Graphs

DFS and Friends

Brad Miller David Ranum[1]

[1] Department of Computer Science
Luther College

12/19/2005

Outline

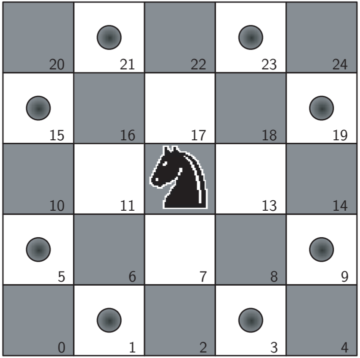
- Depth First Search
- Strongly Connected Components
- Topological Sorting

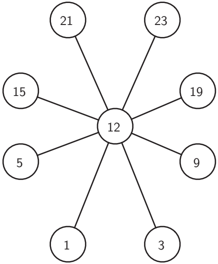
Outline

- Depth First Search
- Strongly Connected Components
- Topological Sorting

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

Legal Moves for a Knight on Square 12, and the Corresponding Graph





Create a Graph Corresponding to All Legal Knight Moves I

```
def knightGraph(bdSize):
    ktGraph = Graph()
    # Build the 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])
                ktGraph.addEdge(nodeId,nid)
    return ktGraph
```

Generate a List of Legal Moves for a Chess Board Position I

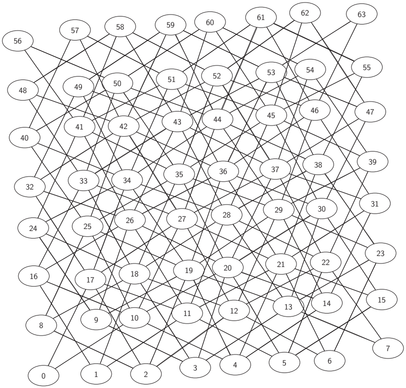
```
def genLegalMoves(x,y,bdSize):
    newMoves = []
    moveOffsets = [(-1,-2),(-1,2),(-2,-1),(-2,1),
                   ( 1,-2),( 1,2),( 2,-1),( 2,1)]:
    for i in moveOffsets:
        newX = x + i[0]
        newY = y + i[1]
        if legalCoord(newX,bdSize) and \
                        legalCoord(newY,bdSize):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

def legalCoord(x,bdSize):
    if x >= 0 and x < bdSize:
        return True
```

Generate a List of Legal Moves for a Chess Board Position II

```
    else:
        return False
```

All Legal Moves for a Knight on an 8×8 Chessboard



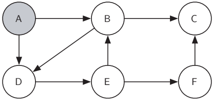
Depth First Search Algorithm for Knights Tour I

```
def knightTour(n,path,u,limit):
        u.setColor('gray')
        path.append(u)
        if n < limit:
            nbrList = orderByAvail(u)
            i = 0
            done = False
            while i < len(nbrList) and not done:
                if nbrList[i].getColor() == 'white':
                    done = knightTour(n+1,
                                      path,
                                      nbrList[i],
                                      limit)
            if not done:  # prepare to backtrack
                path.remove(u)
                u.setColor('white')
        else:
```

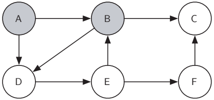
Depth First Search Algorithm for Knights Tour II

```
                done = True
        return done
```

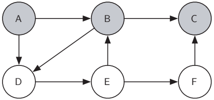
Finding a Path Through a Graph with `knightTour`



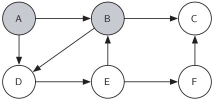
(a) Start with node A



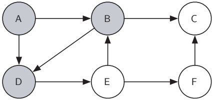
(b) Explore B



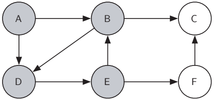
(c) node C is a dead end



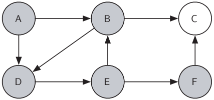
(d) backtrack to B



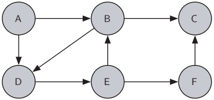
(e)



(f)

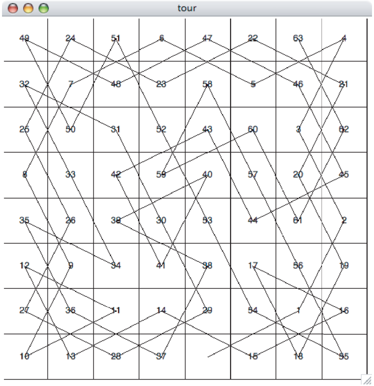


(g)



(h) finish

A Complete Tour of the Board



Selecting the Next Vertex to Visit Is Critical

```
def orderByAvail(n):
    resList = []
    for v in n.getAdj():
        if v.getColor() == 'white':
            c = 0
            for w in v.getAdj():
                if w.getColor() == 'white':
                    c = c + 1
            resList.append((c,v))
    resList.sort()
    return [y[1] for y in resList]
```

General Depth First Search I

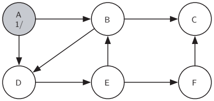
```
def dfs(theGraph):
    for u in theGraph:
        u.setColor('white')
        u.setPred(-1)
    time = 0
    for u in theGraph:
        if u.getColor() == 'white':
            dfsvisit(u)

def dfsvisit(s):
    s.setDistance(0)
    s.setPred(None)
    S = Stack()
    S.push(s)
    while (S.size() > 0):
        w = S.pop()
        w.setColor('gray')
```

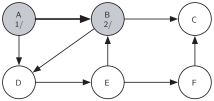
General Depth First Search II

```
for v in w.getAdj():""
    if (v.getColor() == 'white'):
        v.setDistance( w.getDistance() + 1 )
        v.setPred(w)
        S.push(v)
w.setColor('black')
```

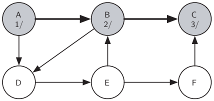
Constructing the Depth First Search Tree



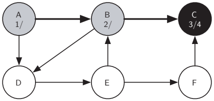
(a)



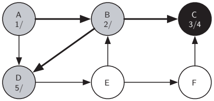
(b)



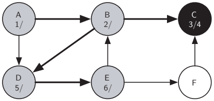
(c)



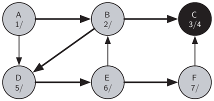
(d)



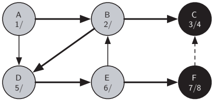
(e)



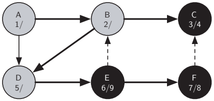
(f)



(g)

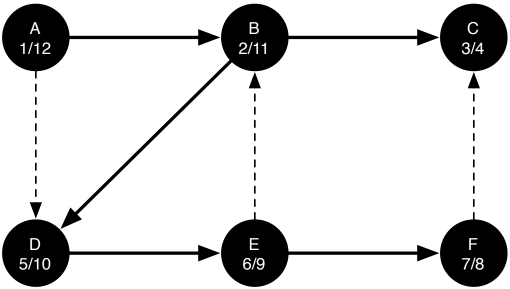


(h)



(i)

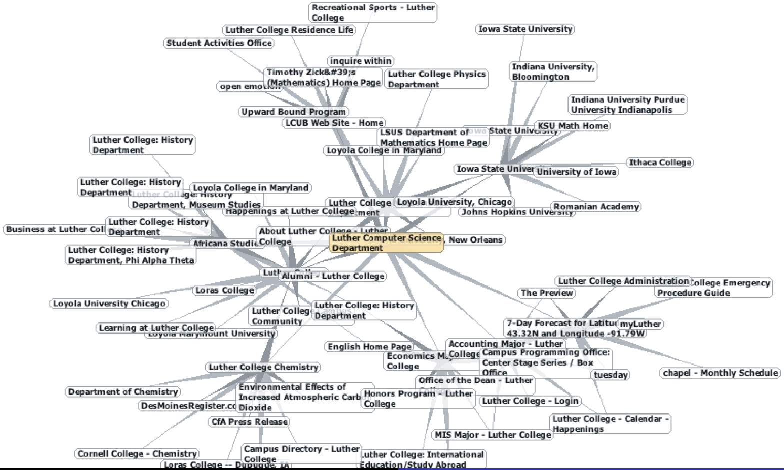
The Resulting Depth First Search Tree



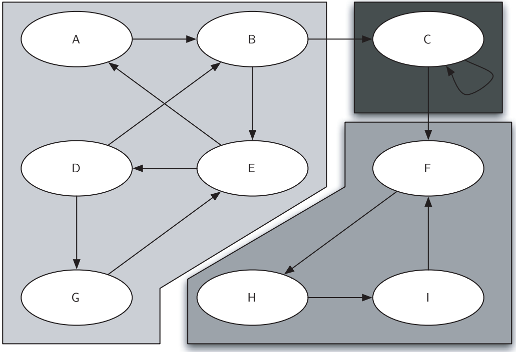
Outline

- Depth First Search
- Strongly Connected Components
- Topological Sorting

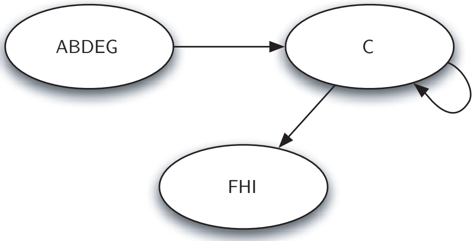
The Graph Produced by Links from the Luther Computer Science Home Page



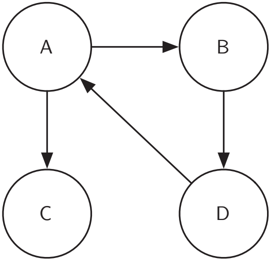
A Directed Graph with Three Strongly Connected Components



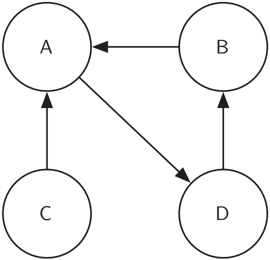
The Reduced Graph



A Graph G and Its Transpose G^T



(m) a graph G

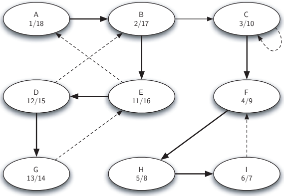


(n) the transposition of G, G^T

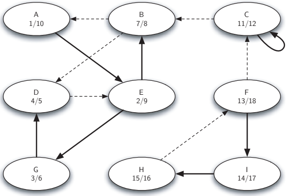
SCC Algorithm

1. Call `dfs` for the graph G to compute the finish times for each vertex.
2. Compute G^T.
3. Call `dfs` for the graph G^T but in the main loop of DFS explore each vertex in decreasing order of finish time.
4. Each tree in the forest computed in step 3 is a strongly connected component. Output the vertex ids for each vertex in each tree in the forest to identify the component.

Computing the Strongly Connected Components

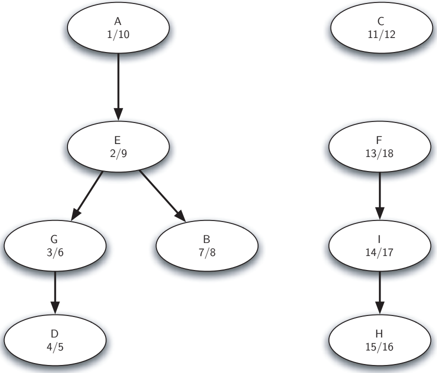


(o) finishing times for the original graph G



(p) finishing times for G^T

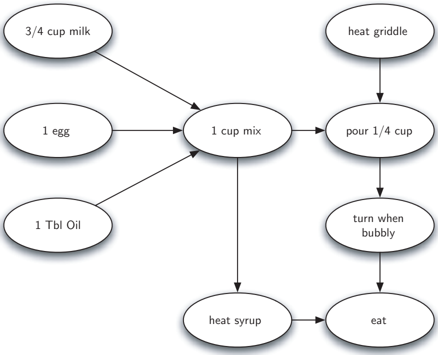
The Strongly Connected Components as a Forest of Trees



Outline

- Depth First Search
- Strongly Connected Components
- Topological Sorting

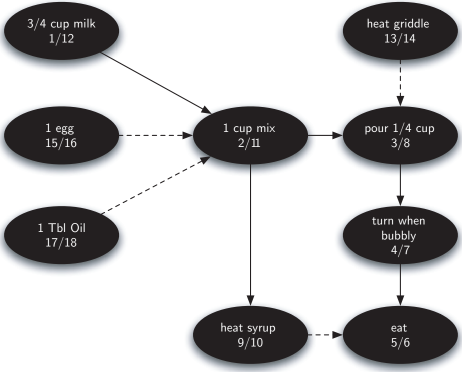
The Steps for Making Pancakes



Topological Sort Algorithm

1. Call `dfs(g)` for some graph `g`. The main reason we want to call depth first search is to compute the finish times for each of the vertices.
2. Order the vertices to a list in decreasing order of finish time.
3. Return the ordered list as the result of the topological sort.

Result of Depth First Search on the Pancake Graph



Result of Topological Sort on Directed Acyclic Graph