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

Sample Answers

The exercises

23. Consider the *subset-sum problem*: Given a set S of positive integers, and a positive integer t, find a subset $S' \subseteq S$ such that $\sum S' = t$, or determine that there is no such subset. Design an exhaustive-search algorithm to solve this problem. Assuming that addition is a constant-time operation, what is the complexity of your algorithm?

    Answer:

    function SUBSETSUM(S, t)
        for each S' in POWERSET(S) **do**
            if $\sum S' = t$ **then**
                return *True*
        return *False*

    How can we systematically generate all subsets of the given set S? For each element $x \in S$, we need to generate all the subset of S that happen to contain x, as well as those that do not. This gives us a natural recursive algorithm:

    function POWERSET(S)
        if $s = \emptyset$ **then**
            return $\{\emptyset\}$
        else
            $x \leftarrow$ some element of S
            $S' \leftarrow S \setminus \{x\}$
            $P \leftarrow$ POWERSET(S')
            return $P \cup \{s \cup \{x\} \mid s \in P\}$

    There are 2^n subsets of S. $\sum S'$ calculates the sum of the elements in S', which requires $O(n)$ time. Hence the brute-force solution is $O(n \cdot 2^n)$.

24. Consider the *partition problem*: Given n positive integers, partition them into two disjoint subsets so that the sum of one subset is the same as the sum of the other, or determine that no such partition exists. Designing an exhaustive-search algorithm to solve this problem seems somewhat harder than doing the same for the subset-sum problem. Show, however, that there is a simple way of exploiting your algorithm for the subset-sum problem (that is, try to *reduce* the partition problem to the subset-sum problem).

    Answer:

    function HASPARTITION(S)
        $sum \leftarrow \sum S$
        if sum is odd **then**
            return *False*
        return SUBSETSUM($S, sum/2$)

25. Consider the *clique problem*: Given a graph G and a positive integer k, determine whether the graph contains a *clique* of size k, that is, G has a complete sub-graph with k nodes. Design an exhaustive-search algorithm to solve this problem.

Answer: Assume $G = (V, E)$ where V is the set of nodes, and E is the set of edges.

```
function HasClique((V, E), k)
    for each U ⊆ V with |U| = k do
        if IsClique(U, E) then
            return True
    return False

function IsClique(U, E)
    for each u ∈ U do
        for each v ∈ U do
            if u ≠ v and (u, v) ∉ E then
                return False
    return True
```

There are $\binom{|V|}{k}$ ways of selecting a subset of k nodes. The function IsClique is called for each such subset, and the cost of each call is $O(k^2|E|)$.

26. Consider the special clique problem of finding triangles in a graph (noting that a triangle is a clique of size 3). Show that this problem can be solved in time $O(|V|^3|E|)$.

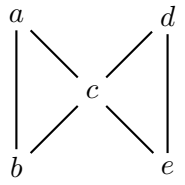
Answer: The number of subsets of size 3 is $\binom{|V|}{3} = \frac{|V|(|V|-1)(|V|-2)}{6}$. For each subset the cost is $O(|E|)$. Altogether the cost of the brute-force approach is $O(|V|^3|E|)$.

27. Draw the undirected graph whose adjacency matrix is

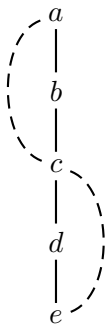
	a	b	c	d	e
a	0	1	1	0	0
b	1	0	1	0	0
c	1	1	0	1	1
d	0	0	1	0	1
e	0	0	1	1	0

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

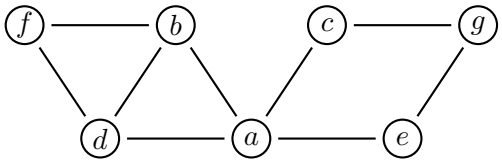
Answer: Here is the graph:



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:



28. Consider this 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:

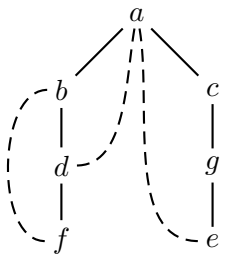
(a) Here is the adjacency matrix:

	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	$\to c \to e$

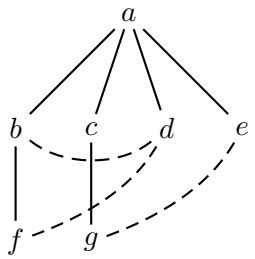
(b) In a depth-first search, the nodes are visited in this order: a, b, d, f, c, g, e.



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.



29. Explain how one can use depth-first search to identify the connected components of an undirected graph. Hint: Number the components from 1 and mark each node with its component number.

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 ← 1
for each v in V do
    if v is marked with 0 then
        DFS(v)
        component ← 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)
```

30. The function CYCLIC is intended to check whether a given undirected graph is cyclic.

```
function CYCLIC(⟨V, E⟩)
    mark each node in V with 0
    count ← 0
    for each v in V do
        if v is marked with 0 then
            cyclic ← HASCYCLES(v)
            if cyclic then
                return True
    return False

function HASCYCLES(v)
    count ← count + 1
    mark v with count
    for each edge (v, w) do                          ▷ w is v's neighbour
        if w's mark is greater than 0 then
            return True                              ▷ w has been visited before
        if HASCYCLES(w) then
            return True                              ▷ a cycle can be reached from w
    return False
```

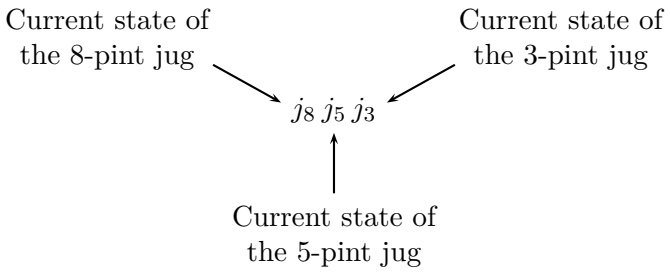
Show, through a worked example, that the algorithm is incorrect. Exercise 35 will ask you to develop a correct algorithm for this problem.

Answer: Take the undirected graph that has just two nodes, a and b, and a single edge. The edge is represented as $\{(a, b), (b, a)\}$ irrespective of whether we use an adjacency matrix or an adjacency list representation. The main functions calls HasCycles(a), which results in a being marked and then the recursive call HasCycles(b) is made. This results in b being marked, and then b's neighbours are considered. Since the neighbour a has already been marked, the algorithm returns *True*, that is, the graph is deemed cyclic, which is wrong.

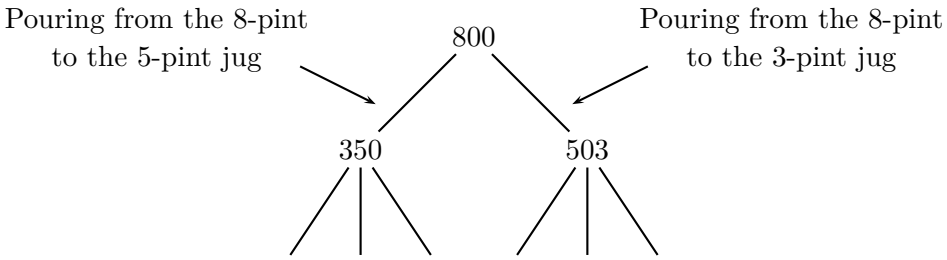
31. (Optional—state space search.) Given an 8-pint jug full of water, and two empty jugs of 5- and 3-pint capacity, get exactly 4 pints of water in one of the jugs by completely filling up and/or emptying jugs into others. Solve this problem using breadth-first search.

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:



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:



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 ← [ ]                                  ▷ Jugs is the set of jugs
    for j ∈ Jugs do
        room[j] ← capacity[j] − s[j]
    for (src, dest) ∈ Jugs^2 with src ≠ dest do
        pour_amount ← min(s[src], room[dest])
        if pour_amount > 0 then
            next ← a copy of s
            next[src] ← next[src] − pour_amount
            next[dest] ← next[dest] + pour_amount
            next_states ← next_states ∪ 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 ← [ ]
    s ← goal_state
    while s ≠ initial_state do
        path ← s, path
        s ← 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 ← [ ]
    Inject(q, initial_state)
    while q is not empty do
        s ← Eject(q)
        if success(s) then
            return Solution(initial_state, s, prev)
```

```
        visited[s] ← True
        for next ∈ NextStates(s) do
            if visited[next] = False then
                prev[next] ← s
                Inject(q, next)
    return False
```

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

32. (Optional—use of induction.) 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).