İSTANBUL SAĞLIK VE TEKNOLOJİ ÜNİVERSİTESİ
MÜHENDİSLİK VE DOĞA BİLİMLERİ FAKÜLTESİ
YAZILIM MÜHENDİSLİĞİ BÖLÜMÜ

YAZ202 – VERİ YAPILARI VE ALGORİTMALAR

- FINAL PROJECT -

GÜRKAN GÖZTEPELİ 210609026
FURKAN AKARÇEŞME 210609008
YAKUPHAN BİLMEZ 210609025

FİNAL

HAZİRAN 2023

ÖZET

Bu proje, bir proje yönetimi problemi için Activity-On-Node (AON) yöntemini kullanarak faaliyetlerin en erken başlama zamanını, en geç bitiş zamanını ve kritik yolunu hesaplamaktır. Bu tür bir hesaplama, bir projenin zamanlamasını ve kaynak yönetimini planlamak için önemlidir. Kod, faaliyetleri ve bağımlılıklarını okuyarak, faaliyetlerin özelliklerini hesaplayarak ve kritik yolu yazdırarak bu hedefi gerçekleştirir.

Proje GitHub Linki : *https://github.com/paysis/vya-final*

Anahtar Kelimeler: *Proje Yönetimi , Activiy-On-Node Yöntemi , En Erken Başlama Zamanı , En Geç Bitiş Zamanı , Kritik Yol*

SUMMARY

This project aims to calculate the earliest start time, latest finish time, and critical path of activities using the Activity-On-Node (AON) method for a project management problem. Such a calculation is important for planning the timing and resource management of a project. The code reads the activities and their dependencies, calculates the properties of the activities, and finally prints the critical path.

Keywords : *Project Management , Activity-On-Node Method , Earliest Start Time , Latest Finish Time , Critical Path*

İÇERİK

Faaliyetlerin özelliklerini tutan "Activity" sınıfı ve faaliyetlerin oluşturduğu ağın özelliklerini (bağımlılıklar, en erken başlama zamanı, en geç bitiş zamanı, kritik yol vb.) tutan "Graph" sınıfı bulunmaktadır. "YAMLParser" sınıfı ise, YAML dosyalarının okunması ve Activity sınıfından nesnelerin oluşturulması için kullanılmaktadır. Bu sınıf, Graph sınıfının oluşturulması için gereklidir ve kodun ana bölümlerinden birisidir. Kodun ana amacı, verilen faaliyetlerin bağımlılıklarını okuyarak Activity-On-Node (AON) yöntemini kullanarak faaliyetlerin en erken başlama zamanını, en geç bitiş zamanını ve kritik yolunu hesaplamaktır.

1.1 Kütüphaneler

Projede, iostream, vector, queue, unordered_set ve unordered_map kütüphaneleri kullanıldı *(Resim1.1)*. İostream girdi/çıktı işlemleri için , vector diziler için , queue sıra yapısı için , unordered_set ve unordered_map ise, hash tabloları için kullanılmıştır. Ayrıca algorithm kütüphanesi, C++'ın standart kütüphanesinin bir parçası olarak gelir ve sıralama, arama, karşılaştırma, değiştirme gibi bir dizi algoritma işlevi sağlar. Bu işlevler, işlemleri kolaylaştırır. Son olarak da, yaml-cpp kütüphanesi kullanılmıştır. Bu kütüphane ise, YAML dosyalarını okumak ve yazmak için kullanılmıştır. YAML, insanların okuması ve yazması kolay olan bir veri serielleştirme formatıdır ve birçok programlama dilinde kullanılabilir. Projede, YAML formatında tanımlanan faaliyetlerin listesini okumak için bu kütüphane kullanılmıştır.

main.cpp

```
1  #include <iostream>
2  #include <vector>
3  #include <queue>
4  #include <unordered_set>
5  #include <unordered_map>
6  #include <algorithm>
7  #include <yaml-cpp/yaml.h>
```

Resim1.1

2.1 Activity Sınıfı

Bu kod (*Resim1.2*), "Activity" sınıfını tanımlar. Bu sınıf, bir etkinliğin özelliklerini içerir. Bu özellikler, etkinlik kimliği ("id"), etkinlik adı ("name"), etkinlik süresi ("duration"), öncü aktivitelerin listesi ("predecessors"), öncü aktivitelerin kontrol edildiği bir küme ("checked_predecessors"), halef aktivitelerin kontrol edildiği bir küme ("checked_successors"), halef aktivitelerin listesi ("successors"), en erken başlama zamanı ("earliestStartTime"), en erken bitiş zamanı ("earliestFinishTime"), en geç başlama zamanı ("latestStartTime"), en geç bitiş zamanı ("latestFinishTime"), etkinlik "slack" değeri ve öncü aktivitelerin en büyük tamamlanma zamanı ("maxOfPredecessors") ve halef aktivitelerin en küçük tamamlanma zamanı ("minOfSuccessors").

Sınıf, iki farklı yapıcı metot içerir. İlk yapıcı metot, yalnızca etkinlik kimliği ("id") parametresi alır ve diğer özellikler varsayılan değerlerle başlatılır. İkinci yapıcı metot, etkinlik kimliği ("id"), etkinlik adı ("name") ve etkinlik süresi ("duration") parametrelerini alır. Bu yapıcı metot, belirtilen özellikleri kullanarak bir etkinlik nesnesi oluşturur.

Tüm özellikler, sınıfın yapıcı metotları tarafından varsayılan değerlerle başlatılır. "minOfSuccessors" özelliği, başlangıçta -1 olarak atanır ve daha sonra hesaplamalar sırasında değiştirilir.

```cpp
using namespace std;

class Activity {
public:
    string id;
    string name;
    int duration;
    vector<Activity*> predecessors;
    unordered_set<Activity*> checked_predecessors;
    unordered_set<Activity*> checked_successors;
    // unordered_map<Activity*, int> checked_predecessors;
    vector<Activity*> successors;
    int earliestStartTime;
    int earliestFinishTime;
    int latestStartTime;
    int latestFinishTime;
    int slack;

    int maxOfPredecessors;
    int minOfSuccessors;

    Activity(string id){
        this->id = id;
        this->name = id;
        this->duration = -1;
        earliestStartTime = earliestFinishTime = latestStartTime = latestFinishTime = slack = maxOfPredecessors = 0;
        minOfSuccessors = -1;
    }

    Activity(string id, string name, int duration) {
        this->id = id;
        this->name = name;
        this->duration = duration;
        earliestStartTime = earliestFinishTime = latestStartTime = latestFinishTime = slack = maxOfPredecessors = 0;
        minOfSuccessors = -1;
    }
};
```

Resim2.1

3.1 Graph Sınıfı

Bu kod (*Resim3.1*), "Graph" sınıfını tanımlar. Bu sınıf, etkinliklerin ve bağımlılıkların bulunduğu bir grafik yapısını temsil eder. Sınıfın özellikleri, etkinliklerin bir listesi ("vertices") ve tüm etkinliklerin en erken tamamlanma zamanını tutan "earliestTotalFinishTime" değeridir.

```
class Graph {
public:
    vector<Activity*> vertices;
    int earliestTotalFinishTime;

    // unordered_set<Activity*> checked_vertices;

    Graph() {
        earliestTotalFinishTime = 0;
    }
```

Resim 3.1

3.1.1 "addActivity"

Sınıf, "addActivity" (*Resim3.1.1*) yöntemi kullanılarak etkinlikler eklemek için tasarlanmıştır. Bu yöntem, bir etkinlik nesnesi alır ve bu etkinliği "vertices" listesine ekler.

```
    void addActivity(Activity* activity) {
        vertices.push_back(activity);
    }
```

Resim 3.1.1

3.1.2 "calculateET"

"calculateET" yöntemi (*Resim3.1.2*), etkinliklerin en erken tamamlanma zamanlarını hesaplamak için kullanılır. Bu yöntem, bir "queue" oluşturur ve tüm etkinlikler üzerinde döngü başlatır. Döngü, her bir etkinliğin öncülerini kontrol eder. Eğer bir etkinliğin öncüsü yoksa, yani tamamlanmışsa, en erken başlama zamanı "earliestStartTime" olarak atanır ve en erken bitiş zamanı "earliestFinishTime" bu zamanın faaliyet süresi eklenmesiyle hesaplanır. Bu etkinlik, "queue" ye eklenir.

Daha sonra, "queue" boş olmadığı sürece, "queue" nin önündeki etkinlik çıkarılır ve bu etkinliğin haleflerinin üzerinde bir döngü başlatılır. Eğer bir halef, öncü etkinliklerin tümüne sahipse, yani tamamlanma zamanları hesaplandıysa, "maxOfPredecessors" değeri, kendisinden bağımlı öncü etkinliklerin en büyük tamamlanma zamanına eşitlenir. Bu halef etkinliğin en erken başlama zamanı, "maxOfPredecessors" değeri olarak atanır ve en erken bitiş zamanı, "maxOfPredecessors" değerine faaliyet süresinin eklenmesiyle hesaplanır. "earliestTotalFinishTime" değeri, bu halef etkinliğin en erken bitiş zamanı ile mevcut "earliestTotalFinishTime" değeri arasındaki maksimum değere eşitlenir. Bu halef etkinlik, "queue" ye eklenir.

Bu işlemler, "queue" boşalana kadar devam eder ve sonunda tüm etkinliklerin en erken tamamlanma zamanları hesaplanır.

```
void calculateET(){
    queue<Activity*> q;
    for(auto& v : vertices){
        if (v->predecessors.empty()){
            v->earliestStartTime = 0;
            v->earliestFinishTime = v->duration;
            q.push(v);
        }
    }

    while(!q.empty()){
        Activity* v = q.front();
        q.pop();


        for(auto& u : v->successors){
            // u->checked_predecessors[v] = v->earliestFinishTime;
            if(u->duration == -1){
                cout << "Error: Activity " << u->id << " has no duration." << endl;
                cout << "Please check the YAML file." << endl;
                exit(1);
            }
            u->checked_predecessors.insert(v);
            u->maxOfPredecessors = max(
                u->maxOfPredecessors,
                v->earliestFinishTime
            );
            if(u->checked_predecessors.size() == u->predecessors.size()){
                u->earliestStartTime = u->maxOfPredecessors;
                u->earliestFinishTime = u->maxOfPredecessors + u->duration;
                this->earliestTotalFinishTime = max(
                    u->earliestFinishTime,
                    this->earliestTotalFinishTime
                );
                q.push(u);
            }
        }
    }
}
```

Resim 3.1.2

3.1.3 "calculateLT"

Bu kod (*Resim3.1.3*), "calculateLT" yöntemini tanımlar. Bu yöntem, etkinliklerin en geç tamamlanma zamanlarını hesaplamak için kullanılır. Bu yöntem, aynı zamanda etkinliklerin "slack" değerini de hesaplar.

Yöntem, "queue" oluşturarak başlar ve tüm etkinlikler üzerinde döngü başlatır. Bu döngü, her bir etkinliğin haleflerini kontrol eder. Eğer bir etkinliğin halefi yoksa, yani hiçbir etkinlik bu etkinlikten sonra gelmiyorsa, en geç bitiş zamanı "earliestTotalFinishTime" olarak atanır ve en geç başlama zamanı bu zamanın faaliyet süresi çıkarılmasıyla hesaplanır. Bu etkinlik, "queue" ye eklenir.

Daha sonra, "queue" boş olmadığı sürece, "queue" nin önündeki etkinlik çıkarılır ve bu etkinliğin öncülerinin üzerinde bir döngü başlatılır. Eğer bir öncü etkinlik, halef etkinliğin tümüne sahipse, yani tamamlanma zamanları hesaplandıysa, "minOfSuccessors" değeri, kendisine bağımlı olan halef etkinliklerin en küçük bitiş zamanına eşitlenir. Bu öncü etkinliğin en geç bitiş zamanı, "minOfSuccessors" değeri olarak atanır ve en geç başlama zamanı, "latestFinishTime" değerinden faaliyet süresinin çıkarılmasıyla hesaplanır. Bu öncü etkinlik, "queue" ye eklenir.

Bu işlemler, "queue" boşalana kadar devam eder ve sonunda tüm etkinliklerin en geç tamamlanma zamanları hesaplanır. Bu hesaplanan en geç tamamlanma zamanları ile en erken tamamlanma zamanları arasındaki fark, her etkinliğin "slack" değerini oluşturur. Etkinliklerin

"slack" değeri, en geç başlama zamanı ile en erken başlama zamanı arasındaki fark olarak tanımlanır ve bu değer, etkinliğin esnekliğini gösterir.

```
void calculateLT(){
    queue<Activity*> q;
    for(auto& v : vertices){
        if(v->successors.empty()){
            v->latestFinishTime = this->earliestTotalFinishTime;
            v->latestStartTime = v->latestFinishTime - v->duration;
            q.push(v);
        }
    }

    while(!q.empty()){
        Activity* v = q.front();
        q.pop();

        for(auto& u : v->predecessors){

            u->checked_successors.insert(v);

            u->minOfSuccessors = u->minOfSuccessors != -1 ? min(
                u->minOfSuccessors,
                v->latestStartTime
            ) : v->latestStartTime;

            if(u->checked_successors.size() == u->successors.size()){
                u->latestFinishTime = u->minOfSuccessors;
                u->latestStartTime = u->latestFinishTime - u->duration;
                q.push(u);
            }
        }

        v->slack = v->latestStartTime - v->earliestStartTime;

    }
}
```

Resim3.1.3

3.1.4 "printCriticalPath"

Bu kod (*Resim3.1.4*), "printCriticalPath" yöntemini tanımlar. Bu yöntem, etkinliklerin "slack" değerine göre kritik yolunu yazdırmak için kullanılır.

Yöntem, "vertices" listesi üzerinde döngü başlatır ve etkinliğin "slack" değerini kontrol eder. Eğer bir etkinliğin "slack" değeri sıfırsa, yani esnekliği yoksa, bu etkinlik kritik yolda yer alır ve "Critical Path:" ifadesi ile birlikte yazdırılır. Etkinliğin adı ve kimliği, parantez içinde yazdırılır.

```
void printCriticalPath() {
    cout << "Critical Path: ";
    bool firstOne = true;
    for (auto& v : vertices) {
        if (v->slack == 0) {
            if(!firstOne){
                cout << " -> ";
            } // to make output look a little bit more pretty

            cout << "(" << v->id << ") " << v->name;
            firstOne = false;
        }

    }
    cout << "." << endl;
}
};
```

Resim3.1.4

Döngü, tüm etkinlikler için tamamlandıktan sonra, kritik yolun sonunda bir nokta yazdırılır. Bu yöntem, kritik yolun görsel olarak kolay anlaşılabilir şekilde yazdırılmasını sağlar.

4.1 YAMLParser Sınıfı

Bu kod (*Resim4.1*), "YAMLParser" sınıfını tanımlar. Bu sınıf, bir YAML dosyasını yüklemek için kullanılır. Sınıfın yapıcısı, bir dosya adı alır ve bu dosyayı yükler.

4.1.1 "calculateLT"

"filename" özelliği (*Resim4.1*), dosya adını tutar ve "data" özelliği, yüklenen YAML verilerini tutar YAML verileri, "YAML::LoadFile" yöntemi ile dosyadan yüklenir. Eğer bir hata oluşursa, "YAML::Exception" fırlatılır ve bu hata yakalanır. Yakalanan hata, ekrana yazdırılır ve program sonlandırılır.

Bu sınıf, YAML dosyalarını yüklemek ve bu dosyalardan veri okumak için kullanılabilir. Bu şekilde, programlar YAML verilerini kullanarak yapılandırma dosyaları veya verileri okuyabilirler.

```
class YAMLParser{
    public:
    YAMLParser(string filename){
        this->filename = filename;
        try{
            this->data = YAML::LoadFile(filename);
        }catch(YAML::Exception& e){
            cout << "File loading error: " << e.what() << endl;
            exit(1);
        }
    }
```

Resim4.1

4.1.2 "parse"

Bu kod (*Resim4.1.2*), "parse" yöntemini tanımlar. Bu yöntem, YAML dosyasından verileri okuyarak bir "Graph" nesnesi oluşturur.

Yöntem, "try-catch" bloğu içinde başlar. "data" özelliği, YAML dosyasından yüklenen verileri tutar. "activities_raw" değişkeni, dosyadan "activities" anahtarına sahip verileri içeren düğümleri alır. Daha sonra, "activities_raw" değişkeni üzerinde bir döngü başlatılır ve her bir etkinlik için aşağıdaki işlemler yapılır:

- Etkinlik özellikleri ("id", "name", "duration") YAML dosyasından alınır.
- Etkinlik süresi negatif ise, bir hata mesajı yazdırılır ve program sonlandırılır.
- Etkinliğin daha önce oluşturulup oluşturulmadığı kontrol edilir. Eğer etkinlik daha önce oluşturulmamışsa, yeni bir etkinlik oluşturulur ve "activities_table" özelliğinde saklanır. Eğer daha önce oluşturulmuşsa, var olan etkinlik alınır ve özellikleri güncellenir.
- Etkinlik, "graph" nesnesine eklenir.
- Etkinliğin bağımlılıkları ("dependencies") YAML dosyasından alınır. Bu bağımlılıklar, "predecessors" özelliği ile etkinliğin öncülerine eklenir ve "successors" özelliği ile etkinliğin haleflerine eklenir.
- Etkinlik, "activities_table" özelliğinde saklanır.

Yöntem, tüm etkinlikler için tamamlandıktan sonra, "graph" nesnesi döndürülür. (*Resim4.1.3*)

Bu yöntem, YAML dosyalarını okumak ve bu dosyalardan veri okumak için kullanılabilir. Bu şekilde, programlar YAML verilerini kullanarak etkinliklerin bağımlılıklarını ve sürelerini okuyabilirler. Bu verileri kullanarak, programlar etkinlikleri ve bağımlılıklarını görselleştirebilir, zamanlama planları oluşturabilir ve kaynakları planlayabilirler.

```
Graph* parse(){
    Graph* graph = new Graph();

    try{
        auto activities_raw = this->data["activities"];
        for(size_t i = 0; i < activities_raw.size(); ++i){
            const YAML::Node& activity_raw = activities_raw[i];

            string id = activity_raw["id"].as<string>();
            string name = activity_raw["name"].as<string>();
            int duration = activity_raw["duration"].as<int>();

            if(duration < 0){
                cout << "Error: Activity " << id << " has a negative duration." << endl;
                cout << "Please check the YAML file." << endl;
                exit(1);
            }

            Activity* activity;
            if(activities_table.find(id) == activities_table.end()){
                activity = new Activity(id, name, duration);
            }else{
                activity = activities_table[id];
                activity->name = name;
                activity->duration = duration;
            }

            graph->addActivity(activity);

            const YAML::Node& dependencies_raw = activity_raw["dependencies"];
```

Resim4.1.2

```
            for(size_t j = 0; j < dependencies_raw.size(); ++j){
                const YAML::Node& dependency_raw = dependencies_raw[j];
                string dependency_id = dependency_raw.as<string>();

                if(activities_table.find(dependency_id) == activities_table.end()){
                    Activity* dependency = new Activity(dependency_id);
                    activities_table[dependency_id] = dependency;
                }
                Activity* dependency = activities_table[dependency_id];

                activity->predecessors.push_back(dependency);
                dependency->successors.push_back(activity);
            }

            activities_table[id] = activity;
        }
    }catch(YAML::Exception& e){
        cout << "File parsing error: " << e.what() << endl;
        exit(1);
    }

    return graph;
}

private:
string filename;
YAML::Node data;
unordered_map<string, Activity*> activities_table;
unordered_set<string> vertices_set;
};
```

Resim4.1.3

5.1 main Sınıfı

Bu kod (*Resim5.1*), bir programın "main" fonksiyonunu tanımlar. Bu fonksiyon, programın ana işlevlerini gerçekleştirir.Fonksiyon, "argc" ve "argv" parametrelerini alır. Bu parametreler, programın komut satırı argümanlarını temsil eder. "argv[1]" parametresi, programın çalıştırılacağı YAML dosyasının adını içerir.

İlk olarak, "argv[1]" adlı YAML dosyası "YAMLParser" sınıfı kullanılarak yüklenir ve "Graph" nesnesi oluşturulur. Bu nesne, etkinliklerin bağımlılıklarını ve sürelerini içerir. Daha sonra, "Graph" nesnesine önceki örnekteki gibi erişilir ve "calculateET" ve "calculateLT" yöntemleri kullanılarak erken başlama ("ES") ve bitiş ("EF") zamanları ile geç başlama ("LS") ve bitiş ("LF") zamanları hesaplanır.

Sonrasında, "printCriticalPath" yöntemi kullanılarak kritik yol yazdırılır. Bu yöntem, etkinliklerin "slack" değerlerine göre kritik yolunu yazdırır. Yorum satırları arasındaki diğer kod parçaları, önceki örnekte olduğu gibi, sabit etkinlikler ve bağımlılıklar tanımlar.

Bu "main" fonksiyonu, YAML dosyasından veri okuma, etkinliklerin bağımlılıklarını hesaplama ve kritik yolun yazdırılması gibi temel program işlevlerini gerçekleştirir. Bu şekilde, programlar etkinlikleri ve bağımlılıklarını görselleştirebilir, zamanlama planları oluşturabilir ve kaynakları planlayabilirler.

```
int main(int argc, char** argv) {

    string argv1_str(argv[1]);
    Graph* g = YAMLParser("../shared/" + argv1_str).parse();

    g->calculateET();
    g->calculateLT();

    g->printCriticalPath();

    return 0;
}
```

Resim5.1

5.1 Sonuç

example.yaml

```yaml
activities:
  - id: A
    name: Activity A
    duration: 7
    dependencies:
  - id: B
    name: Activity B
    duration: 9
    dependencies:
  - id: C
    name: Activity C
    duration: 12
    dependencies:
      - A
  - id: D
    name: Activity D
    duration: 8
    dependencies:
      - A
      - B
  - id: E
    name: Activity E
    duration: 9
    dependencies:
      - D
  - id: F
    name: Activity F
    duration: 6
    dependencies:
      - C
      - E
  - id: G
    name: Activity G
    duration: 5
    dependencies:
      - E
```

Sonuç1.1

Verilen aktiviteler ve bağımlılıklar, bir projenin parçalarını temsil etmektedir. Bu aktivitelerin süreleri ve birbirleriyle olan bağımlılıkları dikkate alındığında, kritik yolun (B) Activity B -> (D) Activity D -> (E) Activity E -> (F) Activity F olduğu belirlenmiştir. Kritik yol, projenin tamamlanması için en uzun süreye sahip olan faaliyetleri gösterir. (*Sonuç1.1*)

example2.yaml

```yaml
activities:
  - id: A
    name: Activity A
    duration: 7
    dependencies:
  - id: B
    name: Activity B
    duration: 9
    dependencies:
  - id: C
    name: Activity C
    duration: 12
    dependencies:
      - A
  - id: D
    name: Activity D
    duration: 8
    dependencies:
      - A
      - B
  - id: E
    name: Activity E
    duration: 9
    dependencies:
      - D
  - id: F
    name: Activity F
    duration: 6
    dependencies:
      - C
      - E
      - A
  - id: G
    name: Activity G
    duration: 5
    dependencies:
      - E
      - F
```

Sonuç1.2

Verilen aktiviteler ve bağımlılıklar, bir projenin parçalarını temsil etmektedir. Bu aktivitelerin süreleri ve birbirleriyle olan bağımlılıkları dikkate alındığında, projenin kritik yolunun (B) Activity B -> (D) Activity D -> (E) Activity E -> (F) Activity F -> (G) Activity G olduğu belirlenmiştir. Kritik yol, projenin tamamlanması için en uzun süreye sahip olan faaliyetleri gösterir ve projenin minimum sürede tamamlanabilmesi için dikkate alınması gereken yoldur. (*Sonuç 1.2*)

6.1 Genel Sonuç

Aktivitelerin birbirlerine olan bağımlılıkları, proje yönetimi ve zaman planlaması açısından büyük bir öneme sahiptir. Bir aktivitenin başlaması veya tamamlanması için önceki aktivitelerin bitirilmesi gerekmektedir. Kritik yol üzerindeki aktiviteler, projenin gecikmesine neden olabilecek riskleri ortaya koyar. Bu nedenle, proje yöneticileri ve ekipleri, kaynakları etkin bir şekilde yönetmek, zamanlamayı optimize etmek ve projenin ilerlemesini takip etmek için bu bağımlılıkları dikkate almalıdır.

Aktivitelerin süreleri ve bağımlılıkları, proje yönetimi sürecinde önemli faktörlerdir. Aktivitelerin paralel olarak yürütülmesi, kaynakların etkin kullanımını sağlar. Ancak, kritik yol üzerindeki aktivitelerin gecikmeye neden olabileceği riskler göz önünde bulundurulmalıdır. Bu şekilde, proje yöneticileri ve ekipleri, projenin zamanında ve başarıyla tamamlanmasını sağlamak için stratejiler geliştirebilirler.

Sonuç olarak, verilen aktivitelerin süreleri ve bağımlılıkları dikkate alındığında belirlenen kritik yol, proje yönetimi ve zaman planlaması açısından büyük bir öneme sahiptir. Bu bilgiler, proje yöneticilerine ve ekiplere, kaynakları etkin bir şekilde yönetme, projenin ilerlemesini takip etme ve gecikmeleri önleme konusunda rehberlik etmektedir. Kritik yolun belirlenmesi, projenin zamanlamasının optimize edilmesi ve başarıyla tamamlanması için stratejilerin geliştirilmesi gerekmektedir.

IN ENGLISH

CONTENT

The code includes an "Activity" class that holds the properties of activities, and a "Graph" class that holds the properties of the network created by activities, such as dependencies, earliest start time, latest finish time, critical path, etc. The "YAMLParser" class is used to read YAML files and create Activity objects. This class is essential for creating the Graph class and is one of the main components of the code. The main purpose of the code is to read the dependencies of given activities, and calculate the earliest start time, latest finish time, and critical path of the activities using the Activity-On-Node (AON) method.

1.1 Libaries

In the project, iostream, vector, queue, unordered_set, and unordered_map libraries were used (Figure 1.1). iostream is used for input/output operations, vector for arrays, queue for a queue structure, and unordered_set and unordered_map for hash tables. Additionally, the algorithm library is part of C++'s standard library and provides a set of algorithm functions such as sorting, searching, comparing, and modifying operations which make operations easier. Finally, the yaml-cpp library was used to read and write YAML files. YAML is a data serialization format that is easy to read and write for humans and can be used in many programming languages. This library was used to read the list of activities defined in YAML format in the project.

2.1 Activity Class

This code (Figure 1.2) defines the "Activity" class. This class contains the properties of an activity. These properties include the activity ID ("id"), activity name ("name"), activity duration ("duration"), a list of predecessor activities ("predecessors"), a set to keep track of checked predecessor activities ("checked_predecessors"), a set to keep track of checked successor activities ("checked_successors"), a list of successor activities ("successors"), earliest start time ("earliestStartTime"), earliest finish time ("earliestFinishTime"), latest start time ("latestStartTime"), latest finish time ("latestFinishTime"), activity slack value and the maximum completion time of predecessor activities ("maxOfPredecessors"), and the minimum completion time of successor activities ("minOfSuccessors").

The class includes two different constructor methods. The first constructor method takes only the activity ID ("id") parameter, and other properties are initialized with default values. The second constructor method takes the activity ID ("id"), activity name ("name"), and activity duration ("duration") parameters. This constructor method creates an activity object using the specified properties.

All properties are initialized with default values by the constructor methods of the class. The "minOfSuccessors" property is initially assigned as -1 and later changed during calculations.

3.1 Graph Class

This code (Figure 3.1) defines the "Graph" class, which represents a graph structure containing activities and their dependencies. The properties of the class are a list of activities ("vertices") and the value "earliestTotalFinishTime" that holds the earliest possible completion time for all activities.

3.1.1 "addActivity"

The class is designed to add activities using the "addActivity" method (Figure 3.1.1). This method takes an activity object and adds it to the "vertices" list.

3.1.2 "calculateET"

The "calculateET" method (Figure 3.1.2) is used to calculate the earliest possible completion times for activities. This method creates a "queue" and starts a loop over all activities. The loop checks the predecessors of each activity. If an activity has no predecessors, i.e., it is completed, the earliest start time is assigned as "earliestStartTime" and the earliest finish time is calculated by adding the activity duration to this time. This activity is then added to the "queue".

Later, as long as the "queue" is not empty, the activity at the front of the "queue" is removed, and a loop is started over its successors. If a successor has all its predecessors, i.e., completion times are calculated, the "maxOfPredecessors" value is set to the largest completion time of its dependent predecessors. The earliest start time of this successor activity is assigned as the "maxOfPredecessors" value, and the earliest finish time is calculated by adding the activity duration to this value. The "earliestTotalFinishTime" value is set to the maximum value between the earliest finish time of this successor activity and the current "earliestTotalFinishTime" value. This successor activity is added to the "queue".

These operations continue until the "queue" is emptied, and eventually all activities' earliest possible completion times are calculated.

3.1.3 "calculateLT"

This code (Figure 3.1.3) defines the "calculateLT" method. This method is used to calculate the latest possible completion times for activities. It also calculates the "slack" value of activities.

The method begins by creating a "queue" and starting a loop over all activities. The loop checks the successors of each activity. If an activity has no successors, i.e., no activity comes after this activity, the latest finish time is assigned as "earliestTotalFinishTime," and the latest start time is calculated by subtracting the activity duration from this time. This activity is added to the "queue".

Later, as long as the "queue" is not empty, the activity at the front of the "queue" is removed, and a loop is started over its predecessors. If a predecessor activity has all its successors, i.e., completion times are calculated, the "minOfSuccessors" value is set to the

smallest finish time of its dependent successors. The latest finish time of this predecessor activity is assigned as the "minOfSuccessors" value, and the latest start time is calculated by subtracting the activity duration from the "latestFinishTime" value. This predecessor activity is added to the "queue".

These operations continue until the "queue" is emptied, and eventually the latest possible completion times for all activities are calculated. The difference between the calculated latest completion times and earliest completion times forms the "slack" value of each activity. The "slack" value of an activity is defined as the difference between the latest start time and earliest start time, and this value indicates the flexibility of the activity.

3.1.4 "printCriticalPath"

This code (Figure 3.1.4) defines the "printCriticalPath" method. This method is used to print the critical path based on the "slack" value of activities.

The method starts a loop over the "vertices" list and checks the "slack" value of each activity. If an activity has a "slack" value of zero, i.e., it is not flexible, it is part of the critical path and printed with the phrase "Critical Path:" along with its name and identifier in parentheses.

After the loop is completed for all activities, a dot is printed at the end of the critical path. This method enables the critical path to be visually printed in an easily understandable way.

4.1 YAMLParser Class

This code (Figure 4.1) defines the "YAMLParser" class. This class is used to load a YAML file. The constructor of the class takes a file name and loads the file.

4.1.1 "calculateLT"

The "filename" property (Figure 4.1) holds the file name, and the "data" property holds the loaded YAML data. The YAML data is loaded from the file using the "YAML::LoadFile" method. If an error occurs, a "YAML::Exception" is thrown and caught. The caught error is printed to the screen and the program is terminated.

This class can be used to load YAML files and read data from them. This way, programs can read configuration files or data using YAML data.

4.1.2 "parse"

This code (Figure 4.1.2) defines the "parse" method. This method creates a "Graph" object by reading data from a YAML file.

The method starts within a "try-catch" block. The "data" property holds the loaded YAML data. The "activities_raw" variable takes the nodes containing data with the "activities" key from the file. Then, a loop is started over the "activities_raw" variable, and the following operations are performed for each activity:

• The activity properties ("id", "name", "duration") are taken from the YAML file.

• If the activity duration is negative, an error message is printed, and the program is terminated.

• The method checks whether the activity has been created before or not. If the activity has not been created before, a new activity is created and stored in the "activities_table" property. If it has been created before, the existing activity is taken and its properties are updated.

• The activity is added to the "graph" object.

• The activity dependencies ("dependencies") are taken from the YAML file. These dependencies are added to the predecessors of the activity with the "predecessors" property and to the successors of the activity with the "successors" property.

• The activity is stored in the "activities_table" property.

After the method is completed for all activities, the "graph" object is returned. (Figure 4.1.3)

This method can be used to read YAML files and data from them. This way, programs can read activity dependencies and durations using YAML data. Using this data, programs can visualize activities and dependencies, create timing plans, and plan resources.

5.1 main Class

This code (Figure 5.1) defines the "main" function of a program. This function performs the main functions of the program. The function takes the "argc" and "argv" parameters, which represent the command-line arguments of the program. The "argv[1]" parameter contains the name of the YAML file that the program will run.

Firstly, the YAML file named "argv[1]" is loaded using the "YAMLParser" class and a "Graph" object is created. This object contains the dependencies and durations of activities. Then, the "Graph" object is accessed as in the previous example, and the early start ("ES") and finish ("EF") times, as well as the late start ("LS") and finish ("LF") times, are calculated using the "calculateET" and "calculateLT" methods.

Then, the "printCriticalPath" method is used to print the critical path. This method prints the critical path based on the "slack" values of the activities. The other code snippets between the comment lines define the fixed activities and dependencies, as in the previous example.

This "main" function performs basic program functions, such as reading data from a YAML file, calculating activity dependencies, and printing the critical path. This way, programs can visualize activities and dependencies, create timing plans, and plan resources.

5.1 Conclusion

The given activities and dependencies represent the parts of a project. Considering the durations of these activities and their dependencies on each other, it has been determined that the critical path is (B) Activity B -> (D) Activity D -> (E) Activity E -> (F) Activity F. The critical path shows the activities that have the longest duration for the completion of the project. (Result 1.1)

The given activities and dependencies represent the parts of a project. Considering the durations of these activities and their dependencies on each other, it has been determined that the critical path of the project is (B) Activity B -> (D) Activity D -> (E) Activity E -> (F) Activity F -> (G) Activity G. The critical path shows the activities that have the longest duration for the completion of the project, and it is the path that should be taken into account for completing the project in the minimum time.

6.1 General Conclusion

The dependencies between activities are of great importance in project management and time planning. Previous activities must be completed for an activity to start or finish. The activities on the critical path reveal the risks that may cause delays in the project. Therefore, project managers and teams should consider these dependencies to manage resources effectively, optimize timing, and track project progress.

The duration and dependencies of activities are essential factors in the project management process. The parallel execution of activities ensures efficient use of resources. However, the risks that the activities on the critical path may cause delays should be taken into account. This way, project managers and teams can develop strategies to ensure the timely and successful completion of the project.

In conclusion, the critical path identified when considering the durations and dependencies of the given activities is of great importance in project management and time planning. This information guides project managers and teams to effectively manage resources, track project progress, and prevent delays. Identifying the critical path requires developing strategies to optimize project timing and successfully complete the project.