

Design and Analysis of Algorithms  
(UE17CS251)  
Semester 4   
Jan-May 2019

CPM & PERT project scheduling

by-  
Shivangi Gupta  
PES1201700274  
4-D

Vinay Jagadeesh  
PES1201700293  
4-D

**Objective**To design a project scheduling CPM and PERT tool based on time and dependencies

**Abstract**A project is an interrelated set of activities that has a definite starting and ending point and it results in a unique product or service. Project management is a scientific way of planning, implementing, monitoring and controlling the various aspects of a project such as time, money, materials, manpower, and other such resources. This project aims to develop an algorithm that can schedule the modules that make-up a project based on time and dependencies.

**Introduction**

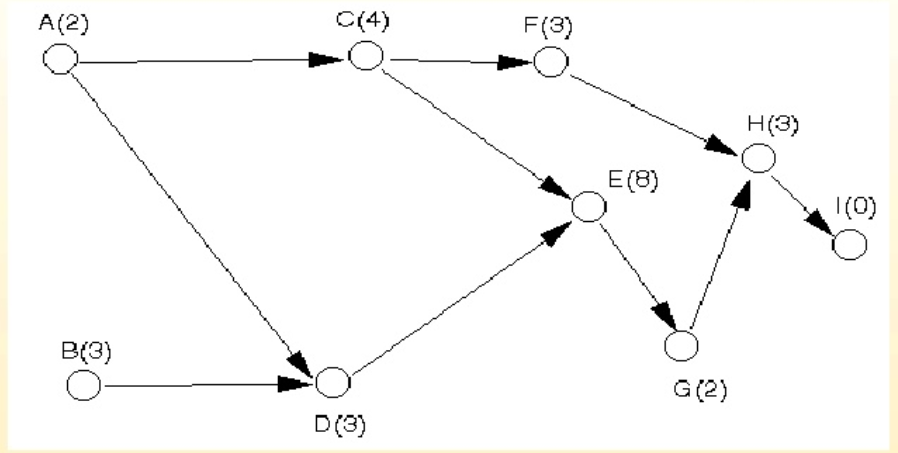
Methods used for network planning are:

* CPM (Critical Path Method)
* PERT (Program Evaluation and Review Technique)

Managing a project with network planning methods involves four steps:

1. Describing the project
2. Diagramming the network
3. Estimating time of completion
4. Monitoring project progress

All projects consist of a number of modules or activities, each of which needs to be implemented for the completion of the project. Before a module can be executed, its preceding activity must be completed. The dependency of one module on another is represented using a graph. An incoming and outgoing arrow represents a module’s predecessor and successor, respectively. The graph must not contain any cycles or cross-edges or the algorithm will fail.  
  
Below is a diagrammatic representation of the network formed by the modules A-I. The value in the brackets represents the time taken to complete the respective module.



The input consists of three entities:

1. ‘n’- the number of modules
2. ‘time[n]’- contains the individual time taken to execute each module
3. dependencies of each module one by one which is then stored as a graph using the adjacency list representation

The output will be the best order of execution of the modules, i.e., how the modules need to be scheduled keeping in mind the time and dependencies constraints, in order to avoid any crashing. If the graph fails to meet the required criterion of being a Directed Acyclic Graph (DAG), then an error will be displayed.

The procedure used to solve this problem is the topological sorting algorithm with minor changes to incorporate the time constraints. Topological sorting of vertices of a **DAG** is an ordering of the vertices v1, v2,....vn in such a way, that if there is an edge directed towards vertex vj from vertex vi, then vi must come before vj. As we know that the source vertex will come after the destination vertex, a stack is used to store previous elements. After completing all nodes, the elements are simply displayed from the stack. There are multiple topological sorting possible for a graph, but this algorithm prioritizes the modules which take lesser time and displays them first.

Time complexity: The above algorithm is simply DFS with an extra stack. Thus, the time complexity is same as DFS which is O(V+E), where V is number of vertices and E is number of edges.

Some real-world applications include:

* Build Systems
* Advanced-Packaging Tool (apt-get)
* Task Scheduling
* Pre-requisite problems
* Accounting functions
* Real-estate project development

**Code**

#include <stdio.h>

#include <stdlib.h>

#include<time.h>

struct node{

int vertex;

struct node\* next;

};

struct stack{

int top;

int capacity;

int\* array;

};

struct node\* createNode(int);

struct stack\* createStack(int);

struct Graph{

int numVertices;

struct node\*\* adjLists;

};

struct Graph\* createGraph(int vertices);

void addEdge(struct Graph\* graph, int src, int dest);

void printGraph(struct Graph\* graph);

void topological(struct Graph\* graph, int \*order, struct stack\* Stack);

int isCyclic(struct Graph\* graph);

int isCyclicUtil(struct Graph\* graph, int, int\*, int\*);

int main(){

clock\_t start, end;

printf("Enter number of modules- ");

int n;

scanf("%d",&n);

printf("Enter the time required to execute each module\n");

int \*time=(int \*)malloc(sizeof(int)\*n);

for(int i=0; i<n; i++){

printf("Time for module %d- ", i+1);

scanf("%d",&time[i]);

}

int big, pos=0;

int \*order=(int \*)malloc(sizeof(int)\*n);

for(int i=0;i<n;i++){

order[i]=i+1;

}

for(int i=0; i<n; i++){

big=0;

for(int j=0; j<n; j++){

if(time[j]>big){

big=time[j];

pos=j;

}

}

order[i]=pos+1;

time[pos]=0;

}

struct stack\* Stack= createStack(n);

printf("\nEnter the dependencies one by one for each module pairwise\nSo if 1 depends on 2, type '1 2' and press enter to type the next pairwise dependency\nEnter 0 0 if all the dependencies are completed- ");

struct Graph\* graph= createGraph(n);

int v1,v2;

scanf("%d%d", &v1,&v2);

while(v1!=0 && v2!=0){

addEdge(graph, v1, v2);

scanf("%d%d",&v1,&v2);

}

start = clock();

if (isCyclic(graph)){

printf("Graph is cyclic. Cannot execute.");

}

else{

topological(graph,order,Stack);

}

end = clock();

printf("\nTime taken %f seconds",(float)(end-start)/CLOCKS\_PER\_SEC);

return 0;  
}

struct node\* createNode(int v){

struct node\* newNode = malloc(sizeof(struct node));

newNode->vertex = v;

newNode->next = NULL;

return newNode;

}

struct stack\* createStack(int capacity){

struct stack\* newStack=(struct stack\*)malloc(sizeof(struct stack));

newStack->capacity=capacity;

newStack->top=-1;

newStack->array=(int\*)malloc((newStack->capacity)\*sizeof(int));

return newStack;

};

void push (struct stack\* Stack, int item){

Stack->array[++Stack->top]=item;

}

struct Graph\* createGraph(int vertices){

struct Graph\* graph = malloc(sizeof(struct Graph));

graph->numVertices = vertices;

graph->adjLists = malloc((vertices+1) \* sizeof(struct node\*));

int i;

for (i = 1; i < vertices+1; i++)

graph->adjLists[i] = NULL;

return graph;

}

void addEdge(struct Graph\* graph, int src, int dest){

// Add edge from dest to src

struct node\* newNode = createNode(src);

newNode->next = graph->adjLists[dest];

graph->adjLists[dest] = newNode;

}

void printGraph(struct Graph\* graph){

int v;

for (v = 1; v < graph->numVertices+1; v++)

{

struct node\* temp = graph->adjLists[v];

printf("\n Adjacency list of vertex %d\n ", v);

while (temp)

{

printf("%d -> ", temp->vertex);

temp = temp->next;

}

printf("\n");

}

}

void topologicalSortUtil(int i, int \*visited, struct Graph\* graph, int \*output, struct stack\* Stack){

visited[i]=1;

struct node\* temp=graph->adjLists[i];

while(temp){

if(visited[temp->vertex]==0){

topologicalSortUtil(temp->vertex, visited, graph, output, Stack);

}

temp=temp->next;

}

push(Stack, i);

}

void topological(struct Graph\* graph, int \*order, struct stack\* Stack){

int\* visited = (int\*)calloc(graph->numVertices+1, sizeof(int));

int\* output = (int\*)calloc(graph->numVertices, sizeof(int));

int k=graph->numVertices-1;

for(int i=0; i<graph->numVertices; i++){

if(visited[order[i]]==0){

topologicalSortUtil(order[i],visited,graph,output,Stack);

}

}

printf("\nThe modules should be scheduled in the following order-\n");

while(k>=0){

printf("%d\n", Stack->array[Stack->top--]);

--k;

}

}

int isCyclic(struct Graph\* graph) {

int\* visited = (int\*)calloc(graph->numVertices, sizeof(int));

int\* recStack = (int\*)calloc(graph->numVertices, sizeof(int));

for (int node = 1; node <= graph->numVertices; node++) {

if (!visited[node] && isCyclicUtil(graph, node, visited, recStack)) return 1;

}

return 0;

}

int isCyclicUtil(struct Graph\* graph, int v, int\* visited, int\* recStack) {

visited[v] = 1;

recStack[v] = 1;

for (struct node\* neighbour = graph->adjLists[v]; neighbour; neighbour = neighbour->next) {

if ((!visited[neighbour->vertex] && isCyclicUtil(graph, neighbour->vertex, visited, recStack)) || recStack[neighbour->vertex]) return 1;

}

recStack[v] = 0;

return 0;

}