KNIGHT’S TOUR

Section Overview

- Explain the Knight’s Tour Problem
- Use Graphs to set up the solution
- View a solution in Python
- Continue on to Depth First Search

Knight’s Tour

- The knight’s tour puzzle is played on a chess board with a single chess piece, the knight.
- The object of the puzzle is to find a sequence of moves that allow the knight to visit every square on the board exactly once.

Knight’s Tour

- We will solve the problem using two main steps:
  - Represent the legal moves of a knight on a chessboard as a graph.
  - Use a graph algorithm to find a path of length **rows×columns−1** where every vertex on the graph is visited exactly once.

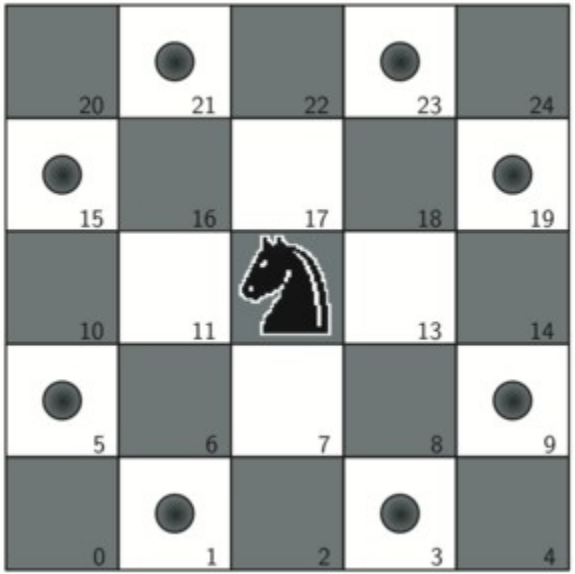
Knight’s Tour

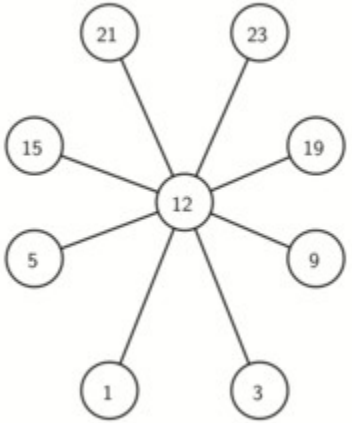
- We will solve the problem using two main steps:
  - Represent the legal moves of a knight on a chessboard as a graph.
  - Use a graph algorithm to find a path of length **rows×columns−1** where every vertex on the graph is visited exactly once.

Knight’s Tour

- To represent the knight’s tour problem as a graph we will use the following two ideas:
  - Each square on the chessboard can be represented as a node in the graph.
  - Each legal move by the knight can be represented as an edge in the graph.

Knight’s Tour





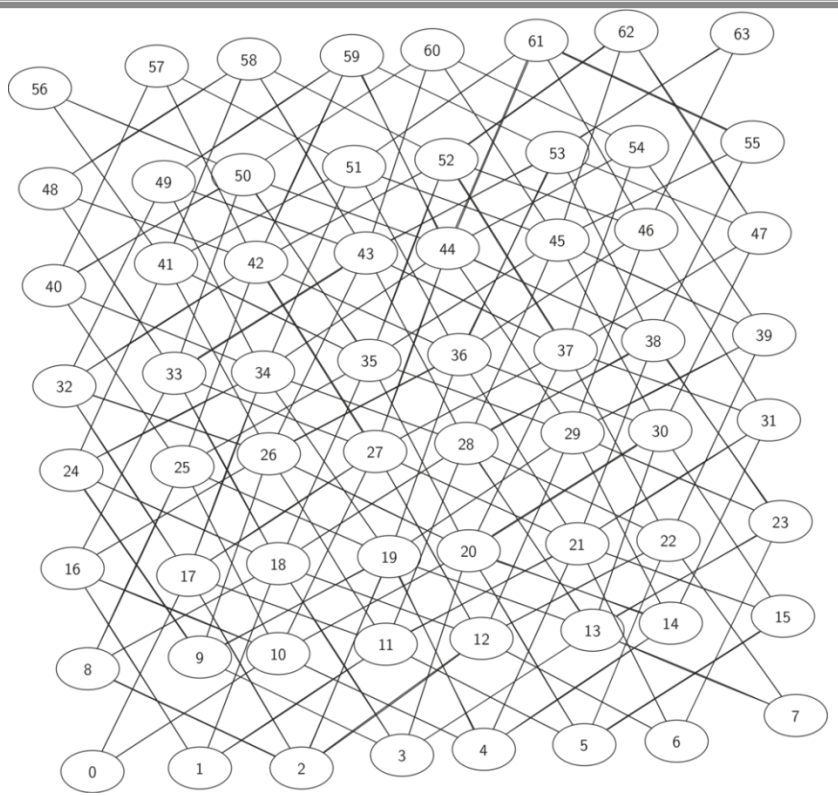
Knight’s Tour

```
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
```

Knight’s Tour

```
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
```

Knight’s Tour



Knight’s Tour

- The search algorithm we will use to solve the knight’s tour problem is called **depth first search** (**DFS**).
- Whereas the breadth first search algorithm discussed in the previous section builds a search tree one level at a time, a depth first search creates a search tree by exploring one branch of the tree as deeply as possible.

Knight’s Tour

- In this section we will look at two algorithms that implement a depth first search.
- The first algorithm we will look at directly solves the knight’s tour problem by explicitly forbidding a node to be visited more than once.
- The second implementation is more general, but allows nodes to be visited more than once as the tree is constructed. (This will be the general DFS discussed later)

Knight’s Tour

- The depth first exploration of the graph is exactly what we need in order to find a path that has exactly 63 edges.
- We will see that when the depth first search algorithm finds a dead end (a place in the graph where there are no more moves possible) it backs up the tree to the next deepest vertex that allows it to make a legal move.

Knight’s Tour

- The **knightTour** function takes four parameters:
  - **n**, the current depth in the search tree
  - **path**, a list of vertices visited up to this point
  - **u**, the vertex in the graph we wish to explore
  - **limit** the number of nodes in the path.
- The **knightTour** function is recursive.

Knight’s Tour

```
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
```

Knight’s Tour

- When the **knightTour** function is called, it first checks the base case condition.
- If we have a path that contains 64 vertices, we return from **knightTour** with a status of True, indicating that we have found a successful tour.
- If the path is not long enough we continue to explore one level deeper by choosing a new vertex to explore and calling **knightTour** recursively for that vertex.

Knight’s Tour

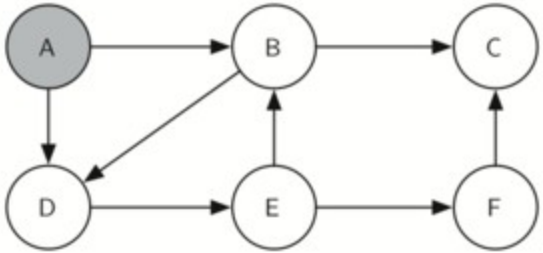
- DFS also uses colors to keep track of which vertices in the graph have been visited.
- Unvisited vertices are colored white, and visited vertices are colored gray.
- If all neighbors of a particular vertex have been explored and we have not yet reached our goal length of 64 vertices, we have reached a dead end.
- When we reach a dead end we must backtrack

Knight’s Tour

- Backtracking happens when we return from **knightTour** with a status of **False**.
- In the breadth first search we used a queue to keep track of which vertex to visit next.
- Since depth first search is recursive, we are implicitly using a stack to help us with our backtracking.
- When we return from a call to **knightTour** with a status of **False** we remain inside the while loop and look at the next vertex in **nbrList**.

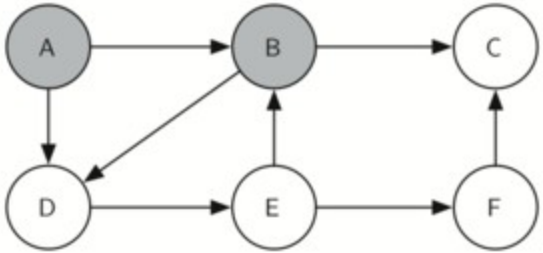
Knight’s Tour

- Knight's tour example



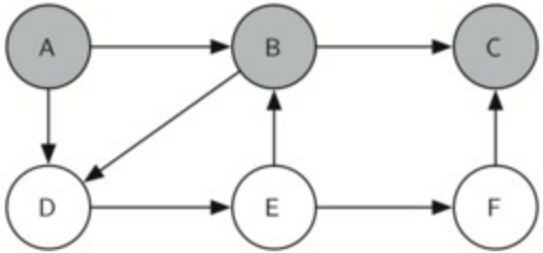
Knight’s Tour

- Knight's tour example



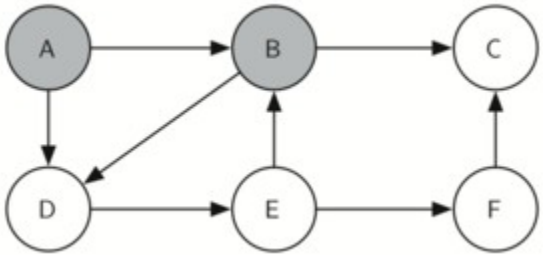
Knight’s Tour

- Knight's tour example



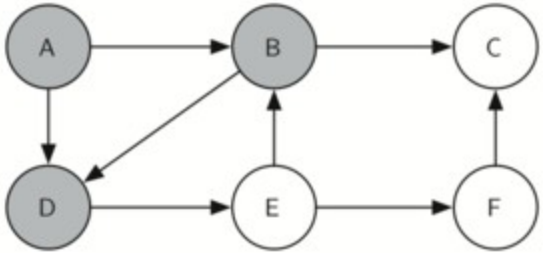
Knight’s Tour

- Knight's tour example



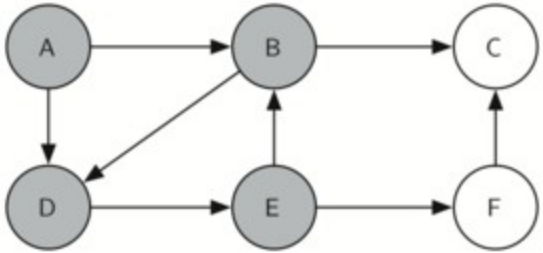
Knight’s Tour

- Knight's tour example



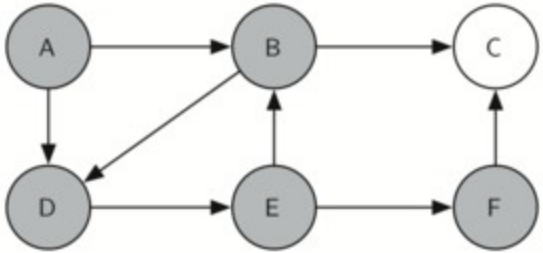
Knight’s Tour

- Knight's tour example



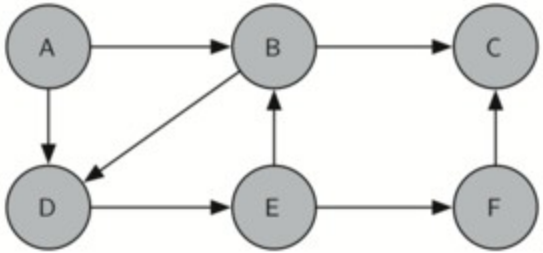
Knight’s Tour

- Knight's tour example



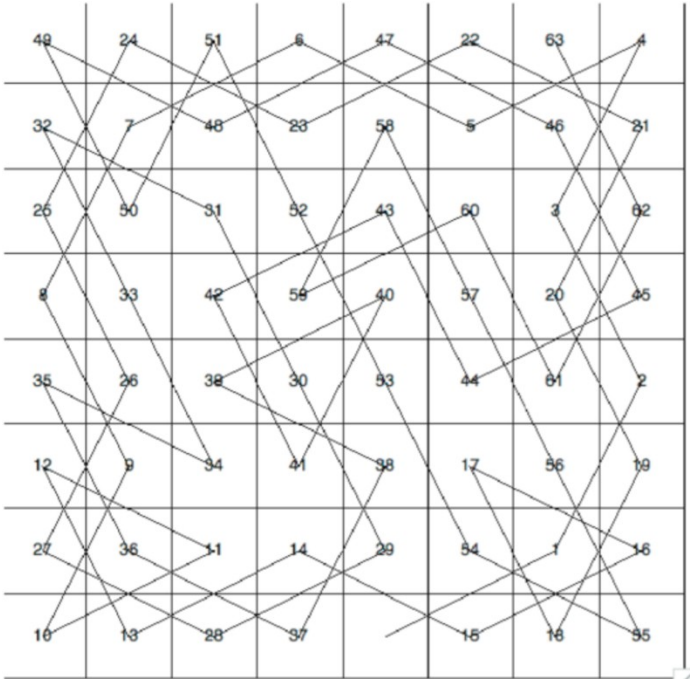
Knight’s Tour

- Knight's tour example



Knight’s Tour

- Knight's tour example



Knight’s Tour

- We’ve gotten a brief overview of possible special use case for depth first search.
- Up next general DFS
- Afterwards full walkthrough implementations