Designing truly novel graph algorithms is a very difficult task. The key to using graph algorithms effectively in applications lies in correctly modeling your problem so you can take advantage of existing algorithms.

### 5.1 Illumination (ACM Regional Contest 2016, USA Southeast)

**Question:** You inherited a haunted house. Its floor plan is an $n \times n$ square grid with $\ell$ lamps in fixed locations $(c_1, r_1), \ldots, (c_\ell, r_\ell)$ and no interior walls. Each lamp can either illuminate its row or its column, but not both simultaneously. The illumination of each lamp extends by $r$ squares in both directions, so a lamp unobstructed by an exterior wall of the house can illuminate as many as $2r + 1$ squares. If a square is illuminated by more than one lamp in its row, or by more than one lamp in its column, the resulting bright spot will scare away ghosts forever, diminishing the value of your property. Is it possible for all lamps to illuminate a row or column, without scaring any ghosts? Note that a square illuminated by two lamps, one in its row and the other in its column, will not scare away the ghosts.

**Modeling the Problem.** When you see a problem of many objects (lamps in this case) and their relationship (in this case, same row or same column but within a certain distance), a natural approach is to interpret it as a graph problem. Thus for each lamp, we create a node.

How should we decide if two nodes are connected? Here we’ll have a potential problem if two lamps can illuminate a common square. This happens if they are in the same row or same column, but within distance of at most $2r$ squares—that’s how we decide if two nodes are connected.

How much is the cost of creating this graph? For each lamp, we’ll need to go through all $\ell - 1$ other lamps to see if they are connected. Thus the cost for graph creation is $O(\ell^2)$. The graph will have $\ell$ nodes and at most $O(\ell^2)$ edges.

Now, for each lamp, we need to decide whether to let it illuminate a row or a column. So there are two choices for a node, and a natural way to formulate it is to view it as a way to color the node either as blue (“Row”) or white (“Column”). The requirement that two lamps can’t illuminate the same square simply means that two connected nodes must be colored differently. Thus you need to find a way to color the nodes with blue and white such that no two connected nodes have the same color.

**Bipartite Checking.** The problem above is a classic graph problem, known as bipartite-checking. In particular, we say that an undirected graph is bipartite if it’s possible to color the graph by using just two colors (say blue and white) such that adjacent nodes have different colors. For example, in Figure 5.1, the left graph is bipartite, justified by the particular way we color it. In contrast, the right graph of Figure 5.1 is non-bipartite, as it contains a cycle $A \rightarrow D \rightarrow G \rightarrow A$ of length 3. This cycle can’t be colored using just two colors: if we color $A$, say, blue, then $D$ must be white, and then $G$ must be blue. But then in the edge $\{A, G\}$, both endpoints have the same color. In general, if a graph contains a cycle of odd length then it’s not bipartite.

Given an undirected graph, we’d like to know if it’s bipartite, and if yes, how to color its nodes.
Finding a way to color. Let’s use the specialization technique to figure out how to attack this problem. In our case, the original problem considers a generic graph, so it’s natural to specialize it to simpler graphs. The simplest graph that we can imagine in this setting is a path, as illustrated in Figure 5.2. In this case it’s easy to color, by alternating the colors node-by-node: the first node is colored white, the second blue, the third white again, the fourth blue, and so on.

![Simple Path Graph](image1)

Figure 5.2: A graph that is a simple path.

Gaining new knowledge and confidence from this simple case, let’s consider a more complex special case: the graph is a tree. The idea for the first special case tells us that we need to find some starting node, color it blue, and then keep alternating the colors. In this second special case, it’s natural to choose the starting node to be the root. This means that we’ll have to color nodes in the next layer white, and so on. In other words, we can color by alternating the colors level-by-level, as illustrated in Figure 5.3.

![Tree Graph](image2)

Figure 5.3: A tree and how to color its nodes. The root is node G.

Emboldened by this new success, we consider an even more complex case: the graph is a forest. In this case coloring is simple: we can color the trees of the forest individually, as illustrated in Figure 5.4.

Finally, we consider coloring a general graph. We need somehow exploit the special cases as stepping stones. How would we use the special cases? How would we, say bring a forest out of a general graph? Eventually
we realize that if we run, say BFS then we can get a BFS forest from the graph. We then can color the BFS forest as in the special case, leading to a potential coloring of the graph. See Figure 5.5 for an illustration.

![Figure 5.4: A forest and how to color its nodes. The roots are node G and node H.](image)

Looking backward, this series of special cases keep bringing ideas to us, one upon another, to realize the full solution. The key point here is to select good special cases, so that we can use them as stepping stones.

**CHECKING CONSISTENCY.** Now, recall that in some graphs, its impossible to color the nodes properly. So we need to check for consistency in our coloring. That is, we’ll check every edge to see if the two endpoints are not of the same color. If the coloring is consistent then the graph is bipartite; otherwise it is not. For example, in Figure 5.6, the coloring is inconsistent, as both nodes C and F are colored blue, and they are connected.

There’s, however, some subtlety in the argument above. Just because one particular way of coloring is inconsistent, it doesn’t mean that every way of coloring will be so. Therefore, in the case of inconsistent coloring, our verdict of non-bipartiteness begs some elaboration. Let’s explain via the concrete example of Figure 5.6; this explanation is actually general and works for any graph. In our particular example, the inconsistency happens at the connected component containing node A; let T be the BFS tree of this connected component. Assume that graph can be colored consistently using two colors; let’s say under this coloring, the root G of T is colored white. But then nodes in the next level of T, namely A and F, must be colored blue, as they are connected to a white node. Then nodes in the next level (D, B, and E) would be colored white, as they are connected to blue nodes, and so on. In other words, this is exactly what our algorithm would color the graph, but this coloring is inconsistent, which is a contradiction.

**AN EXERCISE.** Below is an exercise for you to practice the specialization technique on graphs.
Figure 5.6: A non-bipartite graph (left) and its BFS forest with coloring (right).

**Question:** An articulation vertex of a graph $G$ is a vertex whose deletion disconnects $G$. Let $G$ be a graph with $n$ vertices and $m$ edges. Give an $O(m+n)$ algorithm for finding a vertex of $G$ that is not an articulation vertex—i.e., whose deletion does not disconnect $G$.

### 5.1.1 Islands (ACM Regional Contest 2016, USA Southeast)

**Question:** You are mapping a faraway planet using a satellite. The planet’s surface can be modeled as an $n \times m$ grid. Your satellite has captured an image of the surface. Each grid square is either land (denoted as ‘L’), water (denoted as ‘W’), or covered by clouds (‘C’). Clouds mean that the surface could either be land or water, but you can’t tell. An island is a maximal region of land where every grid cell in the island is reachable from every other by a path that only goes up, down, left or right. Given an image, determine the minimum number of islands that is consistent with the given information.

**Modeling the problem.** At the first glance, it’s tempting to view each cell as a node. However, note that the problem is all about islands, meaning that only L or C cells matter. Thus for each L or C cells, we will create a corresponding node.

How would we decide if two nodes are connected? From the description of island, here you only consider paths that move across adjacent cells. Thus two nodes are connected if the two corresponding cells are adjacent. To tell if a node is L or C, we will color them blue or white respectively. Constructing the graph takes $O(n^2)$ time; it has $O(n^2)$ nodes and edges. See for an example of a grid and its corresponding graph.

![Grid and Graph Example](image)

**Figure 5.7:** A grid (left) and its corresponding graph (right).

From the description of island, it sounds like an island should be some sort of connected components of the graph. Is what we seek simply the number of connected components of this graph? No, if a connected
component consists of only white nodes, all those nodes may be just water, and the connected component is not an island. We can only be certain that a connected component is an island if it contains at least a blue node. For example, we can only say that the graph in Figure 5.7 contains at least an island. Thus we only need to go over all connected components and only count those who have some blue nodes. This takes $O(n^2)$ time.

5.2 Exercises

5.2.1 Snakes and Ladders

Snakes and Ladders is a classic board game. The board consists of an $n \times n$ grid of squares, numbered consecutively from 1 to $n^2$, starting in the bottom left corner and proceeding row by row from bottom to top, with rows alternating to the left and right. Certain pairs of squares in this grid, always in different rows, are connected by either snakes (leading down) or ladders (leading up). Each square can be an endpoint of at most one snake or ladder. See Figure 5.8 for an illustration for the case $n = 4$.

![Snakes and Ladders Board](image)

Figure 5.8: A Snakes and Ladders board. Upward straight arrows are ladders; downward wavy arrows are snakes.

You start with a token in cell 1, in the bottom left corner. In each move, you advance your token up to $k$ positions, for some fixed constant $k$. If the token ends the move at the top end of a snake, it slides down to the bottom of that snake. Similarly, if the token ends the move at the bottom end of a ladder, it climbs up to the top of that ladder. Describe an efficient algorithm to compute the smallest number of moves required for the token to reach the last square of the grid.

5.2.2 Dominators

In graph theory, a node $X$ dominates a node $Y$ if every path from the predefined start node to $Y$ must go through $X$. If $Y$ is not reachable from the start node then node $Y$ does not have any dominator. By definition, every node reachable from the start node dominates itself. Given a directed graph, design an efficient algorithm that finds the dominators of every node where the 0-th node is the start node. See Figure 5.9 for an illustration.
5.2.3 Bug Patching

After releasing a new software, Tinyware Inc. has been producing patches ever since. However, today they realize a big problem with the patches they released. While all patches fix some bugs, they often rely on other bugs to be present to be installed. This happens because to fix one bug, the patches exploit the special behavior of the program due to another bug.

More formally, the situation looks like this. Tinyware has found a total of \( n \) bugs \( B = \{b_1, \ldots, b_n\} \), with \( n \leq 20 \), in their software. They have released \( m \) patches \( p_1, \ldots, p_m \). To apply patch \( p_i \) to the software, the bugs \( B_i^+ \subseteq B \) have to be present in the software, and the bugs \( B_i^- \subseteq B \) must be absent (of course \( B_i^+ \cap B_i^- = \emptyset \)). The patch then fixes the bugs \( F_i^- \subseteq B \) (if they have been present), and introduces the new bugs \( F_i^+ \subseteq B \) (where, again \( F_i^- \cap F_i^+ = \emptyset \)).

Each of \( B_i^+ \), \( B_i^- \), \( F_i^+ \), \( F_i^- \) is represented as an \( n \)-element array. For example, \( B_i^+[k] \) is ‘+’ if bug \( b_k \) has to be present to apply patch \( p_i \), a ‘-’ if bug \( b_k \) has to be absent, and a ‘*’ if it doesn’t matter whether the bug is present or not.

Given the original version of their software, which contains all the bugs in \( B \), Tinyware want to find a sequence of patches which results in a bug-free version. If there are many sequences, they want to find the one with the smallest number of patches.