CS 344: Design and Analysis of Computer Algorithms

(Spring 2022 — Sections 5,6,7,8)

Lecture 14: Disjoint Intervals, Introduction to Graphs, Graph Reductions

The Maximum Disjoint Intervals Problem

Maximum Disjoint Intervals Problem

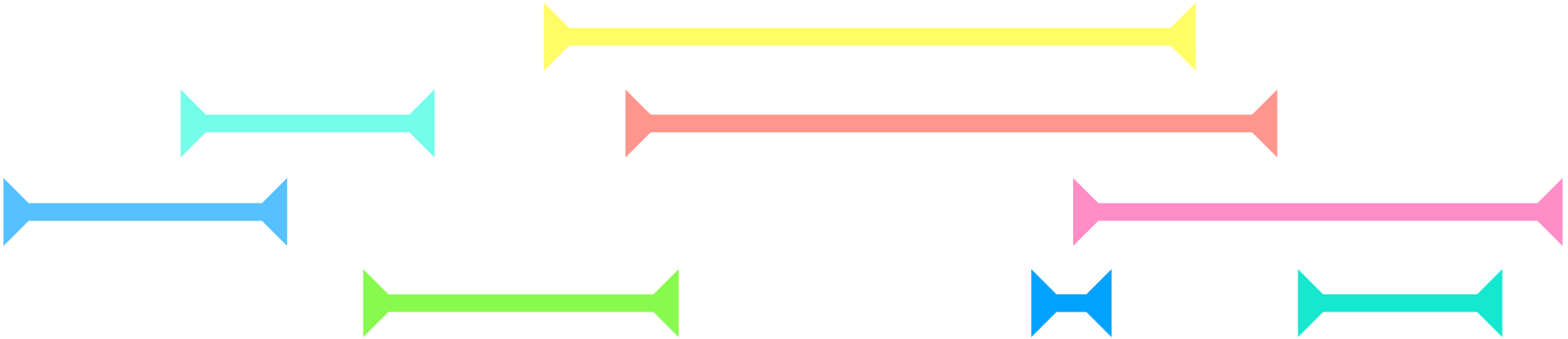
- **Input:**
  - A collection of n intervals specified by their endpoints as $[a_1, b_1], [a_2, b_2], \ldots, [a_n, b_n]$
  - Two intervals are disjoint if they do not share any points
    - E.g. [1,3] [4,5] are disjoint but [1,3] and [2,4] are not.
- **Output:**
  - Maximum number of intervals that are all disjoint from each other

Maximum Disjoint Intervals Problem

- **Example:**

Maximum Disjoint Intervals Problem

- **Example:**



Maximum Disjoint Intervals Problem

- **Example:**

Maximum Disjoint Intervals Problem

- **Example:**

- An optimal solution

Maximum Disjoint Intervals Problem

- **Example:**

- A non-optimal solution

Maximum Disjoint Intervals Problem

- **Example:**

- A wrong solution

Greedy Algorithm?

- A “Greedy” Observation:
  - The interval that finishes first can always be part of the solution
  - Why?

Greedy Algorithm

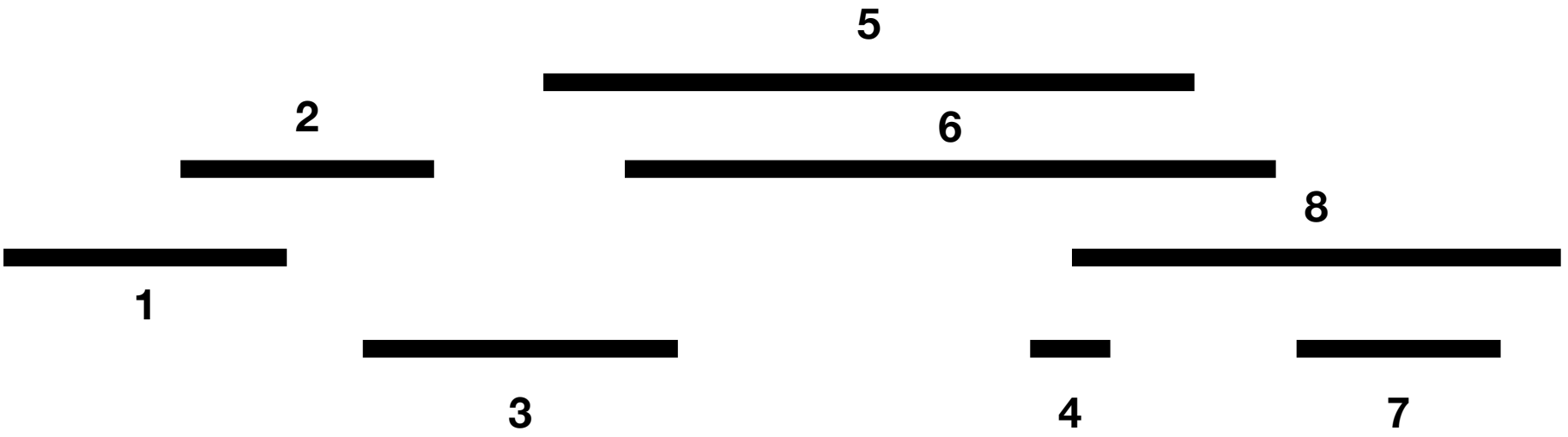
- The input is A[1],…,A[n]:
  - A[i].first gives the start point and A[i].second gives the finish
- Sort the intervals in A based on A[i].second in increasing order
- Pick A[1] in the solution and let p=1.
- For i=2 to n:
  - If A[i].first < A[p].second continue;
  - otherwise, add A[i] to the solution and update p = i.

Example

- **Example:**

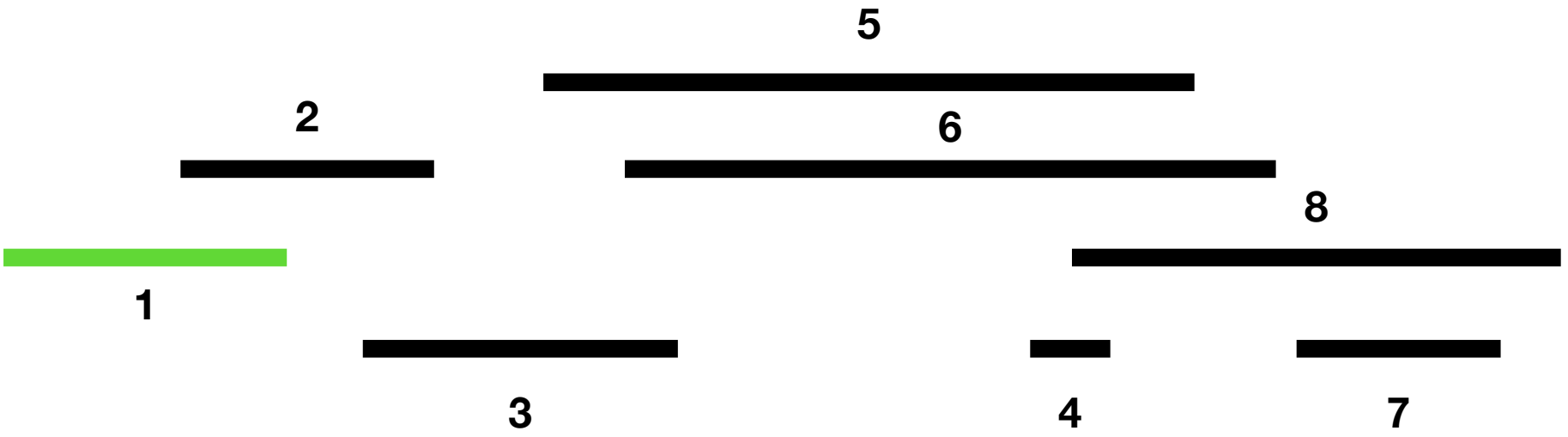
Example

- **Example:**



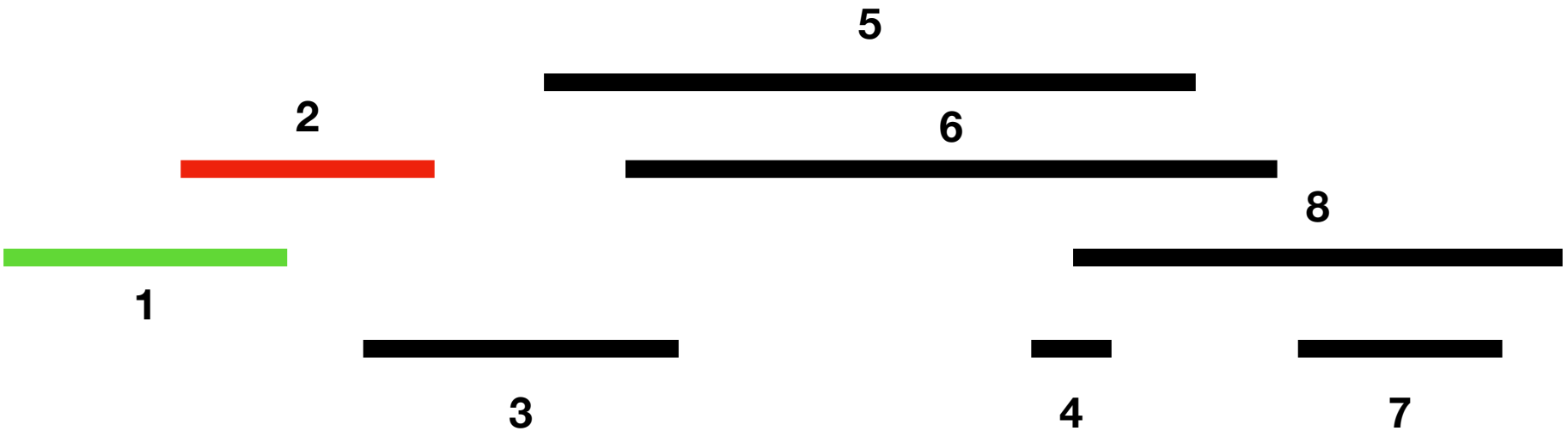
Example

- **Example:**



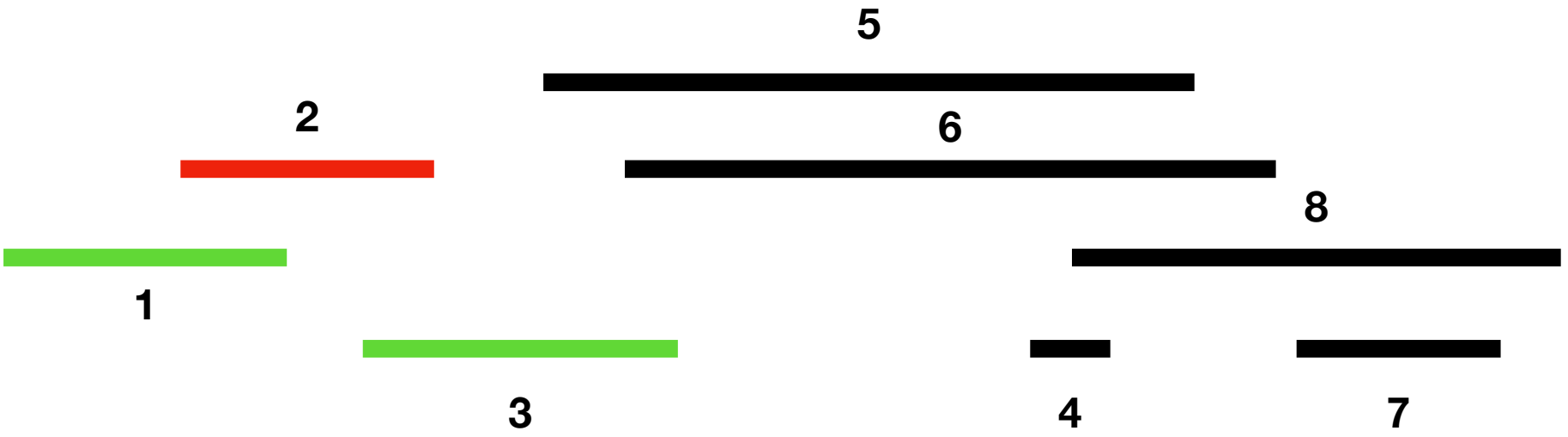
Example

- **Example:**



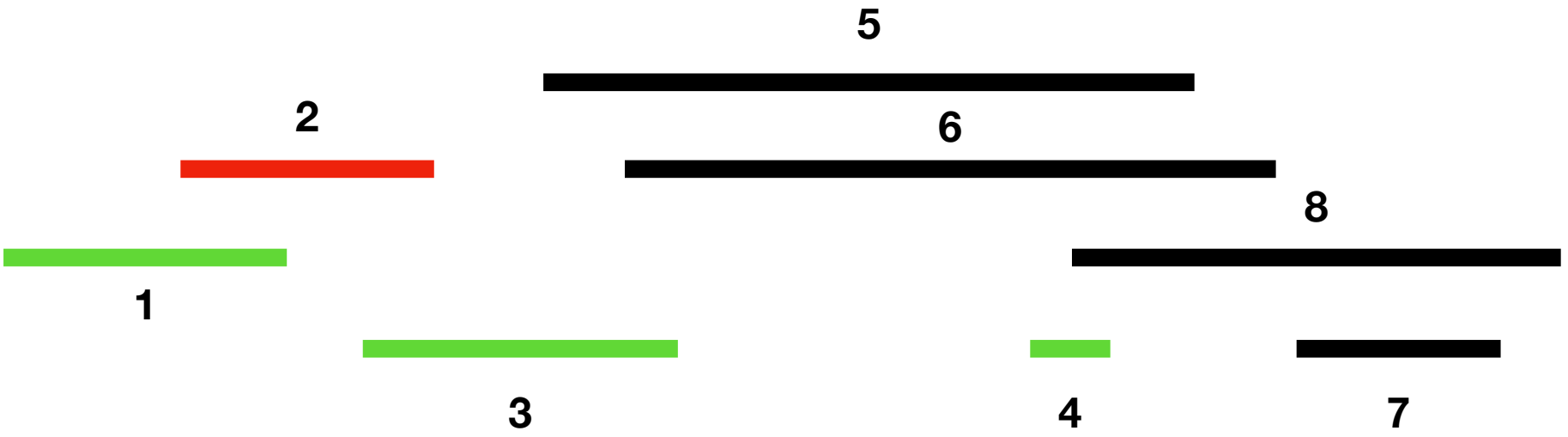
Example

- **Example:**



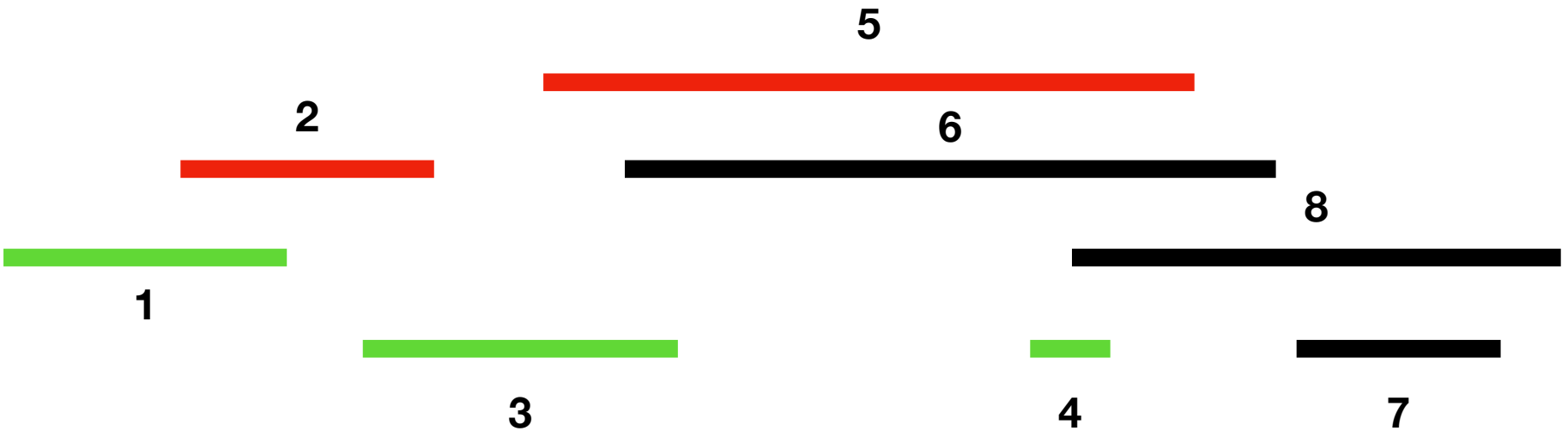
Example

- **Example:**



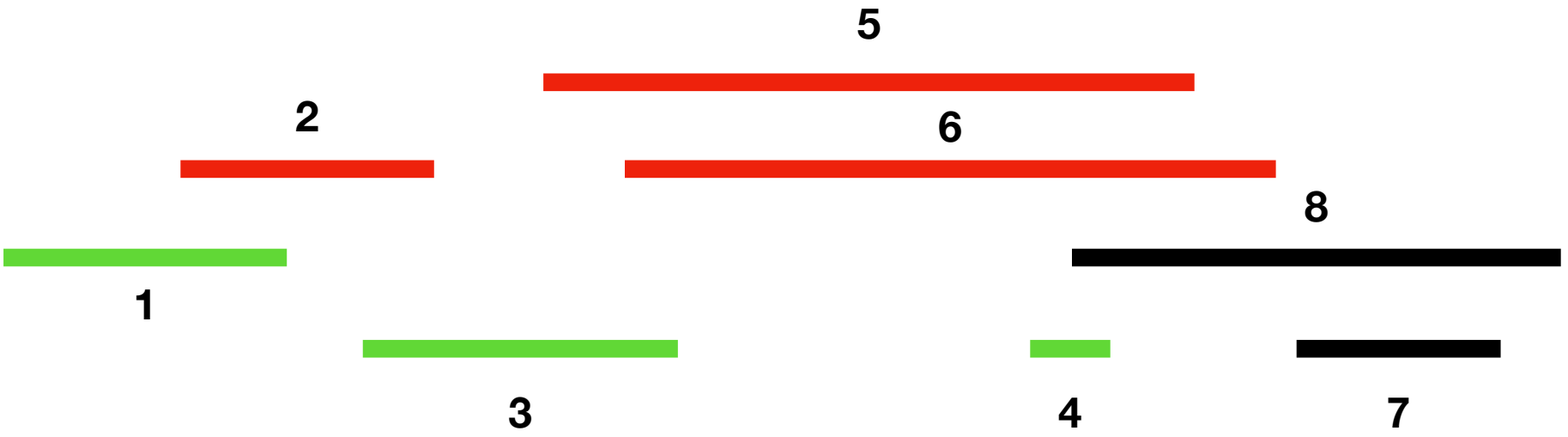
Example

- **Example:**



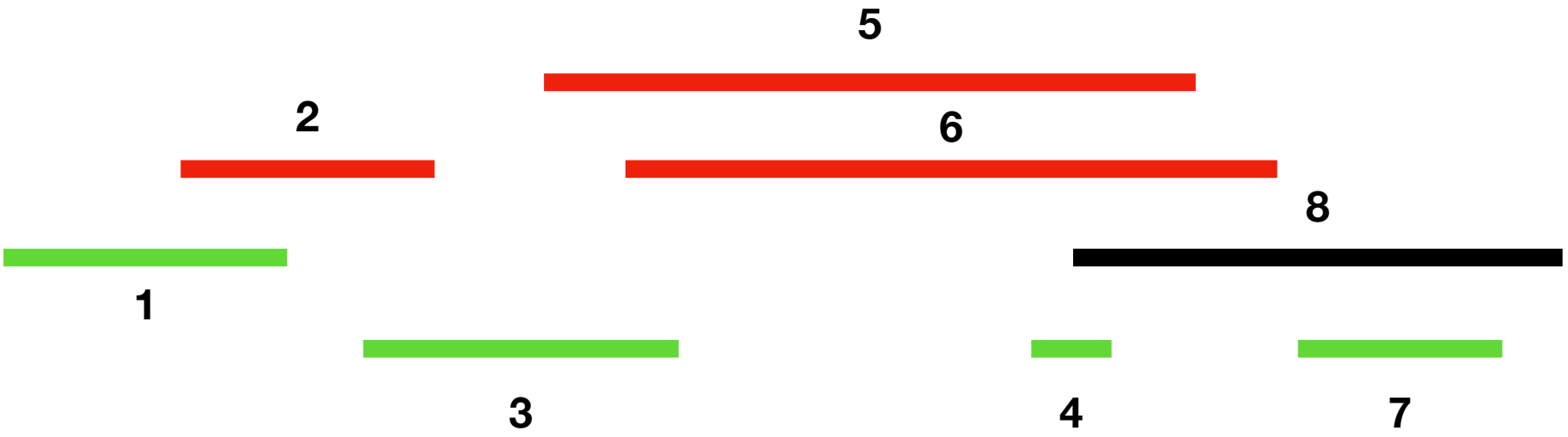
Example

- **Example:**



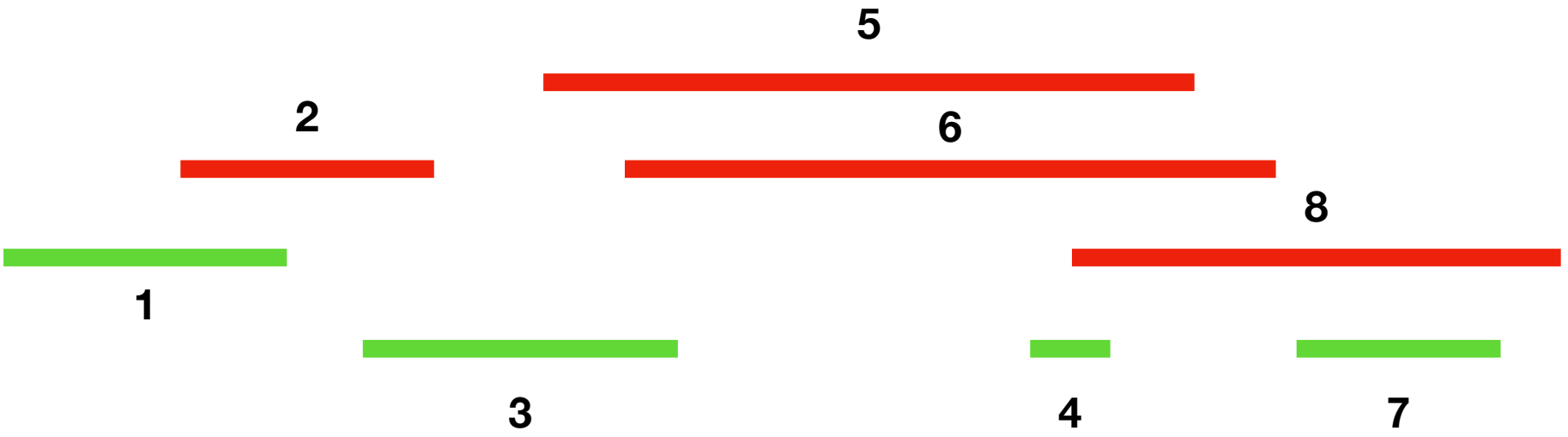
Example

- **Example:**



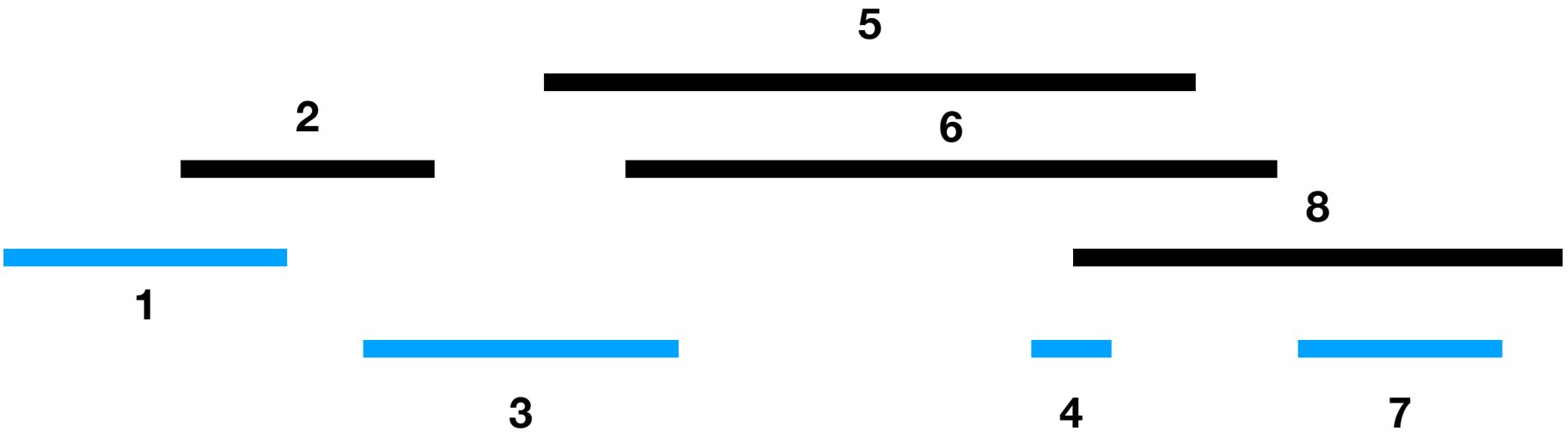
Example

- **Example:**



Example

- **Example:**



Proof of Correctness

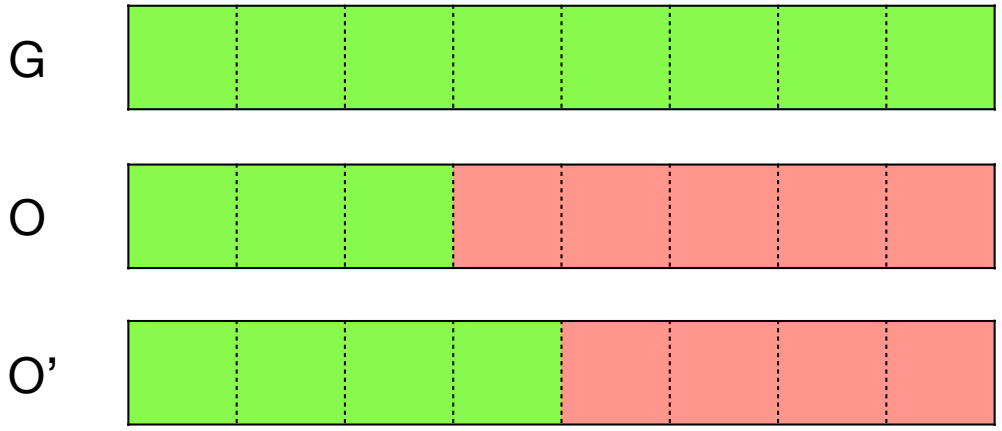
- Let $G = (g_1, g_2, \ldots, g_k)$ be the intervals output by greedy
- First, is G even a valid solution? (i.e., intervals are disjoint)

Proof of Correctness

- Let $G = (g_1, g_2, \ldots, g_k)$ be the intervals output by greedy
- Now, is G optimal?

Proof of Correctness

- Consider the intermediate solution
  $O' = (g_1 = o_1, g_2 = o_2, g_{i-1} = o_{i-1}, g_i, o_{i+1}, \ldots, o_\ell)$



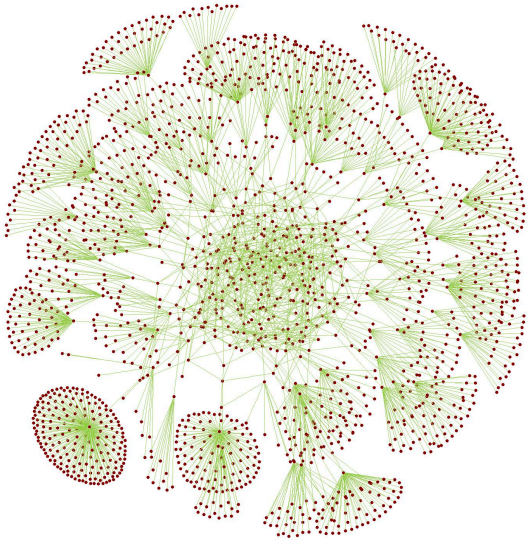
Greedy Algorithm

- The input is A[1],…,A[n]:
  - A[i].first gives the start point and A[i].second gives the finish
- Sort the intervals in A based on A[i].second in increasing order
- Pick A[1] in the solution and let p=1.
- For i=2 to n:
  - If A[i].first < A[p].second continue;
  - otherwise, add A[i] to the solution and update p = i.

Graphs and Graph Algorithms

Graphs are everywhere!

Webgraph



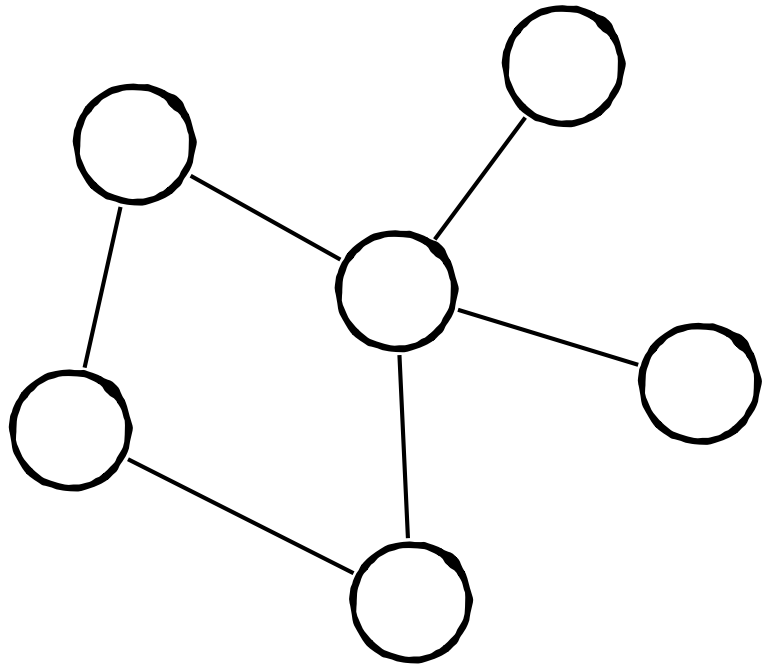
Social networks

Maps

Neurons and synapses

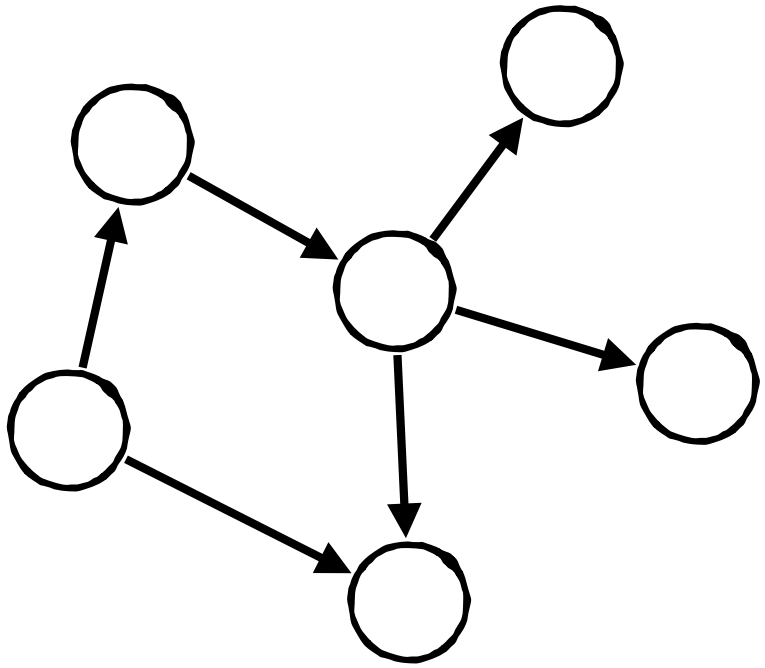
What is a graph?

- Undirected graph $G = (V, E)$
- Vertices are V
- Edges are E
  - connect pairs of vertices together



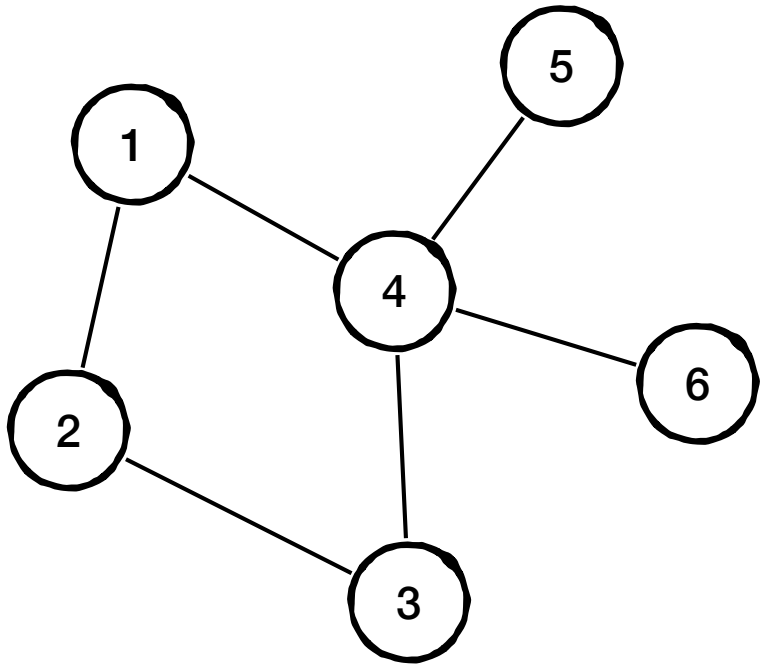
What is a graph?

- Directed graph $G = (V, E)$
- Vertices are V
- Edges are E
  - connect one vertex to another



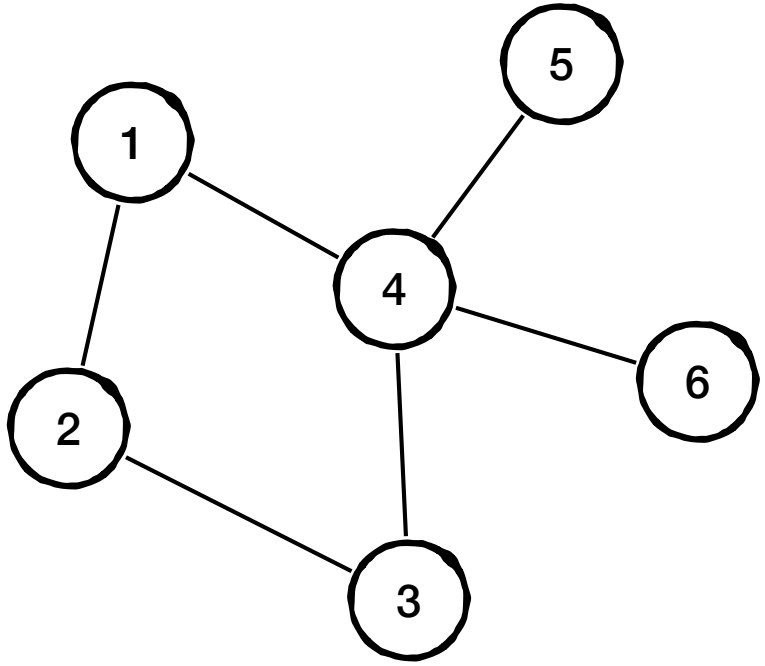
Basic Notation

- n: number of vertices
- m: number of edges



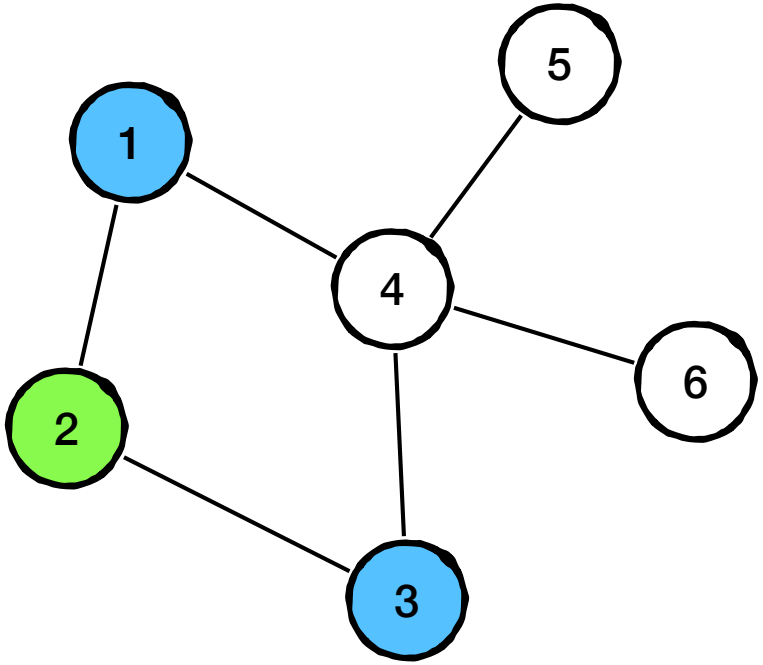
Basic Notation

- n: number of vertices
- m: number of edges
- N(v): neighbors of vertex v
  - vertices that v has an edge to
- deg(v): degree of v
  - number of neighbors of v



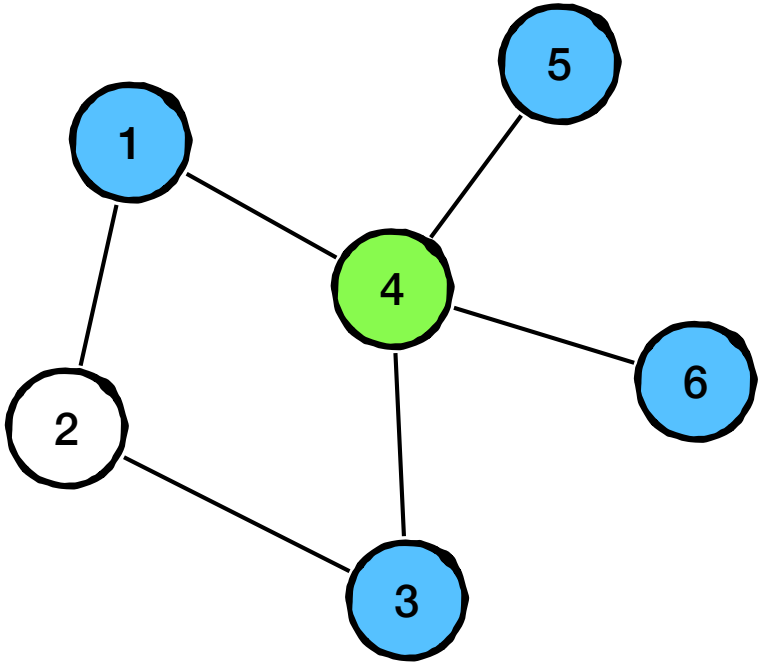
Basic Notation

- n: number of vertices
- m: number of edges
- N(v): neighbors of vertex v
  - vertices that v has an edge to
- deg(v): degree of v
  - number of neighbors of v



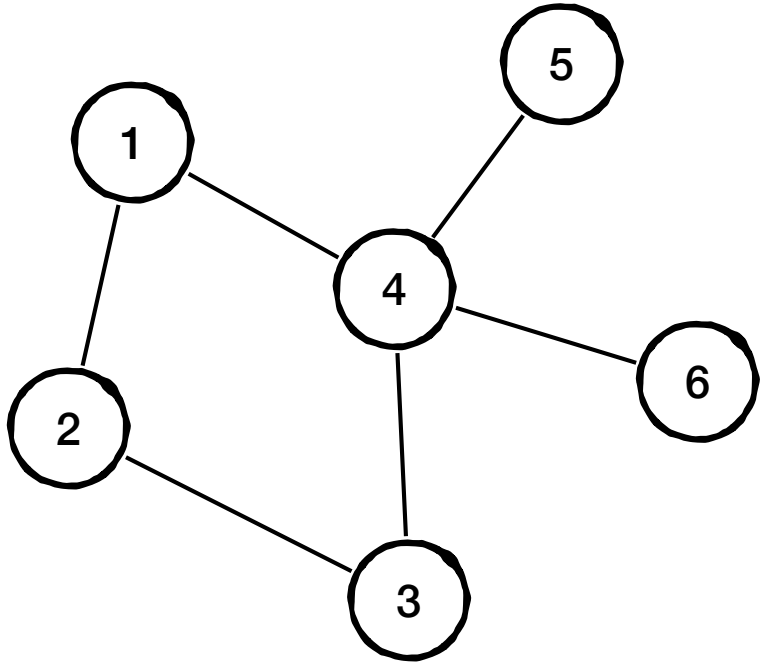
Basic Notation

- n: number of vertices
- m: number of edges
- N(v): neighbors of vertex v
  - vertices that v has an edge to
- deg(v): degree of v
  - number of neighbors of v



Basic Notation

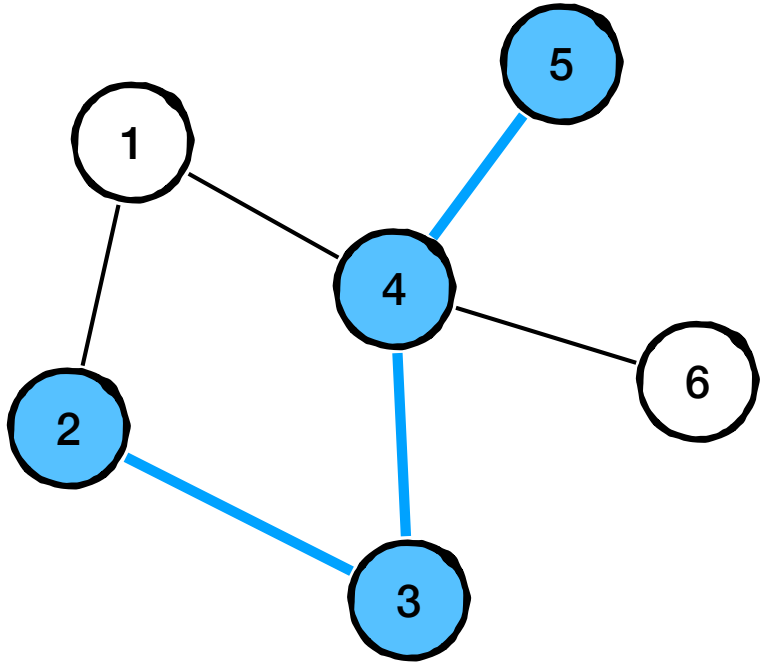
- n: number of vertices
- m: number of edges
- N(v): neighbors of vertex v
  - vertices that v has an edge to
- deg(v): degree of v
  - number of neighbors of v



- Path: a sequence of distinct vertices $v_1, v_2, \ldots, v_k$ where there is an edge from v_i to v_{i+1}

Basic Notation

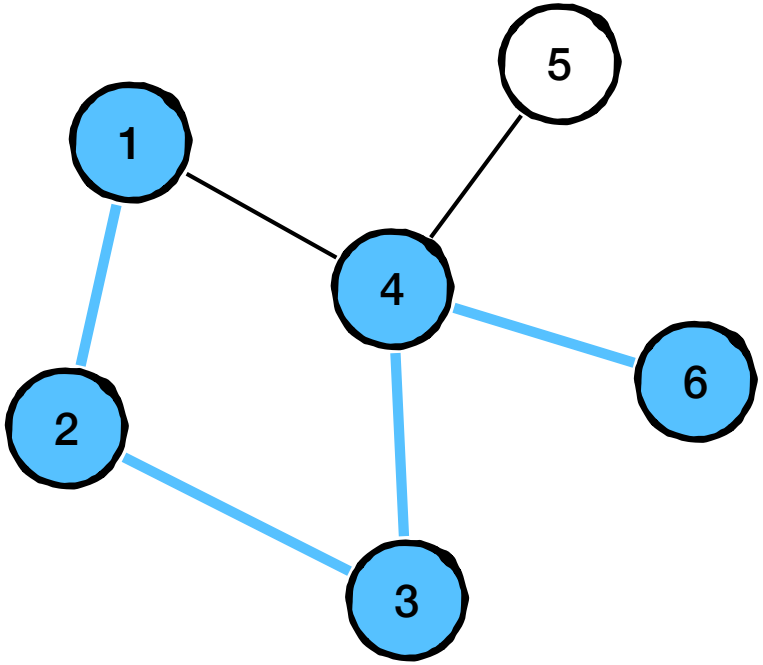
- n: number of vertices
- m: number of edges
- N(v): neighbors of vertex v
  - vertices that v has an edge to
- deg(v): degree of v
  - number of neighbors of v



- Path: a sequence of distinct vertices $v_1, v_2, \ldots, v_k$ where there is an edge from v_i to v_{i+1}

Basic Notation

- n: number of vertices
- m: number of edges
- N(v): neighbors of vertex v
  - vertices that v has an edge to
- deg(v): degree of v
  - number of neighbors of v



- Path: a sequence of distinct vertices $v_1, v_2, \ldots, v_k$ where there is an edge from v_i to v_{i+1}

Graph Algorithms

- Algorithms that solve various problems over graphs:
  - Graph search
  - Topological sorting
  - Minimum (weight) spanning tree
  - Shortest path
  - Network flow
  - …
- Graphs are excellent for modeling different problems

Graph Representation

Graph Representation

- Different ways to represent the input graph $G = (V, E)$
- Adjacency List Representation:
  - For each vertex $v \in V$, store the neighbors $N(v)$ of v in a list.
  - Store all these lists in an array $A[1 : n]$ so for any vertex $v \in V$ the entry $A[v]$ returns the list $N(v)$ of vertex v
  - We can find the list of each vertex in $O(1)$ time and iterate over its neighbors in $O(\deg(v))$ time
  - Reading the entire graph takes $O(n + m)$ time

Graph Representation

- Different ways to represent the input graph $G = (V, E)$
- Adjacency Matrix Representation:
  - A matrix $M[1 : n][1 : n]$ where $M[i][j] = 1$ if and only if there is an edge from vertex i to vertex j; otherwise $M[i][j] = 0$
  - Can find all neighbors of a vertex in $O(n)$ time
  - Can find if there is an edge between two vertices in $O(1)$ time
  - Reading the entire graph takes $O(n^2)$ time

Graph Representation

- Different ways to represent the input graph $G = (V, E)$
- Adjacency Matrix Representation:

We always work with the adjacency list representation unless specified otherwise

  - Can find if there is an edge between two vertices in $O(1)$ time
  - Reading the entire graph takes $O(n^2)$ time

Graph Reductions

Reduction

- Solving a problem using algorithms for another problem in a **black-box** manner.
- A highly effective strategy in the algorithm design:
  - Reduce the task of solving problem A to solving problem B instead

Example

- **Problem 1:** How to prepare hot tea?
  - Input: Tea pot + Tea bag + access to water + access to heat
  - Output: A tea pot of tea
- **Algorithm 1:**
  1. Add water to tea pot
  2. Place tea pot on heat
  3. After 5 minutes add tea bag
  4. After 5 minutes you have a tea pot of tea

Example

- **Problem 2:** How to prepare hot tea starting with a full tea pot?
  - Input: Tea pot of water + Tea bag + access to water + access to heat
  - Output: A tea pot of tea

Example

- **Problem 2:** How to prepare hot tea starting with a full tea pot?
  - Input: Tea pot of water + Tea bag + access to water + access to heat
  - Output: A tea pot of tea
- **Algorithm 2:**
  1. Place tea pot on heat
  2. After 5 minutes add tea bag
  3. After 5 minutes you have a tea pot of tea

Example

- **Problem 2:** How to prepare hot tea starting with a full tea pot?
  - Input: Tea pot of water + Tea bag + access to water + access to heat
  - Output: A tea pot of tea
- **Algorithm 3:**
  1. Run Algorithm 1 except that do not do its first step

Example

- Algorithm 1:
    1. Add water to tea pot
    2. Place tea pot on heat
    3. After 5 minutes add tea bag
    4. After 5 minutes you have a tea pot of tea

Example

- Algorithm 1 may change over time, across different people, …

1. Place tea pot on heat
2. Add water to tea pot
3. After 5 minutes add tea bag
4. After 5 minutes you have a tea pot of tea

Example

- **Problem 2:** How to prepare hot tea starting with a full tea pot?
  - Input: Tea pot of water + Tea bag + access to water + access to heat
  - Output: A tea pot of tea
- **Algorithm 4 (reduction):**
  1. Empty the tea pot
  2. Run any algorithm for Problem 1 to get your hot tea

Graph Reductions

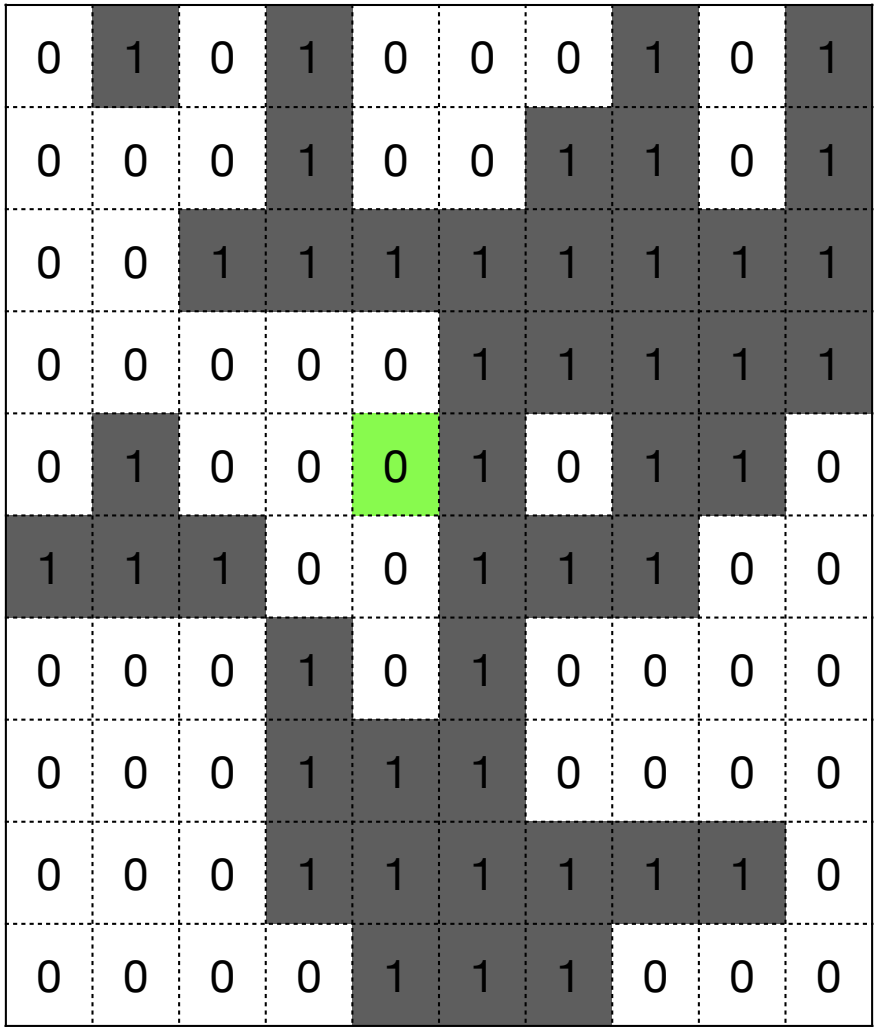
- Model the given problem as a graph problem
  - Specify exactly how you can get your graph from the input of the original problem
- Run any **black-box** algorithm for the graph problem you specified
- Translate back the solution to the graph problem to the solution of your original problem

Example: Fill Coloring Problem

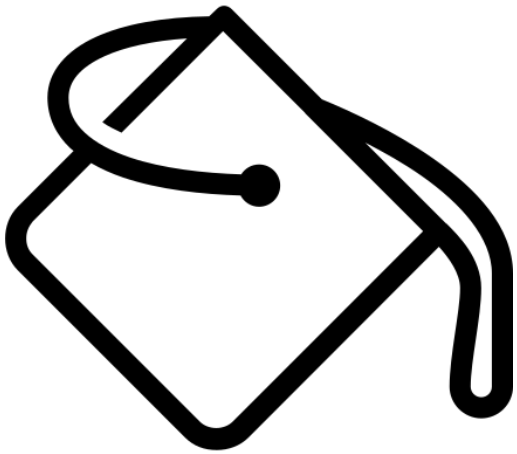
Example: Fill Coloring Problem

0	1	0	1	0	0	0	1	0	1
0	0	0	1	0	0	1	1	0	1
0	0	1	1	1	1	1	1	1	1
0	0	0	0	0	1	1	1	1	1
0	1	0	0	0	1	0	1	1	0
1	1	1	0	0	1	1	1	0	0
0	0	0	1	0	1	0	0	0	0
0	0	0	1	1	1	0	0	0	0
0	0	0	1	1	1	1	1	1	0
0	0	0	0	1	1	1	0	0	0

Example: Fill Coloring Problem

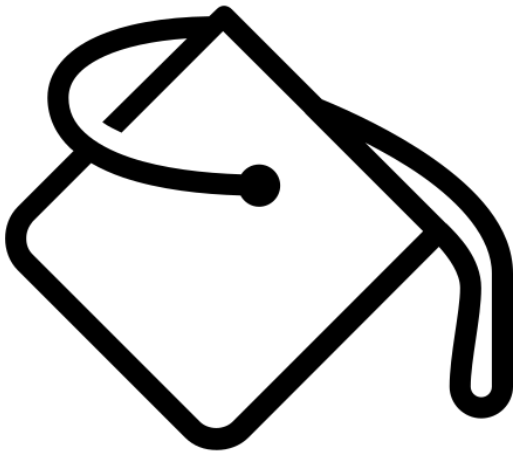


Example: Fill Coloring Problem



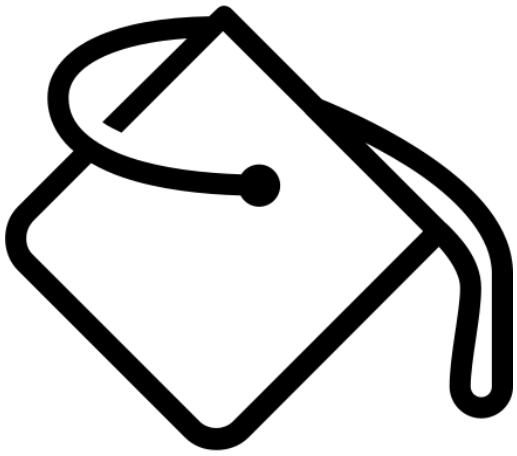
0	1	0	1	0	0	0	1	0	1
0	0	0	1	0	0	1	1	0	1
0	0	1	1	1	1	1	1	1	1
0	0	0	0	0	1	1	1	1	1
0	1	0	0	0	1	0	1	1	0
1	1	1	0	0	1	1	1	0	0
0	0	0	1	0	1	0	0	0	0
0	0	0	1	1	1	0	0	0	0
0	0	0	1	1	1	1	1	1	0
0	0	0	0	1	1	1	0	0	0

Example: Fill Coloring Problem



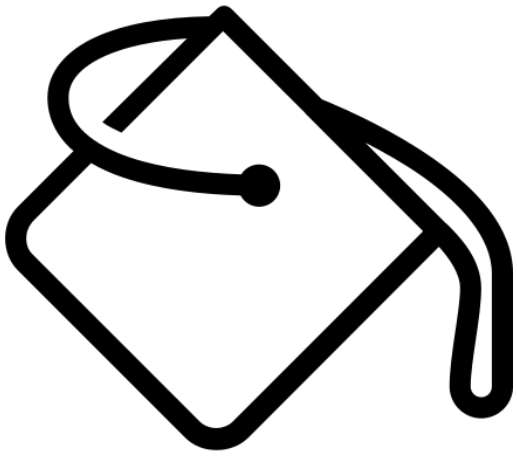
0	1	0	1	0	0	0	1	0	1
0	0	0	1	0	0	1	1	0	1
0	0	1	1	1	1	1	1	1	1
0	0	0	0	0	1	1	1	1	1
0	1	0	0	0	1	0	1	1	0
1	1	1	0	0	1	1	1	0	0
0	0	0	1	0	1	0	0	0	0
0	0	0	1	1	1	0	0	0	0
0	0	0	1	1	1	1	1	1	0
0	0	0	0	1	1	1	0	0	0

Example: Fill Coloring Problem



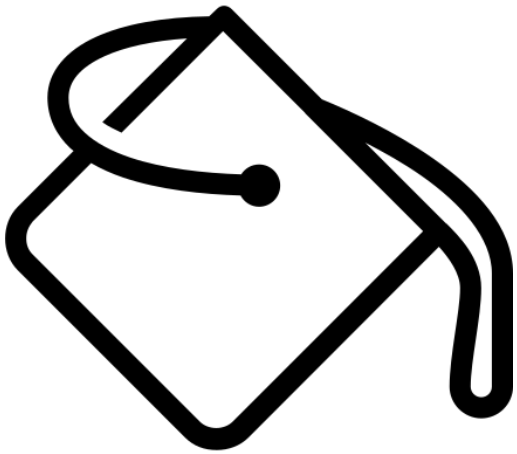
0	1	0	1	0	0	0	1	0	1
0	0	0	1	0	0	1	1	0	1
0	0	1	1	1	1	1	1	1	1
0	0	0	0	0	1	1	1	1	1
0	1	0	0	0	1	0	1	1	0
1	1	1	0	0	1	1	1	0	0
0	0	0	1	0	1	0	0	0	0
0	0	0	1	1	1	0	0	0	0
0	0	0	1	1	1	1	1	1	0
0	0	0	0	1	1	1	0	0	0

Example: Fill Coloring Problem



0	1	0	1	0	0	0	1	0	1
0	0	0	1	0	0	1	1	0	1
0	0	1	1	1	1	1	1	1	1
0	0	0	0	0	1	1	1	1	1
0	1	0	0	0	1	0	1	1	0
1	1	1	0	0	1	1	1	0	0
0	0	0	1	0	1	0	0	0	0
0	0	0	1	1	1	0	0	0	0
0	0	0	1	1	1	1	1	1	0
0	0	0	0	1	1	1	0	0	0

Example: Fill Coloring Problem



0	1	0	1	0	0	0	1	0	1
0	0	0	1	0	0	1	1	0	1
0	0	1	1	1	1	1	1	1	1
0	0	0	0	0	1	1	1	1	1
0	1	0	0	0	1	0	1	1	0
1	1	1	0	0	1	1	1	0	0
0	0	0	1	0	1	0	0	0	0
0	0	0	1	1	1	0	0	0	0
0	0	0	1	1	1	1	1	1	0
0	0	0	0	1	1	1	0	0	0

Example: Fill Coloring Problem

- **Input:** A pixel map and a starting point
  - An input 2D-array A[1:k][1:k] with entries in {0,1}.
  - A single cell i_s, j_s with $A[i_s][j_s] = 0$ as the starting point
- **Output:** Find the cells colored by the fill coloring of this pixel map
  - Color starting point,
  - go to any neighboring cell (left, right, top, down) with 0-value in the matrix and color them, go to the neighbor of any cell colored
  - continue until no new cell can be colored

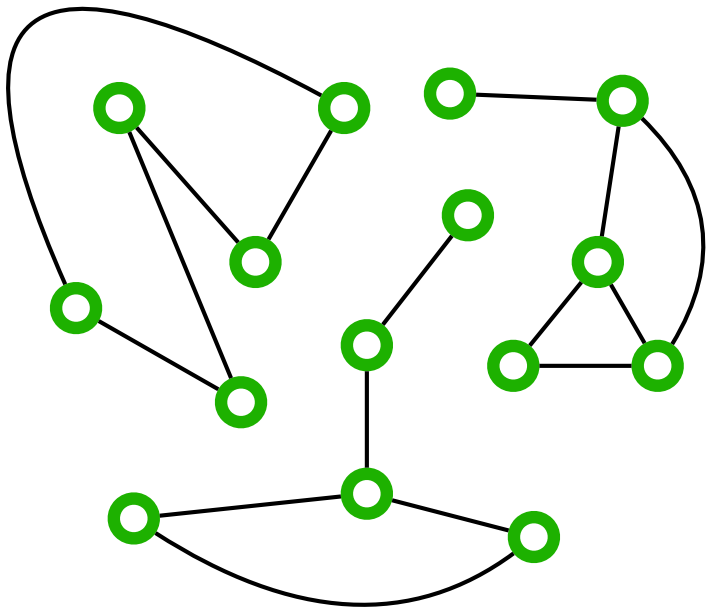
Algorithm?

- Our goal is to design an algorithm for this problem
- We will do this using a graph reduction
- **Graph problem?**
- Graph search

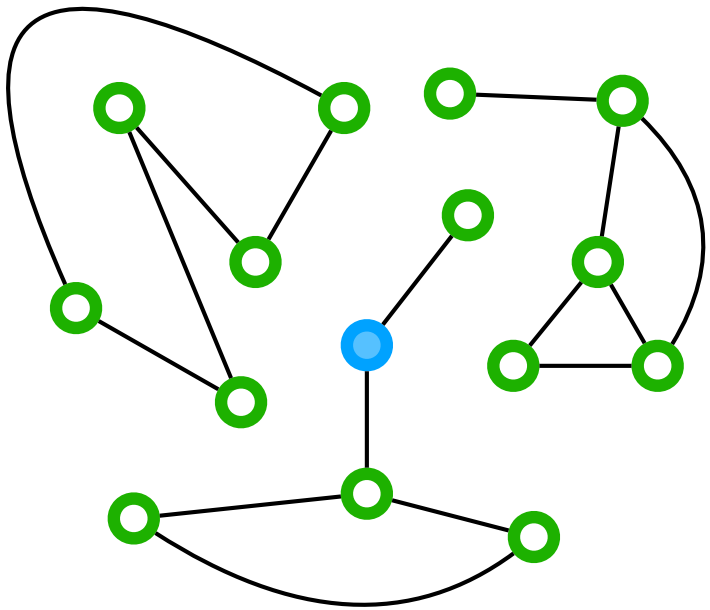
Graph Search

- **Input:**
  - An undirected graph G=(V,E)
  - A starting vertex s
- **Output**
  - All vertices in the connected component of G that contains s
    - Vertices reachable from s in the graph via some path

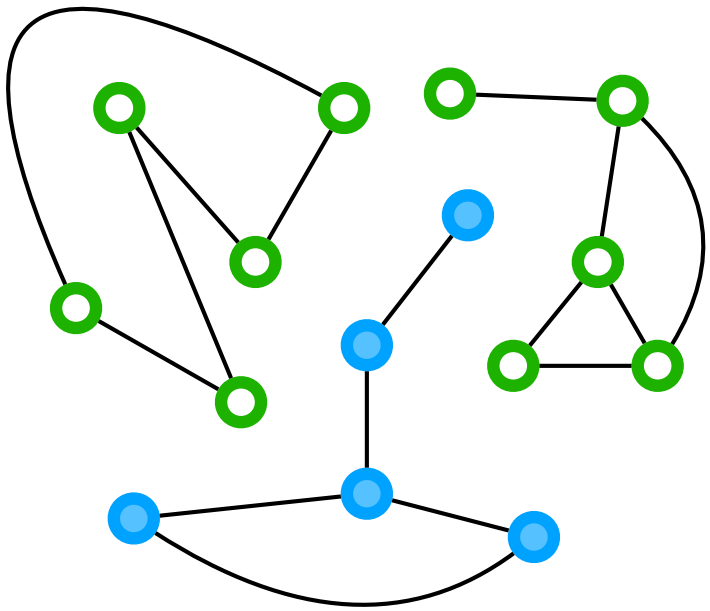
Graph Search: Example



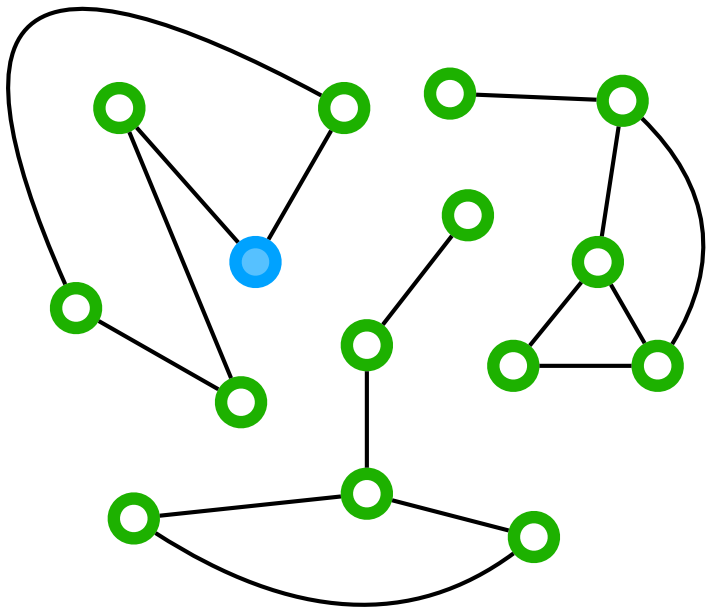
Graph Search: Example



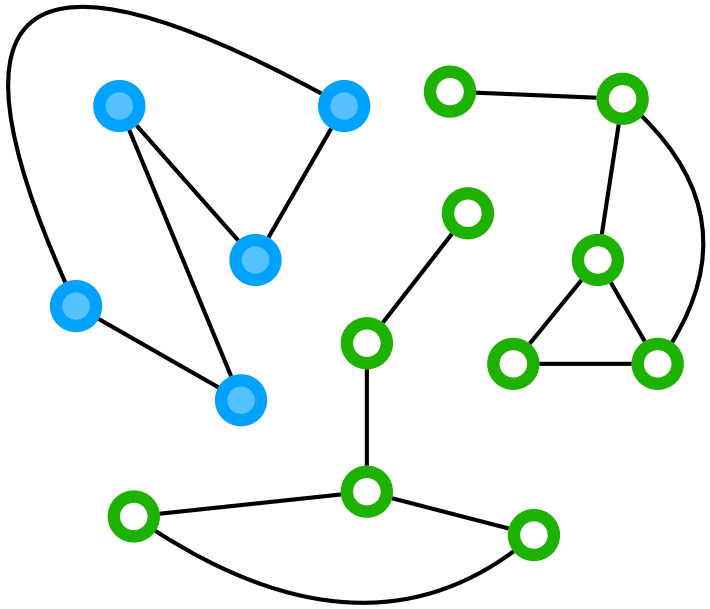
Graph Search: Example



Graph Search: Example



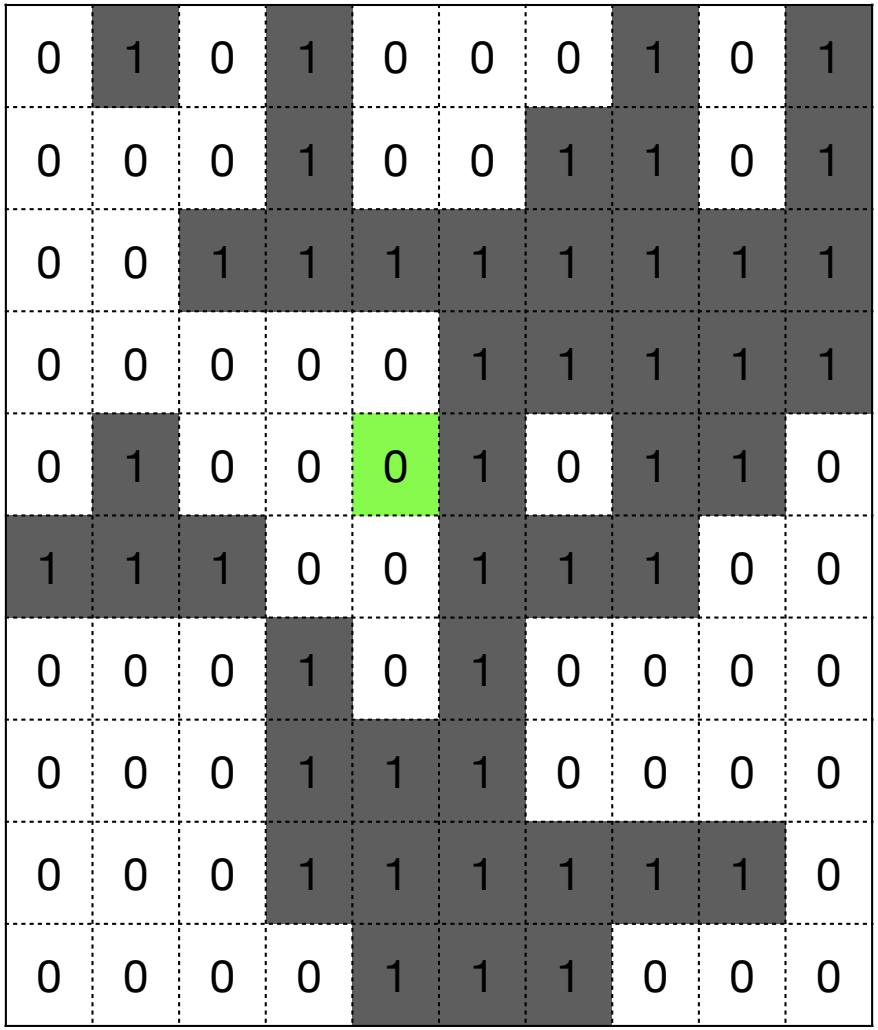
Graph Search: Example



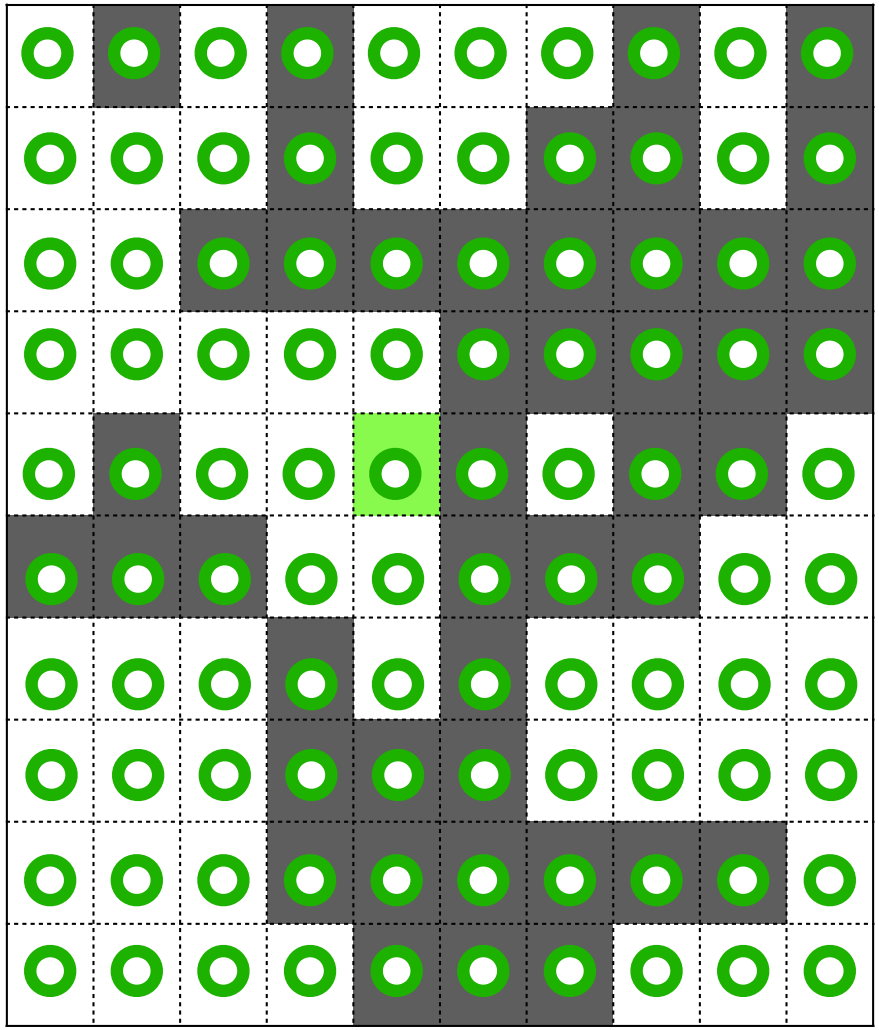
Reducing Fill Coloring to Graph Search

- Algorithm for Fill Coloring (Reduction):
  - Create a graph $G = (V, E)$ with $n = k^2$ vertices as follows:
    - For any cell (i, j) in the array A create a vertex $v_{i,j}$
    - For any neighboring cells (i, j) and (x, y) if both $A[i][j] = A[x][y] = 0$ then add an edge $(v_{i,j}, v_{x,y})$.
  - Define the starting vertex of search as $s = v_{i_s,j_s}$ where (i_s, j_s) is the starting point in the fill coloring problem
  - Run any graph search algorithm on G and s and return all cells (i, j) where their corresponding vertex $v_{i,j}$ is output by the search algorithm

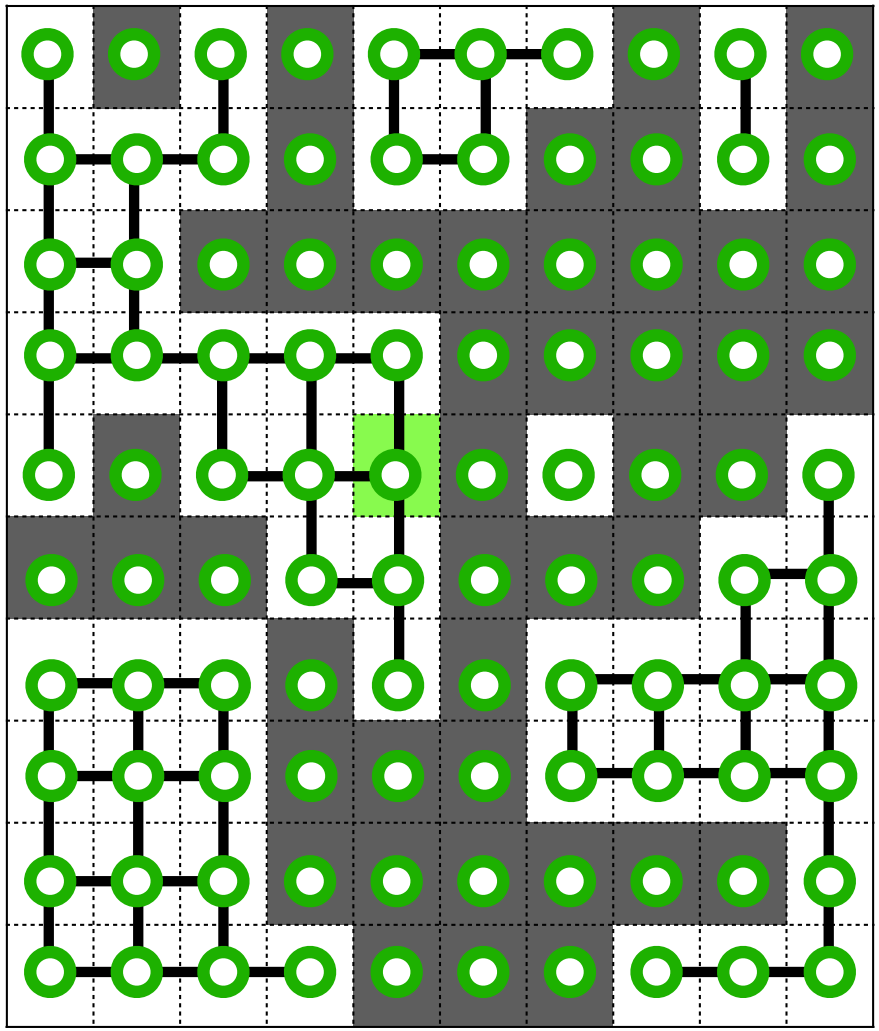
Example of the Reduction



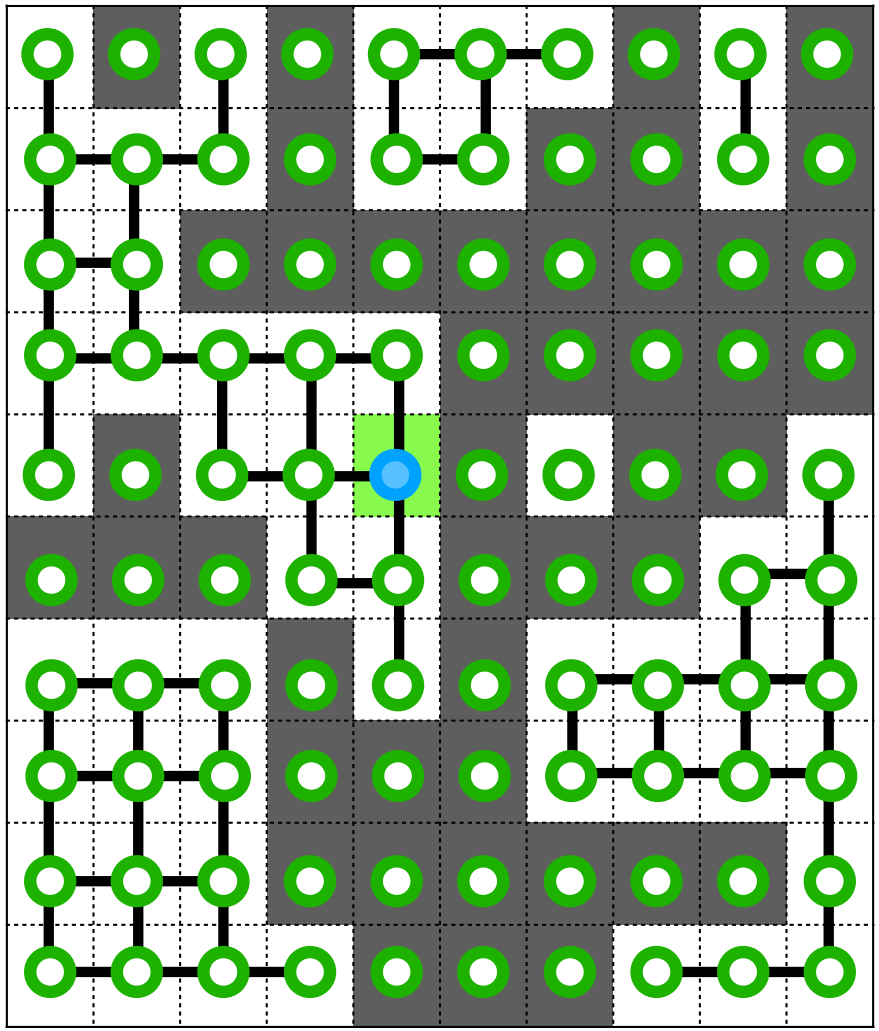
Example of the Reduction



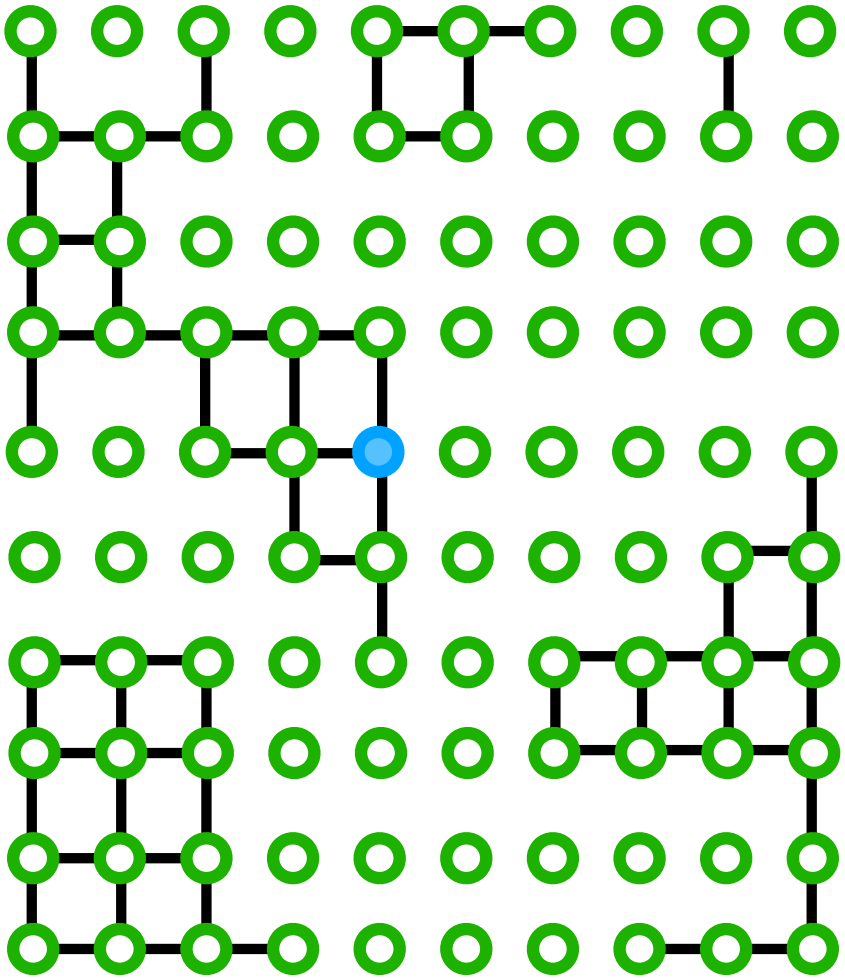
Example of the Reduction



Example of the Reduction

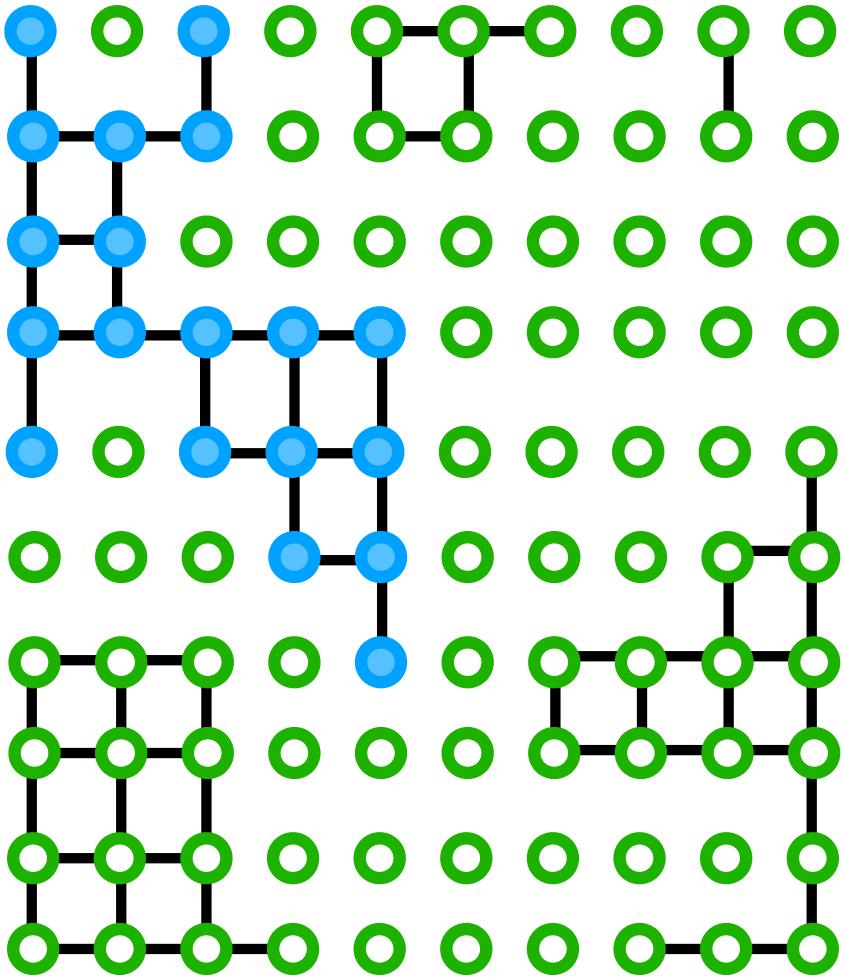


Example of the Reduction

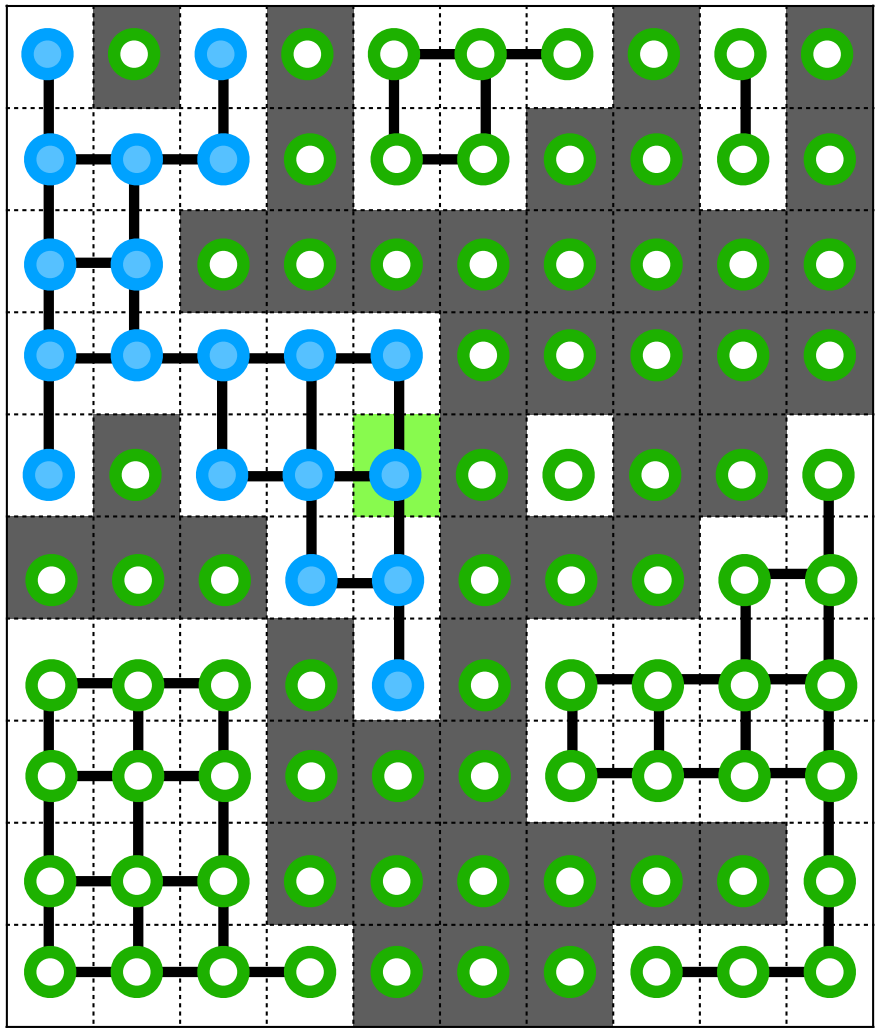


Example of the Reduction

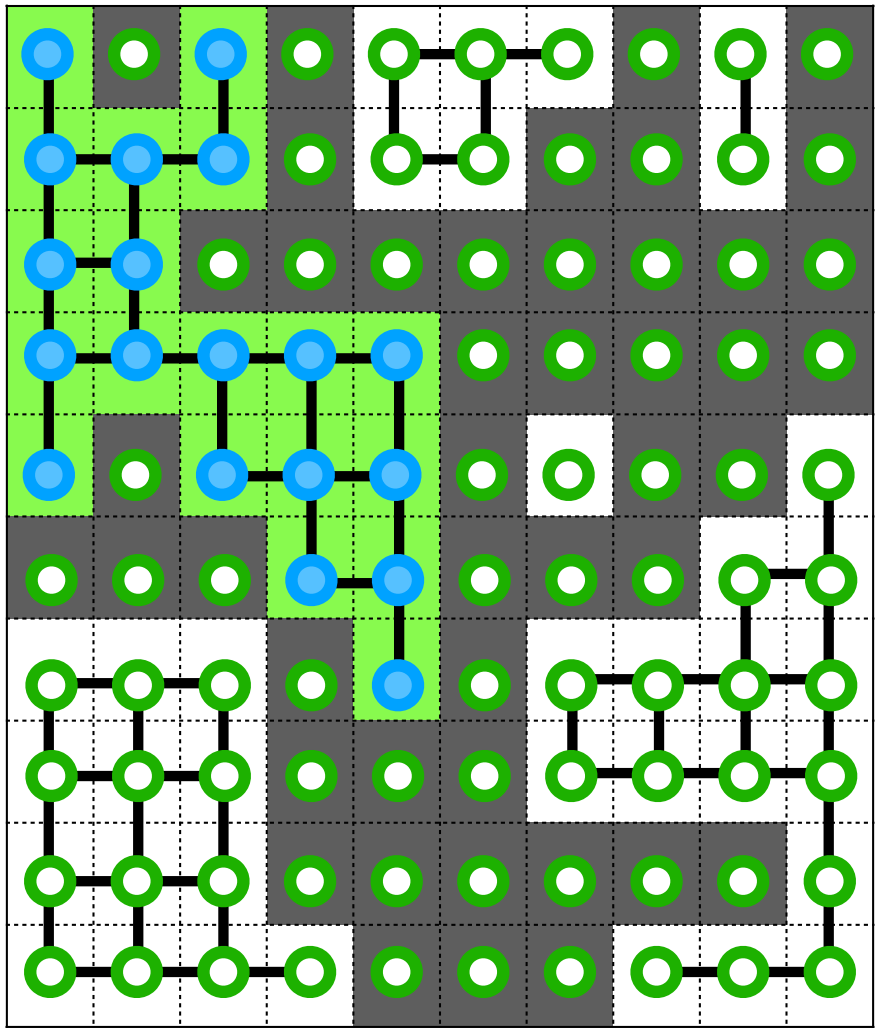
Graph Search Algorithm



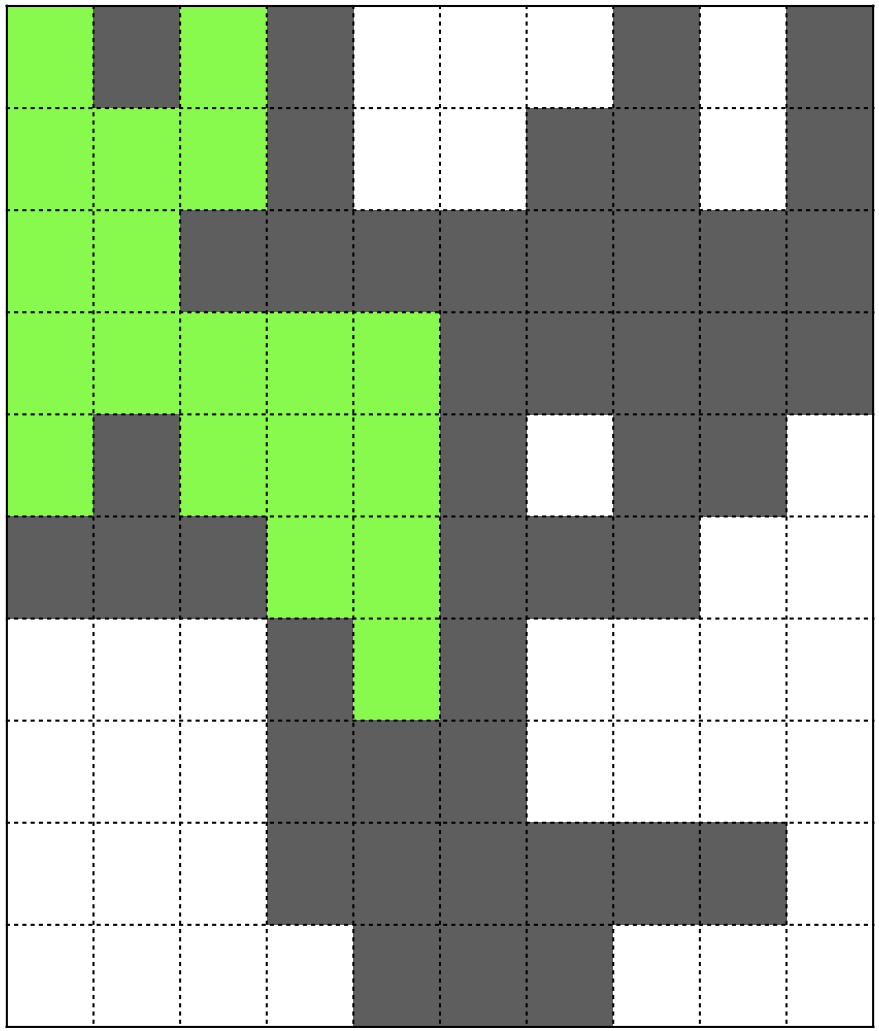
Example of the Reduction



Example of the Reduction



Example of the Reduction



Proof of Correctness

- We prove that the set of cells that should be fill colored is exactly the same as the connected component of the vertex v_{i_s,j_s}

Proof of Correctness

- We prove that the set of cells that should be fill colored is exactly the same as the connected component of the vertex v_{i_s,j_s}
- Part one: any cell that needs to be colored corresponds to a vertex in the connected component of v_{i_s,j_s}
- Part two: any vertex in the connected component of v_{i_s,j_s} corresponds to a cell that needs to be colored

Proof of Correctness

- We prove that the set of cells that should be fill colored is exactly the same as the connected component of the vertex v_{i_s,j_s}
- Part one: any cell that needs to be colored corresponds to a vertex in the connected component of v_{i_s,j_s}
- Part two: any vertex in the connected component of v_{i_s,j_s} corresponds to a cell that needs to be colored

- We should remember to prove **both parts**!

Proof of Correctness

- Part one: any cell that needs to be colored corresponds to a vertex in the connected component of v_{i_s,j_s}
- **Proof:**

Proof of Correctness

- Part two: any vertex in the connected component of v_{i_s,j_s} corresponds to a cell that needs to be colored
- **Proof:**

Runtime Analysis

- Creating the graph $G = (V, E)$ takes $O(k^2)$ time
- This graph has $n = \Theta(k^2)$ & $m = \Theta(k^2)$
- Let $Search(n, m)$ denote the runtime of the best algorithm for doing a graph search on a graph with n vertices and m edges
- Runtime of our algorithm is $O(k^2 + Search(\Theta(k^2), \Theta(k^2)))$
- In the next lecture, we design different algorithms for graph search and show that $Search(n, m) = O(n + m)$
- This makes our runtime $O(k^2)$