Epita:Algo:Cours:Info-Spe:les arbres recouvrants minimum

De EPITACoursAlgo.

Sommaire

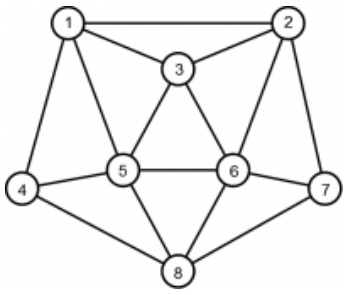
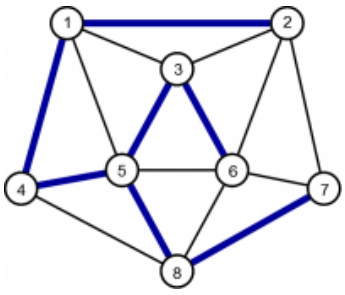
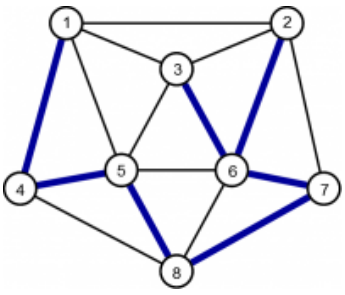
Généralités

Remarques et rappels :

- *Le coût (poids ou valeur) d'un graphe est égal à la somme des coûts des arêtes qui le composent et sera pour un graphe **G** noté Coût(**G**).*
- *Les propriétés suivantes, pour un graphe **G** = < **S**, **A**, **C** > de **n** sommets, sont caractéristiques d'un arbre et sont toutes équivalentes:*
  - ***G** est connexe et sans cycle.*
  - ***G** est connexe avec **n-1** arêtes.*
  - ***G** est connexe et si l'on lui enlève une arête, il ne l'est plus (toute arête est un isthme).*
  - ***G** est sans cycle avec **n-1** arêtes.*
  - ***G** est sans cycle et si l'on lui ajoute une arête, on en crée un.*
  - *Tout couple de sommet **{x,y}** de **S** est relié par une seule chaîne dans **G**.*

Un AR (arbre recouvrant) est, pour un graphe ***G*** connexe, obtenu en construisant un graphe partiel de ***G*** qui est un arbre. c'est à dire connexe et sans cycle. On peut voir sur la figure 1 un graphe connexe avec deux exemples possibles d'AR.

Figure 1. Un graphe connexe non valué et deux arbres recouvrants possibles.

graphe de base	Arbre Recouvrant	Arbre Recouvrant
		

Conditions d'existence:

Supposons un graphe $G = < S, A, C >$ non orienté valué et connexe. Le problème de l'**ARM** (Arbre Recouvrant minimum) consiste à trouver un graphe partiel de G qui est connexe et dont le coût est minimum. Un tel graphe, ne peut avoir de cycle sinon la suppression d'une des arêtes de ce cycle nous fournirait un graphe partiel de coût inférieur et donc mailleur candidat au titre d'ARM. Comme il est connexe, c'est donc un arbre et de coût minimum.

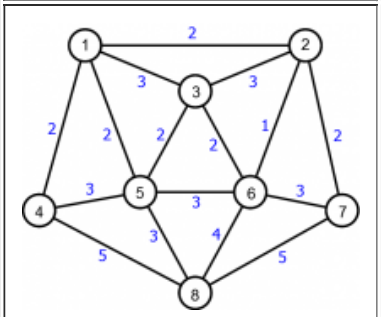
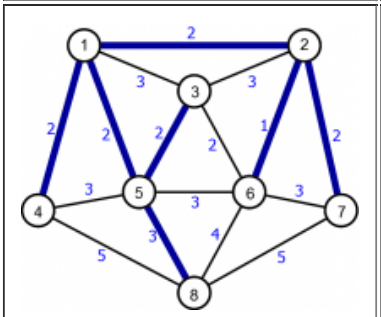
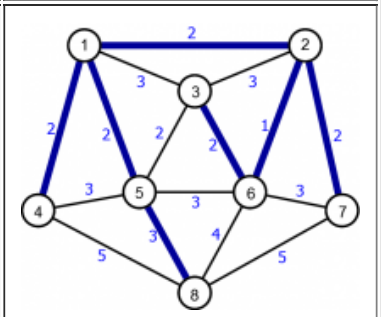
Remarque: Toutefois si le graphe G n'est pas connexe, il est toujours possible de déterminer un graphe partiel de G sans cycle. On obtient alors une fôret recouvrante constituée d'autant d'arbres qu'il y a de composantes connexes dans le graphe G.

Une façon simple de déterminer un arbre recouvrant est de supprimer du graphe d'origine toutes les arêtes qui forment des cycles. Le nombre d'arbres couvrants étant fini, nous sommes assurés d'en trouver un de coup minimum.

Question: est-ce que cet ARM est unique ?

En général non, comme on peut d'ailleurs le voir sur la figure 2 qui présente deux exemples possibles d'ARMs. En revanche, si les coûts des arêtes sont tous distincts, on peut montrer que l'ARM obtenu est unique.

Figure 2. Un graphe connexe valué et deux arbres recouvrants minimum possibles.

graphe de base	Arbre Recouvrant minimum	Arbre Recouvrant minimum
		

Variantes du problème:

Nous allons envisager les deux algorithmes les plus classiques :

1. **Prim** qui proche de **Dijkstra** sur le principe est intéressant pour les graphes peu denses et qui maintient la connexité de l'ensemble des arêtes de l'arbre en constrcution,
2. **Kruskal** qui est très efficace sur les graphes denses et qui lui ne s'attache qu'à maintenir l'acyclicité du graphe en construction.

Prim

Comme dit précédemment, l'algorithme de Prim est très proche de celui de Dijkstra. On part d'un sommet source et ensuite à chaque itération on choisit le sommet libre le plus proche au sens du poids tout en conservant la propriété d'arbre (*connexe et sans cycle*).

Principe

Soit $G = < S, A, C >$ un graphe non orienté valué connexe de n sommets. On se propose de construire un arbre T (graphe partiel de G).

Soit x un sommet de S. Appelons SC l'ensemble des sommets y connectés à x dans T et SR son complémentaire dans S, c'est à dire l'ensemble des sommets de S restants à connecter à x.

- Au départ, SC ne contient que la source (x).
- A chaque étape, on ajoute à SC un sommet y de SR tel que le coût de l'arête $\{x,y\}$ soit minimum. Ce qui traduit l'ajout de l'arête $\{x,y\}$ dans T.
- On s'arrête lorsque SR est vide ($SC = S$).

Remarque: Si le graphe G n'est pas connexe, nous nous arrêterons lorsque SC contiendra tous les sommets accessibles depuis x.

Le choix des arêtes étant que les deux sommets appartiennent à deux ensembles complémentaires, cela nous garantit l'absence de cycle. De plus le coût de l'arête retenue étant minimum, nous sommes aussi assurés du fait que l'arbre ***T*** obtenu est de coût minimum. Nous sommes donc bien en possessions d'un ARM.

Algorithme

L'algorithme va parcourir le graphe ***g*** à partir du sommet source ***x*** et retourner un graphe ***t*** correspondant à l'ARM construit. Pour cela, on utilisera deux tableaux: ***pp*** et ***d*** pour mémoriser pour chaque sommet le sommet le plus proche le reliant à ***x*** et le coût de l'arête entre ces deux sommets. C'est à dire que ***pp[y]*** mémorisera l'arête ***{ y, pp[y] }*** et ***d[y]*** mémorisera coût(***y, pp[y], g***).

Pour éviter des tests inutiles, nous allons étendre les capacités de la fonction de coût pour qu'elle soit définie quelque soit le couple de sommet (***x,y***), soit :

```
coût(x,x,g) = 0 pour un graphe sans boucle


coût(x,y,g) = ∞  si x arête y = faux
```

Nous utiliserons aussi une opération **choisirmin(*M, d*)** qui renverra le sommet $m \in M$ tel que ***d[m]*** est minimum, ainsi que les opérations sur les ensembles.

Pour l'algorithme de Prim, les types de données employés sont les suivants :

```
Constantes

   Nbs = ...                          /* Nombre de sommets du graphe */

Types

   t_vectNbsReel = Nbs reel           /* Vecteur de Nbs réels */
   t_vectNbsEnt  = Nbs entier         /* Vecteur de Nbs entiers */
```

Ce qui donne:

```
Algorithme procédure Prim
Paramètres locaux
   entier x                           /* x est le sommet source */
   graphe g
Paramètres globaux
   graphe t                           /* t est l'arm que l'on construit */
Variables
   t_vectNbsRéel pp                   /* pp[y] est le sommet le plus proche de y */
   t_vectNbsEnt  d                    /* d[y] est le coût de l'arête entre y et pp[y] */
   entier i, y, z, m                  /* y, z et m sont des sommets */
   réel v
   ensemble M
Début
   /* Initialisation de T, des tableaux pp et t et de l'ensemble M */
   t ← graphevide
   M ← ensemblevide
   pour i ← 1 jusqu'à n faire
      d[i] ← coût(x,i,g)
      pp[i] ← x
      M ← ajouter(i,M)
   fin pour
   M ← supprimer(x,M)                        /* CC={x} */
   /* Construction par ajouts successifs de t */
   tant que M <> ensemblevide faire
      m ← choisirmin(M,d)
      M ← supprimer(m,M)                     /* ajout de m à CC */
      z ← pp[m]
      v ← coût(m,z,g)
      t ← ajouterlarête <m,z> de coût v à t
      /* réajustement des valeurs de distance */
      pour i ← 1 jusqu'à d°(m,g) faire
         y ← ième-succ(i,m,g)
         si (y ∈ M) et (coût(m,y,g)<d[y]) alors
            d[y] ← coût(m,y,g)
            pp[y] ← m
         fin si
      fin pour
   fin tant que
Fin Algorithme procédure Prim
```

Exemple

Appliquons l'algorithme de Prim au graphe non orienté valué connexe de la figure 2 en prenant le sommet ***1*** comme sommet source.

Kruskal

L'algorithme de Kruskal s'appuie sur une des propriétés des arbres, qui dit qu'un graphe **G** de **n** sommets est un arbre s'il est sans cycle avec **n-1** arêtes. On part d'un graphe vide et on ajoute **n-1** arêtes tout en respectant la propriété qu'a un arbre d'être *sans cycle*.

Principe

Soit $G = < S, A, C >$ un graphe non orienté valué connexe de n sommets et p arêtes. On se propose de construire un arbre $T=< S, AC, C>$, un graphe partiel de G.

Soit $\{x,y\}$ une arête de A. Appelons AC l'ensemble des arêtes choisies et AR son complémentaire dans A, c'est à dire l'ensemble des arêtes de A restantes.

- Au départ, AC est vide.
- A chaque étape, on ajoute à AC une arête $\{x,y\}$ de AR tel que le coût de l'arête $\{x,y\}$ soit minimum et qu'elle ne crée pas de cycle dans T. Ce qui traduit l'ajout de l'arête $\{x,y\}$ dans T.
- On s'arrête lorsque l'on a ajouté $n-1$ arêtes sans créer de cycle.

Question: Comme savoir si l'ajout de l'arête **{x,y}** *à* **T** *crée un cycle ou non ?*

Il suffit pour cela de savoir s'il existe déjà dans T une chaîne entre x et y. Ce qui traduirait l'appartenance de x et y à une même composante connexe de T. Si ce n'est pas le cas, l'ajout de l'arête $\{x,y\}$ unit les deux composantes en une seule.

Algorithme

Pour cela , l'algorithme de Kruskal va utiliser:

- un vecteur ***pere*** qui associé aux fonctions ***trouver*** et ***réunir*** (cf connexité d'un graphe non orienté évolutif) nous permettra de savoir si deux sommets x et y du graphe T appartiennent ou non à la même composante connexe. Si c'est le cas, cela veut dire qu'ajouter à T l'arête $\{x,y\}$ créerait un cycle, ce que l'algorithme s'abstiendra de faire pour conserver la propriété d'acyclicité d'un arbre.

- un ensemble U contenant les arêtes valuées du graphe G (*au départ* $U=A$).

L'algorithme va donc à chaque tour choisir l'arête de U présentant le plus faible coût. Elle est retirée de U et si elle ne crée pas de cycle, elle est ajoutée à T et comptabilisée.

Quand il en a été ajouté $n-1$, le graphe est constituée des $n-1$ arêtes de plus faible coût ne créant pas de cycle. C'est donc un ARM de G.

Nous utiliserons aussi une opération *choisirmin(U)* qui retournera l'arête valuée $\{x,y,v\}$ de U telle que v est minimum, ainsi que les opérations sur les ensembles.

Pour l'algorithme de Kruskal, les types de données employés sont les suivants :

```
Constantes

   Nbs = ...                          /* Nombre de sommets du graphe */

Types

   t_vectNbsEnt = Nbs entier          /* Vecteur de Nbs entiers */
```

Ce qui donne:

```
Algorithme procédure Kruskal
Paramètres locaux
   graphe g                           /* g est un graphe non orienté valué */
Paramètres globaux
   graphe t                           /* t est l'arm que l'on construit */
Variables
   t_vectNbsEnt pere                  /* pere est le vecteur d'ascendant des sommets */
   ensemble U                         /* U est l'ensemble des arêtes valuées */
   entier i, x, y, rx, ry             /* x, y, rx et ry sont des sommets */
```

```
   réel v                              /* i est le compteur d'arête et v le côut */
Début
   /* Initialisation de T, du vecteur pere et de l'ensemble U */
   t ← graphevide
   U ← ensemblevide
   pour x ← 1 jusqu'à n faire
      pere[x] ← -1
      pour i ← 1 jusqu'à d°(x,g) faire
         y ← i ème-succ-de x dans g
         U ← ajouter({x,y,coût(x,y,g)},U)
      fin pour
   fin pour
   i ← 1                               /* Nombre d'arêtes choisies = 0 */
   /* Construction par ajouts successifs de t */
   tant que i < n faire                /* tant qu'il n'y a pas n-1 arêtes */
      {x,y,v} ← choisirmin(U)          /* choix de la nouvelle arête */
      U ← supprimer({x,y,v},M)
      rx ← trouver(x,pere)
      ry ← trouver(y,pere)
      /* vérification de l'acyclicité */
      si (rx <> ry) alors
         reunir(rx,ry,pere)
         t ← ajouterlarête <m,z> de coût v à t
         i ← i + 1                    /* une arête de plus */
      fin si
   fin tant que
Fin Algorithme procédure Kruskal
```

Exemple

Appliquons l'algorithme de kruskal au graphe non orienté valué connexe de la figure 2.

Edmonds

L'algorithme d'Edmonds a la particularité de déterminer, en partant d'un sommet racine déterminé, un ARM sur un graphe ***Orienté*** valué.

Principe

Soit $G = < S, A, C >$ un graphe orienté valué de n sommets et p arcs, on cherche alors un ARM T de racine r fixée. L'algorithme d'Edmonds va exploiter les deux définitions suivantes d'un AR de G.

1. T est un arbre recouvrant de G (si l'on oublie l'orientation)
2. Tout sommet de G possède un unique prédécesseur, sauf la racine r fixée qui n'en a pas.

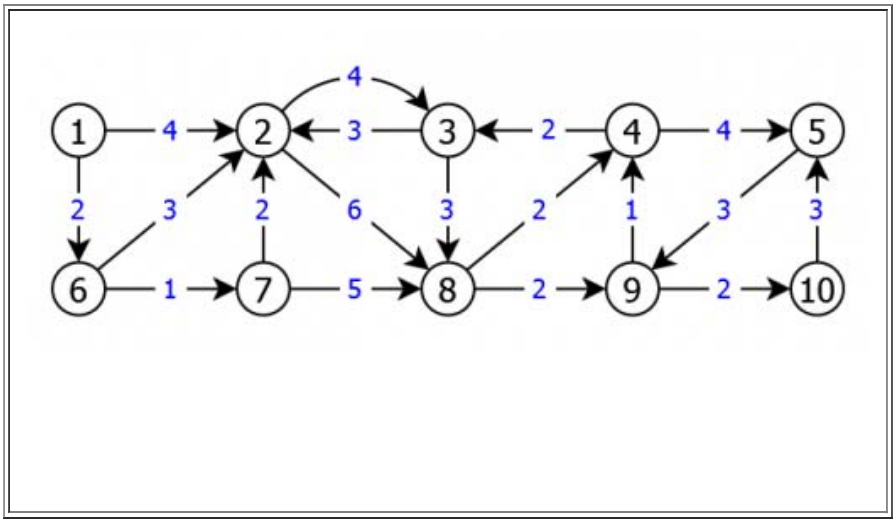
Ces définitions nous amènent à écrire une heuristique qui va construire le graphe partiel $G' = < S, A', C >$ qui a de bonnes chances d'être un arbre. L'idée est de ne conserver pour chaque sommet y de G (exception faite du sommet racine r) son prédécesseur x le plus proche (celui pour lequel l'arc entrant vers le sommet est de plus faible coût), ce qui donne :

```
Algorithme procédure heuristique_Construit_Graphe_partiel
Paramètres locaux
   entier r                          /* r est le sommet racine */
   graphe g
Paramètres globaux
   graphe g'                         /* l'arbre partiel que l'on construit */
Variables
   entier x, y, m                    /* x, y et m sont des sommets */
   réel v
Début
   g' ← graphevide                   /* Initialisation de g' */
   pour y ← 1 jusqu'à n faire
      si y<>r alors
         m ← ième-pred(1,y,g)
         v ← coût(m,y,g)
         pour i ← 2 jusqu'à d°-(y,g) faire
            x ← ième-pred(i,y,g)
            si (coût(x,y,g)<v) alors
               v ← coût(x,y,g)
               m ← x
            fin si
         fin pour
```

```
        g' ← ajouterlarête <m,y> de coût v à g'
      fin si
   fin pour
Fin Algorithme procédure heuristique_Construit_Graphe_partiel
```

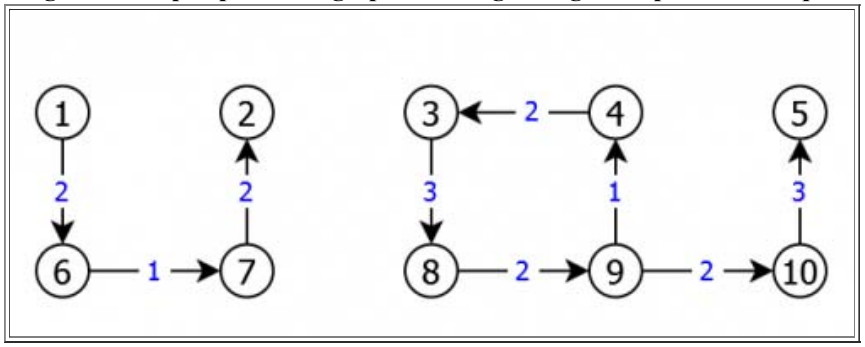
Appliquons cette heuristique sur le graphe ***G*** = < ***S, A, C*** > de la figure 3 en prenant le sommet ***1*** comme racine.

Figure 3. Un graphe orienté valué contenant un circuit et 2 composantes connexes.



Nous aurions alors le graphe partiel ***G'*** = < ***S, A', C***> de la figure 4:

Figure 4. Graphe partiel du graphe de la figure 3 généré par l'heuristique.



Une fois l'heuristique appliqué à ***G***:

- soit ***G'*** est un arbre, c'est donc aussi un ARM et c'est terminé,
- soit ***G'*** n'est pas un arbre et dans ce cas, il possède un cycle et n'est pas connexe. En effet un arbre est connexe et sans cycle avec ***n-1*** arêtes. L'heuristique a retenu une arête par sommet sauf pour la racine choisie du graphe (*le sommet **1** sur l'exemple de la figure 3*), ce qui fait ***n-1*** arêtes. Donc ***G'*** n'étant pas un arbre, il n'est pas connexe et possède au moins un cycle, comme le montre le résultat de l'application figure 4.

*Note: Si **G'** possède des cycles, ce sont aussi des circuits dans la mesure où tous les arcs sont orientés dans le même sens (sinon un sommet aurait deux prédecesseurs). On en déduit la propriété suivante:*

```
P1: Si le graphe partiel G' construit par l'heuristique n'est pas un arbre, il n'est pas connexe et contient au moins un c
```

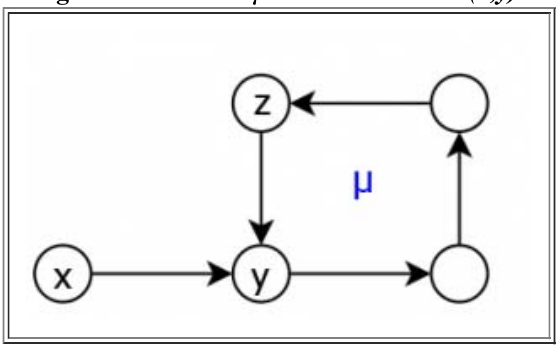
L'algorithme d'Edmonds exploite le graphe ***G'*** grace à la propriété suivante:

```
P2: Soit μ un de circuits du graphe partiel G' construit par l'heuristique, il existe un ARM T qui possède tous les arcs d
```

Cette propriété va nous permettre de contracter le circuit μ en un sommet unique appelé *pseudo_sommet* et de continuer la recherche de notre ARM sur le graphe contracté.

Appelons *arc entrant* dans μ un arc (x,y) avec le sommet y dans μ et x non. Appelons aussi z le sommet prédecesseur de y dans le circuit μ (cf. figure 5).

Figure 5. Un circuit μ et un arc entrant (x,y).



Pour un graphe $G = < S, A, C >$ orienté valué et un circuit μ, on appelle *graphe contracté* $G/\mu = G_1 = <S_1, A_1>$ le graphe obtenu en remplaçant μ par un *pseudo-sommet*. Les arcs incidents vers l'intérieur à ce pseudo-sommet sont les arcs entrants dans μ. Après contraction de μ, T devient alors un arbre T_1 sur G_1 dont le poids est lié à celui de T par la relation:

```
C(T) = C(T1) + C(μ) - C(x,y,G) avec C la fonction de Coût de G
```

Les coûts C_1 de G_1 seront alors modifiés de la façon suivante:

- $C_1(x,y,G_1) = C(x,y,G)$ si (x,y) n'est pas un arc entrant dans μ
- $C_1(x,y,G_1) = C(x,y,G)-C(z,y,G)$ si (x,y) entre dans μ avec z le prédécesseur de y dans μ

Le but de cette modification des coûts des arcs est d'avoir en final:

```
C(T) = C1(T1) + C(μ)
```

Ce qui donne la propriété suivante:

```
P3: T est un ARM de G ssi il en est de même pour sa restriction T1 dans le graphe contracté G1 = G/μ muni des coûts C1.
```

Dès lors, on peut appliquer l'heuristique sur G pour calculer G'. Si ce dernier contient des circuits, on les contracte en corrigeant les coûts des arcs entrants dedans. Cela nous donne le graphe G_1 muni des coûts C_1 sur lequel on recommence le processus (heuristique, contractions éventuelles, etc.).

On obtient alors une suite de graphes contractés G_k accompagné de leur graphe partiel G'_k et l'on n'arrête cela que lorsque le graphe partiel G'_k est un arbre T_k du graphe G_k.

Si le graphe G_k contient la contraction d'un circuit μ du graphe G_{k-1} et que G'_k (le graphe partiel) est un arbre T_k, nous savons par la propriété P_2 que cet arbre possède un unique arc entrant (x,y) dans μ. Si z est le prédécesseur de y dans le circuit μ, On peut alors reconstruire l'arbre T_{k-1} en ajoutant à l'arbre T_k tous les arcs de μ sauf (z,y).

On recommence cette opération de récupération d'arbre intermédiaire en arbre intermédiaire jusqu'à retrouver l'arbre T du graphe G d'origine, ce qui pour l'algorithme d'Edmonds donnerait la structure suivante :

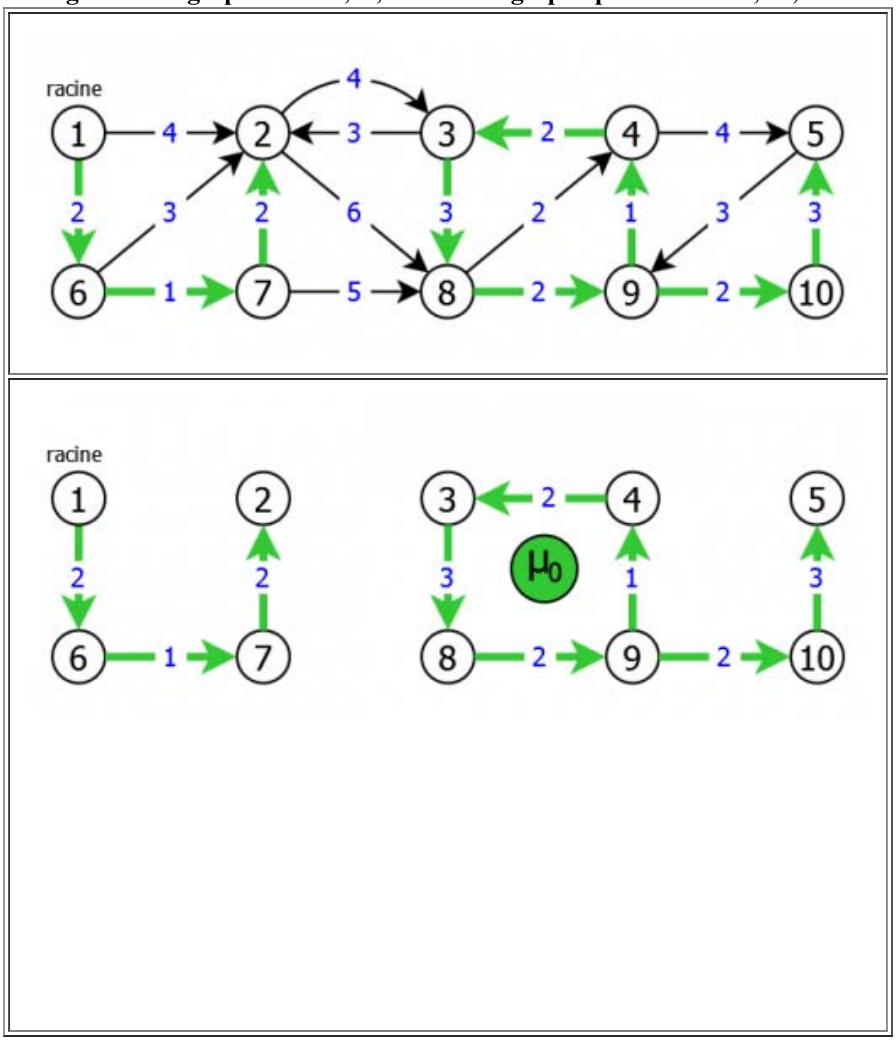
```
G' ← G
```

```
r ← ...      /* choix de r le sommet racine */
faire
   T ← heuristique_Construit_Graphe_partiel (r, G')
   pour chaque circuit µ de T faire
      G' ← contraction de G' sur µ
      Corrections des coûts des arcs entrants dans µ selon la propriété P3
   fin pour
tant que T possède un circuit.
Reconstruction de l'ARM T de G par recupérations successives des arbres intermédiaires
```

Exemple

Essayons de clarifier tout cela à l'aide d'un exemple. Prenons le graphe de la figure 6 représentant notre graphe $G=G_0$ et le graphe partiel G'_0 obtenu après passage par l'heuristique.

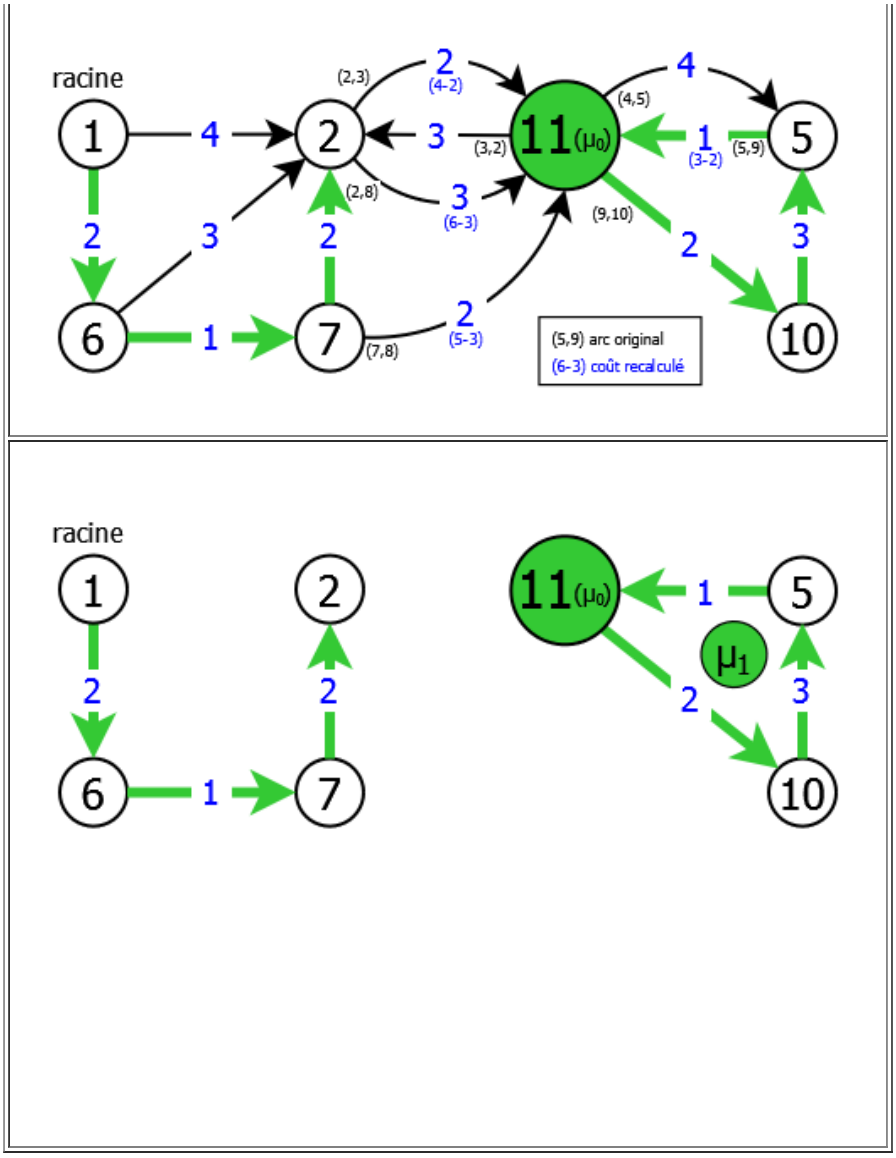
Figure 6. Le graphe G=< S, A, C > et son graphe partiel G'=< S, A', C >.



On constate alors qu'il y a deux composantes et un circuit μ_0 *(3,8,9,4,3)*. Nous allons donc contracter le graphe $G = G_0$ sur le circuit μ_0 en remplaçant ce dernier par un pseudo-sommet ***11***. Nous allons ensuite réajuster les coûts des arcs (2,3), (2,8), (7,8) et (5,9) entrants dans μ_0, et enfin relancer l'heuristique sur le graphe $G_1 = G_0/\mu_0 = < S_1, A_1, C_1 >$ obtenu, ce qui donne (cf. figure 7):

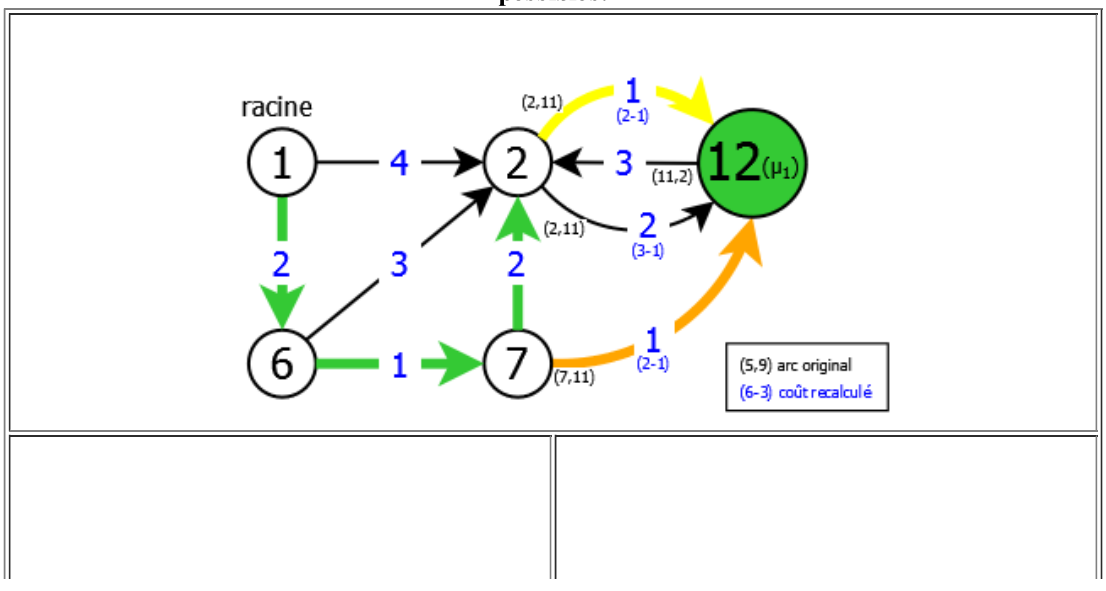
Il est à noter que la contraction d'un graphe sur un circuit peut générer des arcs multiples sur deux mêmes sommets, comme c'est le cas sur cet exemple. En effet il y deux arcs (2,11) remplaçants les arcs (2,3) et (2,8) du graphe précédent.

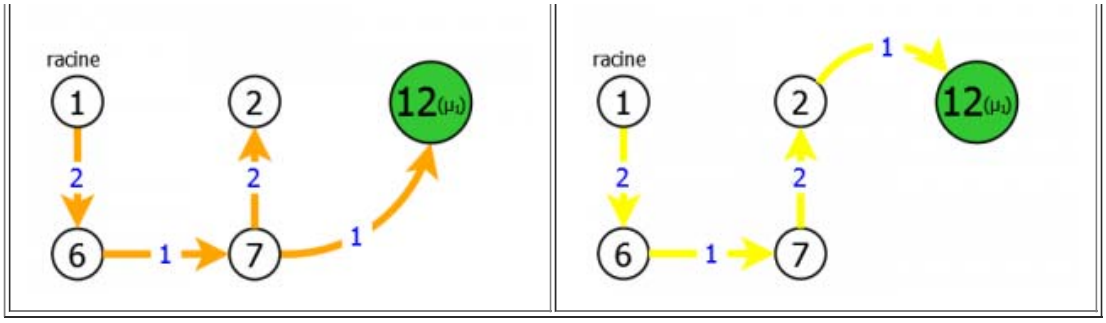
Figure 7. Le graphe $G_1 = < S_1, A_1, C_1 >$ et son graphe partiel $G'_1 =< S_1, A'_1, C_1 >$.



On constate alors qu'il y a encore deux composantes et un circuit μ_1 *(5,11,10,5)*. Nous allons là aussi contracter le graphe G_1 sur le circuit μ_1 en remplaçant ce dernier par un pseudo-sommet *12*. Nous allons enfin réajuster les coûts des arcs (2,11)(les deux) et (7,11) entrants dans μ_1, et enfin relancer l'heuristique sur le graphe $G_2 = G_1/\mu_1 = < S_2, A_2, C_2 >$ obtenu, ce qui donne (cf. figure 8):

Figure 8. Le graphe $G_2 = < S_2, A_2, C_2 >$ et ses deux graphes partiels $G'_2 = < S_2, A'_2, C_2 >$ possibles.

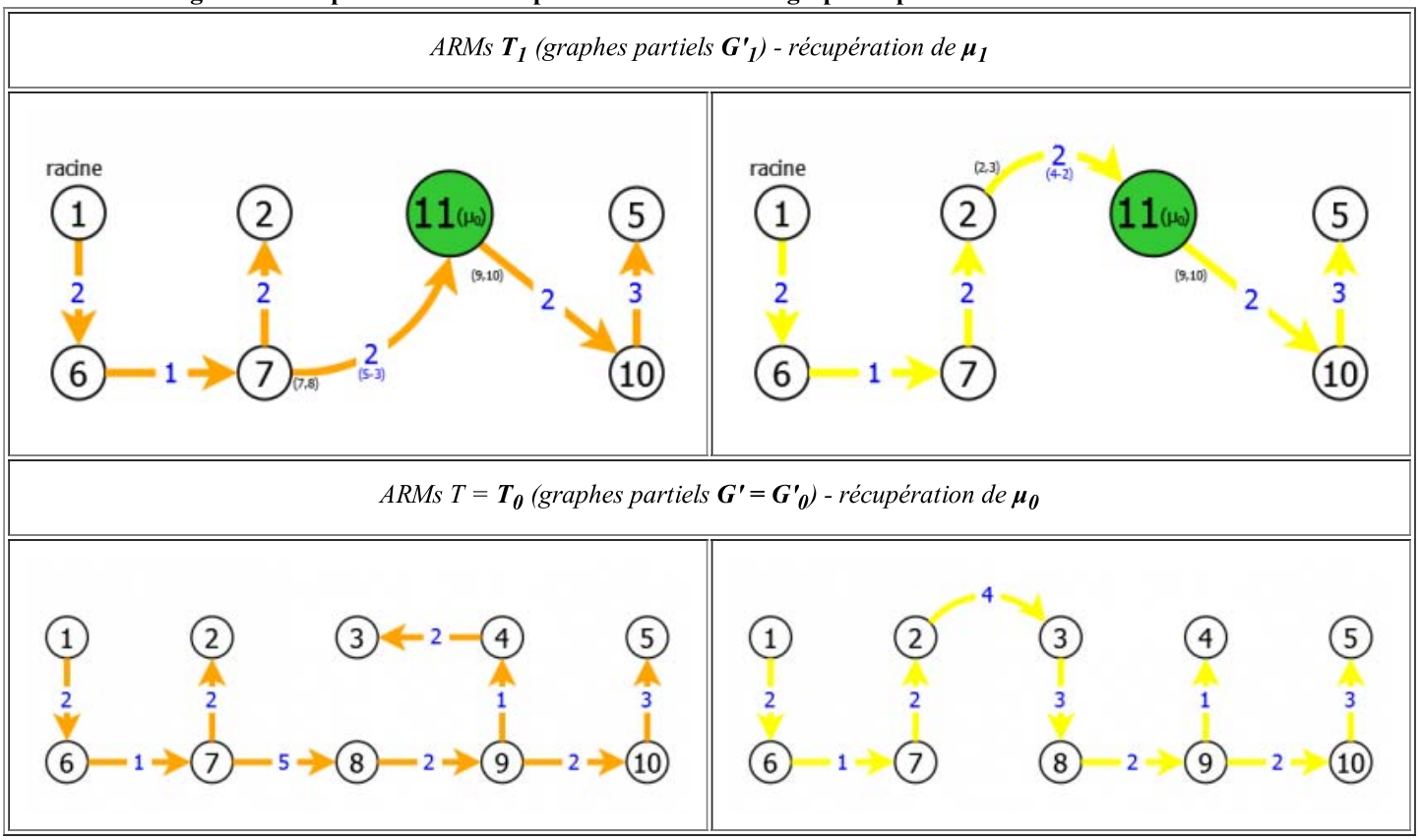




On s'aperçoit alors que le passage de l'heuristique sur le graphe G_2 peut générer deux graphes partiels distincts. En effet, sur les trois arcs incidents au pseudo-sommet ***12*** vers l'intérieur, il y en a deux qui sont de même poids: l'ancien arc (2,11) et l'ancien arc (7,11), tous deux entrants dans le circuit μ_1.

Cela étant, dans les deux cas, le graphe partiel généré G_2 est un arbre T_2. Nous n'avons donc plus qu'à Reconstruire l'ARM $T = T_0$ de $G = G_0$ par recupérations successives des arbres T_1 et T_0, comme le montre la figure 9:

Figure 9. Récupération des deux possibilités d'ARM du graphe G par reconstructions successives.



On peut donc récupérer deux ARMs différents de même poids égal à 20. En effet, les deux paires d'arcs échangables ((7,8),(4,3) et (2,3), (3,8)) ont évidemment le même poids cumulé qui est dans ce cas égal à 7 (5+2 ou 4+3).

Algorithme

Suite à venir...

(Christophe "krisboul" Boullay)

- Dernière modification de cette page le 28 mars 2012 à 13:21.