Overview

- Depth First Search
- Topical sorting
**Objective**

- Understand and be able to apply the depth first search (DFS) algorithm
- Apply Topological Sorting as graph algorithm

**Knights Tour Problem**

- Puzzle played on chess board with single figure, the knight
- Objective: find sequence of moves that allow knight to visit every square on board “exactly” once
- Such sequence is called “tour”
- Upper bound on possible tours is $1.35 \times 10^{35}$
- Use graph search to solve problem
Knights Tour Problem

Solve problem by using two main steps:

• Represent legal moves of knight on chessboard as graph
• Use a graph algorithm to find path of length $rows \times columns - 1$ where every vertex on graph is visited exactly once

• Each square represented as node in graph
• Each legal move represented by edge
Building the Graph

```python
from Graph import Graph

def knightGraph(bdSize):
    ktGraph = Graph()
    for row in range(bdSize):
        for col in range(bdSize):
            nodeId = posToNodeId(row, col, bdSize)
            newPositions = genLegalMoves(row, col, bdSize)
            for e in newPositions:
                nid = posToNodeId(e[0], e[1], bdSize)
                ktGraph.addEdge(nodeId, nid)
    return ktGraph

def posToNodeId(row, column, board_size):
    return (row * board_size) + column
```

Building the Graph

```python
def genLegalMoves(x, y, bdSize):
    newMoves = []
    moveOffsets = [(-1, -2), (-1, 2), (-2, -1), (-2, 1),
                    (1, -2), (1, 2), (2, -1), (2, 1)]
    for i in moveOffsets:
        newX = x + i[0]
        newY = y + i[1]
        if legalCoord(newX, bdSize) and \
        legalCoord(newY, bdSize):
            newMoves.append(((newX, newY)))
    return newMoves

def legalCoord(x, bdSize):
    if x >= 0 and x < bdSize:
        return True
    else:
        return False
```
**Complete Graph**

- 336 edges
- Less connections for vertices on edges of board
- Sparsity:
  - Fully connected graph: 4096 edges
  - Matrix only 8.2% filled

**Depth First Search (DFS)**

- Solve problem width depth first search (DFS) algorithm
- Creates search tree by exploring one branch of the tree as deeply as possible
- We will look at two algorithms:
  1. Directly solves problem by explicitly forbidding a node to be visited more than once
  2. More general, but allows nodes to be visited more than once as the tree is constructed
Implementing Knight’s Tour

• DFS exploration of graph finds path with exactly 63 edges

• When dead end is found (more moves possible)
  • Algorithm backs up tree to next deepest vertex allowing a legal move

```python
from Graph import Graph, Vertex

def knightTour(n, path, u, limit):
    u.setColor('gray')
    path.append(u)
    if n < limit:
        nbrList = list(u.getConnections())
        i = 0
        done = False
        while i < len(nbrList) and not done:
            if nbrList[i].getColor() == 'white':
                done = knightTour(n+1, path, nbrList[i], limit)
                i = i + 1
            if not done:  # prepare to backtrack
                path.pop()
                u.setColor('white')
        else:
            done = True
    return done
```
DFS – Coloring

- DFS uses colors to keep track which vertices have been visited
  - White: unvisited
  - Gray: visited
- If neighbors of particular vertex have been explored && length of vertices < 64 => dead end reached
- If dead end reached => backtrack (Return from knightTour with false)

DFS – Coloring

- Since DFS is recursive, use stack to help with backtracking
- After return from knightTour with status False:
  - Remain inside while loop
  - Look at next vertex in nbrlist
Simple Example

- Following figures show steps of search
- Assume `getConnections` orders nodes in alphabetical order
- Start with calling `knightTour(0,path,A,6)`

**Simple Example**

- `knightTour` starts with node A (a))
- B and D are adjacent to A
- Since B comes next in alphabet, it is chosen next (b))
- Recursively calling `knightTour` explores B
**Simple Example**

- B is adjacent to C and D
- `knightTour` elects to explore C
- C is dead end with no adjacent white notes (c))
- Change color of C back to white (d))
- Backtracks search to vertex B
Simple Example

• when we get to node C the test \( n < \text{limit} \) fails
• \( \Rightarrow \) all nodes in graph exhausted
• return True to indicate that we have made a successful tour of the graph
• return the list, path has the values \([A, B, D, E, F, C]\), which is the order we need to traverse the graph to visit each node exactly once

• Complete tour around 8 x 8 board
Knight’s Tour - Analysis

• Very sensitive to method used to select next vertex
• Example
  • 5 x 5 board, calculate path in 1.5 second
  • 8 x 8 board, up to ½ hour
• Reason: $O(k^N)$, $N$ is number of squares, $k$ is small constant

Knight’s Tour - Analysis

• Root is starting point of search tree
• Then checks each move knight can make
  • 2 legal moves in corner
  • 3 in squares adjacent to corners
  • 8 in middle of board
Knight’s Tour - Analysis

• Figure shows number of possible moves on board
• Next level of tree has again 2 – 8 next possible moves
• Number of possible positions to examine corresponds to number of nodes in search tree

<table>
<thead>
<tr>
<th>2</th>
<th>3</th>
<th>3</th>
<th>4</th>
<th>4</th>
<th>4</th>
<th>4</th>
<th>3</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>4</td>
<td>8</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>6</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>6</td>
<td>6</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>6</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>6</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>6</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>6</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>8</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>4</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>

Knight’s Tour - Analysis

• Number of nodes in binary tree is $2^{N+1} - 1$
• Number much larger for tree with up to 8 nodes
• Use average branch factor to estimate number of child nodes: $k^{N+1} - 1$, $k$ is average branching factor
• Example:
  • 5 x 5 board, tree is 25 levels deep => $N=24$
  • $k=3.8$ => $3.8^{25}-1 = 3.12 \times 10^{14}$
Knight’s Tour - Analysis

• Way to speed up 8 x 8 case => runs in less than 1 second

• `orderByAvail` will be called used instead of `u.getConnections` (shown in previous code)

• Line 10 is critical one, it ensures to select vertex that has *fewest* available moves

• But why not select node that has *most* available moves?

```python
def orderByAvail(n):
    resList = []
    for v in n.getConnections():
        if v.getColor() == 'white':
            c = 0
            for w in v.getConnections():
                if w.getColor() == 'white':
                    c = c + 1
            resList.append((c, v))
    resList.sort(key=lambda x: x[0])
    return [y[1] for y in resList]
```
Knight’s Tour - Analysis

• Problem with using vertex with most available moves => tends to have knight visit middles squares early on
  • Easy for night to get stranded on one side of board and cannot reach other side.
• Visiting squares with fewest available moves first pushes knight to visit squares around edges
• Using intuition is called heuristic!

General Depth First Search

• Implementation extends graph class by adding:
  • Time instance variable and methods dfs and dfsvisit
  • dfs method iterates over all vertices in graph calling dfsvisit on white nodes
  • This ensures all nodes in graph are considered and no vertices are left out of depth first forest
General Depth First Search

from Graph import Graph, Vertex
class DFSGraph(Graph):
    def __init__(self):
        super().__init__()
        self.time = 0

    def dfs(self):
        for aVertex in self:
            aVertex.setColor('white')
            aVertex.setPred(-1)
        for aVertex in self:
            if aVertex.getColor() == 'white':
                self.dfsvisit(aVertex)

    def dfsvisit(self, startVertex):
        startVertex.setColor('gray')
        self.time += 1
        startVertex.setDiscovery(self.time)
        for nextVertex in startVertex.getConnections():
            if nextVertex.getColor() == 'white':
                nextVertex.setPred(startVertex)
                self.dfsvisit(nextVertex)
        startVertex.setColor('black')
        self.time += 1
        startVertex.setFinish(self.time)

• DFS method starts with single vertex `startVertex` and explores all neighboring white vertices as deeply as possible
• `dfsvisit` is almost identical to `bfsexcept`
• `dfsvisit` uses a stack where `bfsexcept` uses queue
  • Not visible in code but implicit of `dfsvisit`
General Depth First Search

- Following sequence of figures illustrates DFS in action
- Dotted lines indicate checked edges but node on other end of edge has already been added to DFS tree
- In the code this is realized by checking that color of the other node is non-white

Search begins at vertex A
- Since all vertices are white algorithm visits vertex A
  1. Set color of vertex A gray => vertex is being explored
  2. Discovery time is set to 1
  3. Neighbors B and D need to be visited as well
  4. Arbitrary decision to visit adjacent nodes in alphabetic order
General Depth First Search

• Vertex B is visited next
  1. Its color is set to gray
  2. Discovery time is set to 2
  3. B is adjacent to C and D
  4. Visit vertex C next

General Depth First Search

• Visiting C brings alg. to end of branch of tree
  1. Color node gray and set discovery time to 3
  2. No adjacent vertices to C
  3. Color vertex black, set finish time to 4
General Depth First Search

- Now return to B and explore nodes adjacent to it
- Only addition vertex is D
  - Visit D and continue search
  - Results in exploring E, which has adjacent vertices B and F
  - B is already colored, thus explore F

F has only adjacent vertex C
- C already colored black
- Nothing else to explore
- Reached end of branch
- Algorithm works its way back to first node
  - setting finish times and
  - coloring vertices black
General Depth First Search

- Start and finishing times are called *parentheses property*
- All children of particular node in DFS
  - Have later discovery time than parent
  - Have earlier finish time than parent
- Figure shows final tree constructed by DFS algorithm
General Depth First Search

- General running time:
  - Loops in dfs run in $O(V)$, since executed once for each vertex in graph
  - Since dfsvisit only called recursively if vertex is white, loop will execute max. once for every edge in graph => $O(E)$
- Total time for DFS is $O(V+E)$

Topological Sorting

- Demonstrate that almost anything can be turned into a graph problem
- Consider problem of stirring up batch of pancakes
  - Recipe: 1 egg, 1 cup of pancake mix, 1 tablespoon oil and $\frac{3}{4}$ cup of milk
  - Heat griddle, mix all ingredients together, and spoon mix onto hot griddle
  - When pancakes start bubbling, turn them over
- Heat up syrup
Topological Sorting

- Here the process is illustrated as a graph

Problem: Know what to do first

- Start by heating griddle or adding any of ingredients to pancake mix
- To make that decision we turn to algorithm called topological sort
Topological Sorting

- Topological sort takes DAG and produces linear ordering of all vertices such that
  - If graph contains edge \((v, w)\) then vertex \(v\) comes before vertex \(w\).
- Other examples besides pancakes:
  - project schedules
  - Multiplying matrices

Algorithm for Topological Sort (adaptation of DFS):

1. Call \(\text{dfs}(g)\) for some graph \(g\). Main reason, call finish times for each vertex
2. Store vertices in a list in decreasing order of finish time
3. Return the ordered list as the result of the topological sort
Topological Sorting

• Tree constructed by DFS

Topological Sorting

• Result of applying topological sorting to graph

• Now we know exactly order in which to make pancakes
Next Steps

• Next lecture on Tuesday: State Machines