmore_neighbors(nbors)) {
    // ... some vertex v …
    vertex v = graph_next_neighbor(nbors);
    // ... and there is  a path from v to target
    if (naive_dfs(G, v, target)) {
      graph_free_neighbors(nbors);
      return true;
    }
  }
  graph_free_neighbors(nbors);
  return false;
}
```

Does it *Always* Work?



- Let's check there is a path from 0 to 3

start	target	nbors
0	3	1, 4
1	3	0, 2, 4
0	3	1, 4
1	3	0, 2, 4
… (this is not promising) …		

- Let's run it

Linux Terminal

```
# gcc … lib/*.c connected.c main.c
# ./a.out 0 3
    Visiting 0
    Visiting 1
    Visiting 0
```

runs forever!

```
bool naive_dfs(graph_t G, vertex start, vertex target) {
  REQUIRES(G != NULL);
  REQUIRES(start < graph_size(G) && target < graph_size(G));
  printf("    Visiting %u\n", start);

  // there is a path from start to target if
  // target == start, or
  if (target == start) return true;
  // there is an edge from start to ...
  neighbors_t nbors = graph_get_neighbors(G, start);
  while (graph_hasmore_neighbors(nbors)) {
    // ... some vertex v …
    vertex v = graph_next_neighbor(nbors);
    // ... and there is  a path from v to target
    if (naive_dfs(G, v, target)) {
      graph_free_neighbors(nbors);
      return true;
    }
  }
  graph_free_neighbors(nbors);
  return false;
}
```

It does **not** Work

- Either the definition is wrong or the code is wrong
- Definition
  - it magically picks the right neighbor v if there is one
    - the magic of "*there is …*"

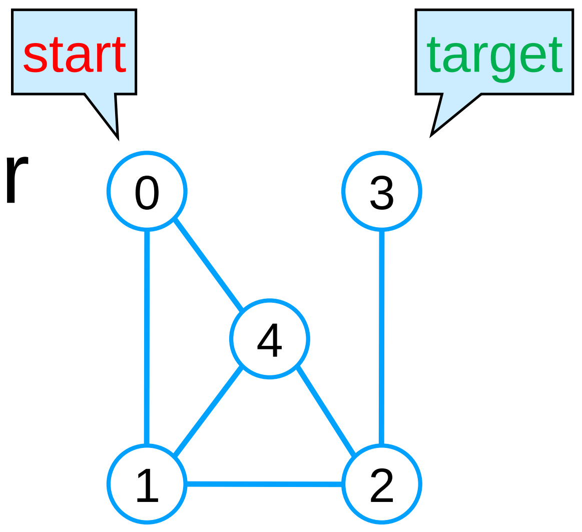
The definition is fine

- Code
  - it must examine the neighbors in some order
    - the first v may not be the right one

There is a path from start to target if
- start == target, or
- there is an edge from start to some vertex v and there is a path from v to target

```
bool naive_dfs(graph_t G, vertex start, vertex target) {
  REQUIRES(G != NULL);
  REQUIRES(start < graph_size(G) && target < graph_si…
  printf("    Visiting %u\n", start);

  // there is a path from start to target if
  // target == start, or
  if (target == start) return true;
  // there is an edge from start to ...
  neighbors_t nbors = graph_get_neighbors(G, start);
  while (graph_hasmore_neighbors(nbors)) {
    // ... some vertex v …
    vertex v = graph_next_neighbor(nbors);
    // ... and there is  a path from v to target
    if (naive_dfs(G, v, target)) {
      graph_free_neighbors(nbors);
      return true;
    }
  }
  graph_free_neighbors(nbors);
  return false;
}
```

Why doesn't it Work?



- The code examines the neighbors in some order
  - it always starts with the same v
    - the first neighbor
  - … even if it has been examined before

start	target	nbors
0	3	1, 4
1	3	0, 2, 4
0	3	1, 4
1	3	0, 2, 4
…		

- The code will never visit the second neighbor (if there is one)
  - it charges ahead with the first neighbor, always
  - if there is a path by only examining first neighbors, it will find it
  - if the path involves some other neighbor, it won't

Recursive Depth-first Search – II

Fixing the Code

- Problems: the code examines the same neighbors over and over

- Solution: **mark** vertices that are being examined
  - only examine a vertex if it is unmarked
  - mark it right away

- How to mark vertices?
  - carry around an array of booleans
    - true = marked
    - false = unmarked

We could use any implementation of **sets**, e.g., hash sets

Fixing the code

- Carry around an array of booleans
- Only run if start is unmarked
- Mark it right away
- Only examine a neighbor if it’s unmarked
  - we need to guard the recursive call

```
bool dfs_helper(graph_t G, bool *mark, vertex start, vertex target) {
  REQUIRES(G != NULL);
  REQUIRES(start < graph_size(G) && target < graph_size(G));
  REQUIRES(!mark[start]);

  mark[start] = true;
  printf("    Visiting %u\n", start);

  // there is a path from start to target if
  // target == start, or
  if (target == start) return true;
  // there is an edge from start to ...
  neighbors_t nbors = graph_get_neighbors(G, start);
  while (graph_hasmore_neighbors(nbors)) {
    // ... some vertex v …
    vertex v = graph_next_neighbor(nbors);
    // ... and there is  a path from v to target
    if (!mark[v] && dfs_helper(G, mark, v, target)) {
      graph_free_neighbors(nbors);
      return true;
    }
  }
  graph_free_neighbors(nbors);
  return false;
}
```

Fixing the Code

- We have modified the prototype of the function
  - but the client should not have to deal with the added details
  - export a **wrapper** instead of dsf_helper

```
bool dfs (graph_t G, vertex start, vertex target) {
  REQUIRES(G != NULL);
  REQUIRES(start < graph_size(G) && target < graph_size(G));

  bool *mark = xcalloc(graph_size(G), sizeof(bool));
  bool connected = dfs_helper(G, mark, start, target);
  free(mark);
  return connected;
}
```

Create the mark array: calloc initializes all positions to false

We must free mark since we calloc'ated it

An Alternative Wrapper

- We can also use a *stack-allocated array* for mark

```
bool dfs(graph_t G, vertex start, vertex target) {
  REQUIRES(G != NULL);
  REQUIRES(start < graph_size(G) && target < graph_size(G));

  bool mark[graph_size(G)];
  for (unsigned int v = 0; v < graph_size(G); v++)
     mark[v] = false;

  return dfs_helper(G, mark, start, target);
}
```

Create the a stack allocated array of size graph_size(G)

We need to initialize it explicitly

But we don't need to free it

- Is this version preferable?
  - stack space is limited
  - for a large graph, the stack may not be big enough
    - **stack overflow**

Does it Work?

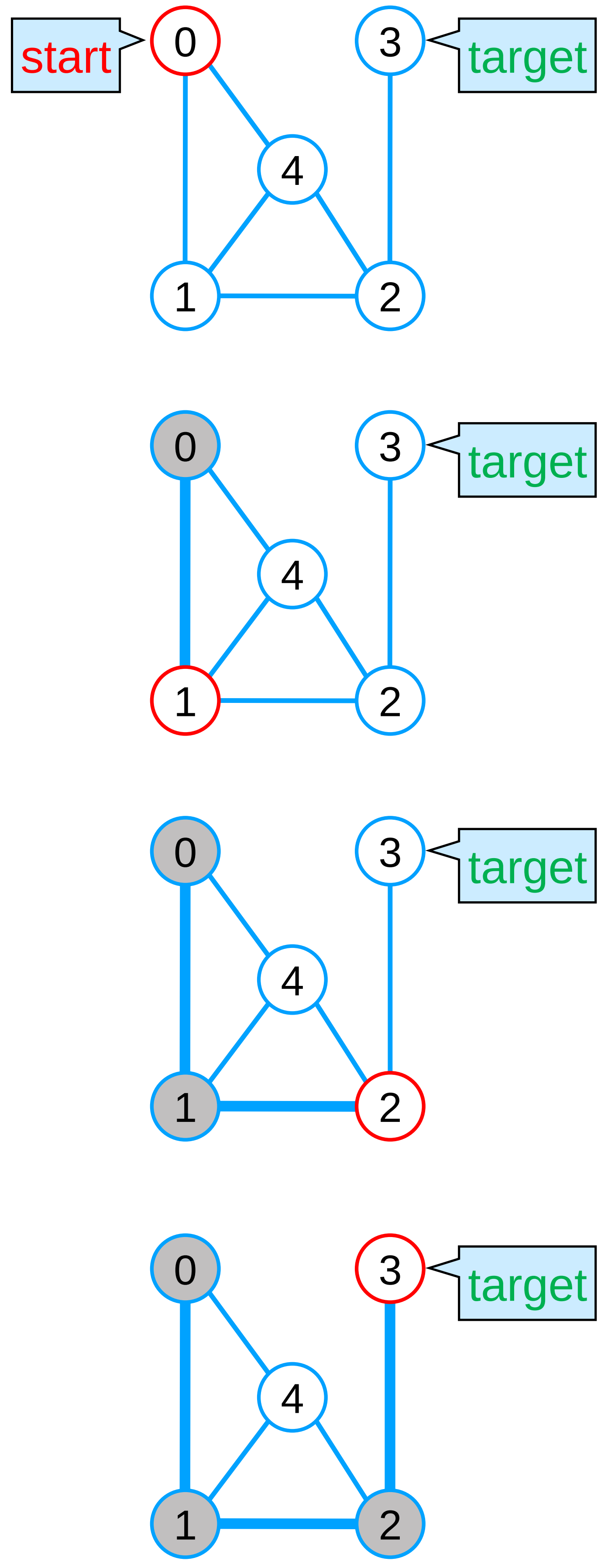
- Let’s check there is a path from 0 to 3

start	target	nbors	marked
0	3	1, 4	0
1	3	0, 2, 4	0, 1
2	3	1, 3, 4	0, 1, 2
3	3	✓	

- Let’s run it

Linux Terminal

```
# gcc … lib/*.c connected.c main.c
# ./a.out 0 3
    Visiting 0
    Visiting 1
    Visiting 2
    Visiting 3
Reachable
```



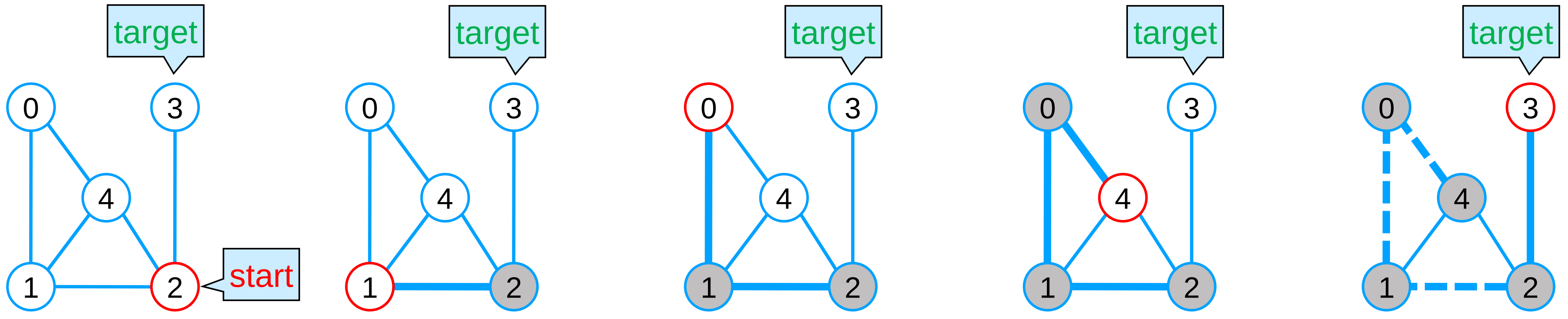
Backtracking

- Let’s check there is a path from 2 to 3

start	target	nbors	marked
2	3	1, 3, 4	2
1	3	0, 2, 4	1, 2
0	3	1, 4	0, 1, 2
4	3	✗ 0, 1, 2	0, 1, 2, 4
3	3	✓	

3 ≠ 4 and all the neighbors of 4 are marked

We **backtrack** to a vertex that has a still unmarked neighbor continue from it



Backtracking

start	target	nbors	marked
2	3	1, 3, 4	2
1	3	0, 2, 4	1, 2
0	3	1, 4	0, 1, 2
4	3	✗ 0, 1, 2	0, 1, 2, 4
3	3	✓	

- *We backtrack to a vertex that has a still unmarked neighbor and continue from it*
- This is achieved by returning **false** from the recursive call
  - the caller will then try the next unmarked neighbor
- Let's run it

Linux Terminal

```
# gcc … lib/*.c connected.c main.c
# ./a.out 2 3
    Visiting 2
    Visiting 1
    Visiting 0
    Visiting 4
    Visiting 3
Reachable
```

```
 …
 while (graph_hasmore_neighbors(nbors)) {
   // ... some vertex v …
   vertex v = graph_next_neighbor(nbors);
   // ... and there is  a path from v to target
   if (!mark[v] && dfs_helper(G, mark v, target)) {
     graph_free_neighbors(nbors);
     return true;
   }
 }
 graph_free_neighbors(nbors);
 return false;
}
```

graph_size	O(1)

Complexity of dfs

- Let’s call dfs on a graph with
  - v vertices,
  - e edges, and
  - implemented using adjacency lists

```
bool dfs (graph_t G, vertex start, vertex target) {
  REQUIRES(G != NULL);
  REQUIRES(start < graph_size(G) && target < graph_size(G));

  bool *mark = xcalloc(graph_size(G), sizeof(bool));
  bool connected = dfs_helper(G, mark, start, target);
  free(mark);
  return connected;
}
```

O(v)

free has constant cost

- The cost of dfs is O(v) plus the cost of dfs_helper

Complexity of dfs_helper

graph_get_neighbors	O(1)
graph_hasmore_neighbors	O(1)
graph_next_neighbor	O(1)
graph_free_neighbors	O(1)

- The body of the loop runs at most 2e times **altogether**
    - ❑ at most 2e calls to graph_next_neighbors
    - ➢ e edges from either endpoint
    - ➢ each endpoint is examined at most once

Just like for graph_print

- There are at most v recursive calls
    - ○ up to v vertices can be marked

In reality, it's more like min(v,e)

- Every operation costs O(1)
- dfs_helper has cost O(e + v)

```
bool dfs_helper(graph_t G, bool *mark, vertex start, vertex target) {
  mark[start] = true;                                     O(1)          O(v)

  if (target == start) return true;                       O(1)          O(v)

  neighbors_t nbors = graph_get_neighbors(G, start);      O(1)          O(v)
  while (graph_hasmore_neighbors(nbors)) {           O(1) ⎫
    vertex v = graph_next_neighbor(nbors);           O(1) ⎪
    if (!mark[v] && dfs_helper(G, mark, v, target)) { O(1) ⎬  O(e)        O(v + e)
      graph_free_neighbors(nbors);                   O(1) ⎪  altogether
      return true;                                        ⎪
    }                                                     ⎪
  }                                                       ⎭
  graph_free_neighbors(nbors);                            O(1)          O(v + e)
  return false;
}                                                                       Tally
```

graph_size	O(1)
graph_get_neighbors	O(1)
graph_hasmore_neighbors	O(1)
graph_next_neighbor	O(1)
graph_free_neighbors	O(1)

Complexity of dfs

- Let's call dfs on a graph with
  - v vertices,
  - e edges, and
  - implemented using adjacency lists

```
bool dfs (graph_t G, vertex start, vertex target) {
  REQUIRES(G != NULL);
  REQUIRES(start < graph_size(G) && target < graph_size(G));

  bool *mark = xcalloc(graph_size(G), sizeof(bool));
  bool connected = dfs_helper(G, mark, start, target);
  free(mark);
  return connected;
}
```

O(v)

O(v + e)

- The cost of dfs is O(v + e)

Complexity of dfs

For a graph with v vertices and e edges

- O(v + e) using the adjacency list implementation

Holds for both sparse and dense graphs

- O(v^2) using the adjacency matrix implementation

Exercise

- AL is more efficient for sparse graphs
  - the most common kind of graphs

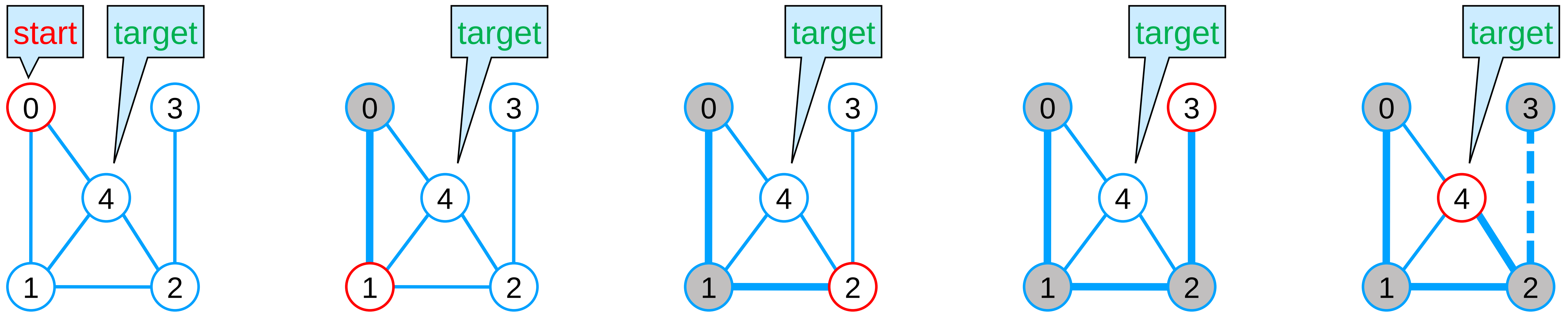
Moving forward, we will always assume an adjacency list implementation

Breadth-first Search

How does dfs Work?

- When calling dfs on 0 and 4, it finds the path 0–1–2–4
  - it also visits 3 and backtracks
- But there is a much shorter path: 0–4
  - dfs does more work than strictly necessary

start	target	nbors	marked
0	4	1, 4	0
1	4	0, 2, 4	0, 1
2	4	1, 3, 4	0, 1, 2
3	4	✗ 2	0, 1, 2, 3
4	4	✓	



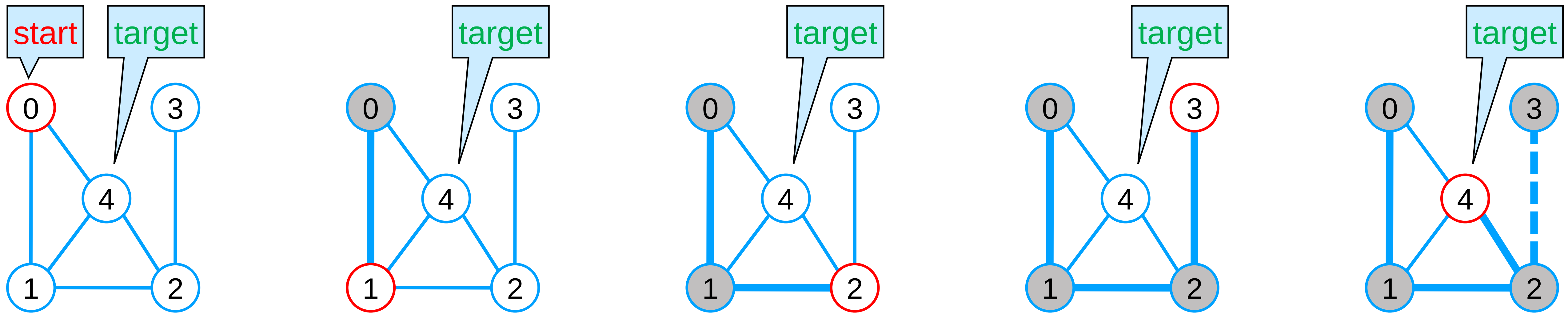
How does dfs Work?

- dfs charges ahead until
  - it finds the target vertex
  - or it hits a dead end
    - then it backtracks to the last choice point

start	target	nbors	marked
0	4	1, 4	0
1	4	0, 2, 4	0, 1
2	4	1, 3, 4	0, 1, 2
3	4	✗ 2	0, 1, 2, 3
4	4	✓	

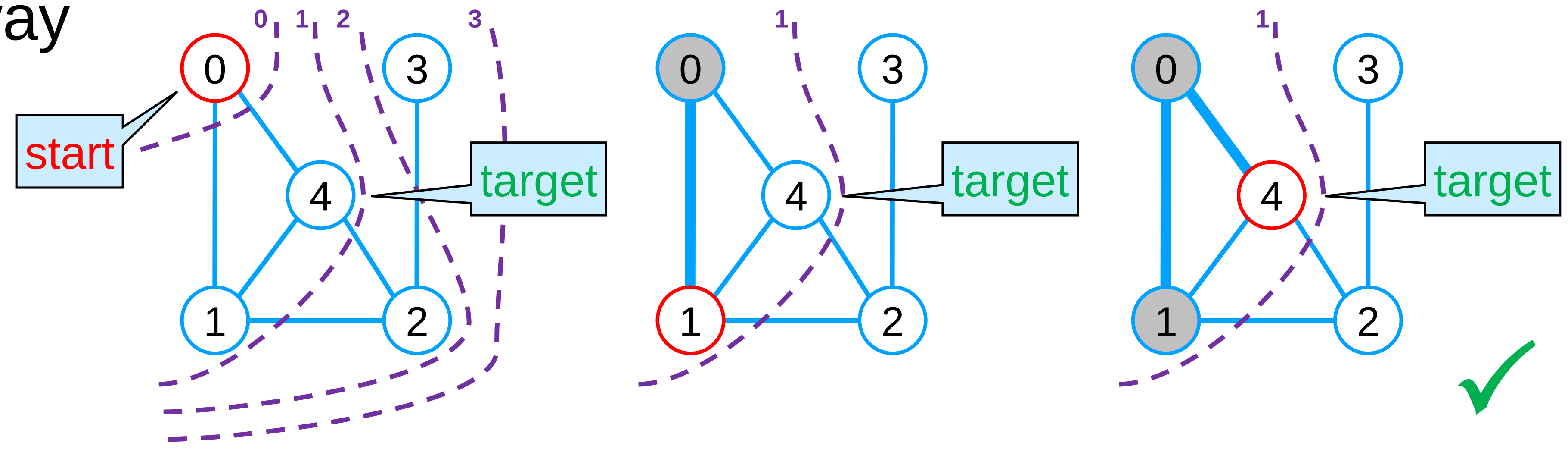
- This strategy is called **depth-first search**

DFS



Breadth-first Search

- To find the shortest path, we need to explore the graph **level by level** from the start vertex
  - first look at the vertices 0 hops away from start,
    - if start == end
  - then look at the vertices 1 hop away from start
  - then 2 hops away
  - then 3 hops away
  - …

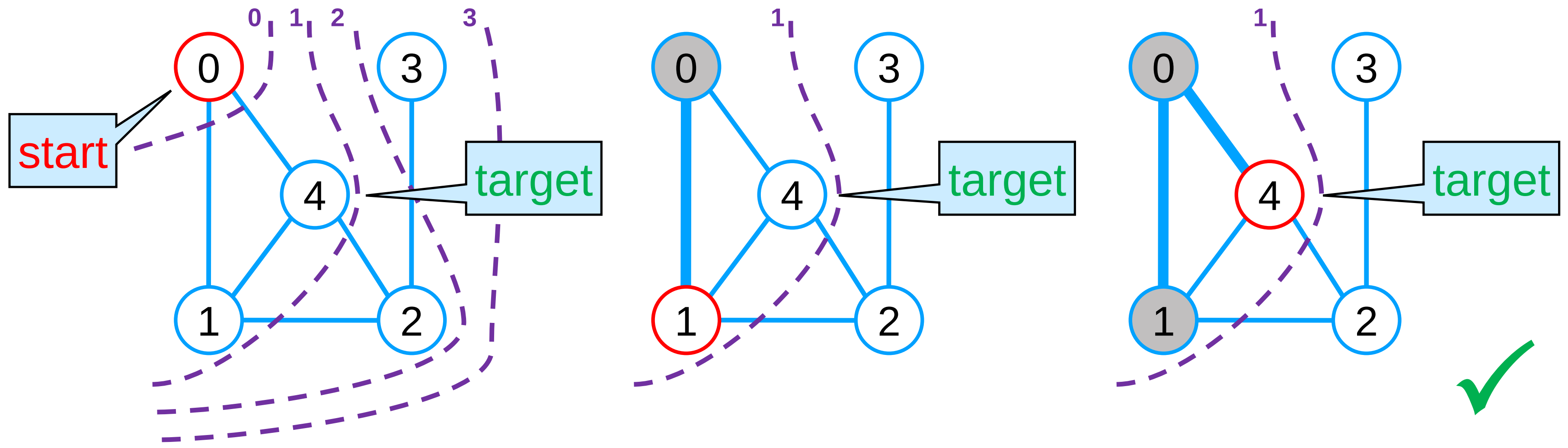


- This strategy is called **breadth-first search**

BFS

Breadth-first Search

- We need to traverse the graph **level by level**



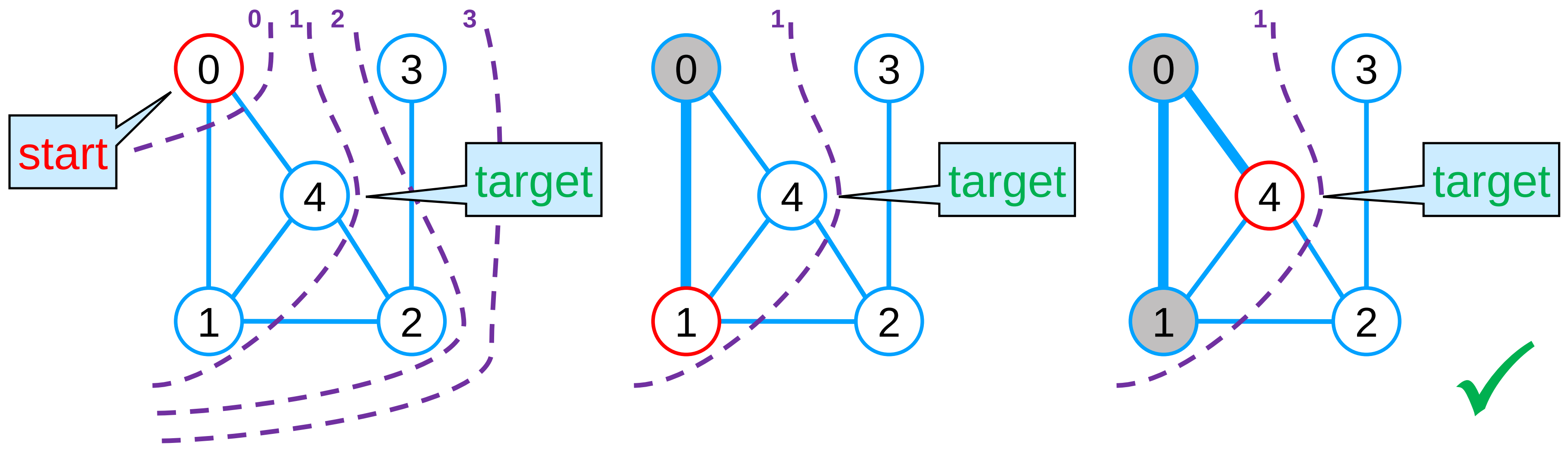
  - When we examine 0, we need to remember that we will have to examine 1 and 4 later
  - When we examine 1, we need to remember we may have to examine 2 later
    - but first we need to look at 4

- We need a **todo list**

Breadth-first Search

- We need a **work list**

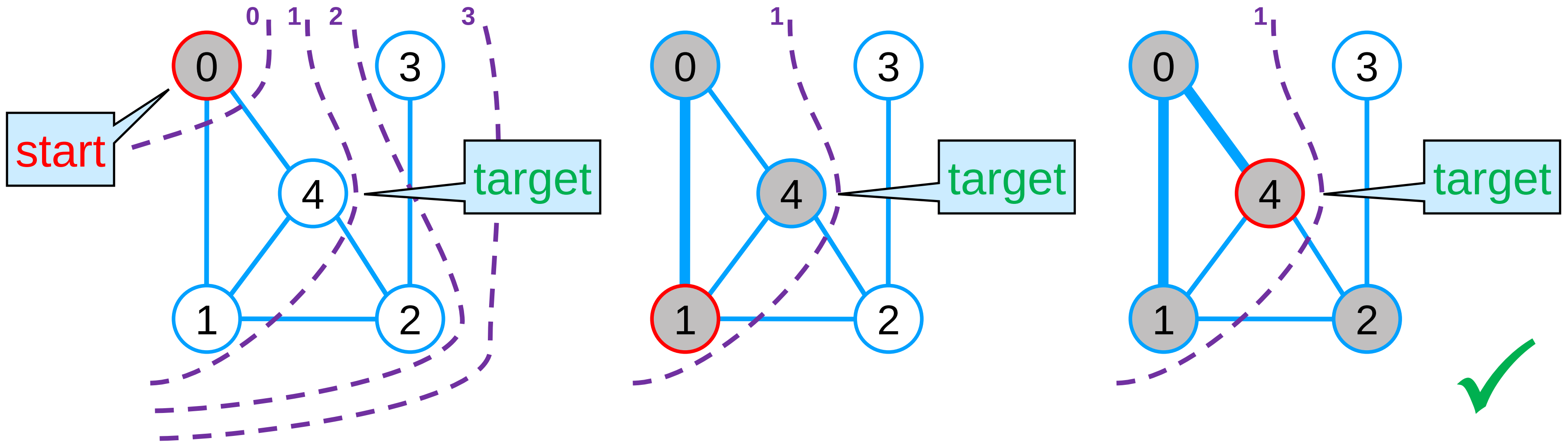
That's what we called todo lists



- We need to traverse the graph level by level
  - finish examining the current level before starting the next level
  - we need to retrieve the vertices inserted the longest time ago
- This work list must be a **queue**
  - older nodes need to be visited before newer nodes

Breadth-first Search

- *This work list must be a* ***queue***



  - start with 0 in the queue
  - at each step, retrieve the next vertex to examine

next	**target**	**queue**	**marked**
	4	0	0
0	4	1, 4	0, 1, 4
1	4	4, 2	0, 1, 4, 2
4	4	✓	

- We mark the vertices so we don't put them in the queue twice
  - either because we examined them already
  - or because they are already in the queue and will be examined later

Implementing BFS

- We need
  - a **queue** where to store the vertices to examine next
  - a **mark array** where to track the vertices we know about
    - either already examined or queued up to be examined

Implementing BFS

- For as long as there are vertices still to be processed
  - retrieve the vertex v inserted in the queue the longest time ago
    - if v is **target**, we are done — there is a path
  - examine each neighbor w of v
    - if w is unmarked add it to the queue and mark it
    - otherwise ignore w – it was already queued up for processing

- if the queue is empty
  - there are no vertices left to process
  - and we have not found a path
  - we are done — there is no path

Implementing BFS – I

Initial setup

```
bool bfs(graph_t G, vertex start, vertex target) {
  REQUIRES(G != NULL);
  REQUIRES(start < graph_size(G) && target < graph_size(G));

  if (start == target) return true;

  // mark is an array containing only start
  bool *mark = xcalloc(graph_size(G), sizeof(bool));
  mark[start] = true;

  // Q initially is a queue containing only start
  queue_t Q = queue_new();
  enq(Q, start);

  …
```

If **start** is **target**, there is a path ✓

calloc initializes every vertex as unmarked

but we want **start** to be marked

Initially only **start** is in the queue

Implementing BFS – II

Traversing the graph

```
 …
while (!queue_empty(Q)) {
   vertex v = deq(Q);
   printf("    Visiting %u\n", v);
   if (v == target) {
     queue_free(Q);
     free(mark);
     return true;
   }
   neighbors_t nbors = graph_get_neighbors(G, v);
   while (graph_hasmore_neighbors(nbors)) {
     vertex w = graph_next_neighbor(nbors);
     if (!mark[w]) {
       mark[w] = true;
       enq(Q, w);
     }
   }
   graph_free_neighbors(nbors);
 }
 …
```

for as long as there are vertices to process

v is the next vertex to process

If v is **target**, there is a path ✓

clean up before returning

examine each neighbor w of v

if w is unmarked mark it and add it to the queue

we are done with the neighbors of v

Implementing BFS – III

Giving up

```
 …
while (!queue_empty(Q)) {
 …
 }
 ASSERT(queue_empty(Q));
 queue_free(Q);
 free(mark);
 return false;
}
```

If there are no more vertices to process

there is no path ✘

clean up before returning

Implementing BFS

- Here’s the overall code

```
bool bfs(graph_t G, vertex start, vertex target) {
  REQUIRES(G != NULL);
  REQUIRES(start < graph_size(G) && target < graph_size(G));

  if (start == target) return true;

  // mark is an array containing only start
  bool *mark = xcalloc(graph_size(G), sizeof(bool));
  mark[start] = true;

  // Q is a queue containing only start initially
  queue_t Q = queue_new();
  enq(Q, start);

  while (!queue_empty(Q)) {
    // v is the next vertex to process
    vertex v = deq(Q);
    printf("    Visiting %u\n", v);
    if (v == target) {   // if v is target return true
      queue_free(Q);
      free(mark);
      return true;
    }
    //   for every neighbor w of v
    neighbors_t nbors = graph_get_neighbors(G, v);
    while (graph_hasmore_neighbors(nbors)) {
      vertex w = graph_next_neighbor(nbors);
      if (!mark[w]) {        // if w is not already marked
        mark[w] = true;        // mark it
        enq(Q, w);             // enqueue it onto the queue
      }
    }
    graph_free_neighbors(nbors);
  }
  ASSERT(queue_empty(Q));
  queue_free(Q);
  free(mark);
  return false;
}
```

Implementing BFS

- This code is **iterative**
  - DFS earlier was recursive
- The code structure is the same as graph_print

```
void graph_print(graph_t G) {
  for (vertex v = 0; v < graph_size(G); v++) {
    printf("Vertices connected to %u: ", v);
    neighbors_t nbors = graph_get_neighbors(G, v);
    while (graph_hasmore_neighbors(nbors)) {
      vertex w = graph_next_neighbor(nbors);
      printf(" %u,", w);
    }
    graph_free_neighbors(nbors);
    printf("\n");
  }
}
```

```
bool bfs(graph_t G, vertex start, vertex target) {
  REQUIRES(G != NULL);
  REQUIRES(start < graph_size(G) && target < graph_size(G));

  if (start == target) return true;

  // mark is an array containing only start
  bool *mark = xcalloc(graph_size(G), sizeof(bool));
  mark[start] = true;

  // Q is a queue containing only start initially
  queue_t Q = queue_new();
  enq(Q, start);

  while (!queue_empty(Q)) {
    // v is the next vertex to process
    vertex v = deq(Q);
    printf("    Visiting %u\n", v);
    if (v == target) {   // if v is target return true
      queue_free(Q);
      free(mark);
      return true;
    }
    //   for every neighbor w of v
    neighbors_t nbors = graph_get_neighbors(G, v);
    while (graph_hasmore_neighbors(nbors)) {
      vertex w = graph_next_neighbor(nbors);
      if (!mark[w]) {        // if w is not already marked
        mark[w] = true;       // mark it
        enq(Q, w);            // enqueue it onto the queue
      }
    }
    graph_free_neighbors(nbors);
  }
  ASSERT(queue_empty(Q));
  queue_free(Q);
  free(mark);
  return false;
}
```

Implementing BFS

- The code structure is the same as graph_print
  - except that we return early if we find a path
- The complexity of bfs is
  - O(v + e) with adjacency lists
  - O(v^2) with adjacency matrices
- same as dfs

```
bool bfs(graph_t G, vertex start, vertex target) {
  REQUIRES(G != NULL);
  REQUIRES(start < graph_size(G) && target < graph_size(G));

  if (start == target) return true;                                O(1)

  // mark is an array containing only start
  bool *mark = xcalloc(graph_size(G), sizeof(bool));               O(v)
  mark[start] = true;                                              O(1)

  // Q is a queue containing only start initially
  queue_t Q = queue_new();                                         O(1)
  enq(Q, start);                                                   O(1)

  while (!queue_empty(Q)) {                                        v times
    // v is the next vertex to process                             }
    vertex v = deq(Q);                                             }
    printf("    Visiting %u\n", v);                                }
    if (v == target) {   // if v is target return true             }
      queue_free(Q);                                               } O(1)
      free(mark);                                                  }
      return true;                                                 }
    }                                                              }
    //   for every neighbor w of v                                 }
    neighbors_t nbors = graph_get_neighbors(G, v);                 }
    while (graph_hasmore_neighbors(nbors)) {                     }
      vertex w = graph_next_neighbor(nbors);                     }
      if (!mark[w]) {       // if w is not already marked        }
        mark[w] = true;       // mark it                         } O(e)
        enq(Q, w);            // enqueue it onto the queue       } altogether
      }                                                          }
    }                                                            }
    graph_free_neighbors(nbors);                                     O(1)
  }
  ASSERT(queue_empty(Q));                                        }
  queue_free(Q);                                                 } O(1)
  free(mark);                                                    }
  return false;                                                  }
}
```

Correctness

- bfs is **correct** if it returns
  - ***true*** when there is a path from **start** to **target**
  - ***false*** when there is no path from **start** to **target**
- It returns in three places

```
bool bfs(graph_t G, vertex start, vertex target) {
  REQUIRES(G != NULL);
  REQUIRES(start < graph_size(G) && target < graph_size(G));

  if (start == target) return true;

  // mark is an array containing only start
  bool *mark = xcalloc(graph_size(G), sizeof(bool));
  mark[start] = true;

  // Q is a queue containing only start initially
  queue_t Q = queue_new();
  enq(Q, start);

  while (!queue_empty(Q)) {
    // v is the next vertex to process
    vertex v = deq(Q);
    printf("    Visiting %u\n", v);
    if (v == target) {   // if v is target return true
      queue_free(Q);
      free(mark);
      return true;
    }
    //   for every neighbor w of v
    neighbors_t nbors = graph_get_neighbors(G, v);
    while (graph_hasmore_neighbors(nbors)) {
      vertex w = graph_next_neighbor(nbors);
      if (!mark[w]) {       // if w is not already marked
        mark[w] = true;       // mark it
        enq(Q, w);            // enqueue it onto the queue
      }
    }
    graph_free_neighbors(nbors);
  }
  ASSERT(queue_empty(Q));
  queue_free(Q);
  free(mark);
  return false;
}
```

Correctness – I

- bfs is correct if it returns
  - ***true*** when there is a path from **start** to **target**
- We need to show that there is a path in this case
  - recall the definition

There is a path from start to target if
- start == target, or
- there is an edge from start to some vertex v and there is a path from v to target

  - we are in the first case

```
bool bfs(graph_t G, vertex start, vertex target) {
  REQUIRES(G != NULL);
  REQUIRES(start < graph_size(G) && target < graph_size(G));

  if (start == target) return true;

  // mark is an array containing only start
  bool *mark = xcalloc(graph_size(G), sizeof(bool));
  mark[start] = true;

  // Q is a queue containing only start initially
  queue_t Q = queue_new();
  enq(Q, start);

  while (!queue_empty(Q)) {
    // v is the next vertex to process
    vertex v = deq(Q);
    printf("    Visiting %u\n", v);
    if (v == target) {   // if v is target return true
      queue_free(Q);
      free(mark);
      return true;
    }
    //   for every neighbor w of v
    neighbors_t nbors = graph_get_neighbors(G, v);
    while (graph_hasmore_neighbors(nbors)) {
      vertex w = graph_next_neighbor(nbors);
      if (!mark[w]) {       // if w is not already marked
        mark[w] = true;       // mark it
        enq(Q, w);            // enqueue it onto the queue
      }
    }
    graph_free_neighbors(nbors);
  }
  ASSERT(queue_empty(Q));
  queue_free(Q);
  free(mark);
  return false;
}
```

Correctness – II

- bfs is correct if it returns
  - ***true*** when there is a path from **start** to **target**
- We need to show that there is a path

There is a path from start to target if

- start == target, or
- there is an edge from start to some vertex v and there is a path from v to target

  - but we have nowhere to point to

```
bool bfs(graph_t G, vertex start, vertex target) {
  REQUIRES(G != NULL);
  REQUIRES(start < graph_size(G) && target < graph_size(G));

  if (start == target) return true;

  // mark is an array containing only start
  bool *mark = xcalloc(graph_size(G), sizeof(bool));
  mark[start] = true;

  // Q is a queue containing only start initially
  queue_t Q = queue_new();
  enq(Q, start);

  while (!queue_empty(Q)) {
    // v is the next vertex to process
    vertex v = deq(Q);
    printf("    Visiting %u\n", v);
    if (v == target) {   // if v is target return true
      queue_free(Q);
      free(mark);
      return true;
    }
    //   for every neighbor w of v
    neighbors_t nbors = graph_get_neighbors(G, v);
    while (graph_hasmore_neighbors(nbors)) {
      vertex w = graph_next_neighbor(nbors);
      if (!mark[w]) {        // if w is not already marked
        mark[w] = true;      // mark it
        enq(Q, w);           // enqueue it onto the queue
      }
    }
    graph_free_neighbors(nbors);
  }
  ASSERT(queue_empty(Q));
  queue_free(Q);
  free(mark);
  return false;
}
```

Correctness – II

There is a path from start to target if
- start == target, or
- there is an edge from start to some vertex v and there is a path from v to target

We need to show there is a path
- *but we have nowhere to point to*

- We need **loop invariants**
  - What do we know about marked vertices?
    - ➢ there is a path from **start** to every marked vertex
  - What do we know about vertices in the queue?
    - ➢ every vertex in the queue is marked

```
bool bfs(graph_t G, vertex start, vertex target) {
  REQUIRES(G != NULL);
  REQUIRES(start < graph_size(G) && target < graph_size(G));

  if (start == target) return true;

  // mark is an array containing only start
  bool *mark = xcalloc(graph_size(G), sizeof(bool));
  mark[start] = true;

  // Q is a queue containing only start initially
  queue_t Q = queue_new();
  enq(Q, start);

  while (!queue_empty(Q)) {
    // v is the next vertex to process
    vertex v = deq(Q);
    printf("    Visiting %u\n", v);
    if (v == target) {   // if v is target return true
      queue_free(Q);
      free(mark);
      return true;
    }
    //   for every neighbor w of v
    neighbors_t nbors = graph_get_neighbors(G, v);
    while (graph_hasmore_neighbors(nbors)) {
      vertex w = graph_next_neighbor(nbors);
      if (!mark[w]) {       // if w is not already marked
        mark[w] = true;      // mark it
        enq(Q, w);           // enqueue it onto the queue
      }
    }
    graph_free_neighbors(nbors);
  }
  ASSERT(queue_empty(Q));
  queue_free(Q);
  free(mark);
  return false;
}
```

Correctness – II

- **Candidate** loop invariants
  - LI 1: there is a path from **start** to every marked vertex
  - LI 2: every vertex in the queue is marked

- INIT
  - LI 1:
    - initially only **start** is marked *by l.7*
    - there is a path from **start** to **start** *by def*
  - LI 2:
    - initially only **start** is in the queue *by l.10*
    - **start** is marked *by l.7*

✓

```
1.  bool bfs(graph_t G, vertex start, vertex target) {
2.    REQUIRES(G != NULL);
3.    REQUIRES(start < graph_size(G) && target < …);

4.    if (start == target) return true;

5.    // mark is an array containing only start
6.    bool *mark = xcalloc(graph_size(G), sizeof(bool));
7.    mark[start] = true;

8.    // Q is a queue containing only start initially
9.    queue_t Q = queue_new();
10.   enq(Q, start);

11.   while (!queue_empty(Q)) {
12.     // v is the next vertex to process
13.     vertex v = deq(Q);
14.     printf("    Visiting %u\n", v);
15.     if (v == target) {   // if v is target return true
16.       queue_free(Q);
17.       free(mark);
18.       return true;
19.     }
20.     //   for every neighbor w of v
21.     neighbors_t nbors = graph_get_neighbors(G, v);
22.     while (graph_hasmore_neighbors(nbors)) {
23.       vertex w = graph_next_neighbor(nbors);
24.       if (!mark[w]) {        // if w is not already marked
25.         mark[w] = true;       // mark it
26.         enq(Q, w);            // enqueue it onto the queue
27.       }
28.     }
29.     graph_free_neighbors(nbors);
30.   }
31.   ASSERT(queue_empty(Q));
32.   queue_free(Q);
33.   free(mark);
34.   return false;
35. }
```

Correctness – II

- **Candidate** loop invariants
  - LI 1: there is a path from **start** to every marked vertex
  - LI 2: every vertex in the queue is marked

- PRES
  - LI 1:

➢ v is in the queue	*by l.13*
➢ it is marked	*by LI 2*
➢ there is a path from **start** to v	*by LI 1*
➢ w is a neighbor of v	*by l.23*
➢ there is a path from **start** to w	*by def*
➢ w gets marked	*by l.25*

  - LI 2:

➢ w gets added to the queue	*by l.26*

```
1.  bool bfs(graph_t G, vertex start, vertex target) {
2.    REQUIRES(G != NULL);
3.    REQUIRES(start < graph_size(G) && target < …);

4.    if (start == target) return true;

5.    // mark is an array containing only start
6.    bool *mark = xcalloc(graph_size(G), sizeof(bool));
7.    mark[start] = true;

8.    // Q is a queue containing only start initially
9.    queue_t Q = queue_new();
10.   enq(Q, start);

11.   while (!queue_empty(Q)) {
12.     // v is the next vertex to process
13.     vertex v = deq(Q);
14.     printf("    Visiting %u\n", v);
15.     if (v == target) {   // if v is target return true
16.       queue_free(Q);
17.       free(mark);
18.       return true;
19.     }
20.     //   for every neighbor w of v
21.     neighbors_t nbors = graph_get_neighbors(G, v);
22.     while (graph_hasmore_neighbors(nbors)) {
23.       vertex w = graph_next_neighbor(nbors);
24.       if (!mark[w]) {        // if w is not already marked
25.         mark[w] = true;       // mark it
26.         enq(Q, w);            // enqueue it onto the queue
27.       }
28.     }
29.     graph_free_neighbors(nbors);
30.   }
31.   ASSERT(queue_empty(Q));
32.   queue_free(Q);
33.   free(mark);
34.   return false;
35. }
```

Correctness – II

There is a path from start to target if
- start == target, or
- there is an edge from start to some vertex v and there is a path from v to target

- We can now prove the correctness of this case
  - v was in the queue *by l.15*
  - so, v is marked *by LI 2*
  - there is a path from **start** to v *by LI 1*
  - v == **target** *by l.17*
  - there is a path from **start** to **target** *by def*

✓

```
1.  bool bfs(graph_t G, vertex start, vertex target) {
2.    REQUIRES(G != NULL);
3.    REQUIRES(start < graph_size(G) && target <  …);

4.    if (start == target) return true;

5.    // mark is an array containing only start
6.    bool *mark = xcalloc(graph_size(G), sizeof(bool));
7.    mark[start] = true;

8.    // Q is a queue containing only start initially
9.    queue_t Q = queue_new();
10.   enq(Q, start);

11.   while (!queue_empty(Q)) {
12.     //@ LI 1: there is a path from start to every marked vertex
13.     //@ LI 2: every vertex in the queue is marked

14.     // v is the next vertex to process
15.     vertex v = deq(Q);
16.     printf("    Visiting %u\n", v);
17.     if (v == target) {   // if v is target return true
18.       queue_free(Q);
19.       free(mark);
20.       return true;
21.     }
22.     //   for every neighbor w of v
23.     neighbors_t nbors = graph_get_neighbors(G, v);
24.     while (graph_hasmore_neighbors(nbors)) {
25.       vertex w = graph_next_neighbor(nbors);
26.       if (!mark[w]) {        // if w is not already marked
27.         mark[w] = true;       // mark it
28.         enq(Q, w);            // enqueue it onto the queue
29.       }
30.     }
31.     graph_free_neighbors(nbors);
32.   }
33.   ASSERT(queue_empty(Q));
34.   queue_free(Q);
35.   free(mark);
36.   return false;
37. }
```

Correctness – III

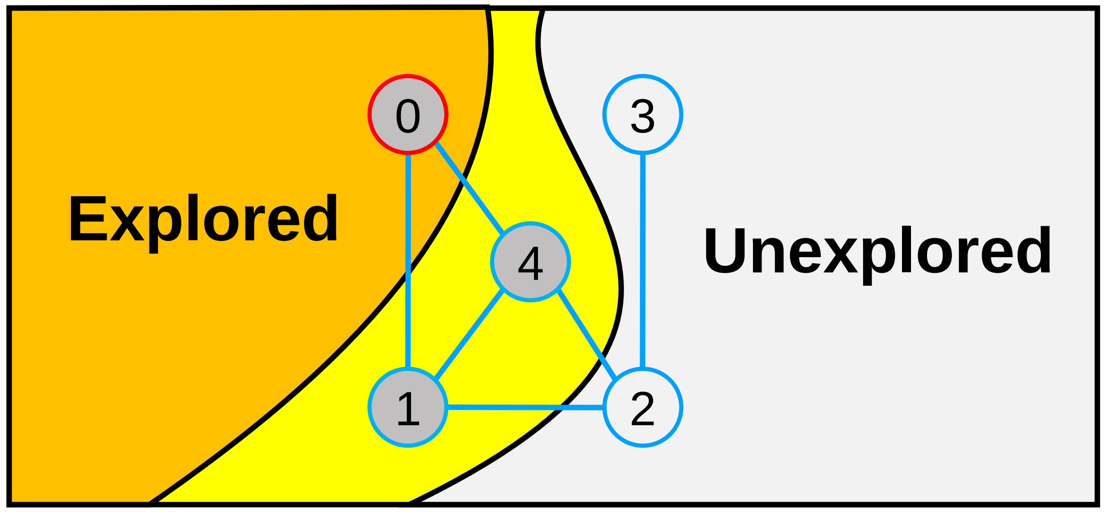
- bfs is correct if it returns
  - ***false*** when there is ***no*** path from **start** to **target**
- LI 1 and LI 2 are insufficient
- We need more insight into the way bfs works

```
bool bfs(graph_t G, vertex start, vertex target) {
  REQUIRES(G != NULL);
  REQUIRES(start < graph_size(G) && target < graph_size(G));

  if (start == target) return true;

  // mark is an array containing only start
  bool *mark = xcalloc(graph_size(G), sizeof(bool));
  mark[start] = true;

  // Q is a queue containing only start initially
  queue_t Q = queue_new();
  enq(Q, start);

  while (!queue_empty(Q)) {
    //@ LI 1: there is a path from start to every marked vertex
    //@ LI 2: every vertex in the queue is marked

    // v is the next vertex to process
    vertex v = deq(Q);
    printf("    Visiting %u\n", v);
    if (v == target) {   // if v is target return true
      queue_free(Q);
      free(mark);
      return true;
    }
    //   for every neighbor w of v
    neighbors_t nbors = graph_get_neighbors(G, v);
    while (graph_hasmore_neighbors(nbors)) {
      vertex w = graph_next_neighbor(nbors);
      if (!mark[w]) {         // if w is not already marked
        mark[w] = true;       // mark it
        enq(Q, w);            // enqueue it onto the queue
      }
    }
    graph_free_neighbors(nbors);
  }
  ASSERT(queue_empty(Q));
  queue_free(Q);
  free(mark);
  return false;
}
```

Correctness – III

- What do the elements of the queue represent?

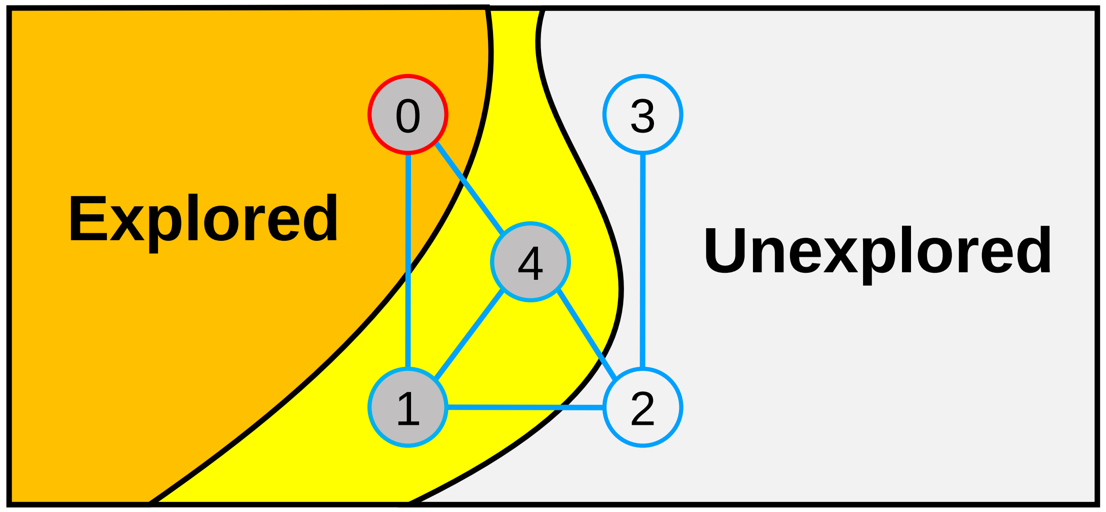
next	target	queue	marked
	4	0	0
0	4	1, 4	0, 1, 4
1	4	4, 2	0, 1, 4, 2
4	4	*Success!*	



  - The **frontier** of the search

```
bool bfs(graph_t G, vertex start, vertex target) {
  REQUIRES(G != NULL);
  REQUIRES(start < graph_size(G) && target < graph_size(G));

  if (start == target) return true;

  // mark is an array containing only start
  bool *mark = xcalloc(graph_size(G), sizeof(bool));
  mark[start] = true;

  // Q is a queue containing only start initially
  queue_t Q = queue_new();
  enq(Q, start);

  while (!queue_empty(Q)) {
    //@ LI 1: there is a path from start to every marked vertex
    //@ LI 2: every vertex in the queue is marked

    // v is the next vertex to process
    vertex v = deq(Q);
    printf("    Visiting %u\n", v);
    if (v == target) {   // if v is target return true
      queue_free(Q);
      free(mark);
      return true;
    }
    //   for every neighbor w of v
    neighbors_t nbors = graph_get_neighbors(G, v);
    while (graph_hasmore_neighbors(nbors)) {
      vertex w = graph_next_neighbor(nbors);
      if (!mark[w]) {        // if w is not already marked
        mark[w] = true;       // mark it
        enq(Q, w);            // enqueue it onto the queue
      }
    }
    graph_free_neighbors(nbors);
  }
  ASSERT(queue_empty(Q));
  queue_free(Q);
  free(mark);
  return false;
}
```

Correctness – III



- All vertices behind the frontier are marked
  - they have been explored
- All vertices beyond the frontier are unmarked
  - they are still unexplored

- Every path from **start** to **target** goes through the frontier

This is a new loop invariant

```
bool bfs(graph_t G, vertex start, vertex target) {
  REQUIRES(G != NULL);
  REQUIRES(start < graph_size(G) && target < graph_size(G));

  if (start == target) return true;

  // mark is an array containing only start
  bool *mark = xcalloc(graph_size(G), sizeof(bool));
  mark[start] = true;

  // Q is a queue containing only start initially
  queue_t Q = queue_new();
  enq(Q, start);

  while (!queue_empty(Q)) {
    //@ LI 1: there is a path from start to every marked vertex
    //@ LI 2: every vertex in the queue is marked

    // v is the next vertex to process
    vertex v = deq(Q);
    printf("    Visiting %u\n", v);
    if (v == target) {    // if v is target return true
      queue_free(Q);
      free(mark);
      return true;
    }
    // for every neighbor w of v
    neighbors_t nbors = graph_get_neighbors(G, v);
    while (graph_hasmore_neighbors(nbors)) {
      vertex w = graph_next_neighbor(nbors);
      if (!mark[w]) {         // if w is not already marked
        mark[w] = true;       // mark it
        enq(Q, w);            // enqueue it onto the queue
      }
    }
    graph_free_neighbors(nbors);
  }
  ASSERT(queue_empty(Q));
  queue_free(Q);
  free(mark);
  return false;
}
```

Correctness – III

- Every path from **start** to **target** goes through the frontier
- When we finally return,
  1. every path from **start** to **target** goes through the frontier
     - ➢ LI 3 hold
  2. the frontier is empty
     - ➢ negation of the loop guard
  - ○ therefore there can't be a path from **start** to **target**
    - ➢ this is the only way (1) can hold
- bfs is correct ✓

```
bool bfs(graph_t G, vertex start, vertex target) {
  REQUIRES(G != NULL);
  REQUIRES(start < graph_size(G) && target < graph_size(G));

  if (start == target) return true;

  // mark is an array containing only start
  bool *mark = xcalloc(graph_size(G), sizeof(bool));
  mark[start] = true;

  // Q is a queue containing only start initially
  queue_t Q = queue_new();
  enq(Q, start);

  while (!queue_empty(Q)) {
    //@ LI 1: there is a path from start to every marked vertex
    //@ LI 2: every vertex in the queue is marked
    //@ LI 3: every path from start to target goes through Q

    // v is the next vertex to process
    vertex v = deq(Q);
    printf("    Visiting %u\n", v);
    if (v == target) {   // if v is target return true
      queue_free(Q);
      free(mark);
      return true;
    }
    //   for every neighbor w of v
    neighbors_t nbors = graph_get_neighbors(G, v);
    while (graph_hasmore_neighbors(nbors)) {
      vertex w = graph_next_neighbor(nbors);
      if (!mark[w]) {        // if w is not already marked
        mark[w] = true;       // mark it
        enq(Q, w);            // enqueue it onto the queue
      }
    }
    graph_free_neighbors(nbors);
  }
  ASSERT(queue_empty(Q));
  queue_free(Q);
  free(mark);
  return false;
}
```

Other Searches

Work List Choice

- bfs uses a **queue** as a work list
  - But the correctness proof does not depend on this
- We get a correct implementation of reachability whatever work list we use

```
bool bfs(graph_t G, vertex start, vertex target) {
  REQUIRES(G != NULL);
  REQUIRES(start < graph_size(G) && target < graph_size(G));

  if (start == target) return true;

  // mark is an array containing only start
  bool *mark = xcalloc(graph_size(G), sizeof(bool));
  mark[start] = true;

  // Q is a queue containing only start initially
  queue_t Q = queue_new();
  enq(Q, start);

  while (!queue_empty(Q)) {
    //@ LI 1: there is a path from start to every marked vertex
    //@ LI 2: every vertex in the queue is marked
    //@ LI 3: every path from start to target goes through Q

 // v is the next vertex to process
    vertex v = deq(Q);
    printf("    Visiting %u\n", v);
    if (w == target) {   // if w is target return true
      queue_free(Q);
      free(mark);
      return true;
    }
    //   for every neighbor w of v
    neighbors_t nbors = graph_get_neighbors(G, v);
    while (graph_hasmore_neighbors(nbors)) {
      vertex w = graph_next_neighbor(nbors);
      if (!mark[w]) {        // if w is not already marked
        mark[w] = true;       // mark it
        enq(Q, w);            // enqueue it onto the queue
      }
    }
    graph_free_neighbors(nbors);
  }
  ASSERT(queue_empty(Q));
  queue_free(Q);
  free(mark);
  return false;
}
```

Work List Choice

- *We get a correct implementation of reachability whatever work list we use*
- **Stack**?
  - The next vertex we process is the **last** we inserted
  - We get an iterative implementation of **depth-first search**
  - Complexity
    - O(v + e) with adjacency lists
    - O(v^2) with adjacency matrices

    because stack and queue operations have the same complexity

```
bool dfs(graph_t G, vertex start, vertex target) {
  REQUIRES(G != NULL);
  REQUIRES(start < graph_size(G) && target < graph_size(G));

  if (start == target) return true;

  // mark is an array containing only start
  bool *mark = xcalloc(graph_size(G), sizeof(bool));
  mark[start] = true;

  // S is a stack containing only start initially
  stack_t S = stack_new();
  push(S, start);

  while (!stack_empty(S)) {
    //@ LI 1: there is a path from start to every marked vertex
    //@ LI 2: every vertex in the stack is marked
    //@ LI 3: every path from start to target goes through S

 // v is the next vertex to process
    vertex v = pop(S);
    printf("    Visiting %u\n", v);
    if (w == target) {  // if w is target return true
      stack_free(S);
      free(mark);
      return true;
    }
    //   for every neighbor w of v
    neighbors_t nbors = graph_get_neighbors(G, v);
    while (graph_hasmore_neighbors(nbors)) {
      vertex w = graph_next_neighbor(nbors);
      if (!mark[w]) {       // if w is not already marked
        mark[w] = true;     // mark it
        push(S, w);         // push it onto the stack
      }
    }
    graph_free_neighbors(nbors);
  }
  ASSERT(stack_empty(S));
  stack_free(S);
  free(mark);
  return false;
}
```

Work List Choice

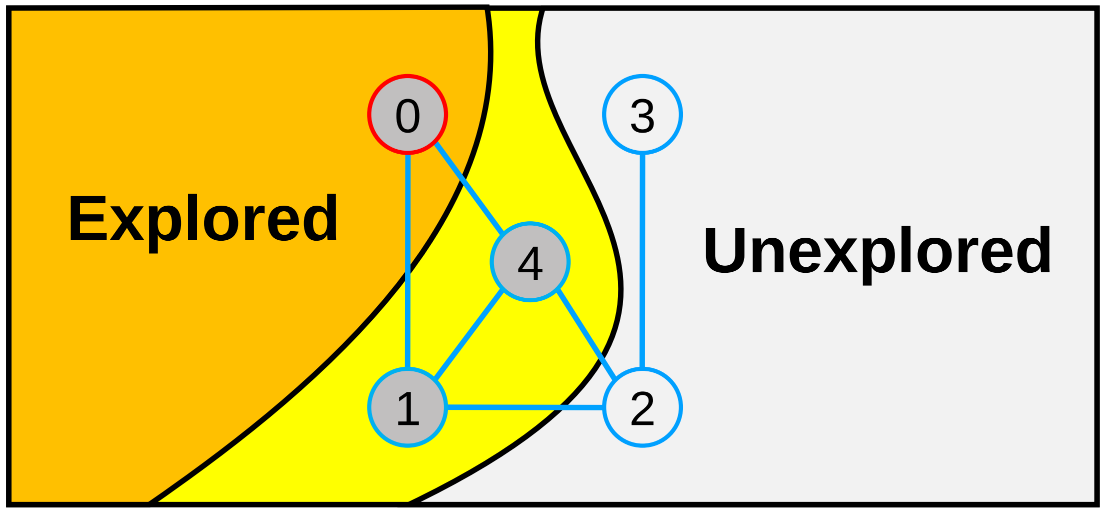
- *We get a correct implementation of reachability whatever work list we use*

- **Priority queues**?
  - The next vertex we process is the **most promising**
  - We get artificial intelligence search algorithms like A*

    pronounced "A star"
    - used in planning problems, game search, …
    - the priority function becomes a heuristic function that tells how good a vertex is
  - Complexity is higher because insertion and removal from a priority queue is not O(1)

Reachability

- All these graph reachability algorithms share the same basic idea



Explore the graph by expanding the frontier

- The difference is the kind of work list they use to remember the vertices to examine next
  - DFS: a stack
  - BFS: a queue
  - A*: a priority queue