Algorithm

Homework 2

駱皓正
r05227124
Dept. of Psy

1. (20) Give an algorithm to detect whether a given undirected graph contains a cycle. If the graph contains a cycle, then your algorithm should output one. (It should not output all cycles in the graph, just one of them.) The running time of your algorithm should be $O(m+n)$ for a graph with n nodes and m edges.

Answer:

There exists a $G = (V, E)$. Here provides an algorithm to output all vertexes of a circle in G. DFS is employed to traverse G and every vertex's parent is recorded concurrently. For every explored vertex u, if there exists a back edge (u, v) which v has been visited and is not u's parent (i.e. ancestor in DSF) then there is a circle in G.

```
bool isCircleDFS(s, G) //implementation w/o recursion
// S is a stack containing vertexes whose neighbors haven't been entirely explored
S = {s}
while S is not empty do
    pop u out of S
    mark u as explored
    foreach edge (u, v) do
        if (v is not marked as explored) then
            mark u as v's parent
            S = S + {v}
        else if (v is marked as explored and not u's parent) then
            print u and v
            while (u's parent w ≠ v) do
                print w
                w = w's parent
            return True //there exists (at least) a circle
return False //there doesn't exist any circle
```

isCircleDFS(*u*, *G*) //implementation with recursion

```
1   mark u as explored
2   foreach edge (u, v) do
3       if (v is not marked as explored) then
4           mark u as v's parent
5           recursively invoke isCircleDFS(v, G)
6       else if (v is marked as explored and not u's parent) then
7           print u and v
8           while (u's parent w ≠ v) do
9               print w
10              w = w's parent
11          Exit //throw exception so as to escape from all of recursions
```

<u>Termination</u>: This algorithm terminates because if G is acyclic then all vertexes will be traversed by DFS procedure; otherwise, if G has a circle then the procedure will print the circle first encountered and exit the procedure.

<u>Correction</u>: Knowing that in a DFS tree T, if there exists a non-tree edge (implying a circle) (u, v), then u or v is an ancestor of the other. So, if an incidence v to u is a non-tree edge, then v must neither be u's child (i.e. not explored) nor u's parent. Line 9 in bool isCircleDFS(s, G) and line 6 in isCircleDFS(u, G) detect the non-tree edge in DFS procedure and output the vertexes of circle in the following lines.

<u>Running Time</u>: By worst case analysis, if there is no circle in G, the running is bounded by O(m+n) after traversal of G.

Reference:
class "graph" power point p.25 and p.35
no coworkers

2. (20) We have a connected graph $G = (V, E)$, and a specific vertex $u \in V$. Suppose we compute a depth-first search tree rooted at u, and obtain a tree T that includes all nodes of G. Suppose we then compute a breadth-first search tree rooted at u, and obtain the same tree T. Prove that G=T. (In other words, if T is both a depth-first search tree and a breadth-first search tree at u, then G cannot obtain any edges that do not belong to T.)

Answer:

$G \rightarrow T$:

Because G is tree, it indicates that there exists only one route form the root u to any other vertexes in G (i.e. there doesn't exist two or more paths from root u to another vertex v in G.) Hence, BFS and DFS procedure will build the same tree.

$T \rightarrow G$: by contradiction

Let G containing an edge (u, v) which doesn't belong to T.

(u, v) is a non-tree edge of T.

If (u, v) is a non-tree edge in BFS then u and v differ by at most one level.

If (u, v) is a non-tree edge in DFS then u or v is an ancestor of the other.

Because BFS and DFS build the same tree, u and v differ by at most one level and u or v is an ancestor of the other. This tells (u, v) is a tree edge. (contradiction!)

There is no edge in G which is not in T.

Reference:

class "graph" power point p.20 and p.25;

https://www.cise.ufl.edu/class/cot5405fa09/index.html

no coworkers

3. (15) Design an O(n lg n) algorithm to compute the depth d for the interval coloring. Given a set of requests $\{1, 2, \ldots, n\}$, i th request corresponds an interval [s(i), f(i)), where start time s(i) and finish time f(i). The depth d of these given intervals is the maximum number of intervals that pass over any single point on the time-line.

Answer:

Here performs interval partitioning algorithm by greedy algorithm. During the procedure, we could, by the way, record how many labels intervals would take. Let z is sufficiently large than d.

ComputeDepth(*R*)

1 $\{I_1, \ldots, I_n\}$ = sort intervals in ascending order of their start times $s(i)$
2 $d = 0$
2 **for** i **from** 1 **to** n **do**
3   exclude the labels of all assigned intervals that are not compatible with I_i
4   **if** (there is a non-excluded label from $\{1, 2, \ldots, z\}$) **then**
5   assign a non-excluded w label to I_i
6   **if** ($w > d$) **then**
7             $d = w$
8 **return** d

Correction: Because the labels are taken by ascending order, the depth d would be renewed once a new label is used, which is fulfilled by line 6.

Running Time: By worst case analysis, the iteration of for loop takes linear time of n and the heap sort takes O(n lg n). Hence, time complexity is bounded by O(n lg n).

Reference:
class "greedy algorithm" power point p.18;
no coworkers

4. (25) We want to execute n jobs on a single machine. Job i has a weight w_i and an execution time t_i, for all $1 \le i \le n$. All of these $2n$ numbers are known in advance. Design a scheduling algorithm that minimizes the total weighted waiting time T, where $T = \sum (i=1, n)\ w_i \times$ (the overall waiting time for job i). The overall waiting time for job i means the interval from the starting time of the whole schedule until the time when job i finished. Justify the correctness of your algorithm.

Answer:

For each w_i, we should arrange an overall waiting time u_i. To minimize T, here pose some rules and check which one would produce min-T.

Given an instance of 4 jobs.

Job	1	2	3	4
Execution time	100	10	2	1
Weight	200	40	8	3

Rules:

(a) Shortest interval first: Process jobs in ascending order of t_i. It might minimize T, since the earlier a job is processed, the more its execution time will be counted for T.

Job	4	3	2	1	T
Execution time	1	2	10	100	
Weight	3	8	40	200	23147
u_i	1	3	13	113	

(b) largest weight first: Process jobs in descending order of w_i. It might minimize T, since the earlier a job is processed, the shorter u_i its w_i will multiply.

Job	1	2	3	4	T
Execution time	100	10	2	1	
Weight	200	40	8	3	25635
u_i	100	110	112	113	

(c) largest weight per unit time first: Process jobs in descending order of $p_i = w_i\,(t_i)^{-1}$. It might minimize T, since we consider the execution time and weight concurrently. We want process a job which has larger w_i and shorter t_i as soon as possible. Since the variable p_i and w_i are in direct proportion and p_i and t_i are in inverse proportion, the larger p_i is, the sooner job i should be processed.

Job	2	3	4	1	T
Execution time	10	2	1	100	
Weight	40	8	3	200	23135
p_i	4	4	3	2	
u_i	10	12	13	113	

Since job 2 and 3 has same p, here change their execution order.

Job	3	2	4	1	T
Execution time	2	10	1	100	
Weight	8	40	3	200	23135
p_i	4	4	3	2	
u_i	2	12	13	113	

This rule which can minimize the T would be the greedy rule.

MinT(*R*, *s*)

```
// f is the finishing time of the last scheduled job
    {p_1, ..., p_n} = sort jobs in descending order of their weight per unit time p_i
    f = s // s is some time at which machine starts
    T = 0
    for i from 1 to n do
        assign job i to the time interval from s(i) = f to f(i) = f + t_i
        u_i = f(i) – s    //overall waiting time for each job
        T = T + w_i × u_i
        f = f + t_i
    return T
```

Termination:

This algorithm terminates because each iteration of for loop (line 4) arrange a job and calculate T.

Correction:

✧ No idle time:

By observation, this greedy algorithm has no idle time by line 5 and line 8. Let O be an optimal algorithm which is allowed with idle time and its output T_O, meanwhile in O, jobs are sorted by descending order of their p_i as well. Let overall waiting time for each job in O is v_i.

Prove by induction.

Basis step:

$$\text{True for } i = 1,\ v_1 \geq u_1$$

$$\text{Since } v_1 = f_v(1) - s = s + t_1 + d - s = t_1 + d,\ d \geq 0$$

$$u_1 = f_u(1) - s = t_1$$

Induction hypothesis:

$$\text{True for } i = k,\ v_k \geq u_k$$

Induction step:

$$\text{Since } v_{k+1} = f_v(k + 1) - s = f_v(k) + t_{k+1} + d - s$$

$$= v_k + s + t_{k+1} + d - s = v_k + t_{k+1} + d\ ,\ d \geq 0$$

$$u_{k+1} = f_u(k + 1) - s = f_u(k) + t_{k+1} - s = u_k + s + t_{k+1} - s = u_k + t_{k+1}$$

$$\text{Hence, } v_{k+1} = v_k + t_{k+1} + d \geq u_k + t_{k+1} = u_{k+1}$$

If our algorithm is not optimal, then T must be larger than T_O.

Since $v_k \geq u_k$, for all $i = k$, then

$$T_O = \sum_{i=1}^{n} w_i v_i \geq T = \sum_{i=1}^{n} w_i u_i$$

Contradiction! Optimal algorithm must have no idle time.

✧ No inversion:

An inversion in this job arrangement is a pair of jobs *i* and *j* such that $s(i) > s(j)$ but their $p_j > p_i$.

First, here proves all schedules of jobs without inversions and idle time have same *T*. Given schedules of jobs without inversions and idle time, these schedules differ at their order. Say they have same p_i but different overall waiting time for each job.

Let there exists $m - k + 1$ jobs which have same *p*. They contribute *T'* to *T*.
Let *s* be the time interval before job *k* starts.

$$T' = \sum_{i=k}^{m} w_i u_i = \sum_{i=k}^{m} p_i t_i u_i = p \sum_{i=k}^{m} t_i u_i$$

$$T' = p \sum_{i=k}^{m} t_i u_i = p \sum_{i=k}^{m} \left[t_i \left(\sum_{j=k}^{i} t_i \ + s \right) \right]$$

$$T' = p \left(s \sum_{i=k}^{m} t_i \ + \sum_{i=k}^{m} t_i^2 \ + \sum_{i,j}^{m} t_i t_j \right),\ i \neq j$$

We can find that *T'* is nothing to do with the order of jobs whose *p* are the same.

Second, here proves there is an optimal schedule with no inversions and no idle time. Let there be an optimal schedule O without idle time, which is done above. If O has an inversion, there is a pair of jobs *i* and *j* such that *j* is scheduled immediately after *i* but their $p_j > p_i$. Let *s* be the time interval before job *i* and *j* starts. Say the schedule is α: $s \rightarrow i \rightarrow j$ or β: $s \rightarrow j \rightarrow i$. O has the schedule α.

$$T_\alpha = T_s + w_i u_i + w_j u_j = T_s + p_i t_i (t_i + s) + p_j t_j (t_j + t_i + s)$$

$$T_\alpha = T_s + p_i (t_i^2 + t_i s) + p_j (t_j^2 + t_i t_j + t_j s)$$

$$T_\beta = T_s + w_j u_j + w_i u_i = T_s + p_j t_j (t_j + s) + p_i t_i (t_i + t_j + s)$$

$$T_\beta = T_s + p_j (t_j^2 + t_j s) + p_i (t_i^2 + t_i t_j + t_i s)$$

$$\because p_j > p_i \Rightarrow p_j t_i t_j > p_i t_i t_j$$

$$\therefore T_\alpha > T_\beta$$

Contradiction! Hence, the optimal schedule with no inversion and no idle time.

✧ My algorithm:

Let's look back to the greedy algorithm I provided above. Line 1 indicates my algorithm is without inversions and line 2, 5 and 8 indicate my algorithm is without idle times. Here proves the algorithm is optimal by exchange argument. Let my algorithm return T when it terminates.

Let O be an optimal schedule with inversions and has T_O. Assume O has no idle time. If O has no inversions, then $T = T_O$. If O has an inversion, let i-j be an adjacent inversion. Swapping i and j won't increase T_O but might decrease T_O and strictly decreases the number of inversions. This contradicts definition of O.

<u>Running Time</u>: By worst case analysis, the iteration of for loop takes linear time of n , calculating p takes linear time of n , and the heap sort takes O(n lg n). Hence, time complexity is bounded by O(n lg n).

Reference:

class "greedy algorithm" power point p.8, 22-27;

no coworkers

5. (25) One of the basic motivations behind the Minimum Spanning Tree Problem is the goal of designing a spanning network for a set of nodes with minimum total cost. Here we explore another type of objective: designing a spanning network for which the most expensive edge is as cheap as possible. Specifically, let $G=(V, E)$ be a connected graph with n vertices, m edges, and positive edge costs that you may assume are all distinct. Let $T=(V, E')$ be a spanning tree of G; we define the bottleneck edge of T to be the edge of T with the greatest cost. A spanning tree T of G is a minimum-bottleneck spanning tree if there is no spanning tree T' of G with a cheaper bottleneck edge.

(a) Is every minimum-bottleneck tree of G a minimum spanning tree of G? Prove or give a counterexample.

(b) Is every minimum spanning tree of G a minimum-bottleneck tree of G? Prove or give a counterexample.

Answer:

(a)

The answer is *false*.

$$\text{Let } G = \{V, E\}$$

$$V = \{v_1, v_2, v_3, v_4\}$$

$$E = \{e_{12}, e_{13}, e_{14}, e_{23}, e_{24}, e_{34}\} = \{1, 2, 3, 3, 4, 5\}$$

One of minimum-bottleneck tree T_{mb} of G contains edges of

$$E'_{mb} = \{e_{13}, e_{14}, e_{23}\} = \{2, 3, 3\}$$

But the minimum spanning tree T_{ms} of G contains edges of

$$E'_{ms} = \{e_{12}, e_{13}, e_{14}\} = \{1, 2, 3\}$$

Hence, very minimum-bottleneck tree of G is not necessary a minimum spanning tree of G.

(b)

The answer is *true*.

Proof by contradiction.

Let T_{ms} of G be the minimum spanning tree. And let T_{mb} of G be the minimum bottleneck tree whose bottleneck is lighter than that of T_{ms}.. Hence, there exists at least an edge e of T_{ms} is heavier than any edges in T_{mb} and e is not contained in T_{mb}.

Here, we add e to T_{mb} and therefore form a circle C in T_{mb}. By circle property indicating that if e is the maximum cost edge in C, then e won't belong to any MST, e will be cut out of T_{mb}. Contradiction! Hence, e doesn't belong to any MST and the T_{ms} is not a MST.

Reference:

class "greedy algorithm" power point p.56-59;

https://www.coursehero.com/sitemap/schools/2339-University-of-Texas/departments/1174-EE/;

no coworkers