Algorithms in Pseude-Code

COMP2721 Algorithms and Data Structures II

Session 2024

```
input  : a graph G = (V, E)
output: a BFS-forest T = (V, F) of G and a BFS-numbering σ : V → ℕ
begin
    i ← 1; F ← ∅; Q ← emptyQueue;
    for v ∈ V do mark v unvisited;
    while there is a unvisited vertex v ∈ V do
        σ(v) ← i; i ← i + 1;
        Q ← enqueue(v, Q); mark v visited;
        while Q is not empty do
            v ← front(Q); Q ← dequeue(Q);                /* remove v from Q */
            for w ∈ N(v) do
                if w is unvisited then
                    F ← F ∪ {vw};
                    σ(w) ← i; i ← i + 1;
                    Q ← enqueue(w, Q); mark w visited
```

Algorithm 1: traditional Breadth-First Search

```
input  : a graph G = (V, E)
output: a BFS-forest T = (V, F) of G and a BFS-numbering σ : V → ℕ
begin
    i ← 1; F ← ∅; Q ← emptyQueue;
    for v ∈ V do mark v unvisited;
    while there is a unvisited vertex v ∈ V do
        σ(v) ← i; i ← i + 1;
        Q ← enqueue(v, Q); mark v visited;
        while Q is not empty do
            v ← front(Q);         /* inspect first element without removing it */
            if v has an unvisited neighbour w then
                F ← F ∪ {vw};
                σ(w) ← i; i ← i + 1;
                Q ← enqueue(w, Q); mark w visited
            else
                Q ← dequeue(Q);                          /* remove first element */
```

Algorithm 2: Breadth-First Search similar to Depth-First Search

```
input  : a graph G = (V, E)
output: a DFS-forest T = (V, F) of G and a DFS-numbering σ : V → ℕ
begin
    i ← 1; F ← ∅; S ← emptyStack;
    for v ∈ V do mark v unvisited;
    while there is a unvisited vertex v ∈ V do
        σ(v) ← i; i ← i + 1;
        S ← push(v, S); mark v visited;
        while S is not empty do
            v ← peek(S);            /* inspect top-element without removing it */
            if v has an unvisited neighbour w then
                F ← F ∪ {vw};
                σ(w) ← i; i ← i + 1;
                S ← push(w, S); mark w visited
            else
                S ← pop(S);                              /* remove top-element */
```

Algorithm 3: Depth-First Search with stack

```
input  : a graph G = (V, E)
output: a DFS-forest T = (V, F) of G and a DFS-numbering σ : V → ℕ
begin
    i ← 1; F ← ∅;
    for v ∈ V do mark v unvisited;
    while there is a unvisited vertex v ∈ V do DFS-visit(v);

procedure DFS-visit (v)
    mark v visited;
    σ(v) ← i; i ← i + 1;
    for w ∈ N(v) do
        if w is unvisited then
            F ← F ∪ {vw};
            DFS-visit(w)
```

Algorithm 4: recursive Depth-First Search

input : a directed acyclic graph (dag) $G = (V, A)$
output: a topological sort $\sigma : V \to \mathbb{N}$ of G

begin
  $i \leftarrow |V|$;
  for $v \in V$ **do** mark v unvisited;
  while there is a unvisited vertex $v \in V$ **do** `TS-visit`(v);

procedure `TS-visit` (v)
  if v is marked stacked **then stop**;
  if v is marked unvisited **then**
    mark v stacked;
    for $w \in N^+(v)$ **do** `TS-visit`(w);
    mark v visited;
    $\sigma(v) \leftarrow i$; $i \leftarrow i - 1$

Algorithm 5: Topological Sort

input : a connected graph $G = (V, E)$ with weights $\ell : E \to \mathbb{N}$
output: a minimum spanning tree $T = (V, F)$ of G

begin
  for $v \in V$ **do** $\pi(v) \leftarrow \infty$;
  choose an arbitrary vertex $u \in V$;
  $U \leftarrow \{u\}$; $F \leftarrow \varnothing$;
  while $N(U) \neq \varnothing$ **do**
    for $v \in N(u) \setminus U$ **do**
      if $\pi(v) > \ell(uv)$ **then**
        $\pi(v) \leftarrow \ell(uv)$; $p(v) \leftarrow u$
    choose $v \in V \setminus U$ with minimal $\pi(v)$;
    $U \leftarrow U \cup \{v\}$; $F \leftarrow F \cup \{p(v)v\}$; $u \leftarrow v$

Algorithm 6: Minimum Spanning Tree (Prim)

input : a connected graph $G = (V, E)$ with weights $\ell : E \to \mathbb{N}$
output: a minimum spanning tree $T = (V, F)$ of G

begin
  sort the edges $e \in E$ by their weights $\ell(e)$;
  $F \leftarrow \varnothing$;
  for $e \in E$ in non-decreasing order **do**
    if $(V, F \cup \{e\})$ is acyclic **then** $F \leftarrow F \cup \{e\}$;

Algorithm 7: Minimum Spanning Tree (Kruskal)

input : a graph $G = (V, E)$ with weights $\ell : E \to \mathbb{N}$ and a source $s \in V$
output: the distances $d(s, v)$ for all $v \in V$ and
a shortest path tree $T = (V, F)$ of G with source s

```
begin
    for v ∈ V do π(v) ← ∞;
    d(s,s) ← 0; π(s) ← 0; U ← {s}; u ← s; F ← ∅;
    while N(U) ≠ ∅ do
        for v ∈ N(u) \ U do
            if π(v) > π(u) + ℓ(uv) then
                π(v) ← π(u) + ℓ(uv); p(v) ← u
        choose v ∈ V \ U with minimal π(v);
        d(s,v) ← π(v);
        U ← U ∪ {v}; F ← F ∪ {p(v)v}; u ← v
```

Algorithm 8: Single-Source Shortest Path (DIJKSTRA)

```
input  : a directed graph G = (V, E), a root r ∈ V, and weights w : E → ℕ
output: F ⊆ E such that (V, F) is a minimum arborescense of G rooted at r,
        if there is one

procedure minimumArborescence(G, r, w)
    F_0 ← ∅; w(r) ← 0;
    for v ∈ V \ {r} do
        w(v) ← ∞;
        for u ∈ N_G^-(v) do
            if w(uv) < w(v) then w(v) ← w(uv); x ← u;
        if w(v) = ∞ then exit;
        else F_0 ← F_0 ∪ {xv};
    if (V, F_0) has a cycle (U, C) then
        shrink; F_1 ← minimumArborescence(G', r, w'); expand
    else F ← F_0;
    return F

procedure shrink
    V' ← {u} ∪ V \ U; E' ← ∅;
    for v ∈ V do w'(uv) ← ∞; w'(vu) ← ∞;
    for xv ∈ E do
        if v ∈ V \ U then
            if x ∈ V \ U then
                E' ← E' ∪ {xv}; w'(xv) ← w(xv) − w(v); p(xv) ← xv
            else
                E' ← E' ∪ {uv};
                if w(xv) − w(v) < w'(uv) then
                    w'(uv) ← w(xv) − w(v); p(uv) ← xv
        if x ∈ V \ U then
            E' ← E' ∪ {xu};
            if w(xv) − w(v) < w'(xu) then
                w'(xu) ← w(xv) − w(v); p(xu) ← xv
    G' ← (V', E')

procedure expand
    F ← C;
    for xv ∈ F_1 do
        F ← F ∪ {p(xv)};
        if v = u then y ← x;
    for x ∈ U do
        if xy ∈ C then F ← F \ {xy};
```

Algorithm 9: Minimum Arborescence

input : an integer $n > 0$ (number of matrices A_i to multiply) and an array $d[0..n]$ (sizes, A_i is a $d[i-1] \times d[i]$ matrix)
output: $N[1, n]$ the minimum number of scalar multiplications

```
begin
    for i ← 1 to n do N[i, i] ← 0;
    for b ← 1 to n − 1 do
        for i ← 1 to n − b do
            k ← i + b; m ← d[i − 1] ∗ d[k]; N[i, k] ← ∞;
            for j ← i to k − 1 do
                N[i, k] ← min(N[i, k], N[i, j] + m ∗ d[j] + N[j + 1, k])
```

Algorithm 10: Matrix Chain-Product

input : a directed graph $G = (V, E)$ with weights $\ell : E \to \mathbb{Z}$ such that G has no cycle of negative length, w.l.o.g. $V = \{1, 2, \ldots, n\}$
output: the distances $d_n(u, v)$ for all $u, v \in V$

```
begin
    for u ∈ V do
        for v ∈ V do
            if u = v then
                d_0(u, v) ← 0
            else
                if uv ∈ E then
                    d_0(u, v) ← ℓ(u, v)
                else
                    d_0(u, v) ← ∞
    for k ← 1 to n do
        for u ∈ V do
            for v ∈ V do
                d_k(u, v) ← min(d_{k−1}(u, v), d_{k−1}(u, k) + d_{k−1}(k, v))
```

Algorithm 11: All Pair Shortest Path (FLOYD/WARSHALL)

fixed : A CFG $G = \langle T, V, P, S \rangle$ in Chomsky normal form is not part of the input.
input : a string $x_1 x_2 \ldots x_n \in T^*$
output: a boolean value indicating whether $x_1 x_2 \ldots x_n$ can be generated by G

begin
- **for** $i \leftarrow 1$ **to** n **do**
  - $V(i,i) \leftarrow \{A \in V \mid (A \rightarrow x_i) \in P\}$
- **for** $b \leftarrow 1$ **to** $n-1$ **do**
  - **for** $i \leftarrow 1$ **to** $n-b$ **do**
    - $k \leftarrow i+b$; $V(i,k) \leftarrow \varnothing$;
    - **for** $j \leftarrow i$ **to** $k-1$ **do**
      - **for** $(A \rightarrow BC) \in P$ **do**
        - **if** $B \in V(i,j)$ **and** $C \in V(j+1,k)$ **then** $V(i,k) \leftarrow V(i,k) \cup \{A\}$;
- **if** $S \in V(1,n)$ **then** accept $x_1 x_2 \ldots x_n$ **else** reject $x_1 x_2 \ldots x_n$;

Algorithm 12: CYK parsing of context-free languages (Cocke/Younger/Kasami)

input : an integer $n > 2$ (number of cities) and a 2-dimensional array $d[1..n, 1..n]$ of pairwise distances
output: a permutation $\pi : \{1,\ldots,n\} \rightarrow \{1,\ldots,n\}$ minimising $d[\pi(n),\pi(1)] + \sum_{i=2}^{n} d[\pi(i-1),\pi(i)]$

begin
- **for** $c \in \{2,\ldots,n\}$ **do** $\ell(\{1,c\},c) \leftarrow d[1,c]$; $p(\{1,c\},c) \leftarrow 1$;
- **for** $k \leftarrow 2$ **to** $n-1$ **do**
  - **for** $S' \in \binom{\{2,\ldots,n\}}{k}$ **do**
    - **for** $c \in S'$ **do**
      - $S \leftarrow S' \cup \{1\}$; $\ell(S,c) \leftarrow \infty$;
      - **for** $a \in S' \setminus \{c\}$ **do**
        - $\ell' \leftarrow \ell(S \setminus \{c\}, a) + d[a,c]$;
        - **if** $\ell' < \ell(S,c)$ **then** $\ell(S,c) \leftarrow \ell'$; $p(S,c) \leftarrow a$;
- $\ell'' \leftarrow \infty$;
- **for** $a \in \{2,\ldots,n\}$ **do**
  - $\ell' \leftarrow \ell(\{1,\ldots,n\},a) + d[a,1]$;
  - **if** $\ell' < \ell''$ **then** $\ell'' \leftarrow \ell'; \pi(n) \leftarrow a$;
- $S \leftarrow \{1,\ldots,n\}$;
- **for** $i \leftarrow n$ **downto** 2 **do**
  - $\pi(i-1) \leftarrow p(S,\pi(i))$; $S \leftarrow S \setminus \{\pi(i)\}$

Algorithm 13: Travelling Sales Person (Bellman/Held/Karp)

input : an array $A[1..n]$ containing a sequence of n numbers $(a_1, a_2, \ldots, a_n)$
output: the array A containing a permutation $(a'_1, a'_2, \ldots, a'_n)$ of the input sequence such that $a'_1 \leqslant a'_2 \leqslant \ldots \leqslant a'_n$

begin
  `quicksort`$(A, 1, n)$

procedure `quicksort` (A, p, r)
  if $p < r$ **then**
    $q \leftarrow$ `partition`(A, p, r);
    `quicksort`$(A, p, q-1)$; `quicksort`$(A, q+1, r)$

function `partition` (A, p, r)
  $x \leftarrow A[p]$; $i \leftarrow p$; $j \leftarrow r$; $k \leftarrow r$;
  while $i < k$ **do**
    if $A[j] \leqslant x$ **then**
      $A[i] \leftarrow A[j]$; $i \leftarrow i+1$; $j \leftarrow i$
    else
      $A[k] \leftarrow A[j]$; $k \leftarrow k-1$; $j \leftarrow k$
  $A[j] \leftarrow x$; **return** j

Algorithm 14: quicksort

input : an array $A[1..n]$ containing a sequence of n numbers $(a_1, a_2, \ldots, a_n)$
output: the array A containing a permutation $(a'_1, a'_2, \ldots, a'_n)$ of the input sequence such that $a'_1 \leqslant a'_2 \leqslant \ldots \leqslant a'_n$

begin
  `mergesort`$(A, 1, n)$

procedure `mergesort` (A, p, r)
  if $p < r$ **then**
    $q \leftarrow \lfloor (p+r)/2 \rfloor$;
    `mergesort`(A, p, q); `mergesort`$(A, q+1, r)$;
    `merge`(A, p, q, r)

procedure `merge` (A, p, q, r)
  $n_{\mathrm{L}} \leftarrow q - p + 1$; $n_{\mathrm{R}} \leftarrow r - q$;
  create arrays $L[1..n_{\mathrm{L}}+1]$ and $R[1..n_{\mathrm{R}}+1]$;
  for $i \leftarrow 1$ **to** n_{L} **do** $L[i] \leftarrow A[p+i-1]$;
  for $j \leftarrow 1$ **to** n_{R} **do** $R[j] \leftarrow A[q+j]$;
  $L[n_{\mathrm{L}}+1] \leftarrow \infty$; $R[n_{\mathrm{R}}+1] \leftarrow \infty$;
  $i \leftarrow 1$; $j \leftarrow 1$;
  for $k \leftarrow p$ **to** r **do**
    if $L[i] \leqslant R[j]$ **then** $A[k] \leftarrow L[i]$; $i \leftarrow i+1$;
    else $A[k] \leftarrow R[j]$; $j \leftarrow j+1$

Algorithm 15: mergesort

```
function empty
    allocate memory for an array h[0..maxHeapSize];
    return (0,h)

function isEmpty (n, h)
    if n = 0 then
        return true
    else
        return false

function minimum (n, h)
    return h[1]

function swap ((n, h), i, j)
    t ← h[i]; h[i] ← h[j]; h[j] ← t;
    return (n, h)

function toggleUp ((n, h), i)
    j ← ⌊i/2⌋;
    if i > 1 and h[j] > h[i] then
        return toggleUp(swap((n, h), i, j), j)
    else
        return (n, h)

function toggleDown ((n, h), i)
    j ← 2 ∗ i;
    if j + 1 ⩽ n and h[j + 1] < h[j] then j ← j + 1;
    if j ⩽ n and h[j] < h[i] then
        return toggleDown(swap((n, h), i, j), j)
    else
        return (n, h)

function insert ((n, h), t)
    h[n] ← t; n ← n + 1;
    return toggleUp((h, n), n)

function deleteMin ((n, h))
    h[1] ← h[n]; n ← n − 1;
    return toggleDown((h, n), 1)
```

Algorithm 16: binary heap

```
function empty
    return nil

function isEmpty(h)
    if h = nil then
        return true
    else
        return false

function minimum(h)
    m ← ∞;
    while h ≠ nil do
        if h.key < m then m ← h.key; i ← h.info;
        h ← h.sibling
    return (m, i)

function link(s, t)
    if s.key < t.key then
        t.sibling ← s.child;
        s.child ← t;
        s.degree ← s.degree + 1;
        return s
    else
        s.sibling ← t.child;
        t.child ← s;
        t.degree ← t.degree + 1;
        return t

function merge(g, h)
    if isEmpty(g) then return h;
    if isEmpty(h) then return g;
    if g.degree < h.degree then
        g.sibling ← merge(g.sibling, h);
        return g
    if g.degree > h.degree then
        h.sibling ← merge(g, h.sibling);
        return h
    x ← merge(g.sibling, h.sibling);
    y ← link(g, h); y.sibling ← nil;
    return merge(x, y)
```

Algorithm 17: binomial heap

```
function insert((k, i), h)
    allocate a new cell at t;
    t.degree ← 0;
    t.key ← k;
    t.info ← i;
    t.child ← nil;
    t.sibling ← nil;
    return merge(t, h)

function deleteMin(h)
    (m, i) ← minimum(h);
    if h.key = m then
        x ← h.sibling;
        t ← h;
        t.sibling ← nil
    else
        x ← h; z ← h;
        while z.sibling.key ≠ m do z ← z.sibling;
        t ← z.sibling;
        z.sibling ← z.sibling.sibling
    s ← t.child;
    y ← nil;
    while s.sibling ≠ nil do
        z ← s;
        z.sibling ← y;
        y ← z;
        s ← s.sibling
    deallocate the cell at t;
    return merge(x, y)
```

Algorithm 18: binomial heap—continued

input : integers $n \geqslant m \geqslant 0$ and arrays $T[1..n]$ (text) and $P[1..m]$ (pattern)
output: the set $S \subseteq \{0, 1, \ldots, n-m\}$ of valid shifts

```
begin
    S ← ∅;
    for s ← 0 to n − m do
        found ← true;
        for i ← 1 to m do
            found ← found ∧ (P[i] = T[s + i])
        if found then S ← S ∪ {s};
```

Algorithm 19: String Matching—Naive Algorithm

input : integers $n \geqslant m \geqslant 0$ and arrays $T[1..n]$ (text) and $P[1..m]$ (pattern)
output: the set $S \subseteq \{0, 1, \ldots, n-m\}$ of valid shifts

begin
  `computeTransitionFunction`;
  $S \leftarrow \varnothing$; $q \leftarrow 0$;
  for $i \leftarrow 1$ **to** n **do**
    $q \leftarrow \delta(q, T[i])$;
    if $q = m$ **then** $S \leftarrow S \cup \{i - m\}$;

procedure `computeTransitionFunction`
  for $q \leftarrow 0$ **to** m **do**
    for $a \in \Sigma$ **do**
      $k \leftarrow \min\{m, q+1\}$;
      while $P_k \not\sqsupseteq P_q a$ **do** $k \leftarrow k - 1$;
      $\delta(q, a) \leftarrow k$

Algorithm 20: String Matching—DFA Matcher

input : integers $n \geqslant m \geqslant 0$ and arrays $T[1..n]$ (text) and $P[1..m]$ (pattern)
output: the set $S \subseteq \{0, 1, \ldots, n-m\}$ of valid shifts

begin
  `computePrefixFunction`;
  $S \leftarrow \varnothing$; $q \leftarrow 0$;
  for $i \leftarrow 1$ **to** n **do**
    while $q > 0$ **and** $P[q+1] \neq T[i]$ **do** $q \leftarrow \pi(q)$;
    if $P[q+1] = T[i]$ **then** $q \leftarrow q + 1$;
    if $q = m$ **then** $S \leftarrow S \cup \{i - m\}$; $q \leftarrow \pi(q)$;

procedure `computePrefixFunction`
  $\pi[1] \leftarrow 0$; $k \leftarrow 0$;
  for $q \leftarrow 2$ **to** m **do**
    while $k > 0$ **and** $P[k+1] \neq P[q]$ **do** $k \leftarrow \pi[k]$;
    if $P[k+1] = P[q]$ **then** $k \leftarrow k + 1$;
    $\pi[q] \leftarrow k$

Algorithm 21: Knuth/Morris/Pratt Algorithm