Depth First Search

Exploring a maze
- A maze can be represented using a graph
- Change in terminology
  - maze ≈ graph
  - passage ≈ edge
  - intersection ≈ vertex
- Methods of exploration
  - Theseus and the Minotaur
  - Trémaux exploration

Trémaux Exploration
- Setup
  - Lights at each intersection
    - Initially off
  - Doors at entrance to each passage
    - Initially closed
    - Window in door allows one to determine if the light is on at the intersection on the other side of the passage
    - Open door when check the light at other intersection
  - Goal
    - Open all doors
    - Turn on all lights

Policy
1. If no doors closed at current intersection, go to step 3. Otherwise, open any door and leave it open.
2. If the intersection at the other end is lit, return to step 1. Otherwise, follow passage to next intersection, unrolling string as you go. Open the door, turn on the light, and return to step 1.
3. If all doors at the current intersection are open, check whether you are at the starting point. If so, stop. If not, use the string to retrace your step, and return to step 1.
Proof by Induction of that Trémaux Exploration visits all intersections

- **Base Case**
  - For a maze with one intersection and no passages, we visit all intersections since we begin at the one intersection and turn on the light

- **Inductive Step**
  - Inductive Hypothesis: Assume algorithm works for some maze with \( n \) intersections
  - Prove that we visit all intersections if we add one more intersection
  - Consider the first passage taken, divide intersections into two subsets
    1. The ones we can get to without returning to the starting position (\( n \) or fewer intersections)
    2. The one we cannot reach without returning to the starting position (\( n-1 \) or fewer intersections)

Depth First Search

- Similar to Trémaux exploration
  - Turning on a light ≈ marking a vertex as visited
  - Opening a door ≈ iterating over the adjacent vertices
  - Leaving a string trail ≈ recursive call
  - Retrace steps following string ≈ return

DFS Implementation

- Use an array of ints to record order of visits
- Also used to determine visit order

```c
#define NOT_VISITED -1
typedef struct {
    int *order;       // array the size of the number of vertices
    int count;
} Search;

void dfsR (Graph *theGraph, Search *info, Edge *curEdge) { 
    int nextStep, curPos = curEdge->to; Edge newEdge;
    info->order[curPos] = info->count++;
    for (nextStep = 0; nextStep < theGraph->vertices; nextStep++) {
        if (theGraph->adj[curPos][nextStep] != 0)
            if (info->order[nextStep] == NOT_VISITED)
                dfsR(g, createTime(curPos, nextStep, &newEdge));
    }
}
```

![Diagram of a maze and traversal](image)
Graph-Search ADT

- DFS is a systematic way of visiting every vertex and every edge
- What happens when graph contains more than one connected component?
  - Create forests

Graph Search

// Uses a self-loop to begin the search of a connected component
// Calls dfsR for each component
Search *graphSearch(Graph *theGraph) {
  int curVertex;
  Edge selfLoop;
  Search *info = malloc(sizeof(Search));
  CHECKP(info);
  info->count = 0;
  info->order = calloc(theGraph->vertices, sizeof(int));
  CHECKP(info->order);

  for (curVertex = 0; curVertex < theGraph->vertices; curVertex++)
    info->order[curVertex] = NOT_VISITED;
  for (curVertex = 0; curVertex < theGraph->vertices; curVertex++)
    if (info->order[curVertex] == NOT_VISITED)
      dfsR(theGraph, info, createEdge(curVertex, curVertex, &selfLoop));

  return info;
}

Search and DFS Properties

- A graph-search function checks each edge and marks each vertex in a graph if and only if the search function marks each vertex and checks each edge in the connected component that contains the start vertex.
- DFS of a graph represented with an adjacency matrix requires time proportional to
- DFS of a graph represented with adjacency lists requires time proportional to