Algorithm Knapsack01(S, maxW)

M <- Array[S.size(), maxW + 1] // 0<=i<=n, 0<=j<=maxW

for w <- 0 to maxW+1 do

M[0, w] <- 0

for k <- 0 to S.size() do

M[k, 0] <- 0

//k <- 0

for k <- 1 to S.size() do

//if w <= maxW then

processKnapsack(S, k, S.elementAtRank(0).w)

Algorithm processKnapsack(S, k, wR)

if k = 0 \/ w = 0 do

return M[k, w]

if M[k, wR] > 0 then

return M[k, wR]

bk <- S.elementAtRank(k-1).benefit()

wk <- S.elementAtRank(k-1).weight()

for w <- 1 to maxW do

if w < wk then

M[k, w] = processKnapsack(S, k-1, w) //M[k-1,w]

else

//return max(M[k-1, w], M[k-1, w - wk] + bk)

M[k, w] = max(processKnapsack(S, k-1, w), processKnapsack(S, k-1, w-wk) + bk)

return M[k, wR]

//Prof's

Algorithm 0-1-KnapsackH(S, k, w)

if k = 0 ˅ w = 0 then // base cases

return 0

else

bk <- S.elemAtRank(k-1).benefit() // benefit of elem k

wk <- S.elemAtRank(k-1).weight() // weight of elem k

if wk > w then // element k does not fit in size w knapsack

return 0-1-KnapsackH(S, k-1, w) // cannot include elem k

else

return max(0-1-KnapsackH(S, k-1, w), // don’t include elem k

0-1-KnapsackH(S, k-1, w-wk) + bk ) // include elem k

--------------------------------------------------------------------------------------------------------------------------------------------------------------------

// 0/1 Knapsack without recursion

Algorithm Knapsack01(S, maxW) O(nMaxW)

n <- S.size()

M <- Array[i, j], 0<=i<=n, 0<=j<=maxW

m[k, l] <- 0, k=0, 0<=l<=maxW [maxW]

m[k, l] <- 0, 0<=k<=n, l=0 [n]

r <- 0

for each (b,w) in S do [n]

r++

for c<-1 to maxW do [maxW]

if w > c then

M[r,c] <- M[r-1, c]

else

lw <- c - w

M[r, c] <- max(M[r-1, c], M[r-1, lw] + b)

maxB <- 0

leftW <- maxW

R <- new Sequence

for i <- n down to 1 do [n]

if leftW > 0 then

(b,w) <- S.elementAtRank(i-1)

if M[i, leftW] <> M[i-1, leftW] do

leftW <- leftW - w

R.insertItem((b,w))

maxB <- maxB + M[i, leftW]

return (R, maxB)

--------------------------------------------------------------------------------------------------------------------------------------------------------------------

// Algorithm determines the subset of objects that produces the largest sum of sizes that is no greater than positive upper limit L

Algorithm Knapsack01(S, L)

n <- S.size()

M <- Array[i, j], 0<=i<=n, 0<=j<=L

m[k, l] <- 0, k=0, 0<=l<=L

m[k, l] <- 0, 0<=k<=n, l=0

r <- 0

for each (b,w) in S do

r++

for c<-1 to L do

if w > c then

M[r,c] <- M[r-1, c]

else

lw <- c - w

M[r, c] <- max(M[r-1, c], M[r-1, lw] + w)

leftW <- maxW

R <- new Sequence

for i <- n down to 1 do

if leftW > 0 then

(b,w) <- S.elementAtRank(i-1)

if M[i, leftW] <> M[i-1, leftW] do

leftW <- leftW - w

R.insertItem((b,w))

return R

--------------------------------------------------------------------------------------------------------------------------------------------------------------------

//DFS

Algorithm DFS(G)

Iterator verticesIter <- G.vertices()

while verticesIter.hasNext() do

v <- verticesIter.nextItem()

setLabel(v, UNEXPLORED)

Iterator edgesIter <- G.edges()

while edgesIter.hasNext() do

e <- edgesIter.nextItem()

setLabel(e, UNEXPLORED)

for each v in verticesIter do

if getLabel(v) = UNEXPLORED then //need to check this when all vertices are initialized to UNEXPLORED?

DFS(G, v)

--------------------------------------------------------------------------------------------------------------------------------------------------------------------

//Using recursion

\_DFS(G, v)

if getLabel(v) = UNEXPLORED then

setLabel(v, VISITED)

Iterator edgesIter <- G.incidentEdges(v)

for each e in edgesIter do

if getLabel(e) = UNEXPLORED then

w <- G.apposite(v, e)

if getLabel(w) = UNEXPLORED then

setLabel(e, DISCOVERY) // draw a straight line

\_DFS(G, w)

else

setLabel(e, BACK) //draw a dot line

else

//e is DISCOVERY or BACK, therefore no update more

--------------------------------------------------------------------------------------------------------------------------------------------------------------------

//Using template method

public abstract class DFS {

protected \_DFS\_Template(G, v){

checkInput(G, v)

initial()

mark(v)

startVisit(v)

Iterator edgesIter <- queryIncidentEdges(v)

for each e in edgesIter do

if isMarked(e) = false then

discoveryEdge(v, e)

w <- G.opposite(v, e)

if isMarked(w) = false then

mark(e)

\_DFS\_Template(G, w)

else

backEdge(v, e)

finishVisit(v)

}

}

--------------------------------------------------------------------------------------------------------------------------------------------------------------------

//Using a stack instead of recursion

\_DFS(G, v)

if G.degree(v) = 0 then

setLabel(v, VISITED)

else

S <- new Stack()

setLabel(v, VISITED)

S.push(v)

while not S.isEmpty() do

v <- S.pop() //S.top() //it should be popped when checked its incidentEdges and pushed them to the stack if UNEXPLORED yet

//setLabel(v, VISITED)

//bDiscovered <- true

for each e in G.incidentEdges(v) do

if getLabel(e) = UNEXPLORED then

setLabel(e, DISCOVERY)

w <- S.opposite(v, e)

if getLabel(w) = UNEXPLORED then

setLabel(w, VISITED)

S.push(w) // just pushed to the stack regardless of having incident edges or not

else

setLabel(e, BACK)

else

// do nothing

--------------------------------------------------------------------------------------------------------------------------------------------------------------------

//find simple path

\_DFS(G, v, z, S)

S.push(v)

for each e in G.incidentEdges(v) do

???

--------------------------------------------------------------------------------------------------------------------------------------------------------------------

//BSF without recursion

Algorithm BFS(G)

Iterator verticesIter <- G.vertices()

while verticesIter.hasNext() do

v <- verticesIter.nextItem()

setLabel(v, UNEXPLORED)

Iterator edgesIter <- G.edges()

while edgesIter.hasNext() do

e <- edgesIter.nextItem()

setLabel(e, UNEXPLORED)

Q <- new Queue()

v <- G.aVertex()

setLabel(v, VISITED)

Q.enqueue(v)

while not Q.isEmpty() do

v <- Q.dequeue()

for each e in G.incidentEdges(v) do

w <- G.opposite(v, e)

if getLabel(w) = UNEXPLORED then

setLabel(e, DISCOVERY)

setLabel(w, VISITED)

Q.enqueue(w)

else

setLabel(e, BACK)

--------------------------------------------------------------------------------------------------------------------------------------------------------------------

//BSF applying the template method

Algorithm BFS\_TemplateMethod(G, startV)

initResult(G) //Template

Iterator verticesIter <- G.vertices()

while verticesIter.hasNext() do

vt <- verticesIter.nextItem()

setLabel(vt, UNEXPLORED)

preInitVertex(vt) //Template

Iterator edgesIter <- G.edges()

while edgesIter.hasNext() do

e <- edgesIter.nextItem()

setLabel(e, UNEXPLORED)

preInitEdge(e) //Template

Q <- new Queue()

//v <- G.aVertex()

setLabel(v, VISITED)

startVertexVisit(G, v) //Template

Q.enqueue(startV)

while not Q.isEmpty() do

v <- Q.dequeue()

for each e in G.incidentEdges(v) do

w <- G.opposite(v, e)

if getLabel(w) = UNEXPLORED then

preDiscoveryTraversal(G, v, e, w) //Template

setLabel(e, DISCOVERY)

setLabel(w, VISITED)

Q.enqueue(w)

postDiscoveryTraversal(G, v, e, w) //Template

else

setLabel(e, BACK)

backTraversal(G, v, e, w) //Template

finishVertexVisit(G, startV) //Template

--------------------------------------------------------------------------------------------------------------------------------------------------------------------

startVertexVisit(G, v)

setLevel(v, 0)

postDiscoveryTraversal(G, v, e, w)

l <- v.getLevel()

setLevel(w, l)

--------------------------------------------------------------------------------------------------------------------------------------------------------------------

Algorithm findPath(G, u, v)

S <- new Stack

minPath <- 0

BFS\_TemplateMethod(G, v)

return minPath

initResult(G)

minVertex <- v

minEdges <- 0

startVertexVisit(G, v)

setParent(v, 0)

setLevel(v, 0)

postDiscoveryTraversal(G, v, e, w)

setParent(w, e)

l <- getLevel(v) + 1

setLevel(w, l)

if w = u /\ l < minEdges then

minVertex <- w

minEdges = l

finishVertexVisit(G, v)

if minVertex = v then

return minPath

//Using the backtracking to find the path with minimum number of edges

S <- new Stack

z <- minVertex

while z <> v do

S.push(z)

e <- z.getParent()

S.push(e)

z <- G.opposite(z, e)

S.push(v)

minPath <- S.elements()

return minPath

--------------------------------------------------------------------------------------------------------------------------------------------------------------------

findCycle(G)

cycle <- new Sequence

startV <- G.aVertex()

BFS\_TemplateMethod(G, startV)

return cycle

startVertexVisit(G, v)

setParent(v, 0)

backTraversal(G, v, e, w)

// How to trace back the cycle???

D1 <- new Dictionary(HT)

D1.insertItem(v, v.getParent())

D2 <- new Dictionary(HT)

D2.insertItem(w, w.getParent())

f <- false

while f = false do

v <- G.opposite(v, v.getParent())

w <- G.opposite(w, w.getParent())

if(v = w)

f <- true

else

p <- D2.findElement(v)

if p <> NO\_SUCH\_KEY then

makeCycle(cycle, D2, p, D1)

f <- true

else

D1.insertItem(v, v.getParent())

p <- D1.findElement(w)

if p <> NO\_SUCH\_KEY then

makeCycle(cycle, D1, p, D2)

f <- true

else

D2.insertItem(w, w.getParent())

makeCycle(cycle, B1, p, B2)

//Output p items from D2

Vs <- B1.keys()

Es <- B1.elements()

i <- 0

for each (k,f) in B1.items() do

cycle.insertLast(k)

cycle.insertLast(f)

i++

if i = p then //just get p items

break

//Output edge e between v & w

cycle.insertFirst(e)

//Output all items from D1

for each (k,f) in B2.items() do

cycle.insertFirst(k)

cycle.insertFirst(f)

--------------------------------------------------------------------------------------------------------------------------------------------------------------------

// Dijkstra algorithm

// Using new methods: set/getLabel(), set/getDistance(), weight(), replaceKey(u: element of key, new key)

Algorithm Dijkstra(G)

//1. Select a start vertex

s <- G.aVertex()

H <- new Heap()

//2. Set other vertices to the infinitive

for each v in G.vertices() do

setLabel(v, UNEXPLORED) //new

if s = v then

H.insertItem(0, s)

else

setDistance(v, infinitive)

H.insertItem(infinitive, v)

//3. Set distance for endpoints of the start vertex's edges by applying the relaxation for other vertices

a. find the smallest distance

b. update the distance of the related endpoint if its is greater than

while not H.isEmpty() do

v <- H.removeMin()

setLabel(v, VISITED) //new

for each e in G.incidentEdges(v) do

u <- G.oppsite(v, e)

if getLabel(u) = UNEXPLORED then //new

d <- getDistance(v) + weight(e)

du <- getDistance(u)

if (d < du) then

setDistance(u, d)

H.replaceKey(u, d)

--------------------------------------------------------------------------------------------------------------------------------------------------------------------

// Dijkstra algorithm (using Locator)

// Using new methods: insert(k, e), replaceKey(u: element of key, new key)

Algorithm Dijkstra(G)

//1. Select a start vertex

s <- G.aVertex()

H <- new Heap()

//2. Set other vertices to the infinitive

for each v in G.vertices() do

setLabel(v, UNEXPLORED) //new

if s = v then

setDistance(v, 0)

else

setDistance(v, infinitive)

l <- H.insert(getDistance(v), v)

H.setLocator(v, l)

//3. Set distance for endpoints of the start vertex's edges by applying the relaxation for other vertices

a. find the smallest distance

b. update the distance of the related endpoint if its is greater than

while not H.isEmpty() do

v <- H.min()

setLabel(v, VISITED) //new

for each e in G.incidentEdges(v) do

u <- G.oppsite(v, e)

if getLabel(u) = UNEXPLORED then //new

d <- getDistance(v) + weight(e)

du <- getDistance(u)

if (d < du) then

setDistance(u, d)

H.replaceKey(getLocator(u), d)

--------------------------------------------------------------------------------------------------------------------------------------------------------------------

//Applying the template method

// --> Just add 1 more method, setParent()

Algorithm Dijkstra(G)

//1. Select a start vertex

s <- G.aVertex()

H <- new Heap()

//2. Set other vertices to the infinitive

for each v in G.vertices() do

setLabel(v, UNEXPLORED) //new

if s = v then

setDistance(v, 0)

//Template method

setParent(v, 0)

else

setDistance(v, infinitive)

l <- H.insert(getDistance(v), v)

H.setLocator(v, l)

//3. Set distance for endpoints of the start vertex's edges by applying the relaxation for other vertices

a. find the smallest distance

b. update the distance of the related endpoint if its is greater than

while not H.isEmpty() do

v <- H.min()

setLabel(v, VISITED) //new

for each e in G.incidentEdges(v) do

u <- G.oppsite(v, e)

if getLabel(u) = UNEXPLORED then //new

d <- getDistance(v) + weight(e)

du <- getDistance(u)

if (d < du) then

setDistance(u, d)

//Template method

setParent(u, e)

H.replaceKey(getLocator(u), d)

--------------------------------------------------------------------------------------------------------------------------------------------------------------------

//The same Dijkstra's algorithm, just differ the way of setting weight to vertices

// and using the locator to check whether it should be updated distance or not

Algorithm Prim(G)

s <- G.aVertex()

H <- new Heap()

//2. Set other vertices to the infinitive

for each v in G.vertices() do

setLabel(v, UNEXPLORED) //new

if s = v then

setDistance(v, 0)

else

setDistance(v, infinitive)

setParent(v, 0)

l <- H.insert(getDistance(v), v)

H.setLocator(v, l)

//3. Set distance for endpoints of the start vertex's edges by applying the relaxation for other vertices

a. find the smallest distance

b. update the distance of the related endpoint if its is greater than

while not H.isEmpty() do

v <- H.min()

setLocator(v, 0) //because v was removed from the heap

for each e in G.incidentEdges(v) do

u <- G.oppsite(v, e)

if getLocator(u) <> 0 then //if the vertex 'u' was visited (not in the heap), it's unnecessary to check more

d <- weight(e)

du <- getDistance(u)

if (d < du) then

setDistance(u, d)

setParent(u, e)

H.replaceKey(getLocator(u), d)

--------------------------------------------------------------------------------------------------------------------------------------------------------------------

Algorithm Kruskal(G)

T <- new Tree

Q <- new Heap

for each v in G.vertices() do

//insert vertex v into T

define a Cloud(v) <- {v}

for each e in G.edges() do

Q.insert(weight(e), e) //{O(m log m) which is O(m log n)}

while T.numEdges() < n-1 do

e <- Q.removeMin()

(u,v) <- G.endVertices(e)

if P.find(u) != P.find(v) then //checking whether u and v are belong to the cloud of MST or not. So what is P?

insert edge e into T

P.union(u,v) //{O(n log n) since a vertex is merged O(log n) times}

return T

--------------------------------------------------------------------------------------------------------------------------------------------------------------------

Baruvka’s Algorithm

--------------------------------------------------------------------------------------------------------------------------------------------------------------------