

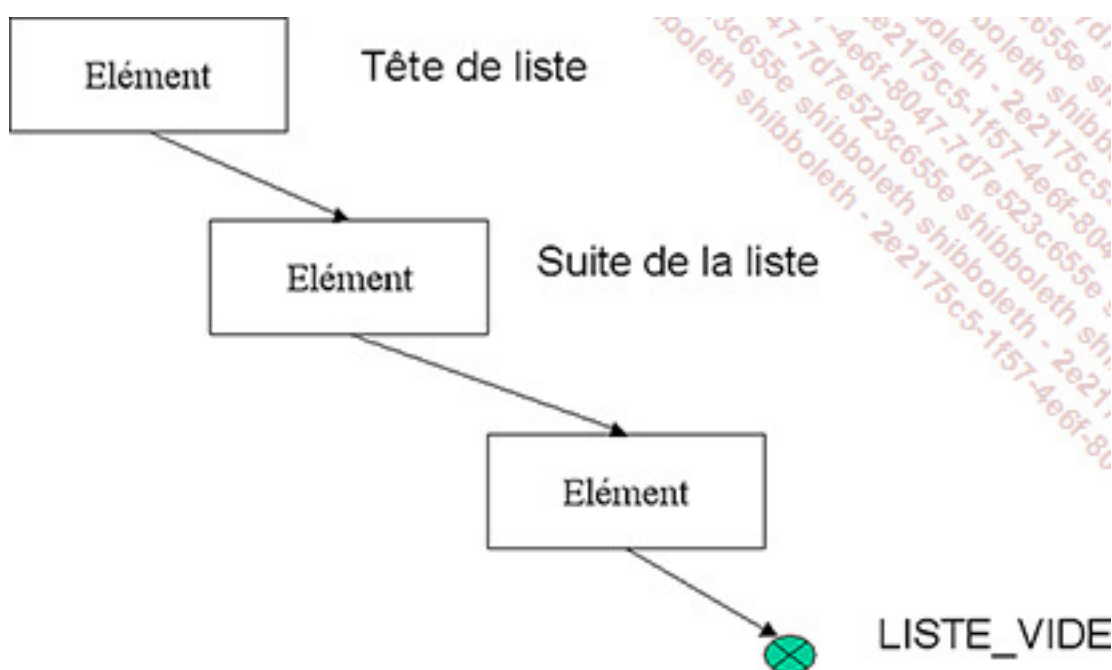
Recherche du plus court chemin

Il existe quantité de situations dans lesquelles on doit rechercher le plus court chemin ; dans le domaine de la logistique, dans celui de la conduite assistée par ordinateur, dans le domaine de la planification et de l'ordonnancement, dans les moteurs de scénario de jeu vidéo...

Avant de présenter trois algorithmes incontournables, chacun d'entre eux naturellement adapté des applications particulières, voici une petite introduction aux graphes.

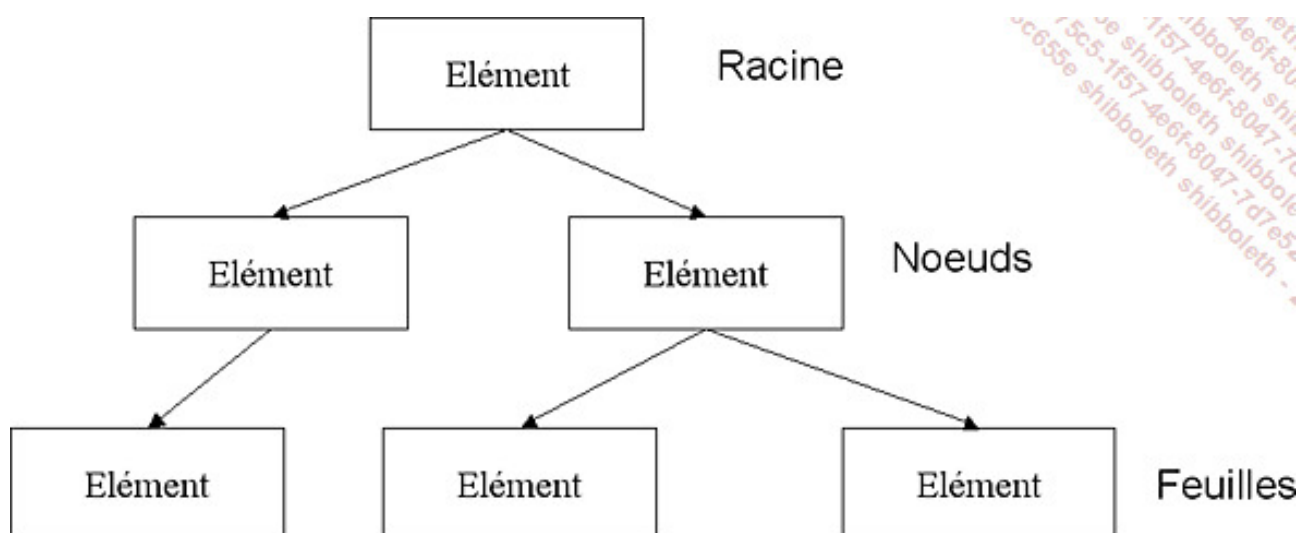
1. Présentation des graphes

Nous avons étudié au chapitre La bibliothèque Standard Template Library une structure dynamique appelée **liste**. Une liste est constituée d'un élément de **tête** (une valeur) rattaché à une liste appelée **suite**. Pour parcourir la liste, on utilise volontiers une fonction récursive qui se rappelle jusqu'à atteindre la liste vide.



En considérant qu'un élément peut avoir plusieurs successeurs on obtient un **arbre** constitué de **nœuds**. Les nœuds ont un seul ancêtre appelé père et un nombre multiple de

descendants appelés fils. Les nœuds terminaux qui n'ont pas de fils sont des **feuilles**, tandis que l'élément au sommet de l'arbre s'appelle **racine**. Il existe des cas particuliers d'arbres, par exemple ceux qui ont au plus deux fils appelés B-arbres ou B-trees, le B signifiant en anglais binary pour le nombre 2.



Un **graphe** généralise encore la construction de l'arbre en admettant qu'un nœud (appelé sommet) compte plusieurs ancêtres.

Les graphes sont formés de deux types d'éléments : des **sommets** et des **arcs**. Les sommets représentent les objets, et les arcs établissent les relations entre ceux-ci. Les arcs peuvent être porteurs de valeurs (ou poids) mais cette caractéristique n'est pas déterminante pour l'étude d'un graphe à proprement parler.

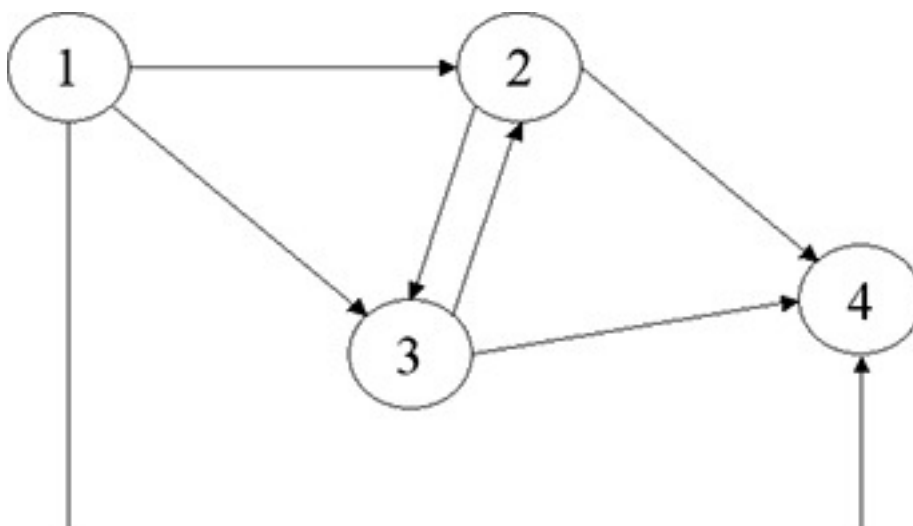
Les graphes peuvent être orientés ou non-orientés. Dans le premier cas, les arcs sont orientés et sont représentés par des flèches. Au besoin, un arc peut être bidirectionnel en étant dédoublé pour représenter chaque sens. La numérotation ou l'identification des sommets n'est pas non plus une caractéristique déterminante, même si c'est usage, du moins dans les phases d'étude d'un algorithme.

Mathématiquement, un graphe est formé d'un ensemble S et d'une relation A sur S . On note $G = (S, A)$. Dans le cas d'un arc orienté, si un arc va du sommet a au sommet b , nous avons :

$$a A b$$

(a en relation avec b). Cette relation n'est pas symétrique et rien n'indique que $b \mathrel{A} a$.

Nous pouvons décrire un graphe orienté G formé de l'ensemble $S = \{ 1 ; 2 ; 3 ; 4 \}$ et de la relation $A = \{ (1 ; 2) (2 ; 3) (2 ; 4) (1 ; 3) (3 ; 2) (3 ; 4) (1 ; 4) \}$.



a. Incidence et adjacence

L'adjacence est une relation entre deux sommets. Si un graphe contient l'arc $(a ; b)$, le sommet b est dit adjacent au sommet a . Dans le cas d'un graphe non-orienté, cette relation d'adjacence est symétrique.

Un graphe dont tous les sommets sont adjacents à tous les autres est dit complet.

L'incidence est une relation entre un sommet et un arc. Dans un graphe orienté, l'arc $(a ; b)$ est incident à partir du sommet a (il part) et est incident au sommet b (il arrive).

b. Chemin d'un graphe

Le chemin d'un graphe est une suite de sommets parcourus en suivant les arcs qui les relient.

Formellement, un chemin d'un sommet a à un sommet a est une suite de sommets $(s_0, s_2 \dots s_k)$ dans laquelle $a = s_0$ et $a = s_k$, et où tous les (s_{i-1}, s_i) sont dans A pour $i = 1, 2, \dots, k$. Un tel chemin contient les arcs $(s_0, s_1) (s_1, s_2) \dots (s_{k-1}, s_k)$, il a une longueur de k . Si un chemin existe de a à a ,

`a` est accessible à partir de `a`.

c. Implémentation des graphes orientés et non orientés

Un moyen très commode pour représenter un graphe consiste à utiliser des listes d'adjacence. Ces listes sont formées à l'aide de deux champs : un sommet et une liste de sommets adjacents.

On peut également utiliser une matrice d'adjacence qui est avantageuse si le graphe est complet, c'est-à-dire si le degré d'adjacence est important.

Voici le code C++ d'un graphe basé sur des listes d'adjacence. Cette implémentation ressemble à celle étudiée au chapitre Les univers de C++ pour le tableur, elle est ici adaptée à du C++ standard.

```
#include <string>
#include <vector>

using namespace std;

/// <summary>
/// Représente un sommet au sein d'un graphe
/// </summary>
class Sommet
{
public:
    /// <summary>
    /// Nom du sommet
    /// </summary>
    string name;

    /// <summary>
    /// Sommets du graphe adjacents
    /// </summary>
    vector<Sommet*> adjacents;

public:
    Sommet()
    {
```

```

        name = "";
    }
};

/// <summary>
/// Représente un graphe dirigé
/// </summary>
class Graphe
{
private:
    /// <summary>
    /// Ensemble des sommets du graphe (membre privé)
    /// </summary>
    vector<Sommet*> g;

public:
    /// <summary>
    /// Constructeur
    /// </summary>
    Graphe()
    {

    }

public:
    /// <summary>
    /// Ajoute un sommet au graphe, avec ses dépendances (sommets adjacents)
    /// </summary>
    void AjouterSommet(string name, vector<string> dependances)
    {
        Sommet* adj;
        Sommet*s = nullptr;
        if ((s = GetSommetByName(name)) == nullptr)
        {
            // le sommet n'existe pas encore
            s = new Sommet();
            s->name = name;
            g.push_back(s);
        }
    }
}

```

```

// ajoute ses dépendances
for (auto i = dependances.begin(); i < dependances.end(); i++)
    if ((adj = GetSommetByName(*i)) != nullptr)
    {
        s->adjacents.push_back(adj);
    }
else
{
    // créé un nouveau sommet adjacent
    adj = new Sommet();
    adj->name = *i;
    g.push_back(adj);
    s->adjacents.push_back(adj);
}
}

public:
    /// <summary>
    /// Identifie le sommet appelé name
    /// </summary>
    Sommet* GetSommetByName(string name)
    {
        for (auto i = g.begin(); i < g.end(); i++)
        {
            Sommet* s = *i;
            if (s->name == name)
                return s;
        }
        return nullptr;
    }

private:
    bool hasError;

public:
    bool HasError()
    {
        return hasError;
    }

```

```
void setError(bool value)
{
    hasError = value;
}

private:
    string errors;

public:
    string getErrors()
    {
        return errors;
    }
    void setErrors(string value)
    {
        errors = value;
    }
};

int main()
{
    Graphe g;

    // sommet 1
    vector<string> adj_s1;
    adj_s1.push_back("2");
    adj_s1.push_back("3");
    adj_s1.push_back("4");

    g.AjouterSommet("1", adj_s1);

    // sommet 2
    vector<string> adj_s2;
    adj_s2.push_back("3");
    adj_s2.push_back("4");

    g.AjouterSommet("2", adj_s2);

    // sommet 3
    vector<string> adj_s3;
```

```

adj_s3.push_back("2");
adj_s3.push_back("4");

g.AjouterSommet("3", adj_s3);

// sommet 4
vector<string> adj_s4;
g.AjouterSommet("4", adj_s4);
}

```

2. Le parcours de graphe en largeur d'abord

Le parcours en largeur d'abord d'un graphe se déroule en visitant tous les sommets adjacents à un sommet donné avant de continuer l'exploration. Une des applications est la détermination des chemins les plus courts.

On commence par choisir un sommet de départ et on le marque de gris : tous les autres sommets du graphe sont initialement marqués de blanc. Ce sommet de départ est placé dans une file.

L'algorithme procède ensuite de la façon suivante : le sommet de départ est placé dans la file. Pour chaque sommet pris en tête de file, on explore tous les sommets qui lui sont adjacents.

Au fur et à mesure que l'on visite un sommet adjacent, on teste son marquage : si la marque du sommet est blanche, c'est qu'il n'a pas encore été parcouru. En ce cas, on le marque de gris pour indiquer qu'il a été visité et on le place à la fin de la file. Si le sommet n'est pas blanc, c'est qu'il a déjà été parcouru et la recherche passe alors au sommet adjacent suivant.

Lorsque tous les sommets adjacents ont été explorés, on le marque de noir pour indiquer qu'on en a fini avec lui. Ce traitement se poursuit jusqu'à ce que la file soit vide. Tous les sommets accessibles à partir du sommet de départ sont alors de couleur noire.



En pratique, deux états blanc et gris suffisent pour dérouler l'algorithme. On cherche à éviter d'atteindre plusieurs fois un sommet dans le cas d'un graphe cyclique.

Ce type de recherche fonctionne également avec des graphes non-orientés.

L'implémentation qui suit traduit cet algorithme en C++ à l'aide de composants de la STL :

```
void Graphe::parcours_largeur(string sommet)
{
    queue<string> visites;
    map<string, Couleur> couleurs;
    map<string, int> nb_saut;

    parcourir(sommet, visites, couleurs, nb_saut);
}

void Graphe::parcourir(string sommet, queue<string>& visites,
    map<string, Couleur>& couleurs, map<string, int>& nb_saut)
{
    couleurs[sommet] = Couleur::Gris;
    visites.push(sommet);

    while (!visites.empty())
        visiter(visites, couleurs, nb_saut);

    // plus court chemin
    cout << "Plus courts chemins depuis " << sommet << " :\n";
    for (auto m : nb_saut)
    {
        cout << m.first << " = " << m.second << endl;
    }
}

void Graphe::visiter(queue<string>& visites, map<string, Couleur>&
    couleurs, map<string, int>& nb_saut)
{

```

```

string s = visites.front();
visites.pop();

cout << "visite du noeud " << s << endl;
Sommet* sommet = GetSommetByName(s);

for (auto ia = sommet->adjacents.begin(); ia
< sommet->adjacents.end(); ia++)
{
    Sommet* adj = *ia;

    if (couleurs.count(adj->name) == 0)
    {
        couleurs.insert(make_pair(adj->name, Couleur::Blanc));
    }

    if (couleurs[adj->name] == Couleur::Blanc)
    {
        couleurs[adj->name] = Couleur::Gris;
        visites.push(adj->name);

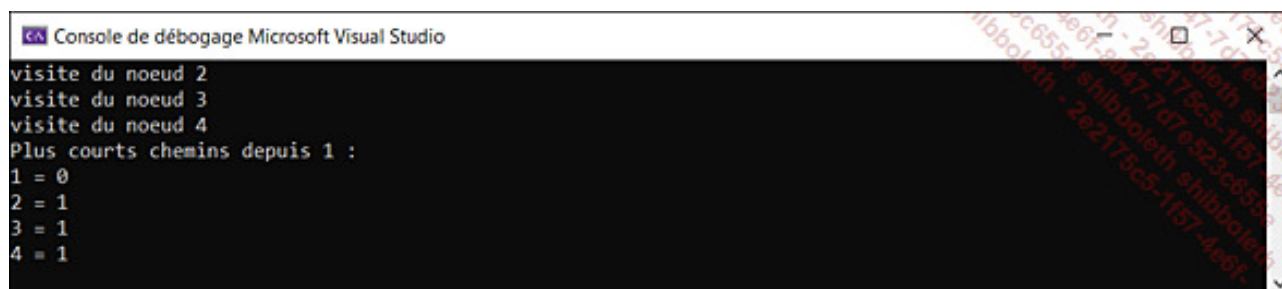
        // plus court chemin
        if (nb_saut.count(s) == 0)
            nb_saut[s] = 0;

        if (nb_saut.count(adj->name) == 0)
            nb_saut[adj->name] = 0;

        nb_saut[adj->name] = nb_saut[s] + 1;
    }
}

couleurs[s] = Couleur::Noir;
}

```



```

Microsoft Visual Studio Debug Console
visite du noeud 2
visite du noeud 3
visite du noeud 4
Plus courts chemins depuis 1 :
1 = 0
2 = 1
3 = 1
4 = 1

```

3. L'algorithme de Dijkstra

L'algorithme de Dijkstra généralise le parcours en largeur d'abord en introduisant la notion de coût ou de poids au niveau des arcs. Au lieu de compter le nombre de sauts pour "passer" d'un sommet à l'autre, on totalise les poids des arcs.

Soit $G = (V, A)$. Chaque arc a une valeur positive, et s est un sommet quelconque.

On pose $S = \{1\}$ (sommet 1).

$C[N \times N]$ est la matrice des coûts.

D contiendra après exécution la plus courte distance de s vers les autres nœuds.

```

S = {1}
pour i = 2 à n
    d[i] = c[1,i]
fin pour

pour i=1 à n-1
    choisir w dans V-{S} / D[w] est minimal
    S = S + {w}
    pour chaque noeud v dans V\S
        D[v] = Min(D[v], D[w] + C[w,v])
    fin pour
fin pour

```

Nous donnons ici une implémentation assez basique d'une classe implémentant un graphe à l'aide d'une matrice d'adjacence. Pour être cohérent entre les notations des

sommets 1 à N et les index des tableaux C++, les indices sont ajustés d'un -1 à dans les méthodes `get_adj()` et `set_adj()`. L'implémentation de la méthode `dijkstra()` est donnée directement dans la définition de la classe :

```
class GrapheMatrice
{
public:
    int* matrice;
    int nb_sommet;
    int* distance;

    int get_adj(int sommet, int adj) {
        return matrice[(sommet-1) * nb_sommet + (adj-1)];
    }

    void set_adj(int sommet, int adj, int valeur) {
        matrice[(sommet - 1) * nb_sommet + (adj - 1)] = valeur;
    }

    GrapheMatrice(int nb_sommet) {
        this->nb_sommet = nb_sommet;
        matrice = new int[nb_sommet * nb_sommet];
        distance = nullptr;
    }

    void dijkstra() {
        cout << "Plus courts chemins" << endl;
        distance = new int[nb_sommet+1];
        const int s1 = 1;
        distance[0] = 0;

        map<int, bool> S;

        for (int i = s1; i <= nb_sommet; i++)
            distance[i] = get_adj(s1,i);

        for (int i = s1; i <= nb_sommet; i++)
        {
            int min = 0;
            int w = 0;
```

```

// choisir w dans V-{S} / D[w] est minimal
for (int w = s1; w <= nb_sommet; w++)
{
    if (S.count(w) == 0 || S[w] == false)
    {
        min = distance[w];
        break;
    }
}
for (int r = s1; r <= nb_sommet; r++)
{
    if ((S.count(r) == 0 || S[r] == false) && distance[r] < min)
    {
        w = r;
        min = distance[r];
    }
}

cout << "w= " << w << "\n";
S[w] = true;

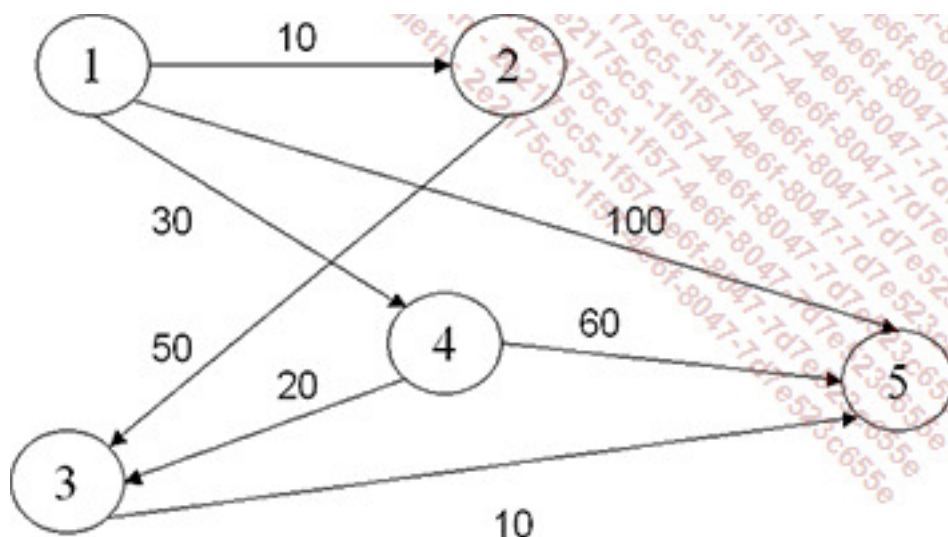
cout << "S = { ";
for (int r = s1; r <= nb_sommet; r++)
    if (S.count(r) > 0 && S[r] == true)
        cout << r << ",";
cout << "}\n";

for (int v = s1; v <= nb_sommet; v++)
{
    if (S.count(v) == 0 || S[v] == false)
    {
        if ((get_adj(w,v) != 1000) &&
            (distance[v] > distance[w] + get_adj(w,v)))
            distance[v] = distance[w] + get_adj(w,v);
    }
    cout << "distance[" << v << "]=" << distance[v] << "\n";
}
}
}

```

};

Nous décidons d'appliquer le programme au graphe suivant :



L'exécution du programme fournit les informations suivantes à partir du sommet 1 :

w	S	D[2]	D[3]	D[4]	D[5]
w=2	S = {1,2}	10	60	30	100
w=4	S = {1,2,4}	10	50	30	90
w=3	S = {1,2,3,4}	10	50	30	60
w=5	S = {1,2,3,4,5}	10	50	30	60

Il est facile de vérifier que pour aller de 1 à 5, on peut passer par 4, 3 et 5, soit un coût de 60. Toutefois, l'algorithme ne fournit pas le chemin !

On remarque que l'algorithme affine progressivement ses résultats, mais qu'il ne peut se prononcer que lorsqu'il a étudié tous les cas de figure.

4. Utilisation d'une méthode avec heuristique : l'algorithme A*

L'algorithme A* est un algorithme de recherche d'un ensemble de chemins formant un arbre au sein d'un graphe pour partir d'un nœud à la recherche d'un autre nœud. Il fonctionne avec un heuristique qui classe chaque sommet en estimant la meilleure route à travers ce sommet.

L'algorithme commence au sommet sélectionné ; il détermine pour ce sommet un coût à partir du coût de sommet (soit 0 pour le nœud initial). A* estime la distance vers le sommet cible à partir du sommet courant.

L'heuristique, formé en combinant cette estimation et le coût, est assigné au chemin depuis le sommet. Ce nœud est placé dans une file à priorité habituellement appelée `open`.

Ensuite, l'algorithme dépile le prochain nœud de la file `open`. Le nœud dépilé est celui avec la priorité la plus faible, donc l'heuristique le plus faible. Si la file est vide, il n'y a pas de chemin disponible. Si le nœud est le sommet cible, A* construit et affiche les différents chemins puis s'arrête.

Si le sommet n'est pas le sommet cible, de nouveaux nœuds sont créés pour chaque sommet adjacent. Pour chaque nœud, A* calcule le coût et l'enregistre. Les coûts s'accumulent de proche en proche.

L'algorithme maintient également une liste fermée `closed` de nœuds adjacents déjà visités. Si les nouveaux sommets générés sont déjà dans la liste avec un coût égal ou inférieur, ces nœuds ne seront plus visités. Si un nœud dans la liste des fermés correspond au nouveau nœud, mais a été enregistré avec un coût supérieur, celui-ci est également supprimé des fermés et le traitement continue.

Ensuite, une estimation de la distance vers le nouveau nœud est ajoutée au coût pour former l'heuristique du nouveau nœud. Celui-ci est ajouté à la liste des ouverts, à moins qu'il ne s'y trouve déjà.

Lorsque toutes ces étapes ont été accomplies pour les nœuds adjacents, le nœud original est ajouté à la liste des fermés. Le processus se répète dans son ensemble en dépilant le prochain nœud.

La fonction heuristique $h(n)$ doit être prédominante sur $c(n)$ - le coût réel - pour que l'algorithme se distingue nettement de l'algorithme de Dijkstra.

Toutefois, la construction de cette fonction dépend de la forme du graphe. Si le graphe a une forme de grille - l'algorithme est utilisé dans ce cas pour rechercher un bon chemin malgré des obstacles - la distance euclidienne ou la distance de Manhattan donnent de bons résultats. Si le graphe a une forme quelconque, il est possible de définir une distance comme différence entre l'identifiant du nœud pour lequel on estime la distance avec l'identifiant du nœud cible. Cela procure une topologie au graphe.

a. Implémentation en C++

Nous commençons l'implémentation par une classe `AEtoileSommet`. Celle-ci est abstraite pour faciliter sa mise en place conjointe avec la classe `AGraphe`.

```
/*
 * AEtoileSommet classe abstraite représentant un sommet pour A*
 */
class AEtoileSommet
{
public:
    // identifiant du sommet
    int ID;

    AEtoileSommet* path_parent;
    float cost_from_start;
    float estimated_cost_to_goal;

    // coût du sommet (en partie estimé)
    float get_cost()
    {
        return cost_from_start + estimated_cost_to_goal;
    }

    // compare le coût du sommet avec le coût d'un autre sommet
    int compareTo(AEtoileSommet* other)
    {
        float this_value = get_cost();
```



```

    float other_value = other->get_cost();
    float v = this_value - other_value;
    return (v > 0) ? 1 : (v < 0) ? -1 : 0;
}

// Obtient le coût entre ce sommet et le sommet adjacent spécifié
virtual float get_cost(AEtoileSommet* node) = 0;

// Obtient le coût estimé entre ce sommet et le sommet spécifié
// Le coût estimé doit être inférieur au véritable coût. Plus il s'en
approche, meilleure est l'estimation.
virtual float get_estimated_cost(AEtoileSommet* node) = 0;

// Obtient la liste des sommets adjacents
virtual vector<AEtoileSommet*> get_sommets_adjacents() = 0;
};

```

Suit la déclaration de la classe `AGraphe` avec un double rôle : c'est elle qui porte la matrice d'adjacence sous forme de pointeurs vers `AEtoileSommet`, mais aussi la matrice de coûts.

```

/*
 * AGraphe implémente un graphe par matrice d'adjacence pondérée
 * Les sommets sont déclarés comme AEtoileSommet*, classe abstraite
 */
class AGraphe
{
public:
    float** graphe; // matrice de coûts
    int N; // nombre de sommets
    AEtoileSommet** sommets; // matrice d'adjacence

    AGraphe(int n)
    {
        N = n;
        graphe = new float*[N];
        sommets = new AEtoileSommet*[N];
    }
};

```

À présent, voici la classe `ASommet` qui implémente la classe abstraite `AEtoileSommet`. On remarque dans la méthode `get_estimated_cost()` un coefficient 1 sur lequel nous reviendrons. Dans la méthode `get_sommets_adjacents()` l'affichage des sommets est mis en commentaire, cela peut être utile d'activer ce code, soit de façon permanente soit à l'aide d'un flag `is_debug`, pour suivre pas à pas l'exécution du programme.

```
class ASommet :
public AEtoileSommet
{
public:
    AGraphe* graphe;

    ASommet(int id, AGraphe* g)
    {
        ID = id;
        graphe = g;
    }

    // Obtient le coût entre ce sommet et le sommet adjacent spécifié
    float get_cost(AEtoileSommet* node)
    {
        return graphe->graphe[ID][((ASommet*)node)->ID];
    }

    // Obtient le coût estimé entre ce sommet et le sommet spécifié
    float get_estimated_cost(AEtoileSommet* node)
    {
        return 1 * abs(ID - ((ASommet*)node)->ID);
    }

    // Obtient des sommets adjacents
    vector<AEtoileSommet*> get_sommets_adjacents()
    {
        vector<AEtoileSommet*> v;
        //cout << "Adj de " << ID+1 << " : ";
        for (int i = 0; i < graphe->N; i++)
            if (graphe->graphe[ID][i] < 1000)
            {
```

```

        //cout << graphe->sommets[i]->ID+1 << " ";
        v.push_back(graphe->sommets[i]);
    }
    //cout << "\n";
    return v;
}
};

```

L'algorithme de recherche a besoin d'une liste à priorité. Bien que plusieurs structures issues de la STL puissent convenir, nous proposons ici la réalisation d'une classe `p_queue` tout simplement basée sur `vector` :

```

/*
 * p_queue file à priorité
 * se base sur vector
 */
class p_queue :
    public vector<AEtoileSommet*>
{
public:
    // affiche les éléments
    void show(string name)
    {
        cout << name << " { ";
        for (auto it = begin(); it < end(); it++)
            cout << (*it)->ID << ((it < end() - 1) ? ", " : "") << " ";
        cout << " } " << endl;
    }

    // supprime l'élément le plus faible
    AEtoileSommet* remove_first()
    {
        auto t = *(end()-1);
        erase(end()-1);
        return t;
    }

    // supprime un élément
    void remove(AEtoileSommet* object)

```

```

{
    for (auto it = begin(); it < end(); it++)
    {
        if ((*it) == object)
        {
            this->erase(it);
            break;
        }
    }
}

// teste l'appartenance d'un élément
bool contains(AEtoileSommet* object)
{
    for (auto it = begin(); it < end(); it++)
    {
        if ((*it) == object)
            return true;
    }
    return false;
}

// ajoute un élément en fonction de son poids
void add(AEtoileSommet* object)
{
    if (contains(object))
        return;

    for (int i = 0; i < size(); i++)
    {
        if (object->compareTo(at(i)) >= 0) // <=
        {
            this->insert(begin() + i, object);
            return;
        }
    }
    push_back(object);
}
};

```

La classe `AEtoileRecherche` implémente l'algorithme proprement dit :

```
class AEtoileRecherche
{
public:
    // Construit le chemin, sans inclure le sommet de départ
    vector<AEtoileSommet*> construire_chemin(AEtoileSommet* node)
    {
        vector<AEtoileSommet*> path;
        while (node->path_parent != nullptr)
        {
            path.push_back(node);
            node = node->path_parent;
        }
        return path;
    }

    // Trouve le chemin du sommet de départ vers le sommet
    d'arrivée.
    // Une liste des sommets est retournée, ou vide si le chemin
    n'est pas trouvé.
    vector<AEtoileSommet*> rechercher_chemin(AEtoileSommet*
startNode, AEtoileSommet* goalNode)
    {
        p_queue open_list;
        p_queue closed_list;

        startNode->cost_from_start = 0;
        startNode->estimated_cost_to_goal =
startNode->get_estimated_cost(goalNode);
        startNode->path_parent = nullptr;

        open_list.add(startNode);

        while (!open_list.empty())
        {
            cout << "taille de ouvert " << open_list.size()
<< " taille de ferme " << closed_list.size() << endl;
            //openList.show("open");
            //closedList.show("closed");
```

```

cout << "\n";

AETOileSommet* node = open_list.remove_first();

if (node == goalNode)
{
    // construit le chemin du départ à l'arrivée
    return construire_chemin(goalNode);
}

vector<AETOileSommet*> neighbors =
node->get_sommets_adjacents();
for (int i = 0; i < neighbors.size(); i++)
{
    AETOileSommet* neighborNode = neighbors.at(i);

    bool isOpen = open_list.contains(neighborNode);
    bool isClosed = closed_list.contains(neighborNode);

    float costFromStart = node->cost_from_start +
node->get_cost(neighborNode);

    // vérifie si le voisin n'a pas été traversé ou si
    // un chemin plus court vers ce sommet est trouvé.
    if ((!isOpen && !isClosed) ||
        costFromStart < neighborNode->cost_from_start)
    {
        neighborNode->path_parent = node;
        neighborNode->cost_from_start = costFromStart;
        neighborNode->estimated_cost_to_goal =
neighborNode->get_estimated_cost(goalNode);

        if (isClosed)
            closed_list.remove(neighborNode);

        if (!isOpen)
            open_list.add(neighborNode);
    }
}
closed_list.add(node);

```

```

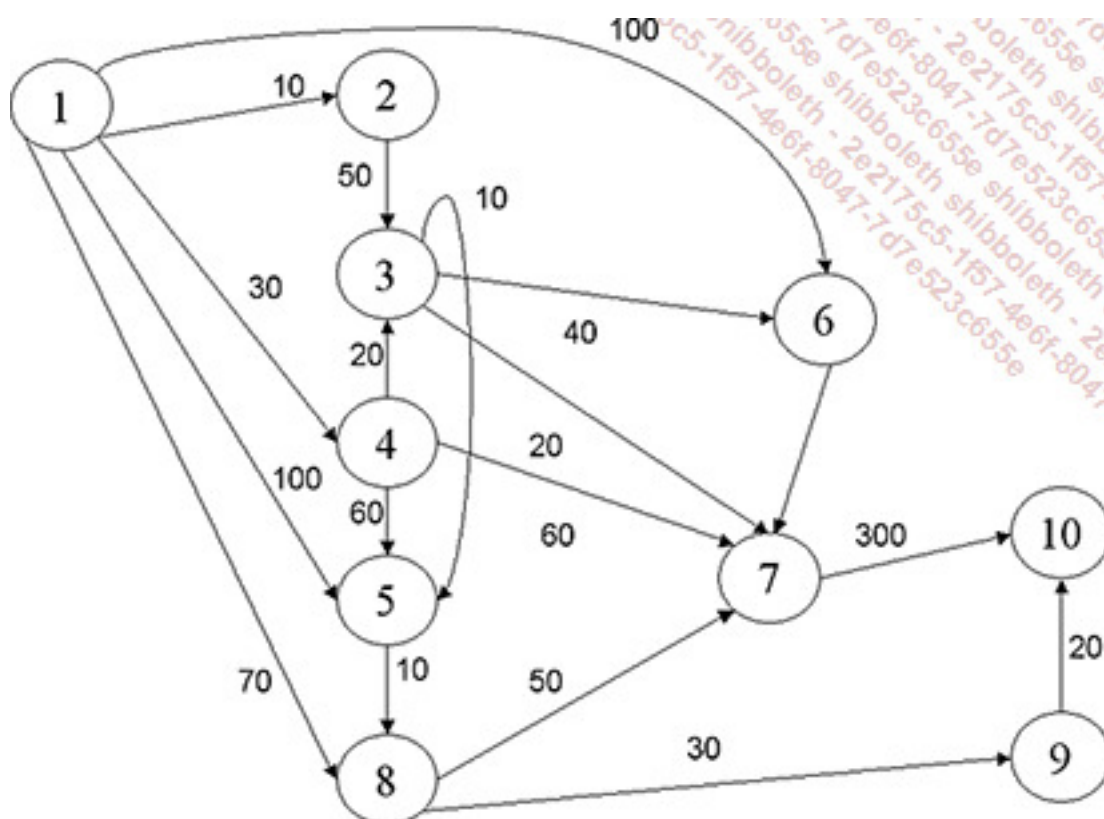
    }

    // pas de chemin trouvé
    return vector<AEtoileSommet*>();
}
};

```

b. Exécution

Nous allons tester A* à partir d'un graphe présentant plusieurs ramifications :



Il existe plusieurs chemins pour aller du sommet 1 au sommet 10, avec des coûts sur les premiers arcs qui pourraient induire l'algorithme en erreur et donc le ralentir. Par exemple le chemin (1 ; 2) représente un coût de 10, ce qui est prometteur comparé aux arcs (1 ; 8) ou (1 ; 6), mais en fin de compte le coût total s'avérera élevé.

Voici la fonction C++ qui paramètre ce graphe et lance l'exécution de l'algorithme A* pour aller de 1 à 10 :

```

void AEtoile()
{
    AGraphe g(10);
    // 1 2 3 4 5 6 7 8 9 10
    g.graphe[0] = new float[] {1000, 10, 1000, 30, 100, 100,
1000, 70, 1000, 1000}; // 1
    g.graphe[1] = new float[] {1000, 1000, 50, 1000, 1000, 1000,
1000, 1000, 1000, 1000}; // 2
    g.graphe[2] = new float[] {1000, 1000, 1000, 1000, 10, 40,
1000, 1000, 1000, 1000}; // 3
    g.graphe[3] = new float[] {1000, 1000, 20, 1000, 60, 1000,
20, 1000, 1000, 1000}; // 4
    g.graphe[4] = new float[] {1000, 1000, 1000, 1000, 1000,
1000, 1000, 10, 1000, 1000}; // 5
    g.graphe[5] = new float[] {1000, 1000, 1000, 1000, 1000,
1000, 10, 1000, 1000, 1000}; // 6
    g.graphe[6] = new float[] {1000, 1000, 1000, 1000, 1000,
1000, 1000, 1000, 1000, 300}; // 7
    g.graphe[7] = new float[] {1000, 1000, 1000, 1000, 1000,
1000, 50, 1000, 30, 1000}; // 8
    g.graphe[8] = new float[] {1000, 1000, 1000, 1000, 1000,
1000, 1000, 1000, 1000, 20}; // 9
    g.graphe[9] = new float[] {1000, 1000, 1000, 1000, 1000,
1000, 1000, 1000, 1000, 1000}; // 10

    for (int i = 0; i < g.N; i++)
        g.sommets[i] = new ASommet(i, &g);

    AEtoileRecherche a;

    int debut = 1;
    int fin = 10;

    cout << "chemin pour aller de " << debut << " vers " << fin
<< endl;

    auto l = a.rechercher_chemin(g.sommets[debut - 1],
g.sommets[fin - 1]);
    for (int i = l.size()-1; i >= 0; i--)

```



```

{
    auto s = l.at(i);
    cout << (s->ID + 1) << " [" << s->cost_from_start
    << "]" << endl;;
}
}

```

La classe `ASommet` utilise la fonction suivante comme heuristique :

```
1 * Math.abs(ID - ((Sommet) node).ID);
```

Étant donné les poids des arcs présents dans le graphe (les valeurs varient entre 10 et 200), la fonction $h(n)$ est négligeable face à $c(n)$ et l'algorithme se conduit un peu comme Dijkstra. Il étudie jusqu'à cinq chemins sur dix sommets pour déterminer le meilleur.

```

Microsoft Visual Studio Console de débogage
chemin pour aller de 1 vers 10
taille de ouvert 1 taille de ferme 0

taille de ouvert 5 taille de ferme 1
taille de ouvert 5 taille de ferme 2
taille de ouvert 5 taille de ferme 3
taille de ouvert 5 taille de ferme 4
taille de ouvert 4 taille de ferme 5
taille de ouvert 4 taille de ferme 6
taille de ouvert 3 taille de ferme 7
taille de ouvert 2 taille de ferme 8
taille de ouvert 1 taille de ferme 9

8 [70]
9 [100]
10 [120]

```

Lorsque la fonction `get_estimated_cost()` comprend un facteur 100, l'effet de heuristique est beaucoup plus évident : l'exécution donne un score identique avec un temps de recherche plus court, puisque l'algorithme est incité à explorer d'abord les sommets d'indices (identifiants) proches du sommet d'arrivée, 10 :

```
// Obtient le coût estimé entre ce sommet et le sommet spécifié
```

```
float get_estimated_cost(AEtoileSommet* node)
{
    return 100 * abs(ID - ((ASommet*)node)->ID);
}
```

```
Console de débogage Microsoft Visual Studio
chemin pour aller de 1 vers 10
taille de ouvert 1 taille de ferme 0

taille de ouvert 5 taille de ferme 1

taille de ouvert 6 taille de ferme 2

taille de ouvert 6 taille de ferme 3

8 [70]
9 [100]
10 [120]
```

Ces résultats prouvent que l'heuristique guide l'algorithme, malgré des coûts qui semblent de prime abord prohibitifs : l'arc reliant le sommet 1 au sommet 8 coûte 70 alors que d'autres arcs partant du sommet 1 ont des coûts inférieurs à 30.

Dans le cas d'un graphe « classique », la fonction heuristique n'est pas facile à doser, et il n'est pas certain que l'algorithme A* donne de meilleurs résultats que Dijkstra, d'autant qu'un heuristique mal évalué peut conduire à un chemin dont le coût est supérieur « au plus court chemin ».

Dans le cas d'un graphe matriciel, les heuristiques basés sur la notion de distance donnent en général des résultats de très bonne qualité, au pire ils s'avèrent identiques à Dijkstra en termes de qualité et de performance.