Department of Computing and Information Systems
COMP90038 Algorithms and Complexity Tutorial Week 5

Sample answers

The exercises

1. *Sparse* graphs $\langle V, E\rangle$ are ones for which $|E| \in O(|V|)$, that is, the number of edges stay, asymptotically, in the order of the number of nodes. Which representation is better for sparse graphs, adjacency matrices or adjacency lists?

   Answer: Adjacency lists are more appropriate for sparse graphs, as the space requirements are less ($O(|V|+|E|)$ as opposed to $O(|V|^2)$), and graph algorithms that are variants of graph traversal can capitalize on this.

2. If we have a finite collection of (infinite) straight lines in the plane, those lines will split the plane into a number of (finite and/or infinite) regions. Two regions are *neighbours* iff they have an edge in common. Show that, for any number of lines, and no matter how they are placed, it is possible to colour all regions, using only two colours, so that no neighbours have the same colour.

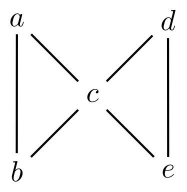
   Answer: We argue this inductively. It is clear that two colours suffice in the case of a single line. Now assume two colours suffice for n lines ($n \geq 1$). When we add line number $n+1$, we can modify the current colouring as follows. On one side of the new line, change the colour of each region. On the other side, do nothing. This gives a valid colouring. Namely, consider two arbitrary neighbouring regions. Either their common border is part of the new line, or it isn't. If it is, then the regions formed a single region before the new line was added, and since the colour of one side was reversed, the two parts now have different colours. If their border is not part of the new line, the two regions were on the same side of the new line. But then they must have different colours, as that was the case before the new line was added (they may have swapped their colours, but that's fine).

3. Draw the undirected graph whose adjacency matrix is

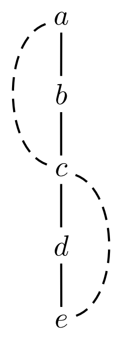
	a	b	c	d	e
a	0	1	1	0	0
b	1	0	1	0	0
c	1	1	0	1	1
d	0	0	1	0	1
e	0	0	1	1	0

   Starting at node a, traverse the graph by depth-first search, resolving ties by taking nodes in alphabetical order.

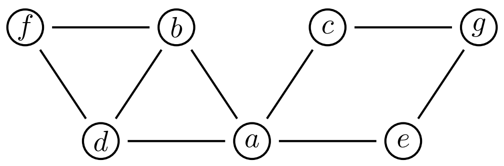
   Answer: Here is the graph:



   In a depth-first search, the nodes are visited in this order: a, b, c, d, e. The depth-first search tree looks as follows:


a
b
c
d
e

4. Consider the following graph:



(a) Write down the adjacency matrix representation for this graph, as well as the adjacency list representation (assume nodes are kept in alphabetical order in the lists).

(b) Starting at node a, traverse the graph by depth-first search, resolving ties by taking nodes in alphabetical order. Along the way, construct the depth-first search tree. Give the order in which nodes are pushed onto to traversal stack, and the order in which they are popped off.

(c) Traverse the graph by breadth-first search instead. Along the way, construct the depth-first search tree.

Answer:

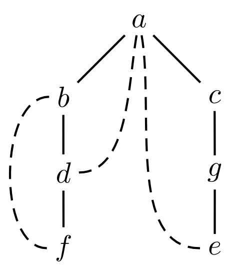
(a) Here is the adjacency matrix:

	a	b	c	d	e	f	g
a	0	1	1	1	1	0	0
b	1	0	0	1	0	1	0
c	1	0	0	0	0	0	1
d	1	1	0	0	0	1	0
e	1	0	0	0	0	0	1
f	0	1	0	1	0	0	0
g	0	0	1	0	1	0	0

The adjacency list representation:

a	$\to b \to c \to d \to e$
b	$\to a \to d \to f$
c	$\to a \to g$
d	$\to a \to b \to f$
e	$\to a \to g$
f	$\to b \to d$
g	$\to c \to e$

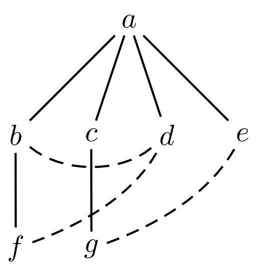
(b) In a depth-first search, the nodes are visited in this order: a, b, d, f, c, g, e.



The order of actions: push a, push b, push d, push f, pop f, pop d, pop b, push c, push g, push e, pop e, pop g, pop c, pop a. The traversal stack develops like so:

$$\begin{array}{ll} f_{4,1} & e_{7,4} \\ d_{3,2} & g_{6,5} \\ b_{2,3} & c_{5,6} \\ a_{1,7} & \end{array}$$

(c) In a breadth-first search, the nodes are visited in this order: a, b, c, d, e, f, g.



5. In the lectures, we discussed how to adapt depth-first traversal to decide whether an undirected graph is cyclic. We identified this complication: Suppose we are at node v and the call DfsExplore(v) takes us to the previously un-visited node w. As part of the cycle test we wanted to check whether w has some neighbour that has already been visited (and if so, classify the graphs as cyclic). However, v is a neighbour of w and v has been visited. So using this approach, every graph that has an edge will be deemed cyclic!

   Write an algorithm to classify all edges of an undirected graph, so that depth-first tree edges can be distinguished from back edges. Then show how this algorithm can be adapted to decide whether an undirected graph is cyclic.

   Answer: Here is how we can classify the edges:

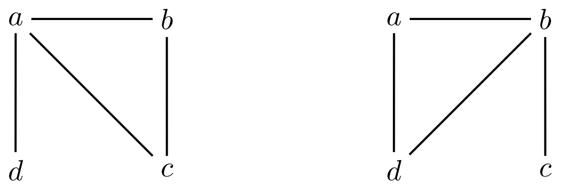
```
function ClassifyEdges($\langle V, E\rangle$)
    mark each node in $V$ with 0
    $count \leftarrow 0$
    for each $v$ in $V$ do
        if $v$ is marked 0 then
            DfsExplore($v$)

function DfsExplore($v$)
    $count \leftarrow count + 1$
    mark $v$ with $count$
    for each edge $(v, w)$ do
        if $w$ is marked with 0 then
            classify $(v, w)$ (and thereby $(w, v)$) as “tree edge”
            DfsExplore($w$)
        else
            if $(v, w)$ (and thereby $(w, v)$) is not a “tree edge” then
                classify $(v, w)$ (and thereby $(w, v)$) as “back edge”
```

   The rest is easy. We just change depth-first exploration slightly:

```
function DfsExplore($v$)
    $count \leftarrow count + 1$
    mark $v$ with $count$
    for each edge $(v, w)$ do                              ▷ $w$ is $v$’s neighbour
        if $w$ is marked with 0 then
            DfsExplore($w$)
        else
            if $(v, w)$ is a back edge then
                halt, and return “cyclic”
```

6. Explain how one can use breadth-first search to see if a graph has cycles. Which of the two traversals, depth-first and breadth-first, will be able to find cycles faster? (If there is no clear winner, give an example where one is better, and another example where the other is better.)

   Answer: A graph has a cycle iff its breadth-first forest contains a cross edge. Sometimes depth-first search finds a cycle faster, sometimes not. Below, on the left, is a case where depth-first search finds the cycle faster, before all the nodes have been visited. On the right is an example where breadth-first search finds the cycle faster.



7. Explain how one can use depth-first search to identify the connected components of a graph.

   Answer: Instead of marking visited node with consecutive integers, we can mark them with a number that identifies their connected component. More specifically, replace the variable *count* with a variable *component*. In *dfs* remove the line that increments *count*. As before, initialise each node with a mark of 0 (for "unvisited"). Here is the algorithm:

```
mark each node in $V$ with 0 (indicates not yet visited)
$component \leftarrow 1$
for each $v$ in $V$ do
    if $v$ is marked with 0 then
        DFS($v$)
        $component \leftarrow component + 1$

function DFS($v$)
    mark $v$ with $component$
    for each vertex $w$ adjacent to $v$ do
        if $w$ is marked with 0 then
            DFS($w$)
```

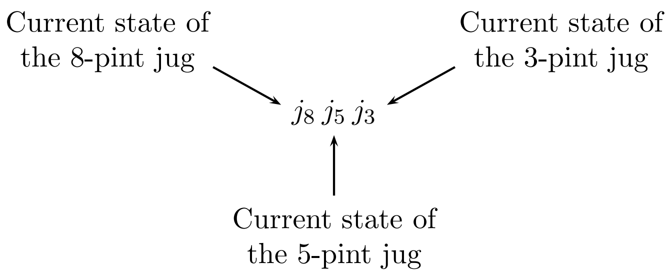
8. Design an algorithm to check whether an undirected graph is 2-colourable, that is, whether its nodes can be coloured with just colours in such a way that no edge connects two nodes of the same colour. Hint: Adapt one of the graph traversal algorithms.

   Answer: Not available

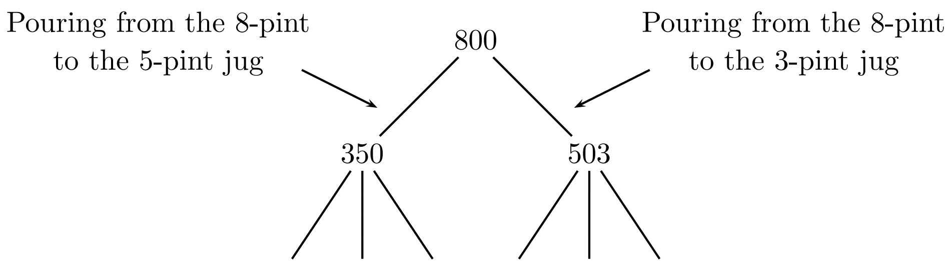
9. Given an 8-pint jug full of water, and two empty jugs of 5- and 3-pint capacity, get exactly 4 pints of water in one of the jugs by completely filling up and/or emptying jugs into others. Solve this problem using breadth-first search.

   Answer: This is commonly known as the "three-jug problem" which can be naturally represented as a graph search/traversal problem. The problem search space is represented as a graph, which is constructed "on-the-fly" as each node is dequeued.

   Each node represents a single problem "state", labelled as a string $j_8 j_5 j_3$ of digits:


Current state of the 8-pint jug
Current state of the 3-pint jug
$j_8\, j_5\, j_3$
Current state of the 5-pint jug

The initial state of the problem is “800”, which means there are 8 pints of water in the 8-pint jug, and no water in the others. A state of “353” denotes that there are 3 pints of water in the 8-pint jug, 5 pints of water in the 5-pint jug, and 3 pints of water in the 3-pint jug. For example, the top part of the search space graph is:


Pouring from the 8-pint to the 5-pint jug
800
Pouring from the 8-pint to the 3-pint jug
350
503

Note that only one action can be performed in a state transition, that is, water can be poured from only one jug between states.

Unlike other examples of graph construction seen so far in the subject, the entire graph of this search space need not be explicitly constructed. Rather, each new state-node is constructed and added to the traversal queue (from the configuration of the current state) as we go.

When each node is dequeued, it is checked to see if it contains a jug with 4 pints of water—that is our “goal” state.

Here is the “plain English” algorithm:

- Create a singleton queue with the initial state of 800.
- While the queue is not empty:
  - Dequeue the current state from the queue
  - Return the path to the current state if it contains a jug with 4 pints; halt
  - Mark the current state as visited.
  - For each state s that is possible from the current state:
    - Add s to the queue.
- Return False if the goal state was not reached.

Below we make this more precise by giving some pseudo-code. The first function finds all the possible ways we can extend a given state s.

```
function NextStates($s$)
    $next\_states \leftarrow [\,]$                                    ▷ $Jugs$ is the set of jugs
    for $j \in Jugs$ do
        $room[j] \leftarrow capacity[j] - s[j]$
    for $(src, dest) \in Jugs^2$ with $src \neq dest$ do
```

```
        $pour\_amount \leftarrow \min(s[src], room[dest])$
        **if** $pour\_amount > 0$ **then**
            $next \leftarrow$ a copy of $s$
            $next[src] \leftarrow next[src] - pour\_amount$
            $next[dest] \leftarrow next[dest] + pour\_amount$
            $next\_states \leftarrow next\_states \cup next$
    **return** $next\_states$
```

The next function will extract the full solution once a satisfying goal state has been found. We make use of a dictionary *prev* that maps a given state to the state from which it was reached. By keeping such a traversal record, we can obtain a path from the goal state to the initial state, thus giving us a sequence of pours to conduct in order to get our 4 pints of water.

```
function SOLUTION($initial\_state, goal\_state, prev$)
    $path \leftarrow [\,]$
    $s \leftarrow goal\_state$
    while $s \neq initial\_state$ do
        $path \leftarrow s, path$
        $s \leftarrow prev[s]$
    return $initial\_state, path$
```

Here then is the breadth-first traversal. The parameters are the initial state and a predicate, *success*, which tests whether a given state is a successful goal state. (In our case, that means whether some jug in the state holds 4 pints.)

```
function BFS($initial\_state, success$)
    $prev \leftarrow [\,]$
    INJECT($q, initial\_state$)
    while $q$ is not empty do
        $s \leftarrow$ EJECT($q$)
        if $success(s)$ then
            return SOLUTION($initial\_state, s, prev$)
        $visited[s] \leftarrow True$
        for $next \in$ NEXTSTATES($s$) do
            if $visited[next] = False$ then
                $prev[next] \leftarrow s$
                INJECT($q, next$)
    return $False$
```

Here is a shortest solution path: 800, 350, 323, 620, 602, 152, 143.