TP série N°4: optimisation du temps d'exécution d'une application «critique»

Christoph Samuel – Jankowiak Matthias

I-Position du problème:

1- L'objectif de ce TP est **d'estimer la durée maximale** pour laquelle il est possible retarder le **démarrage d'un thread,** sans pour autant augmenter la durée optimale de **décision globale.** En effet, dans le cadre de déploiement d'une application qui automatise la commande d'une batterie antimissiles, **l'estimation du coût des chemins de coût** est **primordiale** pour la sécurité du projet. Ainsi, une estimation préalable du coût est un facteur déterminant dans la **prise de décision de équipe d'ingénierie logicielle.**

2- Le problème posé peut se ramener à un **problème de recherche de chemin à coût minimal/critique d'un graphe orienté** représentatif de l'application qui automatise la commande d'une batterie antimissiles, en effet, on veut assurer une **estimation maximalisée de chemin** avec un **coût global minimum.** Il nous faut alors déterminer **le chemin possédant le plus faible coût,** nous permettant de parcourir un graphe G du sommet DEBUT jusqu'au sommet FIN, sans pour autant, augmenter la durée optimale de la **décision globale.**

Code thread	Durée maximale	thread(s) précédent(s)
X_1	4	-
X_2	8	-
X_3	1	-
X_4	1	X_3
X_5	6	X_1
X_6	3	X_1
X_7	5	X_2
X_8	3	X_5, X_6 , X_7
X_9	1	X_4
X_{10}	2	X_9
X_{11}	2	X_8
X_{12}	5	X_{10}, X_{11}

3- Le **problème réel** dans le cadre de la théorie des graphes consiste à choisir entre plusieurs algorithmes parmi lesquels on retrouve l'algorithme de **Bellman** et l'algorithme de **Dijkstra.** Nous avons choisi dans le cadre du TP, de s'appuyer sur l'algorithme de **Bellman,** en effet, sa **stratégie est plus adapté** dans notre cas, Bellman recherche le chemin optimal depuis un **sommet source** donné dans un graphe orienté, en choisissant un sommet dont **«tous les prédécesseurs ont déjà été marqués»** contrairement a Dijkstra qui choisit le sommet **«le plus proche».**

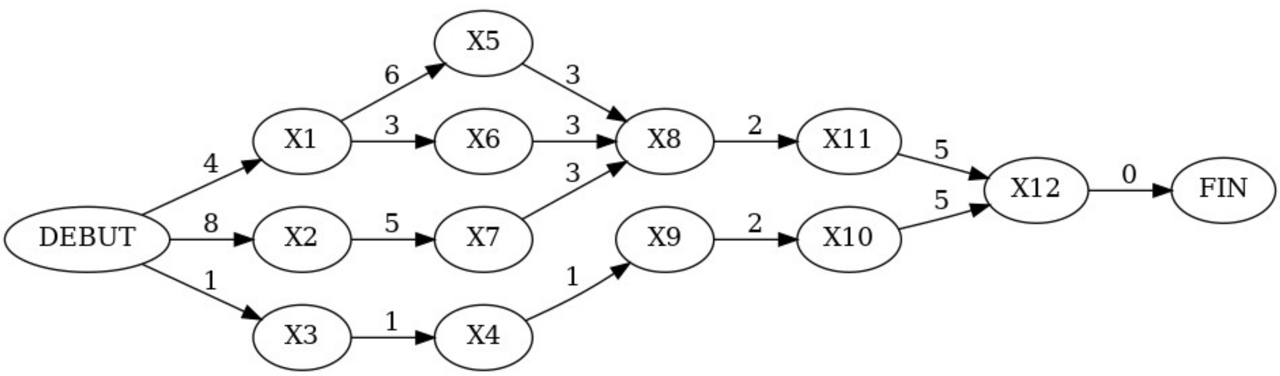
II-Réalisation:

1- Pour définir notre modèle de graphe, on utilisera les **conventions suivantes,** les **sommets du graphe représentent les threads** d'application et à chaque sommet, on associe 3 données, le **code** du thread, sa **date d'activation au plus tôt** et sa **date d'activation au plus tard.**

Les arcs représentent les **relations de précédence entre les threads,** l'arc (x,y) signifie que le thread y est précédée immédiatement du thread x, le **coût associé à un arc** (x,y) correspond à la **durée maximum d'exécution du thread** y, deux sommets particuliers sont à distinguer, l'un appelé Début, représente un thread virtuel qui initialise le cycle: sa durée est nulle, il n'est précédée d'aucun thread, de plus sa date au plutôt et sa date au plus tard sont égales à 0, l'autre, appelé Fin, représente un thread virtuel qui termine le cycle : sa durée est nulle; il ne précède aucun thread. Le modèle de graphe retenu est un **graphe orienté G = (S,A)** définit comme suit.

S = { DEBUT, X1, X2, X3, X4, X5, X6, X7, X8, X9, X10, X11, X12, FIN },
A ={ (DEBUT, X1), (DEBUT, X2), (DEBUT, X3), (X3, X4), (X1, X5), (X1, X6),
(X2, X7), (X3, X4), (X5, X8), (X6, X8), (X7, X8),(X4, X9), (X9, X10),
(X8, X11), (X10, X12), (X11, X12), (X12, FIN) },

Le **modèle de graphe est représenté** tel que les sommets sont définit a l'aide de leurs noms (DEBUT, X1, … , X12, FIN) la **durée en unité de temps de chaque nœuds** sera référencée sur les **arétes allant au nœud** en question pour ce qui concerne les dates au plus tôt et au plus tard d'activation des threads, elle serons référencé directement **dans le terminal lors de la compilation** et non lors de la génération du graphe avec GraphViz. Le modèle de graphe est représenté ci-après.



Le modèle sera complet dés lors que sera calculé pour chaque threads, **sa date de départ au plus tôt** et **sa date de départ au plus tard,** nous affinerons ce modèle au fur et a mesure de l'exercice afin d'obtenir le **modèle complet** de notre graphe.

2- Pour préserver la **relation de précédence** on doit imposer qu'une **décision** ne peut être prise **qu'une fois** prise **toutes les décisions qui la précèdent.** Dans le modèle de graphe proposé, la longueur d'un chemin est calculée en faisant **la somme de tous les sommets qui s'inscrivent le long de ce chemin.**

Donc, la détermination de la **date au plus tôt d'une décision** T se ramène au **calcul de la longueur** Lt du **chemin le plus long** entre les sommets DEBUT et T. D'où la formule suivante de calcul de la date au plus tôt d'une décision:

$$\boxed{D\text{+tôt}(T) = LT}$$

Nous remarquons par ailleurs qu'avec **l'algorithme de Dijkstra,** également vu en cours, nous obtenons, avec le code suivant le **chemin minimal de notre graphe** ainsi que sont coût.Il est important de noter que **l'algorithme de Bellman** est utilisé pour les graphes qui peuvent contenir des **cycles de coût négatifs** et **Dijkstra** est utilisé pour les graphes qui n'ont **pas de cycles de coût négatifs.**

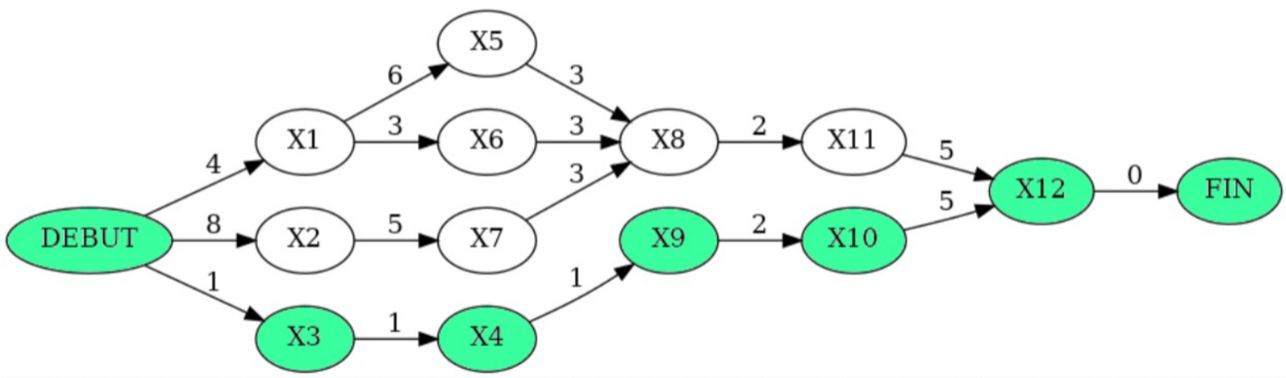
```cpp
// Déclaration des variables pour stocker les résultats de l'algorithme de Dijkstra.
const int num_vertices = 14;
vector<Vertex> p(num_vertices(g));
vector<int> d(num_vertices(g));

// Exécution de l'algorithme de Dijkstra.
dijkstra_shortest_paths(g, DEBUT, predecessor_map(make_iterator_property_map(p.begin(),
get(vertex_index, g))).distance_map(make_iterator_property_map(d.begin(), get(vertex_index, g))));

// Affichage dans le terminal.
cout << "Le chemin minimal du graphe est : \n";
cout << vertex_names[source] << " -> " << vertex_names[target] << endl;
cout << "Son temps d'exécution est de : \n" << distance[target] << endl;

// Coloration des noeuds du graphe en vert, pour le chemin minimal.
Vertex current = target;
while (current != source)
{
    Vertex next = predecessor[current];
    dp.property("color", "green")(edge(next, current, g));
    current = next;
}

```

code C++ pour l'algorithme de Dijkstra

```
-VirtualBox:~/Bureau/TP4 - CHRISTOPH Samuel - JANKOWIAK Matthias$ g++ tp4.cpp -o tp4.o
-VirtualBox:~/Bureau/TP4 - CHRISTOPH Samuel - JANKOWIAK Matthias$ ./tp4.o
```

```
Le chemin minimal du graphe est : DEBUT -> X3 -> X4 -> X9 -> X10 -> X12 -> FIN
Son temps d'éxécution en unité de temps est de : 10
```

En effet, le chemin minimal **de notre graphe** est DEBUT → X3 → X4 → X9 → X10 → X12 → FIN et a **pour durée** 10 unités de temps. Nous pouvons le voir avec l'affichage des résultats suivant ainsi que du graphe qui lui est associé.



chemin minimal de notre graphe

3- Afin de calculer les dates au plus tôt d'activation des threads et la durée optimale du cycle d'exécution, on avons décider d'appliquer **l'algorithme de Bellman,** on obtient donc avec la formule précédente, le **tableau suivant:**

Code thread	Durée (en unité de temps)	thread(s) précédente(s)	Date au plus tôt
DEBUT	0	-	0
X1	4	DEBUT	4
X2	8	DEBUT	8
X3	1	DEBUT	1
X4	1	X3	2
X5	6	X1	10
X6	3	X1	7
X7	5	X2	13
X8	3	X5, X6, X7	16
X9	1	X4	3
X10	2	X9	5
X11	2	X8	18
X12	5	X10, X11	23

La durée optimale de la décision globale est **obtenue en calculant la date au plus tôt** du thread FIN, nous pouvons, de plus observer ses résultats dans notre terminal. D'après la question 2, elle est égale à **longueur** du chemin le plus long **partant** du sommet DEBUT et **atteignant** le sommet FIN, elle est, dans notre cas de:

date+tôt(FIN) = 23 (unités de temps)

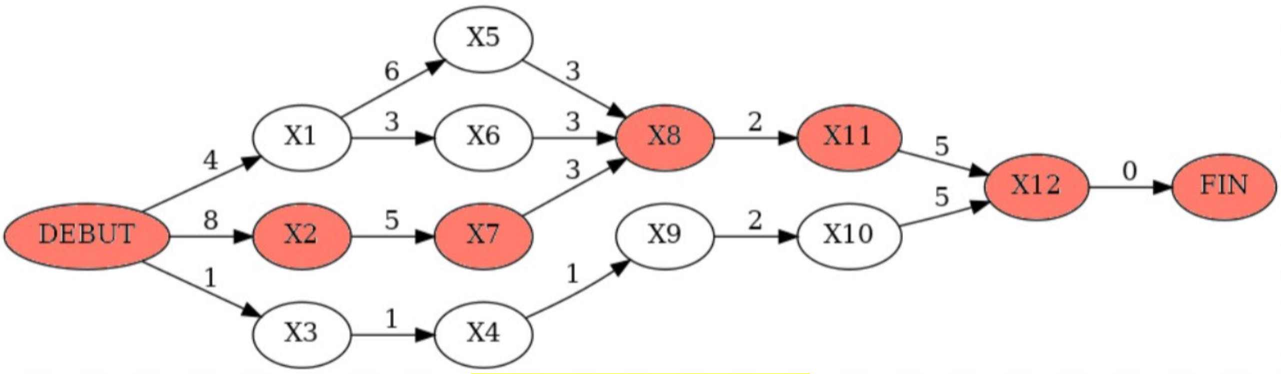
```
Les dates au plus tôt :
La date au plus tôt pour atteindre le noeud X1 est : 4
La date au plus tôt pour atteindre le noeud X2 est : 8
La date au plus tôt pour atteindre le noeud X3 est : 1
La date au plus tôt pour atteindre le noeud X4 est : 2
La date au plus tôt pour atteindre le noeud X5 est : 10
La date au plus tôt pour atteindre le noeud X6 est : 7
La date au plus tôt pour atteindre le noeud X7 est : 13
La date au plus tôt pour atteindre le noeud X8 est : 16
La date au plus tôt pour atteindre le noeud X9 est : 3
La date au plus tôt pour atteindre le noeud X10 est : 5
La date au plus tôt pour atteindre le noeud X11 est : 18
La date au plus tôt pour atteindre le noeud X12 est : 23
```

Terminal affichant les dates au plus tôt pour l'ensemble des threads

4- Est considérée comme **critique** toute décision partielle T telle que date+tôt(T) = date+tard (T). Tout retard au niveau d'une telle décision entraîne la **remise en cause de la durée optimale** du thread globale calculée lors de la question 2.

```
-VirtualBox:~/Bureau/TP4 - CHRISTOPH Samuel - JANKOWIAK Matthias$ g++ tp4.cpp -o tp4.o
-VirtualBox:~/Bureau/TP4 - CHRISTOPH Samuel - JANKOWIAK Matthias$ ./tp4.o
```

```
Le chemin critique du graphe est : DEBUT -> X2 -> X7 -> X8 -> X11 -> X12 -> FIN
Son temps d'éxécution en unité de temps est de : 23
```



chemin critique du graphe

Le chemin traversant les sommets correspondant aux **décisions critiques** est appelé **chemin critique:** c'est le chemin le long du graphe partant du sommet DEBUT et atteignant le sommet FIN. La **détermination des threads critique** ne pourra être déterminée seulement après la question 6 (cf fin étude de cas).

5- Il convient de tenir le raisonnement suivant ***« L'objectif est d'estimer la durée maximale dont il est possible retarder le démarrage d'un thread sans pour autant augmenter la durée optimale de la décision globale».*** Pour cela, on doit garantir que la **date de fin** d'un thread de décision doit **précéder la date de début** de tout les threads **de décision suivantes.** D'où le calcul pour chaque thread de décision T du chemin le plus long entre le sommet FIN et le sommet qui représentant T.

6- Notant LTF, la **longueur du plus long chemin** entre le sommet représentant la décision T et la tâche FIN. La date de début au plus tard de la décision T est calculée à l'aide de la formule:

$$\boxed{\text{Date+tard}(T) = (23 - LTF\ )}$$

La longueur L est calculée en appliquant l'algorithme de **Bellman-Dual** la **complexité** est en **O(| S || A |)** ou S est le nombre de sommets et A est la nombre d'arcs, le résultat de la **trace d'exécution** est exposée dans le terminal après compilations de notre programme, le terminal nous affiche les dates aux plus tard suivantes.

```
Les dates au plus tard :
La date au plus tard pour atteindre le noeud X12 est : 23
La date au plus tard pour atteindre le noeud X11 est : 18
La date au plus tard pour atteindre le noeud X10 est : 18
La date au plus tard pour atteindre le noeud X9 est : 16
La date au plus tard pour atteindre le noeud X8 est : 16
La date au plus tard pour atteindre le noeud X7 est : 13
La date au plus tard pour atteindre le noeud X6 est : 13
La date au plus tard pour atteindre le noeud X5 est : 13
La date au plus tard pour atteindre le noeud X4 est : 15
La date au plus tard pour atteindre le noeud X3 est : 14
La date au plus tard pour atteindre le noeud X2 est : 8
La date au plus tard pour atteindre le noeud X1 est : 10
```

Terminal affichant les dates au plus tard pour l'ensemble des threads

Au niveau de notre code nous avons utilisé la librairie **BoostGraph** en C++ pour appréhender la compilation des algorithmes de **Bellman-Ford, Bellman-Dual** et Dijkstra, voici, dans ce qui suit notre code détaillé, d'une part part étapes et d'une seconde part a l'aide de multiples commentaires disponibles dans le fichier cpp (tp4.cpp).

```cpp
// Pour std::cout
#include <iostream>
using namespace std;
// Inclusion de la librairie boost.
#include <boost/graph/dijkstra_shortest_paths.hpp>
#include <boost/graph/bellman_shortest_paths.hpp>
#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/reverse_graph.hpp>
#include <boost/graph/graph_utility.hpp>
#include <boost/graph/graph_traits.hpp>
#include <boost/graph/graphviz.hpp>
#include <boost/lexical_cast.hpp>
using namespace boost;

```

Ajout des librairies et include nécessaire à la compilation des algorithmes

```cpp
// Dans la structure, on définit les propriété des vertex, c'est à dire des noeux.
struct VertexProperties
{
    // Un identifiant désigné par une chaîne de caractéres.
    string id;
    // Un constructeur par défaut si aucune donnée n'est rentrée lors de la création du noeud.
    VertexProperties() : id(0) {}
    // La fonction qui permet d'assigner les valeur saisie par l'utilisateur au noeud correspondant.
    VertexProperties(string i) : id(i) {}
};

```

Définition des sommets du graphe

```cpp
// Création de la propriété nécessaire a la création du coût de chaques arêtes.
typedef property<edge_weight_t, int> EdgeWeightProperty;
// Passage des paramètres nécéssaires à adjacency_list afin d'obtenir le graph désiré.
typedef adjacency_list<listS, vecS, directedS, VertexProperties, EdgeWeightProperty> Graph;
// Définiton du type sommets.
typedef typename graph_traits<Graph>::vertex_descriptor Vertex;
// Définiton du type arêtes.
typedef typename graph_traits<Graph>::edge_descriptor Edge;

```

Déclaration du type global de notre graphe

```cpp
    // Déclaration de tous les noeuds de nom : "X1", d'id : 1 dans le Graphe g.
    Vertex DEBUT = add_vertex(VertexProperties("DEBUT"),g);
    Vertex X1 = add_vertex(VertexProperties("X1"),g);
```

Déclaration des sommets

```cpp
    // Déclaration de tous les arcs et leurs coûts : arc X1 qui est dirigé vers arc X5 de coût 6 dans le graphe g.
    add_edge(DEBUT, X1, EdgeWeightProperty(4),g);
    add_edge(DEBUT, X2, EdgeWeightProperty(8),g);
```

Déclaration des arrêtes du graphe

```cpp
// Fonction d'application.
int main()
{
    // Création du graphe.
    Graph g;

```

Début de la fonction d'application

```cpp
    // Initialisation d'une dynamic_properties pour afficher les noeuds et arcs de notre Graphe g.
    dynamic_properties dp;
    // L'élément dp récupére chaque noeuds et arcs et leurs donnent un nom (ou un coût) pour la création du graphe en png.
    dp.property("node_id", get(vertex_index, g));
    dp.property("label", get(&VertexProperties::id, g));
    dp.property("label", get(edge_weight, g));
    dp.property("weight", get(edge_weight, g));

```

Initialisation des dynamic properties

```cpp
    // Déclaration des variables pour stocker les résultats de l'algorithme de Bellman.
    vector<int> distance(num_vertices);
    vector<Vertex> predecessor(num_vertices);

    // Exécution de l'algorithme de Bellman.
    bellman_ford_shortest_paths(g, num_vertices, vertex_distance_map(vertex_cost_map).weight_map(get(edge_weight, g))
    .distance_inf(numeric_limits<int>::max()).distance_zero(0), &distance[0], &predecessor[0],root_vertex(source));

    // Affichage dans le terminal.
    cout << "Le chemin critique du graphe est : \n";
    cout << vertex_names[source] << " -> " << vertex_names[target] << endl;
    cout << "Son temps d'exécution est de : \n" << distance[target] << endl;

    // Coloration des noeuds du graphe en rouge, pour le chemin critique.
    Vertex current = target;
    while (current != source)
    {
        Vertex next = predecessor[current];
        dp.property("color", "red")(edge(next, current, g));
        current = next;
    }

```

Définition et utilisation de l'algorithme de Bellman-Ford

```cpp
    // Initialisation du graphe initials sous graph.dot.
    string graphDot = "graph.dot";
    ofstream FGraph(graphDot.c_str());

    // Création de l'image initiale sous graphe.png contenant le graphe g.
    write_graphviz_dp(FGraph, g, dp);
    system("dot -Grankdir=LR -Tpng graph.dot > graph.png");

    return 0;
}
```

Création de l'image sous png à l'aide de la librairie GraphViz et fin du programme

III-Bilan/Conclusion:

1- Nous avons appris grâce à ce TP que certains **problèmes réels d'ingénierie logicielle,** sont traduisibles en un **problème de recherche de chemin à coût minimal de la théorie des graphes**. Dans notre cas, recherche le chemin optimal d'un graphe à l'aide d'un algorithme adapté, celui de **Bellman,** nous a permis de mieux comprendre son fonctionnement et ses différentes étapes. Il en devient donc simple de résoudre de tels problèmes, à l'aide du **modèle de graphe** et des outils à dispositions.

2- Nous retenons également cette façon de faire, que nous pourrions vraisemblablement être confrontés à ce **type de problématique** dans le futur si nous sommes amenés à travailler dans le **domaine d'ingénierie logicielle** et la recherche le chemin optimal dans le cadre d'une application dite **«critique».**