**CMPE 300 ANALYSIS OF ALGORITHMS**

###### MIDTERM ANSWERS

1. The fractional knapsack problem is as follows:

There is a knapsack of capacity *C*. There are *n* objects, b0, b1, ..., bn-1. Each object *i* has a weight (or, volume) wi and a value vi (0 ≤ *i* ≤ *n*-1).

We want to place the objects or fractions of the objects in the knapsack, without exceeding the capacity, such that the total value of the objects in the knapsack is maximized.

1. The formal definition is as follows:

function Knapsack (V[0:n-1], W[0:n-1], C, F[0:n-1])

Sort V[0:n-1] and W[0:n-1] in decreasing order of V[i]/W[i] values

for i ← 0 to n-1 do

F[i] ← 0

endfor

RemainCap ← C

i ← 0

if W[0] ≤ C then

Fits ← true

else

Fits ← false

endif

while (Fits) and (i ≤ n-1) do

F[i] ← 1

RemainCap ← RemainCap – W[i] (\*)

i ← i+1

if W[i] ≤ RemainCap then

Fits ← true

else

Fits ← false

endif

endwhile

if i ≤ n-1 then

F[i] ← RemainCap / W[i]

endif

end

1. Given the data below

C=10

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | b0 | b1 | b2 | b3 | b4 | b5 | b6 | b7 |
| w | 8 | 2 | 4 | 1 | 2 | 6 | 10 | 50 |
| v | 32 | 32 | 25 | 10 | 9 | 18 | 55 | 100 |

the output will be

Place w1=2 (f1=1) of b1

Place w3=1 (f3=1) of b3

Place w2=4 (f2=1) of b2

Place w6=3 (f6=0.3) of b6

The total value in the knapsack is 1.0\*32+1.0\*10+1.0\*25+0.3\*55=83.5

1. Basic operation: Comparison in sorting + (\*) statement in the loop

Then, W(n) = n\*log n + n ∈ Θ(n\*log n)

1. The difference of 0/1 knapsack from the fractional knapsack is that fraction of an object cannot be put in the knapsack; the entire object must be put. That is, fi = 0 or 1 for all *i*.

In the formulation in part (b), the only difference is: instead of “0 ≤ fi ≤ 1”, we have “fi ∈ {0,1}”.

Example:

C=6

|  |  |  |  |
| --- | --- | --- | --- |
|  | b0 | b1 | b2 |
| w | 4 | 3 | 3 |
| v | 5 | 3 | 3 |

the output will be

Place w0=4 (f0=1) of b0

The total value in the knapsack is 1.0\*5=5

However, there is a better solution:

Place w1=3 (f1=1) of b1

Place w2=3 (f2=1) of b2

The total value in the knapsack is 1.0\*3+1.0\*3=6

1. Visit Unvisited neighbors Backtrack

A B,C,D,E

B C,E

C D,E,F,H

D F,G

F G

G H

H --- to G

G (returned) --- to F

F (returned) --- to D

D (returned) --- to C

C (returned) E

E --- to C

C (returned) --- to B

B (returned) --- to A

A (returned) ---

So, the order of visits: A,B,C,D,F,G,H,E

The DFS tree is:

A

B

C

D E

F

G

H

1. Visit Unvisited neighbors Enqueue

A B,C,D,E

B,C,D,E B,C,D,E

B (dequeue) ---

C (dequeue) F,H F,H

F,H

D (dequeue) G G

G

E (dequeue) ---

F (dequeue) ---

H (dequeue) ---

G (dequeue) ---

So, the order of visits: A,B,C,D,E,F,H,G

The BFS tree is:

A

B C D E

F H G

1. The representation of the graph is as follows:

A → B → C → D → E

B → A → C → E

C → A → B → D → E → F → H

D → A → C → F → G

E → A → B → C

F → C → D → G

G → D → F → H

H → C → G

**Node A is visited**. **Link A-B is accessed**. Since B was not visited before, A is made to point to the next neighbor C and B is visited.

**Node B is visited**. **Link B-A is accessed**. A was visited before. B is made to point to the next neighbor C. **Link B-C is accessed**. Since C was not visited before, B is made to point to the next neighbor E and C is visited.

**Node C is visited**. **Link C-A is accessed**. A was visited before. C is made to point to the next neighbor B. **Link C-B is accessed**. B was visited before. C is made to point to the next neighbor D. **Link C-D is accessed**. Since D was not visited before, C is made to point to the next neighbor E and D is visited.

**Node D is visited**. **Link D-A is accessed**. A was visited before. D is made to point to the next neighbor C. **Link D-C is accessed**. C was visited before. D is made to point to the next neighbor F. **Link D-F is accessed**. Since F was not visited before, D is made to point to the next neighbor G and F is visited.

And so on.

Let n : number of nodes

m: number of edges

Basic operation: Visiting a node and Accessing an edge

Number of node visits: Since each node is visited exactly once (shown in bold above), this is *n*.

Number of edge accesses: For each edge u–v, the edge is accessed (examined) exactly twice. This can be seen in the example execution above (shown in bold above). In the adjacency list, there is an u→v link and a v→u link. When we are at node u, we access u→v link and then make u to point to the next neighbor. When we are at node v, we access v→u link and then make v to point to the next neighbor. So, there are exactly two accesses for each edge in the graph. The total number of edge accesses is 2*m*.

Then, W(n) = n+2m ∈ Θ (n+m)

1. We can view the algorithm as having three steps. Let T1 denote the number of first assignment operation, T2 the number of second assignment operation, and T3 the number of assignment operations in the recursive call. Then

A(n) = E[T] = E[T1] + E[T2] + E[T3]

E[T1] is equal to *n*. E[T3] is equal to A(n/2).

We can condition T2 on a random variable X that denotes the value of the last element of the list (i.e. L[n-1]). Then we have

The if condition checks whether an element is less than and equal to the last element of the list or not. We can observe the following:

If X=1 (the last element is the smallest element), then E[T2|X=1]=1

If X=2 (the last element is the next smallest element), then E[T2|X=2]=2

...

If X=n (the last element is the largest element), then E[T2|X=n]=n

Assume that P(X=i)=1/n, for all i. Then,

Then,

, A(1)=2

By Master theorem, A(n)∈Θ(n).