Note: All exercises ending with a ☆ have previously appeared on an exam.

Part I
Graph Searching

Graph implementation

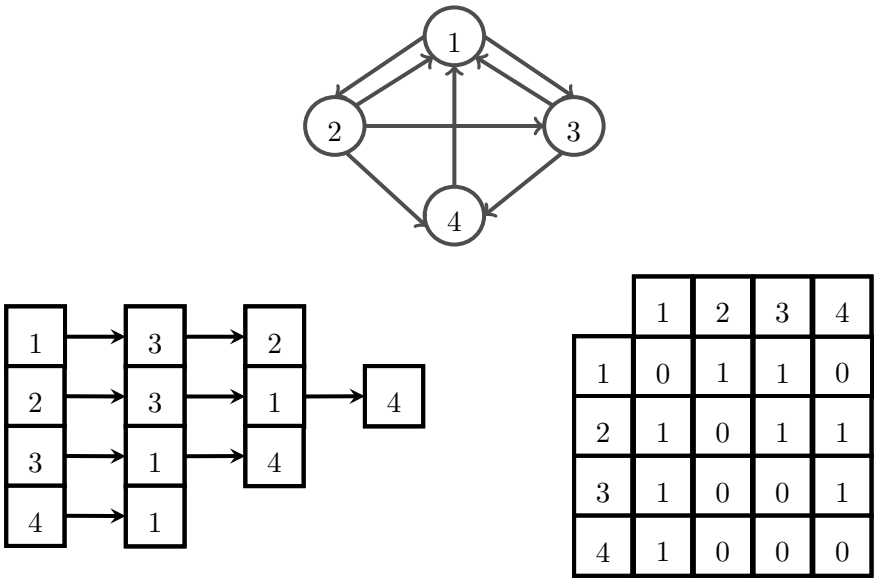


Figure 1: Example of a directed graph with its adjacency list and matrix

Big O notation

$$f(x) = O(g(x)) \text{ (as } x \to \infty) \Leftrightarrow \exists c > 0;\, x_0 \in \mathbb{R} : \forall x \geq x_0 : |f(x)| \leq c\,|g(x)|$$

The function $g(x)$ provides an upper bound on the growth rate of $f(x)$.

1 Graph searching (p. 4)

1.1 Graph representations (p. 4)

1. Given an adjacency list representation of a directed graph $G = (V, E)$, give an $O(|V| + |E|)$ algorithm to obtain an adjacency list representation such that each of the $|V|$ lists are sorted.

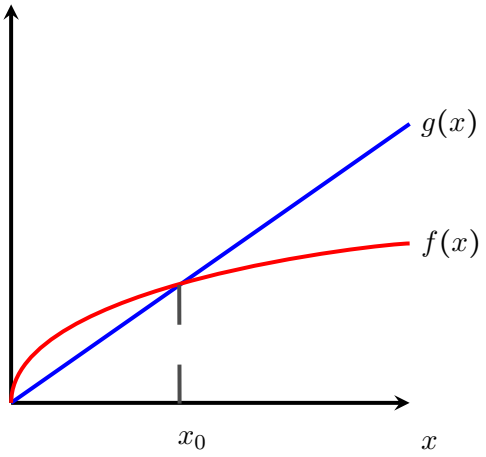


Figure 2: Example of 2 functions where $f(x) \in O(g(x))$

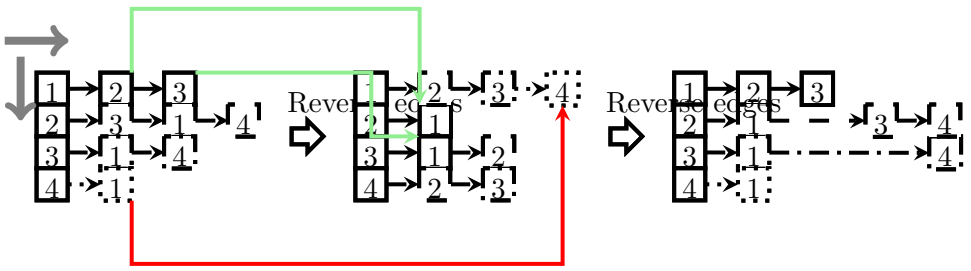
Solution: Reverse all edges in the following manner

Create $G' = (V, E')$
for all $v \in V$ **do**
  for all $u \in$ adjacency list of v **do**
    Add v to back of adjacency list of u in G'
  end for
end for

This creates G', where the edges in E are reversed to form E'. Note that the edges of the adjacency lists of E' are sorted, since all $v \in V$ were traversed in a sorted order (from top to bottom). The algorithm is clearly $O(|V|+|E|)$: $O(|V|)$ for creating $|V|$ lists plus $O(|E|)$ for adding elements to these lists.

Repeat the algorithm above on G', resulting in G''. The adjacency lists of G'' are now sorted.

Example:

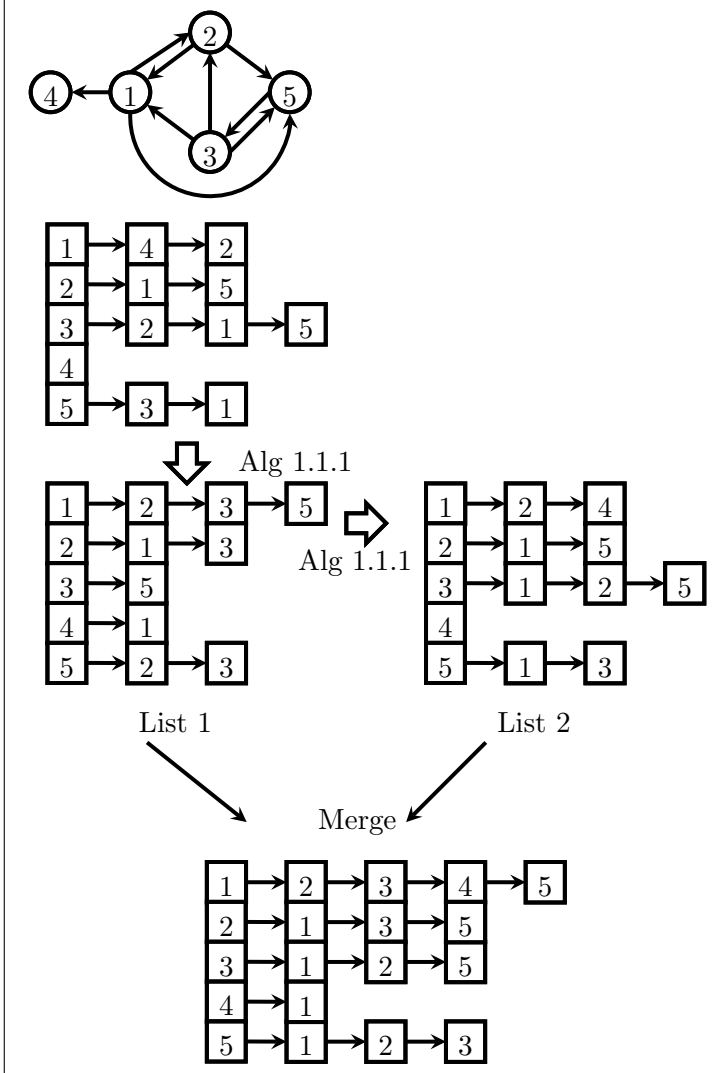


2. Let $G = (V, E)$ be a directed graph, give an $O(|V| + |E|)$ algorithm to obtain the graph

$G' = (V, E')$ with $(u, v) \in E'$ if and only if $(v, u) \in E$ or $(u, v) \in E$.

Solution: We start by reversing G with the reversing algorithm from exercise 1.1.1 ($O(|V| + |E|)$). This gives us $G_1 = (V, E_1)$ with $(v, u) \in E_1$ if $(u, v) \in E$. Note again that the adjacency lists of G_1 are already sorted. We now reverse the graph G_1 such that the adjacency lists of G are also sorted.
We copy the nodes to form a new graph $G' = (V, E')$ ($O(|V|)$), where the adjacency lists of G and G_1 are merged with each other. This can be done in $O(|V| + |E|)$ because the adjacency lists of G and G_1 are sorted.

Merging two sorted lists can be done as follows: we place a pointer at the leftmost element of the two lists. If the elements are equal to one another, we move both pointers to the right, otherwise we only move the pointer of the smaller element to the right. Every time we move a pointer(s), we write the element it/they pointed to to the merged list.



3. Let A be the adjacency matrix representation of an undirected graph $G = (V, E)$. Let $(A^3)_{ij}$ be entry (i, j) of A^3. What does $\frac{1}{6}\sum_{i=1}^{|V|}(A^3)_{ii}$ represent? ☆

Solution: Note: we don't allow edges from a vertex to itself. By induction on k one can show the following:

THEOREM 1.1 Let $G = (V, E)$ be a graph and A its adjacency matrix. Let $k \in \mathbb{N}$. Then $(A^k)_{ij}$ is the number of paths from i-th to j-th vertex in G of length k (where A^0 is the identity matrix).

Using this theorem we can see that $\sum_{i=1}^{|V|}(A^3)_{ii}$ is the number of paths in the shape

of a triangle. However, as G is undirected, each of these triangles is counted 6 times (3 possible start vertices and 2 directions of walking: clockwise and counterclockwise). Therefore, $\frac{1}{6}\sum_{i=1}^{|V|}(A^3)_{ii}$ represents the number of unique triangles/ number of undirected 3-cycles in a graph G.
Remark: analogously, $\frac{1}{3}\sum_{i=1}^{|V|}(A^3)_{ii}$ represents the number of unique directed triangles/ 3-cycles in a directed graph.

4. Consider the adjacency matrix representation A of a directed graph G. Define $v \in V$ as a universal sink if v has $|V|-1$ incoming edges and no outgoing edges. Develop an $O(|V|)$ algorithm to determine whether G contains a universal sink. ☆

Solution: By definition, we have:

- if $A_{ij} = 0$, for $i \neq j$, then j cannot be a sink; and
- if $A_{ij} = 1$ then i cannot be a sink.

Based on these two facts, we get an $O(|V|)$ algorithm:

Step 1: picking a candidate

```
i = 1, j = 1
while max(i, j) ≤ |V| do
    if i = j then
        if A_ij = 1 then
            j = j + 1
        else
            i = i + 1
        end if
    else
        if A_ij = 1 then
            i = i + 1
        else
            j = j + 1
        end if
    end if
end while
```

Step 2: checking whether j is a sink

```
if j > |V| then
    return false
else if A_j1 + ··· + A_j|V| > 0 then
    return false
else if A_1j + ··· + A_|V|j < |V| − 1 then
    return false
else
    return true
```

end if

The algorithm is $O(|V|)$:

- The while loop in step 1 takes $O(|V|)$ steps, as every iteration either i or j is increased by 1.
- Each iteration of the while loop is $O(1)$.
- Step 2 clearly is $O(|V|)$.

5. Give an $O(|V|^3)$ algorithm which determines for all pairs $u, v \in V$ the number of common neighbors of u and v in a directed graph $G = (V, E)$. ☆

Solution: Denote by A_i the i-th row of a matrix A. We can then use the following algorithm:

(Optional: Create A, the adjacency matrix of G)
for $i = 1, \ldots, |V|$ **do**
  for $j = 1, \ldots, |V|$ **do**
    $neighbours(i, j) = A_i \cdot A_j$
  end for
end for

Creating A is clearly $O(|V|^2)$. Multiplying two rows of A is $O(|V|)$ (it takes $|V|$ multiplications and $|V| - 1$ additions). As the algorithm does this $|V|^2$ times, the complexity of the algorithm is $O(|V|^3)$. Note, we can also let the second for loop run from j till i as $neighbours(i, j) = neighbours(j, i)$.
Essentially, we thus have $neighbours(i, j) = (AA^T)_{ij}$.

We can also give a solution using the adjacency list representation:

(Optional:
for $i = 1, \ldots, |V|$ **do**
  Create L_i, the adjacency list of i
end for)
Sort $L_1, \ldots, L_{|V|}$ using algo. 1.1.1
for $i = 1, \ldots, |V|$ **do**
  for $j = 1, \ldots, |V|$ **do**
    $neighbours(i, j)$ = number of common elements in L_i and L_j
  end for
end for

The first for loop and sorting the edges take $O(|V| + |E|) \leq O(|V|^2)$ time. Counting the number of common elements of two (sorted!) adjacency lists can be done in $O(|V|)$. The total time complexity is $O(|V|^3)$.

2 Breadth-first Search (p. 5)

Algorithm 1 BFS(G, s)

```
for all u ∈ V ∖ {s} do
    color(u) = WHITE; d(u) = ∞; π(u) = NIL
end for
color(s) = GRAY; d(s) = 0; π(s) = NIL

while Q ≠ ∅ do
    u = HEAD(Q)
    for all v ∈ Γ(u) do
        if color(v) = WHITE then
            color(v) = GRAY; d(v) = d(u) + 1; π(v) = u
            ENQUEUE(Q, v)
        end if
    end for
    DEQUEUE(Q)
    color(u) = BLACK
end while
```

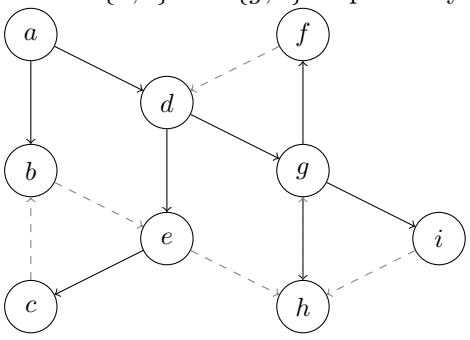
Theorems

THEOREM 2.1. Given $G = (V, E)$ a graph with $s \in V$ the source vertex, the BFS algorithm returns a tree containing all vertices reachable from s and $d(u) = \delta(s, u)$ upon termination, where $\delta(s, u)$ gives the length of the shortest simple path from s to u in G (that is, the number of edges needed to get from s to u).

2.1 BFS searches (p. 8)

1. Determine the BFS tree of the graph depicted in Figure 3 (p. 6) if we reverse the order of the vertices in the adjacency lists of a and d.

Solution: In Figure 3 (p. 6), nodes were visited in alphabetic order, with source $s = a$. Here we visit the neighbors of a and d in orders $\{d, b\}$ and $\{g, e\}$ respectively. This results in the following predecessor tree:

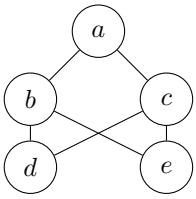


and the queue contents during the BFS run:

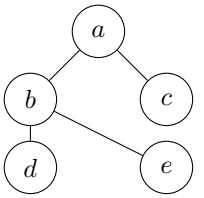
Visit	Q (after visit)
	a
a	d, b
d	b, g, e
b	g, e
g	e, f, h, i
e	f, h, i, c
f	h, i, c
h	i, c
i	c
c	$\varnothing$

2. Give an example of a graph G and source vertex s such that $E' \subseteq E$ forms a tree on V such that the unique path from s to v, for all $v \in V$, has length $\delta(s, v)$, but that cannot be produced as the predecessor tree by the BFS algorithm, no matter how the vertices were sorted in the adjacency lists.

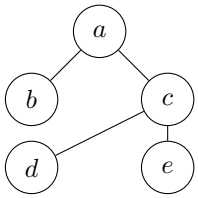
Solution: Running the BFS algorithm on the following graph with source vertex a



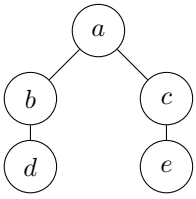
can result in



or



but never



as we can only visit the d and e nodes through either b or c (depending on the order of the adjacency list of a).

3. A graph G is bipartite if V can be partitioned into V_1 and V_2 such that if $(u, v) \in E$,

then $u \in V_1$ and $v \in V_2$ or vice versa. Give an efficient algorithm to test whether G is bipartite.

Solution: *Hint: the only thing that matters is that there exists an edge connecting u and v, not the direction of said edge. You can make the graph undirected in $O(|V| + |E|)$ time using exercise 1.1.2. This is still deemed efficient and the problem of choosing a starting node disappears. (Note that G is bipartite if and only if there exist no cycles of odd length in the undirected graph.)*

Use one color to denote each partition, for example RED for V_1 and BLUE for V_2. You can also solve this using even and odd distances $d(u)$ (where even=RED and odd=BLUE). However, coloring nodes makes this a bit more intuitively clear.

Then implement the following on top of BFS:

- Color the source node s RED
- Color every node $u \in \Gamma(s)$ BLUE
- Color every neighbour of each u in the step above RED
- ...

Repeat this until every node has an assigned color (bipartite graph) or a conflict is detected where a neighbor of a node u already has the same color as the one assigned to u. Note, if the graph is not connected, i.e. if it has multiple connected components, we have to repeat this algorithm on the other connected components. The graph is then bipartite if and only if each connected component is bipartite.

4. Let the diameter of a graph $G = (V, E)$ be $\max_{u,v \in V} \delta(u, v)$ (for those u, v couples with $\delta(u, v)$ finite) Give an algorithm to determine the diameter, what is the time complexity of your algorithm?

Solution: *(Optional)* Run the BFS algorithm for each node $s \in V$. For each invocation BFS(s, G), note the maximum distance encountered, which is equal to $d(u)$, with $u \in V$ the node discovered/visited last in the run.
The diameter is equal to the maximum of $d(u)$ over all runs.

Time complexity is $O(|V|(|V| + |E|))$ or approximately $O(|V||E|)$ as $|V| \leq |E|$ for most graphs.

5. Give an $O(|V|)$ algorithm to find the diameter in an undirected acyclic connected graph. ☆

Solution: We first note that an undirected acyclic connected graph (undirected tree) $G = (V, E)$ has exactly $|V| - 1$ undirected edges. Therefore, BFS algorithm runs on such a graph in $O(|V|)$ time. We now use the following algorithm:

(a) Pick a source vertex $s \in V$ randomly and execute BFS.

(b) Pick a vertex t with $d(t)$ maximal.

(c) Execute BFS with t as starting vertex.

(d) Pick a vertex u with $d(u)$ maximal.

(e) Return $d(u)$.

Steps (a) and (c) run in $O(|V|)$ time by the discussion above. Steps (b) and (d) clearly run in $O(|V|)$.

6. We have three containers whose sizes are 10 pints, 7 pints, and 4 pints, respectively. The 7-pint and 4-pint containers start out full of water, but the 10-pint container is initially empty. We are allowed one type of operation: pouring the contents of one container into another, stopping only when the source container is empty or the destination container is full. We want to know if there is a sequence of pourings that leaves exactly 2 pints in the 7- or 4-pint container. Model this as a graph problem: give a precise definition of the graph involved and state the specific question about this graph that needs to be answered. What algorithm should be applied to solve the problem? ☆

Solution: We construct a graph $G = (V, E)$ corresponding to this problem as follows. We denote each vertex of V as (a, b, c) with a, b, c corresponding to the amount of water in 10-, 7- and 4-pint container respectively. Thus, we set $V = \{(a, b, c) | a, b, c \geq 0, a \leq 10, b \leq 7, c \leq 4, a + b + c = 11\}$. We let $((a, b, c), (d, e, f)) \in E$ if and only if it is possible to go in one step from vertex (a, b, c) to (d, e, f) according to the rules described in the problem. We now run BFS on G with $(0, 7, 4)$ as the starting vertex and stop if we come across a vertex of the form $(a, 2, c)$ or $(a, b, 2)$. Using BFS algorithm we get a shortest path from $(0, 7, 4)$ to $(2, 7, 2)$:

$$(0, 7, 4), (4, 7, 0), (10, 1, 0), (6, 1, 4), (6, 5, 0), (2, 5, 4), (2, 7, 2).$$

Extra: A different example is the one with containers of volume 10,6,5 pints, where the 6- and 5-pint containers are initially full and we need to get 8 pints of water in the 10-pint container.

7. Consider a graph $G = (V, E)$ and two sets $V_1, V_2 \subset V$ with $V_1 \cap V_2 = \varnothing$. Give an $O(|V| + |E|)$ algorithm to find $\delta(V_1, V_2) = \min_{u \in V_1, v \in V_2} \delta(u, v)$. ☆

Solution: We can use a modified version of the BFS algorithm (this solution) or modify the graph (next solution):

for each vertex $u \in V \setminus V_1$ **do**
  $color(u)$ = WHITE, $d(u) = \infty$, $\pi(u)$ = NIL
end for
for each vertex $u \in V_1$ **do**
  $color(u)$ = GRAY, $d(u) = 0$, $\pi(u)$ = NIL
end for
$Q = V_1$

while $Q \neq \varnothing$ **do**
  u = HEAD(Q)
  for each $v \in \Gamma(u)$ **do**
    if $color(v)$ = WHITE **then**
      $color(v)$ = GRAY; $d(v) = d(u) + 1$; $\pi(v) = u$
      ENQUEUE(Q, v)
    end if
  end for
  DEQUEUE(Q)
  $color(u)$ = BLACK
end while
return $\min_{v \in V_2} d(v)$

Analogously to the discussion on time complexity of BFS algorithm, we have that this algorithm is $O(|V| + |E|)$. Intuitively, the algorithm is BFS but with all of V_1 collapsed to a single starting vertex.
A similar solution is the following:

Set $V' = V \cup \{s\}$, $E' = E \cup \{(s, v) | v \in V_1\}$ and $G' = (V', E')$.
Run BFS on G' with starting vertex s.
return $\min_{v \in V_2} d(v) - 1$

8. Adapt the BFS algorithm such that it not only determines the shortest path length $d(v)$ for all the nodes v reachable from s, but also the number of paths $M(v)$ of length $d(v)$ between s and v. ☆

Solution: A simple modification to BFS algorithm yields:

for all $u \in V \setminus \{s\}$ **do**
  $color(u)$ = WHITE; $d(u) = \infty$; $\pi(u)$ = NIL, $M(v) = 0$
end for
$color(s)$ = GRAY; $d(s) = 0$; $\pi(s)$ = NIL, $M(s) = 1$

while $Q \neq \varnothing$ **do**
  u = HEAD(Q)
  for all $v \in \Gamma(u)$ **do**

```
        if color(v) = WHITE then
            color(v) = GRAY; d(v) = d(u) + 1; π(v) = u, M(v) = M(u)
            ENQUEUE(Q, v)
        else if d(u) + 1 = d(v) then
            M(v) = M(v) + M(u)
        end if
    end for
    DEQUEUE(Q)
    color(u) = BLACK
end while
```

Note that if $d(u) + 1 = d(v)$ and $color(v) \neq$ WHITE, then we must have $color(v) =$ GRAY.

Extra

1. Indicate how we can solve the Word Ladder Problem using graph searching methods. The Word Ladder Problem: Suppose we start with a word in the English dictionary. We need to check if we can transform this word into another English word of the same length using the following rules: we may gradually change the starting word one letter at a time, such that at each step the word we have is an existing English word. For example we can go from "cat" to "paw" as follows: cat, mat, maw, paw.

Solution: Suppose we start with an n-letter word. Set $V = \{l_1 l_2 \dots l_n | n \in \mathbb{N} \setminus 0, l_i$ is a letter $\forall i = 1, \dots, N$ and $l_1 l_2 \dots l_n$ is a word$\}$ and
$E = \bigcup_{i=1}^{n} \{(l_1 \dots l_n, l_1 \dots l_{i-1} L l_{i+1} \dots l_n) | L \neq l_i, L$ is a letter and $l_1 \dots l_{i-1} L l_{i+1} \dots l_n$ is a word$\}$, then run BFS/DFS.