Algorithms Booklet

2010

Note: in all algorithms, unless stated otherwise, the input graph G is represented by adjacency lists.

1 Graph Searching

1.1 Breadth First Search

Algorithm 1: BFS(G, s)

Input: $G = (V, E)$ is a (directed or undirected) graph;
$s \in V$ is a designated vertex of G.
Output: $d : V \to \mathbb{N} \cup \{\infty\}$ is the distance from s function;
$\pi : V \to V \cup \{\text{NIL}\}$ is the predecessor function.

```
1  foreach vertex u ∈ V \ {s} do
2      color[u] ← WHITE                          // WHITE means: not yet in queue
3      d[u] ← ∞
4      π[u] ← NIL
5  color[s] ← GRAY                               // GRAY means: in queue
6  d[s] ← 0
7  π[s] ← NIL
8  Q ← an empty queue
9  Enqueue(Q, s)
10 while Q is not empty do
11     u ← head of Q
12     foreach (out)neighbor v of u do           // i.e., (u, v) ∈ E
13         switch color[v] do
14             case WHITE:
15                 color[v] ← GRAY
16                 d[v] ← d[u] + 1
17                 π[v] ← u
18                 Enqueue(Q, v)
19                 mark (u, v) as a tree edge
20             case GRAY:
21                 if d[u] = d[v] then
22                     mark (u, v) as a cross edge
23                 else                          // necessarily d[v] = d[u] + 1
24                     mark (u, v) as a forward edge
25             case BLACK:
26                 mark (u, v) as a back edge    // can only happen in directed graphs
27     Dequeue(Q)
28     color[u] ← BLACK                          // BLACK means: already out of queue
29 return d, π
```

1.2 Depth First Search

Algorithm 2: DFS(G)

Input: $G = (V, E)$ is a (directed or undirected) graph.
Output: $d, f : V \to \mathbb{N}$ are the discovery and finish functions;
$\pi : V \to V \cup \{\text{NIL}\}$ is the predecessor function.

```
1  foreach vertex u ∈ V do color[u] ← WHITE        // WHITE means: not yet discovered
2  time ← 0
   // The first DFS-Visit may start in a designated vertex s, like in BFS.
3  foreach vertex u ∈ V do
4      if color[u] = WHITE then
5          π[u] ← NIL                                  // u is a root of its DFS tree
6          DFS-Visit(u)
7  return d, f
```

Procedure DFS-Visit(u)

Input: u is a WHITE vertex in a graph G.
Result: $d[u], f[u]$ and $\pi[u]$ are calculated while searching the subtree of u.

```
1  color[u] ← GRAY                          // GRAY means: discovered, not yet finished
2  time ← time + 1
3  d[u] ← time
4  foreach (out)neighbor v of u do                                   // i.e., (u, v) ∈ E
5      switch color[v] do
6          case WHITE:
7              mark (u, v) as a tree edge
8              π[v] ← u
9              DFS-Visit(v)
10         case GRAY:
11             mark (u, v) as a back edge
12         case BLACK:
13             if d[v] > d[u] then
14                 mark (u, v) as a forward edge
15             else
16                 mark (u, v) as a cross edge
17 color[u] ← BLACK                        // BLACK means: finished with u's subtree
18 time ← time + 1
19 f[u] ← time
```

2 Minimum Spanning Trees

2.1 Kruskal

Algorithm 3: Kruskal(G, w)

Input: $G = (V, E)$ is a connected, undirected graph;
$w : E \to \mathbb{R}$ is an edge weight function.
Output: $T = (V, A)$ is a minimum spanning tree of G with respect to w.

```
A ← ∅
foreach vertex v ∈ V do
    Make-Set(v)                          // initialize the Union-Find data structure
Sort E by non-decreasing weight w
foreach edge (u, v) ∈ E do               // according to the order computed above
    if Find-Set(u) ≠ Find-Set(v) then
        A ← A ∪ {(u, v)}
        Union(u, v)
return (V, A)
```

2.2 Prim

Algorithm 4: Prim(G, w, r)

Input: $G = (V, E)$ is a connected, undirected graph;
$w : E \to \mathbb{R}$ is an edge weight function;
$r \in V$ is an arbitrary vertex of G.
Output: $T = (V, A)$ is a minimum spanning tree of G with respect to w, rooted at r.

```
Q ← an empty priority queue
foreach vertex v ∈ V \ {r} do
    key[v] ← ∞
key[r] ← 0
π[r] ← NIL
insert V to Q using key as priorities
while Q is not empty do
    u ← Extract-Min(Q)
    foreach neighbor v of u do
        if v ∈ Q and w(u, v) < key[v] then
            π[v] ← u
            key[v] ← w(u, v)             // also call Decrease-Key to update Q
A ← {(π[u], u) | u ∈ V \ {r}}
return (V, A)
```

3 Single Source Shortest Paths

Procedure Initialize-Single-Source(G, s)

Input: $G = (V, E)$ is a (directed or undirected) graph;
$s \in V$ is a designated vertex of G.
Result: Distance function d and predecessor function π are initialized.

```
foreach vertex v ∈ V do
    d[v] ← ∞
    π[v] ← NIL
d[s] ← 0
```

Procedure Relax(u, v, w)

Input: $u, v \in V$ are two vertices of $G = (V, E)$ and $(u, v) \in E$;
$w : E \to \mathbb{R}$ is an edge weight function.
Result: Distance function d and predecessor function π may be updated.

```
1 if d[v] > d[u] + w[u, v] then
2     d[v] ← d[u] + w[u, v]
3     π[v] ← u
```

3.1 Dijkstra

Algorithm 5: Dijkstra(G, s, w)

Input: $G = (V, E)$ is a (directed or undirected) graph;
$s \in V$ is a designated vertex of G;
$w : E \to \mathbb{R}^+$ is a *non-negative* edge weight function.
Output: $d : V \to \mathbb{R}^+$ is the distance from s function;
$\pi : V \to V \cup \{\text{NIL}\}$ is the predecessor function.

```
1  Initialize-Single-Source (G, s)
2  S ← ∅
3  Q ← an empty priority queue
4  insert V to Q using d as priorities
5  while Q is not empty do
6      u ← Extract-Min(Q)
7      S ← S ∪ {u}
8      foreach (out)neightbor v of u do
9          Relax (u, v, w)        // also call Decrease-Key to update Q if d is changed
10 return d, π
```

3.2 Bellman–Ford

Algorithm 6: Bellman–Ford(G, s, w)

Input: $G = (V, E)$ is a (directed or undirected) graph;
$s \in V$ is a designated vertex of G;
$w : E \to \mathbb{R}$ is an edge weight function.
Output: NIL if G contains a negative cycle reachable from s; otherwise
$d : V \to \mathbb{R}$ is the distance from s function;
$\pi : V \to V \cup \{\text{NIL}\}$ is the predecessor function.

```
1  Initialize-Single-Source (G, s)
2  for i ← 1 to |V| − 1 do
3      foreach edge (u, v) ∈ E do  Relax (u, v, w)
   // Check for negative cycles
4  foreach edge (u, v) ∈ E do
5      if d[v] > d[u] + w[u, v] then                    // can Relax further?
6          return NIL
7  return d, π
```

4 All Pairs Shortest Paths

4.1 Matrix "Multiplication"

Function Extend-Shortest-Paths(D, W)

Input: D is the $n \times n$ distance matrix;
W is the $n \times n$ edge weight matrix.
Output: D' is the extended $n \times n$ distance matrix.

```
for i ← 1 to n do
    for j ← 1 to n do
        d'_ij ← min {d_ik + w_kj | k ∈ {1, ..., n}}
return D'
```

Algorithm 7: Slow-All-Pairs-Shortest-Paths(W)

Input: W is the $n \times n$ edge weight matrix.
Output: D is the $n \times n$ distance matrix.

```
// The algorithm maintains n − 1 distance matrices D^(1), ..., D^(n−1).
D^(1) ← W
for m ← 2 to n − 1 do
    D^(m) ← Extend-Shortest-Paths(D^(m−1), W)
return D^(n−1)
```

Algorithm 8: Faster-All-Pairs-Shortest-Paths(W)

Input: W is the $n \times n$ edge weight matrix.
Output: D is the $n \times n$ distance matrix.

```
D^(0) ← W
for k ← 1 to ⌈log_2(n − 1)⌉ do        // in terms of the slow algorithm, k = log_2 m
    D^(k) ← Extend-Shortest-Paths(D^(k−1), D^(k−1))
return D^(⌈log_2(n−1)⌉)
```

4.2 Floyd–Warshall

Algorithm 9: Floyd–Warshall(W)

Input: W is the $n \times n$ edge weight matrix.
Output: D is the $n \times n$ distance matrix.

```
D^(0) ← W
for k ← 1 to n do                                   // compute D^(k) based on D^(k−1)
    for i ← 1 to n do
        for j ← 1 to n do
            d_ij^(k) ← min{d_ij^(k−1), d_ik^(k−1) + d_kj^(k−1)}
return D^(n)
```

4.3 Johnson

Algorithm 10: Johnson(G, w)

Input: $G = (V, E)$ is a (directed or undirected) graph;
$w : E \to \mathbb{R}$ is an edge weight function.
Output: NIL if G contains a negative cycle; otherwise
$d : V \times V \to \mathbb{R}$ is the distance function;
$\pi : V \times V \to V \cup \{\text{NIL}\}$ is the predecessor function.

```
1  V' ← V ∪ {s}                                              // s is a new vertex
2  E' ← E ∪ {(s, v) | v ∈ V}
3  G' ← (V', E')
4  foreach vertex v ∈ V do  w(s, v) ← 0                      // extend weight function to G'
5  if Bellman-Ford(G', s, w) = NIL then
6      return NIL                                            // G has some negative cycle
7  else
8      h ← the distance function computed by the above Bellman-Ford
9      foreach edge (u, v) ∈ E' do
10         w'(u, v) ← w(u, v) + h[u] − h[v]                   // necessarily non-negative
11     foreach vertex u ∈ V do
12         (δ_u, π_u) ← Dijkstra(G, u, w')
13         foreach vertex v ∈ V do
14             d(u, v) ← δ_u[v] + h[v] − h[u]
15             π(u, v) ← π_u[v]
16     return d, π
```

5 Maximum Flow

5.1 Ford–Fulkerson

Function Construct-Residual-Network(N, f)

Input: $N = (V, E, c, s, t)$ is an (s, t)-flow network with edge capacity function c;
$f : E \to \mathbb{R}$ is a valid (s, t)-flow in N.
Output: $N_f = (V, E_f, c_f, s, t)$ is the residual network of N with respect to f.

```
1  E_f ← ∅
2  foreach edge (u, v) ∈ E or (v, u) ∈ E do
3      c_f[u, v] ← c[u, v] − f[u, v]
4      if c_f[u, v] > 0 then
5          E_f ← E_f ∪ {(u, v)}
6  return (V, E_f, c_f, s, t)
```

Algorithm 11: Ford–Fulkerson(N)

Input: $N = (V, E, c, s, t)$ is an (s, t)-flow network with edge capacity function c.
Output: $f : E \to \mathbb{R}$ is a maximum (s, t)-flow in N.

```
1  foreach edge (u, v) ∈ E do                          // initialize f to the zero flow
2  |  f[u, v] ← 0
3  |  f[v, u] ← 0
4  repeat
5  |  N_f ← Construct-Residual-Network(N, f)
6  |  if there exists a path p : s ⇝ t in N_f then      // augment the flow along p
7  |  |  select such a path p                          // algorithm for finding p is not specified
8  |  |  c_f(p) ← min {c_f[u, v] | (u, v) ∈ p}
9  |  |  foreach edge (u, v) ∈ p do
10 |  |  |  f[u, v] ← f[u, v] + c_f(p)
11 |  |  |  f[v, u] ← −f[u, v]
12 until f does not change
13 return f
```

5.2 Edmonds–Karp

Algorithm 12: Edmonds–Karp(N)

Input: $N = (V, E, c, s, t)$ is an (s, t)-flow network with edge capacity function c.
Output: $f : E \to \mathbb{R}$ is a maximum (s, t)-flow in N.

```
1  foreach edge (u, v) ∈ E do                          // initialize f to the zero flow
2  |  f[u, v] ← 0
3  |  f[v, u] ← 0
4  repeat
5  |  N_f ← Construct-Residual-Network(N, f)
6  |  if there exists a path p : s ⇝ t in N_f then
7  |  |  select a shortest such path p, using BFS(N_f, s)   // w.r.t. number of edges
8  |  |  c_f(p) ← min {c_f[u, v] | (u, v) ∈ p}
9  |  |  foreach edge (u, v) ∈ p do
10 |  |  |  f[u, v] ← f[u, v] + c_f(p)
11 |  |  |  f[v, u] ← −f[u, v]
12 until f does not change
13 return f
```

5.3 Dinic

Function Construct-Layered-Network(N)

Input: $N = (V, E, c, s, t)$ is an (s, t)-flow network with edge capacity function c.
Output: $L = (V', E', c, s, t)$ is the layered network of N.

```
1  G ← (V, E)
2  (d, π) ← BFS(G, s)
3  V' ← {v ∈ V | d[v] ≤ d[t]}
4  E' ← {(u, v) ∈ E | d[v] = d[u] + 1}          // keep just tree and forward edges
   L ← (V', E', c, s, t)
5  return L
```

Function Compute-Blocking-Flow(L)

Input: $L = (V, E, c, s, t)$ is a layered network of an (s, t)-flow network with capacities c.
Output: $g : E \to \mathbb{R}$ is a blocking (s, t)-flow in L.

```
1  g ← the zero flow on all edges of E
2  state ← INITIALIZE
3  repeat
4      switch state do
5          case INITIALIZE:
6              v ← s
7              p ← an empty path starting at s
8              state ← ADVANCE
9          case ADVANCE:
10             if v has out-degree zero then state ← RETREAT
11             else
12                 select some edge (v, w) ∈ E
13                 v ← w
14                 append w to p
15                 if v = t then state ← AUGMENT     // otherwise it remains ADVANCE
16         case AUGMENT:
17             c(p) ← min {c[u, v] | (u, v) ∈ p}
18             foreach edge (u, v) ∈ p do
19                 g[u, v] ← g[u, v] + c(p)
20                 c[u, v] ← c[u, v] − c(p)
21                 if c[u, v] = 0 then remove (u, v) from L        // the edge was saturated
22             state ← INITIALIZE
23         case RETREAT:
24             if v = s then state ← DONE
25             else
26                 let u be the predecessor of v on p
27                 remove v from p
28                 remove (u, v) from L
29                 v ← u
30                 state ← ADVANCE
31 until state = DONE
32 return g
```

Algorithm 13: Dinic(N)

Input: $N = (V, E, c, s, t)$ is an (s, t)-flow network with edge capacity function c.
Output: $f : E \to \mathbb{R}$ is a maximum (s, t)-flow in N.

```
1  foreach edge (u, v) ∈ E do                          // initialize f to the zero flow
2      f[u, v] ← 0
3      f[v, u] ← 0
4  repeat
5      N_f ← Construct-Residual-Network(N, f)
6      L_f ← Construct-Layered-Network(N_f)
7      if there exists a path p : s ⇝ t in L_f then     // such p is also in N_f
8          g ← Compute-Blocking-Flow(L_f)
9          foreach edge (u, v) ∈ E do
10             f[u, v] ← f[u, v] + g[u, v]
11             f[v, u] ← −f[u, v]
12 until f does not change
13 return f
```

6 String Matching

6.1 Finite Automaton

Function Compute-Transition-Function(P)

Input: $P[1 \dots m]$ is a pattern of length m over the alphabet Σ.
Output: $\delta : Q \times \Sigma \to Q$ is the transition function of the finite automaton accepting P.

```
1  for j ← 0 to m do
2      foreach a ∈ Σ do
3          k ← min{m, j + 1}
4          while P[1 ... k] is not a suffix of P[1 ... j]a do
5              k ← k − 1
6          δ(q_j, a) ← q_k
7  return δ
```

Algorithm 14: Finite-Automaton-Matcher(T, P)

Input: $T[1 \dots n]$ is a text of length n;
$P[1 \dots m]$ is a pattern of length m.
Output: $I \subseteq \{1, \dots, n\}$ is the set of all indices i such that $T[(i - m + 1) \dots i] = P$.

```
1  I ← ∅
2  δ ← Compute-Transition-Function(P)
3  q ← q_0
4  for i ← 1 to n do
5      q ← δ(q, T[i])
6      if q = q_m then
7          I ← I ∪ {i}
8  return I
```

6.2 Knuth–Morris–Pratt

Function Compute-Prefix-Function(P)

Input: $P[1 \dots m]$ is a pattern of length m.
Output: $\pi : \{1, \dots, m\} \to \{0, \dots, m-1\}$ is the prefix function.

```
1 π[1] ← 0
2 k ← 0
3 for j ← 2 to m do
4 |   while k > 0 and P[k+1] ≠ P[j] do
5 |   |   k ← π[k]
6 |   if P[k+1] = P[j] then
7 |   |   k ← k+1
8 |   π[j] ← k
9 return π
```

Algorithm 15: KMP(T, P)

Input: $T[1 \dots n]$ is a text of length n;
$P[1 \dots m]$ is a pattern of length m.
Output: $I \subseteq \{1, \dots, n\}$ is the set of all indices i such that $T[(i-m+1) \dots i] = P$.

```
1  I ← ∅
2  π ← Compute-Prefix-Function(P)
3  j ← 0
4  for i ← 1 to n do
5  |   while j > 0 and P[j+1] ≠ T[i] do
6  |   |   j ← π[j]
7  |   if P[j+1] = T[i] then
8  |   |   j ← j+1
9  |   if j = m then                          // P fully matches here
10 |   |   I ← I ∪ {i}                       // T[i] is the last letter of P
11 |   |   j ← π[j]
12 return I
```