1. Problem 26 in Jeff Erickson's chapter on basic graph algorithms. Only part (a).

Solution: We will use graph modeling. We will adopt a convention that the bottom left of the maze has coordinate $(1,1)$ (and thus the top right has coordinate (n,n)); other conventions are fine as well.

- First define a set $W \subseteq \{1..n\}^2 \times \{1..6\}^2$ as follows.

  For each value $t \in \{1..6\}$ on the top of the die, there are four possible values $\ell \in \{1..6\}$ for the left-side of the die, corresponding to four different rotations of the die. Then a *candidate* vertex $(i,j,t,\ell) \in \{1..n\}^2 \times \{1..6\}^2$ means that the die is at position (i,j) with value t on top, and ℓ on the left.

  We include (i,j,t,ℓ) in W if and only if (i,j,t,ℓ) represents a *valid* configuration, i.e., it is possible to have t on top and ℓ on the left side, and furthermore either $L[i,j] = 0$ or $(L[i,j] > 0$ and $t = L[i,j])$.

  Additionally, we will add two auxiliary vertices called S and T, the *source* and *target* vertices whose function will be explained below.

  In summary, the set of vertices is $V = W \cup \{S,T\}$.

- We define the edge set E as follows.

  For each $(i,j,t,\ell) \in V$, there are at most four incident edges. For each direction (north, south, east, west), rolling the die in that direction gives a unique tuple (i',j',t',ℓ'). If $(i',j',t',\ell') \in V$, then $(i,j,t,\ell)(i',j',t',\ell') \in E$.

  Additionally, we will have edges between S and all vertices of the form $(1,1,t,\ell)$, and edges between T and all vertices of the form (n,n,t,ℓ).

  All edges are undirected, since rolling is reversible.

- We need to determine if we can reach a vertex representing a die in the target position (i.e., of the form (n,n,t,ℓ)) from a vertex representing a die in the initial position (i.e., of the form $(1,1,t,\ell)$).

  While one can do this by solving reachability for each initial position vertex and checking if a target position vertex can be reached from one of the initial position vertices, the addition of vertices S and T will allow us to simplify to solving a single reachability problem: observe that a target position vertex can be reached from an initial position vertex if and only if T can be reached from S.

  Thus *we need to determine if T can be reached from S*.

- We can solve this problem by running Basic (aka Whatever-First) Search from S and checking of T is reachable.

- Constructing the graph by naive brute force can be done in $O(n^2)$: There are at most $24n^2 + 2 = O(n^2)$ possible vertices and the validity of each candidate vertex can be checked in constant time; furthermore, there are $O(n^2)$ edges, since each vertex can have at most 4 incident edges, and the validity of each candidate edge can also be checked in constant time.

  As a consequence of the preceding analysis, $V + E = O(n^2)$, so running Basic Search takes $O(V+E) = O(n^2)$ time, giving a total runtime of $\boldsymbol{O(n^2)}$. ■

Rubric: Standard graph reduction rubric. Maximum 8 points if rotation of die was not taken into account. No penalty for taking the OR of multiple usages of Basic Search.

Rubric (Standard rubric for graph reduction problems): For problems out of 10 points:

- \+ 1 for correct vertices, *including English explanation for each vertex*
- \+ 1 for correct edges
  - — ½ for forgetting "directed" if the graph is directed
- \+ 1 for stating the correct problem (in this case, *reachability*)
  - — "Breadth-first search" is not a problem; it's an algorithm!
- \+ 1 for correctly applying the correct algorithm (in this case, *Basic (Whatever-First) Search*)
  - — ½ for using a slower or more specific algorithm than necessary
- \+ 1 for time analysis in terms of the input parameters.
- \+ 5 for other details of the reduction
  - — If your graph is constructed by naive brute force, you do not need to describe the construction algorithm; in this case, points for vertices, edges, problem, algorithm, and running time are all doubled.
  - — Otherwise, apply the appropriate rubric, *including Deadly Sins*, to the construction algorithm. For example, for a solution that uses dynamic programming to build the graph quickly, apply the standard dynamic programming rubric.

2. This question is about cycles in graphs.

   - Describe a linear time algorithm that given a *directed* graph $G = (V, E)$ and a node $s \in V$ outputs a directed cycle containing s if there is at least one, or correctly states that there is no directed cycle containing s.

Solution: This algorithm is based on the following idea. If there is a cycle C containing s then consider the edge entering s in C. Say it is (u, s). Then $C - (u, s)$ is a path from s to u. Thus u is reachable from s. Similarly if an in-neighbor u (that is, $(u, s) \in E$)) is reachable from s then the path from s to u together with the edge (u, s) is a cycle. One can use this observation to derive the algorithm which computes the set S of all nodes reachable from s and checks if any of them has an edge to s. Alternatively, one can check for each in-neighbor u of s whether it is in S.

```
CheckAndOutputCycle(G, s):
  Compute S ← rch(G, s) using Basic Search
  Let T be out-tree rooted at s found by the search
  For each edge (u, s) ∈ In(s) do
      If (u ∈ S) then
          Let C be cycle formed by path from s to u in T and the edge (u, s)
          Output C
  return "NO cycle in G containing s"
```

The running time is $O(m + n)$ since we only do one search and simply check whether the in-neighbors are in the reachable set. Finding the cycle C via the path in T also straight forward. ■

Solution: This algorithm is based on DFS. The correctness of this algorithm follows the same reasoning as above and via the properties of DFS.

```
CheckAndOutputCycleViaDFS(G, s):
  Compute S ← rch(G, s) via DFS
  Let T be out-tree rooted at s found by DFS
  If there is a back edge (u, s) discovered during DFS then
      Let C be cycle formed by path from s to u in T and the edge (u, s)
      Output C
  Else
      return "NO cycle in G containing s"
```

The running time is $O(m+n)$ by the same reasoning as the previous solution. ■

- Describe a linear time algorithm that given an *undirected* graph $G = (V, E)$ and a node $s \in V$ outputs a cycle containing s if there is at least one, or correctly states that there is no cycle containing s.

Solution: This algorithm is based on following idea. Let X be the set of neighbors of s in G, that is $X = \{u \mid (u,s) \in E\}$. Suppose there is a cycle C containing s. Let $e_1 = (u_1, s), e_2 = (u_2, s)$ be the two edges incident to s where $u_1 \neq u_2$ and $u_1, u_2 \in X$. One sees that there is a path connecting u_1 and u_2 in the graph $G' = G - s$. Conversely if G' has a path between $u_1, u_2 \in X$ then that path together with the edges (u_1, s) and (u_2, s) gives a cycle containing s. Thus G has a cycle containing s if and only if there are two neighbors of s in the same connected component of $G' = G - s$.

```
CheckAndOutputCycle(G, s):
  X is set of neighbors of s in G
  G' ← G − s
  Compute connected components of G' using Basic Search
  If two distinct nodes u_1, u_2 ∈ X are in same connected component
      Output cycle C by concatenating path from u_1 to u_2 in G' with (u_1, s), (s, u_2).
  Else
      return "NO cycle in G containing s"
```

It may not be completely obvious how one can check whether two neighbors of s are in same connected component but we can do this in $O(n)$ time as follows.

The connected component algorithm can be modified to output for each for each vertex u a number $\alpha(u)$ where $\alpha(u)$ is the index of the connected component (as discovered by the Basic Search algorithm in some order). Once we have that information we initialize an array A of size k with -1's where k is the number of connected components. For each vertex $u \in X$ in some order we check if $A[\alpha(u)]$ is -1. If it is then we set $A[\alpha(u)] = u$. If $A[\alpha(u)] \neq -1$ while examining u we know that some other neighbor of s is in the same connected component as u and we know its identity. The total work is $O(n)$.

One can see that finding a path between u_1 and u_2 that are in the same connected component in G' takes at most $O(m + n)$ time (in fact it can be done in $O(n)$ time if one keeps track of a spanning tree for each connected component when finding them in G').

In summary the running time is $\boldsymbol{O(m+n)}$. ■

Solution: This is the algorithm from the second solution for *directed* graphs, modified to use *non-tree* edges instead of *back* edges since G is now undirected.

```
CheckAndOutputCycleViaDFS(G, s):
  Compute S ← rch(G, s) via DFS
  Let T be tree rooted at s found by DFS
  If there is a non-tree edge (u, s) discovered during DFS then
      Let C be cycle formed by path from s to u in T and the edge (u, s)
      Output C
  Else
      return "NO cycle in G containing s"
```

As before the running time is $\boldsymbol{O(m+n)}$. ■

- Describe a linear-time algorithm that given a *directed* graph outputs all the nodes in G that are contained in some cycle. More formally you want to output

$$S = \{v \in V \mid \text{there is some cycle in } G \text{ that contains v}\}.$$

Solution: Observe that a node s is in a cycle iff s is in a strongly connected component of G of size at least 2. Thus one can find all the strongly connected components and output the vertex sets of those which are of size at least 2. We saw a linear time algorithm in lecture to compute all strongly connected components of a given directed graph. Note that we are not required to identify a cycle for each vertex.

```
OutputAllNodesInCycles(G):
  Compute strongly connected components S_1, S_2, ..., S_k of G
  S ← ∅
  For i = 1 to k do
      If (|S_i| ≥ 2) then S ← S ∪ S_i
  Output S
```

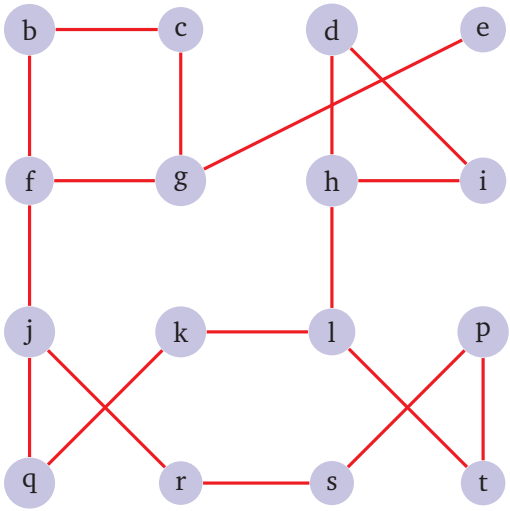
Since $S \subseteq V$ the entirety of the for loop can be done in $O(n)$ time, so the overall running time of this algorithm is dominated by the running time of the algorithm for computing strongly connected components, i.e., $\boldsymbol{O(m+n)}$. ■

Rubric:

- 4 points: 3 points for correct algorithm and 1 point for run time justification.
- 4 points: 3 points for correct algorithm and 1 point for run time justification.
- 2 points: 1 point for correct algorithm and 1 point for run time justification.

3. Given an *undirected* connected graph $G = (V, E)$ an edge (u, v) is called a cut edge or a bridge if removing it from G results in two connected components (which means that u is in one component and v in the other).

   - What are the cut-edges in the graph shown in the figure?



Solution: These cut-edges are (g, e), (f, j), (l, h). ■

   - Given G and edge $e = (u, v)$ describe a linear-time algorithm that checks whether e is a cut-edge or not. What is the running time to find all cut-edges by trying your algorithm for each edge? No proofs necessary for this part.

Solution: To check whether $e = (u, v)$ is a cut edge, we simply run Basic Search from u in the graph $G' = G \setminus e$. If v is not reachable from u it means that u and v are in different connected components in G', and hence e is a cut edge. Suppose v is reachable from u in in G' then e is not a cut edge. Running Basic Search take $O(m + n)$ time and doing this for each of the m edges results in a total time of $\boldsymbol{O(m^2)}$ time.

```
FindAllCutEdges(G):
  S ← ∅
  For each edge e = (u, v) in G do
      G' ← G − e
      Compute rch(G', u) using Basic Search
      If (v ∉ rch(G', u))
          S ← S ∪ {e}

  return S
```

■

- Consider *any* spanning tree T for G. Prove that every cut-edge must belong to T. Conclude that there can be at most $(n-1)$ cut-edges in a given graph. How does this information improve the algorithm to find all cut-edges from the one in the previous step?

Solution: Consider any edge $e \notin T$. Removing e does not disconnect the graph since T remains and connects all vertices. Thus any cut edge must be in T.

To improve our algorithm to find all cut edges, instead of testing every edge, we can simply test every edge from an arbitrary spanning tree T. One can compute a spanning tree T by doing Basic Search from any vertex u. Since there are only $(n-1)$ edges in T, and we can do the computation described in the previous part to test whether an edge $e \in T$ is a cut edge or not in $O(m+n)$ time. Thus the overall time is $\boldsymbol{O(mn)}$, which is an improvement when $n = o(m)$.

```
FindAllCutEdgesTake2(G):
  S ← ∅
  Compute a spanning tree T by running Basic Search from some vertex w ∈ V
  For each edge e = (u, v) in T do
      G' ← G − e
      Compute rch(G', u) using Basic Search
      If (v ∉ rch(G', u))
          S ← S ∪ {e}
  return S
```

■

- Prove that an edge e is contained in some cycle of G if and only if it is *not* a cut edge.

Solution:

($\Rightarrow$) If $e = (u, v)$ is contained in a cycle C of G, then $P = C \setminus e$ is a path in $G \setminus e$, meaning that the endpoints u and v remain connected in $G \setminus e$, and hence e cannot be a cut-edge.

($\Leftarrow$) Suppose that $e = (u, v)$ is not a cut edge. Then, there is some path P in $G \setminus e$ from u to v, so $e \notin P$. Then, $C = P \cup e$ forms a cycle in G containing e as desired.

■

- Let $s \in V$ be a vertex in G. Prove that there is a cycle containing s in G if and only if there is some edge e incident to s such that e is not a cut edge of G.

Solution:

($\Rightarrow$) Let C be the cycle containing s. Then, there must be two edges e, e' in C that are incident to s. Both e, e' are in C and hence, from the previous part, neither can be cut edge. Thus s is incident to an edge (in fact two) that are not cut edges.

($\Leftarrow$) Suppose $e = (s, u)$ is a non-cut-edge incident to s. Then by the previous part, we know that e is contained in a cycle of G, meaning that s is contained in that cycle as well.

■

- Assuming that there is a linear-time algorithm to find all the cut edges of G (as outlined in the subsequent parts) describe a linear time algorithm to find *all* vertices in G that are in some cycle. This is the same problem as in 2(c) but in undirected graphs.

Solution: First, run the algorithm to find all cut edges Z of G. Let $G' = (V, Z)$ be the graph on V induced by the edges Z. We can form G' in linear time. Let $\deg_G(s)$ be the degree of s in G and let $\deg_{G'}(s)$ be the degree of s in G'. We can compute the degrees easily in linear time by going over the adjacency lists. From the previous part we see that a vertex s is contained in a cycle iff $\deg_G(s) \neq \deg_{G'}(s)$. Thus, by scanning the degrees of the vertices, in $O(n)$ time we can find all vertices in G that are in a cycle. ■

Solution: Alternatively, we consider the graph $G'' = (V, E \setminus Z)$. We can form this graph in linear time. The isolated vertices in G'' (the ones with degree 0) are precisely the ones which are *not* in a cycle.

```
FindAllVerticesInCyles(G):
  Use known algorithm to find all cut edges Z in G
  Obtain graph G'' by removing edge set Z from G
  S ← {v ∈ V | deg_{G''}(v) ≠ 0}
  return S
```

■

Rubric: Give half credit for minor errors.

- 1 point each for Parts 1 & 2
- 2 points each for Parts 3–6.