Algorithms & Datastructures
Laboratory Exercise Sheet 12

Wolfgang Pausch <wolfgang.pausch@uibk.ac.at>
Heiko Studt <heiko.studt@uibk.ac.at>
René Thiemann <rene.thiemann@uibk.ac.at>
Tomas Vitvar <tomas.vitvar@uibk.ac.at>

June 15th, to be discussed on June 22nd

Exercise 1) State Space Search The graph in Figure 1 represents a state space where vertices $0, 1, 2, ..., 19$ denote states and edges denote transitions between states. Each state transition has associated costs of that transition (which can be ignored for this exercise, except for `SuccessorFn`). For the following implementation tasks use and/or modify the code skeleton given below.

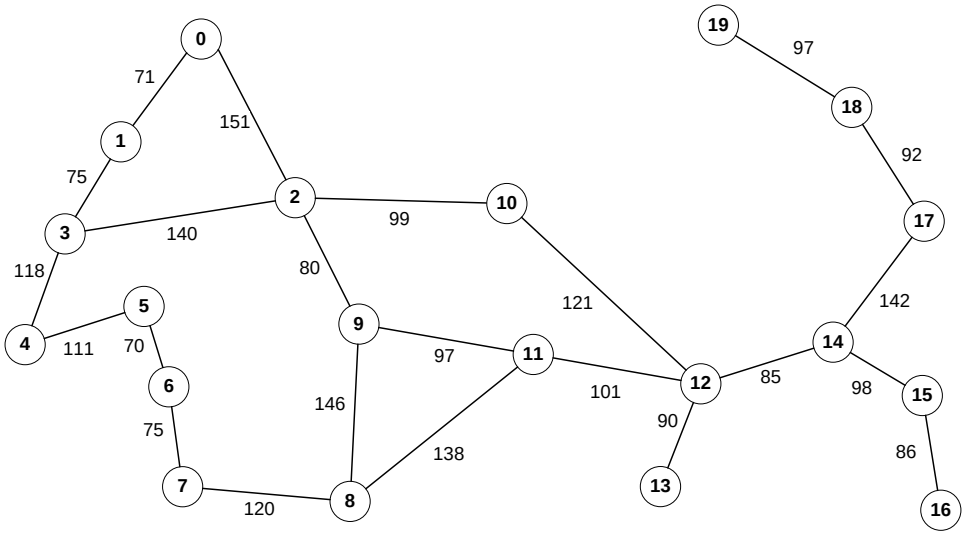

Figure 1: State Space

1. Implement a method `SuccessorFn(State x)` which returns a set of successor states for the state `x`. Note, that the state space is a non-directed graph, thus the successor function should always take into account the complete incidence matrix.

2. Implement a method `findPath_BFS(int initial, int goal)` using breadth-first search and `findPath_DFS(int initial, int goal)` using depth-first search. The methods should return a path (i.e, sequence of states) from an initial state to a goal state together with the total length. Use previously implemented successor function in both implementations and take care of repeating states.

3. What happens if you do not take care of repeating states?

4. Implement a method `findPath_DFS(int initial, int goal, int depth_limit)` using a depth-limited search. Depth-limited search is a depth-first search with some predetermined depth limit (that is, the algorithm stops when it reaches the depth limit in the search tree).

5. Implement a method `findPath_IDS(State initial, State goal)` using iterative deepening depth-first search. Iterative deepening depth-first search is a strategy that uses depth-limited search and iteratively increases the limit until it finds a solution.

6. Prove that iterative deepening depth-first search is faster than breath-first search.

```
public class State {

    int id;
    // implement me!
}
```

```
public class StateSpace {

    int[][] data;

    public StateSpace() {
        data = new int[20][20];

        data[0][1]   = 71;  data[0][2]   = 151;
        data[1][3]   = 75;  data[2][3]   = 140;
        data[3][4]   = 118; data[4][5]   = 111;
        data[5][6]   = 70;  data[6][7]   = 75;
        data[7][8]   = 120; data[8][9]   = 146;
        data[2][9]   = 80;  data[2][10]  = 99;
        data[9][11]  = 97;  data[8][11]  = 138;
        data[11][12] = 101; data[10][12] = 121;
        data[12][13] = 90;  data[12][14] = 85;
        data[14][15] = 98;  data[15][16] = 86;
        data[14][17] = 142; data[17][18] = 92;
        data[18][19] = 97;

        data[1][0]   = 71;  data[2][0]   = 151;
        data[3][1]   = 75;  data[3][2]   = 140;
        data[4][3]   = 118; data[5][4]   = 111;
        data[6][5]   = 70;  data[7][6]   = 75;
        data[8][7]   = 120; data[9][8]   = 146;
        data[9][2]   = 80;  data[10][2]  = 99;
        data[11][9]  = 97;  data[11][8]  = 138;
        data[12][11] = 101; data[12][10] = 121;
        data[13][12] = 90;  data[14][12] = 85;
        data[15][14] = 98;  data[16][15] = 86;
        data[17][14] = 142; data[18][17] = 92;
        data[19][18] = 97;
    }

    public void print() {
        System.out.print(String.format("%3s |"," "));
        for (int i = 0; i < data[0].length; i++)
            System.out.print(String.format("%5d",i));
        System.out.println();
        for (int i = 0; i < data[0].length + 1; i++)
```

```
            System.out.print(String.format("%s","-----"));
        System.out.println();

        for (int i = 0; i < data[0].length; i++) {
            System.out.print(String.format("%3d |",i));
            for (int j = 0; j < data[i].length; j++) {
                System.out.print(String.format("%5d",data[i][j]));
            }
            System.out.println();
        }
    }

    public List<State> successorFn(State x) {
        // implement me!
    }

    public State findPath_DFS(int initial, int goal) {
        // implement me!
    }

    public State findPath_DFS(int initial, int goal, int depth_limit) {
        // implement me!
    }

    public State findPath_BFS(int initial, int goal) {
        // implement me!
    }

    public State findPath_IDS(int initial, int goal) {
        // implement me!
    }

}
```

Exercise 1) Solution

1. `State` class and `SuccessorFn(State x)`:

```
public class State {

    int id;
    State parent;
    int path_cost;
    int depth;

    public State(int id, State parent, int path_cost, int depth) {
        this.id = id;
        this.parent = parent;
        this.path_cost = path_cost;
        this.depth = depth;
    }

}
```

```
    public List<State> successorFn(State x) {
        ArrayList<State> states = new ArrayList<State>();
        for (int j = 0; j < data[x.id].length; j++)
            if (data[x.id][j] != 0)
                states.add(new State(j, x, x.path_cost + data[x.id][j
                    ], x.depth + 1));
        return states;
    }
```

2. `findPath_BFS(int initial, int goal)` (repeating states are handled using the `closed` list):

```
    public State findPath_BFS(int initial, int goal) {
        // create structures
        Hashtable<Integer, State> closed = new Hashtable<Integer,
            State>();
        Queue<State> q = new LinkedList<State>();
        State s = new State(initial, null, 0, 0);
        q.add(s);
        closed.put(s.id, s);

        // search the space
        while (!q.isEmpty()) {
            s = q.poll();
            if (s.id == goal)
                return s;

            List<State> successors = successorFn(s);
            for (State x : successors)
                if (closed.get(x.id) == null) {
                    q.add(x);
                    closed.put(x.id, x);
                }
        }
        return null;
    }
```

3. `findPath_DFS(int initial, int goal)`: it is the depth-limited search (see below) where the limit is not specified (we indicate that the limit is not specified by the value of -1).

```
public State findPath_DFS(int initial, int goal) {
    // unlimited depth
    return findPath_DFS(initial, goal, -1);
}
```

4. If you do not take care of repeating states, the algorithm would never end in cases where the path does not exist. Otherwise the algorithm would search states that have already been searched several times.

5. `findPath_DFS(int initial, int goal, int depth_limit)`:

```
public State findPath_DFS(int initial, int goal, int depth_limit)
    {
    // create structures
    Hashtable<Integer, State> closed = new Hashtable<Integer,
        State>();
    Stack<State> stack = new Stack<State>();
    State s = new State(initial, null, 0, 0);
    stack.add(s);
    closed.put(s.id, s);

    // search the space
    while (!stack.empty()) {
        s = stack.pop();
        if (s.id == goal)
            return s;

        if (depth_limit == -1 || s.depth < depth_limit) {
            List<State> successors = successorFn(s);
            for (State x : successors)
                if (closed.get(x.id) == null) {
                    stack.push(x);
                    closed.put(x.id, x);
                }
        }
    }
    return null;
}
```

6. `findPath_IDS(State initial, State goal)`: The algorithm increases the limit by one and performs depth-limited search on that value. As a maximum possible length of a path between two states in the given graph is 19 (without much of the thinking), the algorithm performs a maximum of 19 iterations. However, proper studying of the graph will reveal that the maximum length of a path between any state to any other state is 9 (so called diameter). The maximum number of iterations in this particular could case could thus be set to this value.

```
public State findPath_IDS(int initial, int goal) {
    // the longest path between two states on the given graph is
        19
    // although using diameter it is possible to discover that
    // any state can be reached from any state in 9 steps
    for (int d = 0; d < data[0].length - 1; d++) {
        State s = findPath_DFS(initial, goal, d);
        if (s != null)
            return s;
    }
    return null;
}
```

7. Proof: Let d be the depth of the search tree where the algorithm finds a solution, and b be a maximum number of states generated by the successor function (so called the *branching factor*). In iterative depth-first search (IDS), the algorithm generates nodes at the bottom level once, those on the next to bottom level are generated twice, etc. up to the children of the root, which are generated d times. The total number of nodes generated is $N(IDS) = (d)b + (d-1)b^2 + ... + (1)b^d$ which gives $O(b^d)$.

   In breath-first search (BFS), the root generates b nodes at the first level, b^2 at the second level, b^3 at the third level, etc. If the solution is at the depth d, the algorithm would generate all nodes at the last level but not a node that is the goal node (the goal itself is not expanded) generating $b^{d+1} - b$ at the level $d+1$. Thus, the total number of nodes generated is $N(BFS) = b + b^2 + ... + b^d + (b^{d+1} - b)$ which gives $O(b^{d+1})$.

   Note that BFS generates some nodes at $d+1$ level whereas IDS does not. From this reason, the IDS is faster than BFS (despite the fact that IDS searches repeated states). For example, for $b = 10$ and $d = 5$ you have: $N(IDS) = 40+400+3000+20000+100000 = 123450$, $N(BFS) = 10+100+1000+10000+100000+999990 = 1111100$. IDS is a preferred method for state spaces of large size without a knowledge about the depth of the solution.

Exercise 2) Strongly Connected Components

1. Given an example of a graph having at least two strongly connected components.
2. Apply the algorithm from the lecture to find all strongly connected components of the graph given in Figure 4. Show the steps of the algorithm.

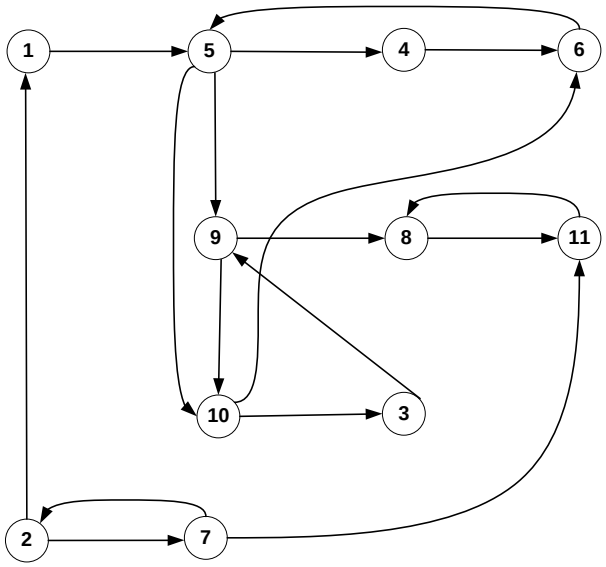

Figure 2: Graph

Exercise 2) Solution

1. An example of a graph having strongly connected components $C1$ and $C2$.

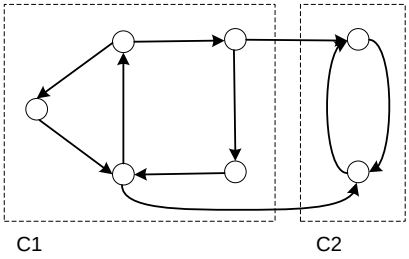

Figure 3: Strongly connected components

2. For a graph G perform following steps:

   (a) Perform a DFS on G and store the finishing times.

v	1	2	3	4	5	6	7	8	9	10	11
$d[v]$	1	19	13	3	2	4	20	8	7	12	9
$f[v]$	18	22	14	6	17	5	21	11	16	15	10

   (b) Create a transpose graph G' from G by reversing directions of all edges in G.

   (c) Start a DFS on G' beginning with vertices with the largest finishing times.

v	1	2	3	4	5	6	7	8	9	10	11
$d[v]$	5	1	13	9	7	8	2	19	12	11	20
$f[v]$	6	4	14	10	18	17	3	22	15	16	21
$parent[v]$			9	6		5	2		10	6	8

   Hence, the DFS-trees with roots 1, 2, 5, and 8 represent the four SCCs.

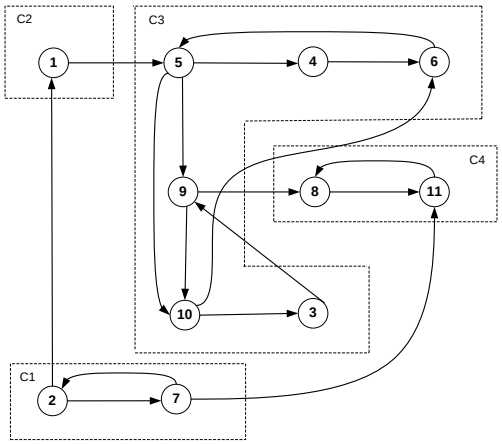

Figure 4: Strongly connected components of the graph.