AI Lab Solutions - MSc(CA) Sem III

Generated: AI Lab Solutions — concise code & explanation

This PDF contains concise, ready-to-run Python solutions for common practical problems listed in the provided lab slips. Each problem includes a short explanation and sample code. Use Python 3.8+ to run the code. Where helpful, sample usage or small notes are included.

1) Dijkstra's Algorithm (shortest path from single source). [10 marks]

```
import heapq

def dijkstra(graph, source):
    # graph: {u: [(v, w), ...], ...}
    dist = {node: float('inf') for node in graph}
    dist[source] = 0
    parent = {source: None}
    pq = [(0, source)]
    while pq:
        d, u = heapq.heappop(pq)
        if d>dist[u]: continue
        for v,w in graph[u]:
            nd = d + w
            if nd < dist[v]:
                dist[v] = nd
                parent[v] = u
                heapq.heappush(pq, (nd, v))
    return dist, parent

# Example:
# g = {1: [(2,2),(3,4)], 2: [(3,1)], 3: []}
# print(dijkstra(g,1))
```

2) Depth First Search (recursive) — find path from initial to goal. [10–20 marks]

```
def dfs(graph, start, goal, visited=None, path=None):
    if visited is None: visited=set()
    if path is None: path=[]
    visited.add(start)
    path = path + [start]
    if start == goal:
        return path
    for nbr in graph.get(start, []):
        if nbr not in visited:
            res = dfs(graph, nbr, goal, visited, path)
            if res:
                return res
    return None

# Example:
# g = {1:[2,3],2:[4],3:[5],4:[],5:[]}
# print(dfs(g,1,5))
```

3) Breadth First Search — shortest path in unweighted graph. [10 marks]

AI Lab Solutions (continued)

Generated: AI Lab Solutions — concise code & explanation

```
from collections import deque
def bfs_shortest_path(graph, start, goal):
    q = deque([start])
    parent = {start: None}
    while q:
        u = q.popleft()
        if u == goal:
            path=[]
            cur=goal
            while cur is not None:
                path.append(cur)
                cur=parent[cur]
            return list(reversed(path))
        for v in graph.get(u,[]):
            if v not in parent:
                parent[v]=u
                q.append(v)
    return None
```

4) Selection Sort (ascending). [10 marks]

```
def selection_sort(arr):
    a = arr[:]
    n = len(a)
    for i in range(n):
        min_idx = i
        for j in range(i+1,n):
            if a[j] < a[min_idx]:
                min_idx = j
        a[i], a[min_idx] = a[min_idx], a[i]
    return a
# Example: selection_sort([64,25,12,22,11])
```

5) Kruskal's Minimum Spanning Tree (MST). [20 marks]

```
def find(parent, i):
    if parent[i]!=i:
        parent[i]=find(parent,parent[i])
    return parent[i]

def union(parent, rank, x, y):
    xroot, yroot = find(parent,x), find(parent,y)
    if rank[xroot] < rank[yroot]:
        parent[xroot]=yroot
    elif rank[xroot] > rank[yroot]:
        parent[yroot]=xroot
    else:
        parent[yroot]=xroot
        rank[xroot]+=1

def kruskal(nodes, edges):
    # nodes: list of nodes, edges: [(u,v,w), ...]
```

```
    parent = {n:n for n in nodes}
    rank = {n:0 for n in nodes}
    mst=[]
    edges_sorted = sorted(edges, key=lambda x: x[2])
    for u,v,w in edges_sorted:
        if find(parent,u)!=find(parent,v):
            union(parent, rank, u, v)
            mst.append((u,v,w))
    return mst
```

6) Prim's algorithm (minimum spanning tree). [10–20 marks]

```
import heapq
def prim(graph, start):
    # graph: {u: [(v,w),...]}
    visited=set([start])
    edges = []
    for v,w in graph[start]:
        heapq.heappush(edges, (w, start, v))
    mst=[]
    while edges:
        w,u,v = heapq.heappop(edges)
        if v in visited: continue
        visited.add(v)
        mst.append((u,v,w))
        for x,c in graph[v]:
            if x not in visited:
                heapq.heappush(edges,(c,v,x))
    return mst
```

7) A* Search (graph search with heuristic). [20 marks]

```
import heapq
def a_star(start, goal, neighbors, h):
    # neighbors(n) -> list of (n2, cost)
    open_set = [(h(start), 0, start, None)]
    came_from = {}
    gscore = {start: 0}
    closed = set()
    while open_set:
        f, g, node, parent = heapq.heappop(open_set)
        if node in closed: continue
        came_from[node] = parent
        if node == goal:
            path=[]
            cur=node
            while cur is not None:
                path.append(cur)
                cur=came_from[cur]
            return list(reversed(path))
        closed.add(node)
        for nei, cost in neighbors(node):
            if nei in closed: continue
```

```
        tentative = g + cost
        if tentative < gscore.get(nei, float('inf')):
            gscore[nei]=tentative
            heapq.heappush(open_set, (tentative + h(nei), tentative, nei, node))
    return None
```

8) Iterative Deepening Depth First Search (IDDFS). [20 marks]

```
def dls(graph, node, goal, depth, visited):
    if depth==0 and node==goal: return [node]
    if depth>0:
        visited.add(node)
        for v in graph.get(node,[]):
            if v not in visited:
                res = dls(graph, v, goal, depth-1, visited)
                if res:
                    return [node]+res
        visited.discard(node)
    return None

def iddfs(graph, start, goal, max_depth=50):
    for d in range(max_depth+1):
        res = dls(graph, start, goal, d, set())
        if res:
            return res
    return None
```

9) 8-Puzzle using A* (sketch). State as tuple of 9 numbers; heuristic = Manhattan distance.

```
# State represented as tuple of length 9, 0 = blank.
from heapq import heappush, heappop

def manhattan(state):
    dist=0
    for i,val in enumerate(state):
        if val==0: continue
        goal_pos = (val-1)//3, (val-1)%3
        cur_pos = i//3, i%3
        dist += abs(goal_pos[0]-cur_pos[0]) + abs(goal_pos[1]-cur_pos[1])
    return dist

def neighbors_8p(state):
    idx = state.index(0)
    r,c = divmod(idx,3)
    for dr,dc in [(-1,0),(1,0),(0,-1),(0,1)]:
        nr, nc = r+dr, c+dc
        if 0<=nr<3 and 0<=nc<3:
            nidx = nr*3+nc
            lst = list(state)
            lst[idx], lst[nidx] = lst[nidx], lst[idx]
            yield tuple(lst), 1
```

AI Lab Solutions (continued)

Generated: AI Lab Solutions — concise code & explanation

```
def a_star_8p(start, goal=(1,2,3,4,5,6,7,8,0)):
    open_set = [(manhattan(start), 0, start, None)]
    came = {}
    g = {start:0}
    closed=set()
    while open_set:
        f, cost, s, parent = heappop(open_set)
        if s in closed: continue
        came[s]=parent
        if s==goal:
            path=[]
            cur=s
            while cur:
                path.append(cur)
                cur=came[cur]
            return list(reversed(path))
        closed.add(s)
        for ns, c in neighbors_8p(s):
            if ns in closed: continue
            ng = g[s]+c
            if ng < g.get(ns, float('inf')):
                g[ns]=ng
                heappush(open_set, (ng+manhattan(ns), ng, ns, s))
    return None
```

10) Tic-Tac-Toe + Minimax with Alpha-Beta (playable). [20 marks]

```
# Minimal Tic-Tac-Toe and Minimax (X = maximizing player)
def check_winner(board):
    wins = [(0,1,2),(3,4,5),(6,7,8),(0,3,6),(1,4,7),(2,5,8),(0,4,8),(2,4,6)]
    for a,b,c in wins:
        if board[a] and board[a]==board[b]==board[c]:
            return board[a]
    if all(board): return 'Draw'
    return None

def minimax(board, player, alpha=-float('inf'), beta=float('inf')):
    winner = check_winner(board)
    if winner=='X': return 1, None
    if winner=='O': return -1, None
    if winner=='Draw': return 0, None
    best_move=None
    if player=='X':
        value = -float('inf')
        for i in range(9):
            if not board[i]:
                board[i]=player
                score,_ = minimax(board, 'O', alpha, beta)
                board[i]=None
                if score>value:
                    value=score; best_move=i
                alpha=max(alpha,value)
                if alpha>=beta: break
        return value, best_move
```

```
    else:
        value = float('inf')
        for i in range(9):
            if not board[i]:
                board[i]=player
                score,_ = minimax(board, 'X', alpha, beta)
                board[i]=None
                if score<value:
                    value=score; best_move=i
                beta=min(beta,value)
                if alpha>=beta: break
        return value, best_move
```

11) Water-Jug Problem (BFS over states). [20 marks]

```
from collections import deque

def water_jug(cap_a, cap_b, target):
    start = (0,0)
    q = deque([start])
    parent = {start: None}
    while q:
        a,b = q.popleft()
        if a==target or b==target:
            path=[]
            cur=(a,b)
            while cur:
                path.append(cur)
                cur=parent[cur]
            return list(reversed(path))
        states = []
        # fill A
        states.append((cap_a, b))
        # fill B
        states.append((a, cap_b))
        # empty A
        states.append((0, b))
        # empty B
        states.append((a, 0))
        # pour A->B
        transfer = min(a, cap_b-b)
        states.append((a-transfer, b+transfer))
        # pour B->A
        transfer = min(b, cap_a-a)
        states.append((a+transfer, b-transfer))
        for s in states:
            if s not in parent:
                parent[s]= (a,b)
                q.append(s)
    return None
```

12) AO* Algorithm (sketch):
AO* is a best-first search for AND-OR graphs. Represent nodes with OR

children sets and AND nodes that require all children solved. Maintain heuristics and backpropagate solved costs. Implementations are more involved; for practical examination, explain node expansion, cost propagation and show a small hand-solved example.

This document contains concise, exam-oriented solutions. If you want full step-by-step explanations for specific slips (e.g., a particular graph for Dijkstra or DFS with given initial/goal), tell me which slip number(s) you want and I'll produce expanded, annotated solutions for those exact problems.