Quelques exemples de situations informatiques menant à une problématique mathématique et inversement

Malika More

(malika.more@uca.fr)

Avec l'informatique, des maths plus discrètes ?
Limoges - 26 Janvier 2018

1 Introduction

Dans cet article, j'ai choisi d'illustrer les interactions entre les mathématiques et l'informatique avec trois types de situations, qui me paraissent illustrer trois points de rencontre importants entre les deux disciplines.
Tout d'abord, je présente une situation liée à la représentation de l'information. En effet, cette notion est au cœur même de la science informatique et c'est l'occasion d'insister sur une différence notable avec les mathématiques : les nombres représentés en informatique ne sont pas identiques aux nombres manipulés en mathématiques, et il est nécessaire d'être conscient des différences. En second lieu, j'ai choisi de traiter d'une méthode de preuve incontournable en informatique, et très peu utilisée en mathématiques scolaires, qui est la recherche exhaustive. Enfin, je propose quelques exemples de preuves d'algorithmes, puisque la preuve est le cœur de l'activité mathématique, et qu'il est par conséquent important de souligner qu'il y a de jolies preuves en informatique, où une mathématicienne peut se faire plaisir.

2 Représentation de l'information

En informatique, l'un des objectifs principaux de l'enseignant me semble devoir être de faire toucher du doigts aux débutants, quel que soit leur niveau d'étude, que dans cette science, on ne manipule pas des objets, mais seulement des représentations de ces objets.
Du point de vue numérique par exemple, il est essentiel de comprendre la différence entre un réel et un flottant, l'objet informatique correspondant. À ce sujet, je renvoie au compte-rendu de l'atelier d'Emmanuel Beffara et Guillaume François.
Pour ma part, je propose un court exercice lié à la cryptographie qui permet à mes étudiants de première année de DUT Informatique de (re)découvrir le sens de la représentation en base dix des entiers.

Autour de la cryptographie Les protocoles cryptographiques fonctionnent le plus souvent selon le schéma général ci-dessous :

— Le message clair est représenté par une suite d'entiers de taille fixée :

1161 1411 7099

— Ces nombres sont chiffrés par l'expéditeur et le message obtenu est envoyés.
— Le destinataire reçoit le message et le déchiffre.
— Si tout va bien, la suite d'entiers obtenue est identique à celle composant le message clair :

1161 1411 7099

Bien entendu, avant et après l'application d'un protocole cryptographique, une étape de transformation d'un texte en une suite d'entiers et réciproquement est nécessaire pour pouvoir échanger des « messages secrets » textuels. Une façon simplifiée de présenter cette étape est la suivante :

— À partir d'un texte :

t r u c

— Chaque caractère est remplacé par un code numérique (ici le code ASCII) :

116 114 117 99

— Ces codes sont collés les uns aux autres :

116114117099

— Et le résultat est découpé en morceaux de taille adaptée au calcul cryptographique particulier qu'on souhaite effectuer (ici des nombres de 4 chiffres) :

1161 1411 7099

Ces opérations sont exécutées dans l'ordre inverse après l'étape de déchiffrement pour retrouver le texte.

Des opérations arithmétiques simples, basées sur la représentation en base dix des entiers permettent de coller des nombres les uns aux autres et de découper un nombre en paquets de chiffres de longueur donnée :

$$117 \times 10^3 + 99 = 117099$$

$$116114117099 = 11611411 \times 10^4 + 7099$$

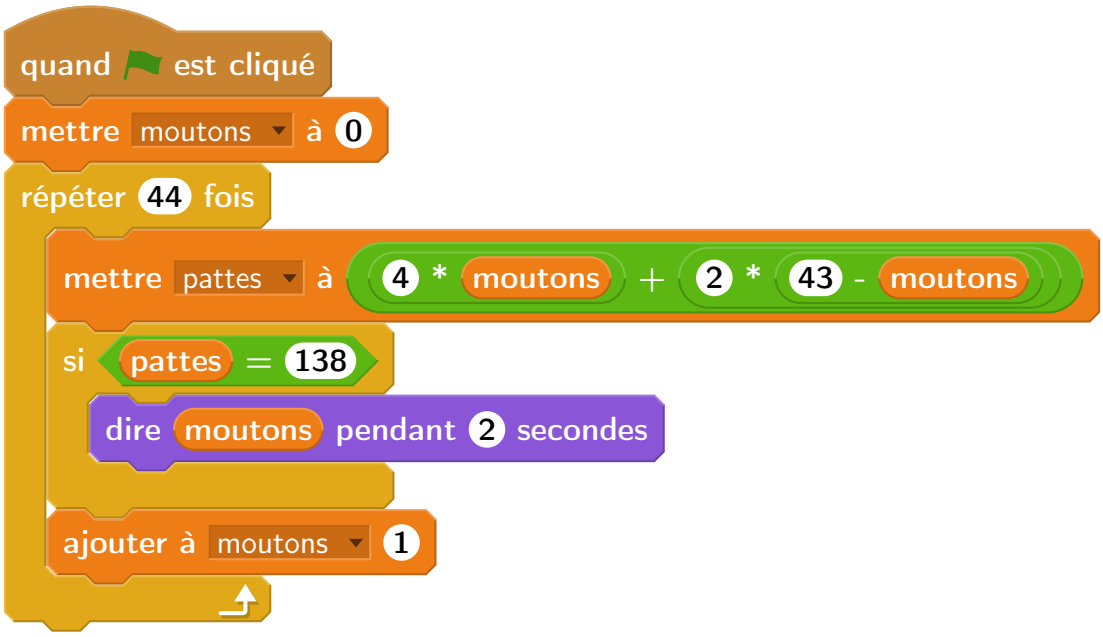
Une alternative informatique est de traiter les nombres comme des chaînes de caractères, en utilisant la concaténation et l'extraction de sous-chaînes à la place des calculs arithmétiques, mais aucun étudiant ne le fait. Peut-être pensent-ils que seule l'utilisation d'outils mathématiques est autorisée en cours de mathématiques.

3 Recherche exhaustive

Dans cette partie, je plaide pour une large utilisation d'exercices mettant en jeu une recherche exhaustive. Pour commencer, j'explique pourquoi cela me semble judicieux, et ensuite je présente un exemple qui permet d'illustrer l'ensemble des arguments présentés précedemment.

3.1 La force brutale

Voici d'abord un exercice de collège, pour lequel une résolution algébrique est en général attendue en cours de mathématiques : « Il y a dans un champ des moutons et des oies. Je compte 43 têtes et 138 pattes. Quel est le nombre de moutons ? » Mais, puisque des éléments d'algorithmique et de programmation sont maintenant enseignés à partir du Cycle 4 au collège, il est envisageable que certains élèves en viennent à proposer une résolution algorithmique :



Le programme proposé ci-dessus ne prétend pas être un modèle, il s'agit simplement d'un exemple illustrant une résolution par exploration exhaustive.

De même, au lycée, il est maintenant envisageable de proposer un exercice du type suivant : « Résoudre dans $\mathbb{N}^3$ l'équation $a^2+b^2+c^2=26853$ avec $a \leq b \leq c$. ». Dans ce cas, l'élève n'a pas d'autre possibilité de résolution que de tenter une exploration exhaustive de l'ensemble des triplets potentiellement solutions. Après une analyse mathématique du problème, on pourrait ainsi obtenir par exemple comme réponse à l'exercice un programme `Python` tel que :

```
N=26853
R=math.ceil(math.sqrt(N))
for a in range(R) :
   for b in range(a,R) :
      for c in range(b,R) :
         M=a*a+b*b+c*c
         if M==N :
            print(a,b,c)
```

Ainsi, dans le cadre de l'introduction d'une part d'algorithmique et de programmation dans les enseignements de mathématiques au collège et au lycée, un objectif raisonnable pourrait être qu'une situation comme décrite ci-dessous devienne banale pour les élèves, en adaptant bien entendu le contexte et le niveau de difficulté selon la classe.

> Étudier une propriété $P(x)$ pour x appartenant à un ensemble fini E :
> — Existence, universalité, comptage, énumération...
> — Reconnaître cette situation, formaliser le problème et le résoudre en mettant en œuvre une **recherche exhaustive**

Bien entendu, il y a au moins deux conditions nécessaires à cela :

— que les élèves aient l'occasion de pratiquer souvent cette approche sur de nombreux objets qui s'y prêtent naturellement (maths discrètes en général, arithmétique, graphes, etc.) ;

— qu'il soit clair que c'est une méthode de preuve légitime en mathématiques et naturelle dans un monde où il existe des ordinateurs pour faire les calculs à notre place.

La raison pour laquelle je défends ce type d'exercices est qu'ils me semblent mettre en jeu de nombreux éléments d'une culture de base en informatique :

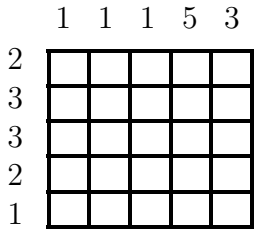
— représentation numérique de l'information : très simple (nombres entiers) ou pas (voir paragraphe suivant) ;
— algorithmique et programmation : itération, condition et variable-compteur ;
— limites de la recherche exhaustive : explosion combinatoire et complexité.

Pour finir, je tiens à souligner que les notions mathématiques ne sont pas absentes : il s'agit bien sûr principalement de mathématiques discrètes, dans lesquelles les aspects combinatoires et énumératifs jouent un rôle important, et des éléments de raisonnement qui vont avec.

3.2 Exemple d'utilisation

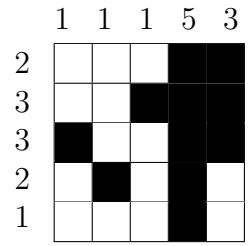
La tomographie est un ensemble de techniques qui ont pour but de reconstruire un objet en volume de façon non destructive, à partir de mesures donnant l'épaisseur des différentes tranches de cet objet selon une direction donnée, et à partir de plusieurs directions d'observation. La tomographie est utilisée en particulier en imagerie médicale.

Nous allons nous intéresser dans ce paragraphe à un cas très particulier, la **tomographie discrète en** $2D$. Dans cette version simplifiée, on considère que l'objet à reconstruire est une image composée de pixels noirs ou blancs, par exemple de taille 5×5. La couleur exacte de chaque pixel est inconnue, mais on connaît, pour chaque ligne et pour chaque colonne, le nombre de pixels noirs.
Ainsi, une instance du problème est par exemple :

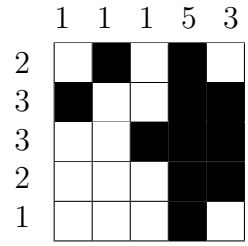


Ce qui signifie qu'il y a 2 pixels noirs sur la première ligne, 3 sur la deuxième, etc. Et qu'il y a 1 pixel noir sur la première colonne, ainsi que sur la deuxième et la troisième, etc.
Diverses questions peuvent être posées : existence d'une solution, calcul d'une solution, nombre de solutions, énumération de toutes les solutions, etc.

Dans notre exemple, une solution possible (c'est-à-dire une image satisfaisant toutes les contraintes) est :

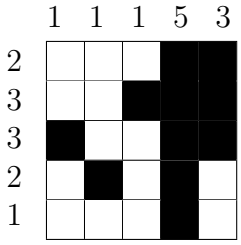


Voici une autre solution possible :



Je propose ici une résolution de cet exercice en plusieurs étapes, chacune intéressante en elle-même : modéliser le problème, énumérer les candidats solutions, mettre en œuvre une recherche exhaustive et enfin expérimenter les limites de la méthode utilisée.

Modéliser L'étape de modélisation consiste tout d'abord à remplacer chaque pixel blanc par un 0 et chaque pixel noir par un 1, puis à mettre bout à bout les lignes de bits obtenues, par exemple en plaçant la ligne du haut à gauche. On obtient ainsi une suite de 25 bits, qu'on peut voir comme un nombre écrit en binaire.



	1	1	1	5	3
2	0	0	0	1	1
3	0	0	1	1	1
3	1	0	0	1	1
2	0	1	0	1	0
1	0	0	0	1	0

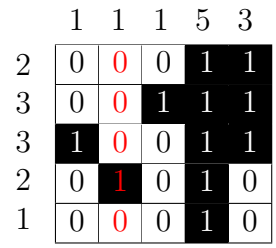
0001100111100110101000010

Ensuite, à partir de cette suite de bits, il est nécessaire de pouvoir extraire ceux qui correspondent à une ligne ou à une colonne donnée.
Concernant les lignes, il s'agit de bits consécutifs. En numérotant les lignes de 1 (en haut) à 5, la ligne i correspond aux bits de poids $2^{5\times(5-i)+4}$ (à gauche) à $2^{5\times(5-i)}$ (à droite).

	1	1	1	5	3
2	0	0	0	1	1
3	0	0	1	1	1
3	1	0	0	1	1
2	0	1	0	1	0
1	0	0	0	1	0

0001100111100110101000010

Pour les colonnes, le calcul est un peu plus compliqué puisque les bits correspondants ne sont pas consécutifs dans le nombre obtenu, mais séparés par 4 autres bits. Ainsi, en numérotant les colonnes de 1 (à gauche) à 5, la colonne j correspond aux bits de poids $2^{24-(j-1)}$ (en haut), $2^{24-(j-1)-5}$, $2^{24-(j-1)-10}$, $2^{24-(j-1)-15}$ et $2^{24-(j-1)-20}$ (en bas).

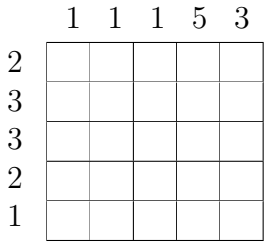


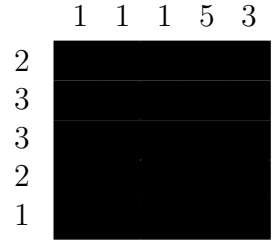
0001100111100110101000010

0	$(1,2)$	23
0	$(2,2)$	18
0	$(3,2)$	13
1	$(4,2)$	8
0	$(5,2)$	3

De la même façon, il est bien sûr possible, mais pas nécessaire pour notre objectif, de calculer la position du bit qui correspond à un pixel (i, j) quelconque de l'image.

Énumérer Avec la modélisation choisie, énumérer les candidats solutions revient à énumérer tous les nombres qu'on peut écrire avec 25 bits, c'est à dire tous les entiers entre 0 et $2^{25} - 1$. On remarque que 0 correspond à une image ne contenant aucun pixel noir et $2^{25} - 1$ à une image ne contenant que des pixels noirs.





00000000000000000000000000 11111111111111111111111

0 $2^{25} - 1$

Mettre en œuvre La mise en œuvre d'un algorithme de recherche exhaustive des solutions nécessite encore de compter, pour un candidat et un numéro de ligne (ou de colonne) donnés, le nombre de pixels noirs correspondants, c'est à dire le nombre de bits valant 1 sur les positions correspondantes du nombre. Je ne détaille pas cette partie, les fonctions `CpteLigne` et `CpteColonne` sont supposées faire le travail. Nous obtenons à la fin un algorithme du type de celui ci-dessous, que j'ai rédigé en syntaxe `Python` (j'ai choisi de compter le nombre de solutions).

```
Lignes = [2,3,3,2,1]
Colonnes = [1,1,1,5,3]
NombreSolutions=0
for Candidat in range(2**25) :
    Test = True
    for i in range(5) :
      if CpteLigne(i+1,Candidat) != Lignes[i] :
        Test = False
      if CpteColonne(i+1,Candidat) != Colonnes[i] :
        Test = False
    if Test :
       NombreSolutions=NombreSolutions + 1
print(NombreSolutions)
```

Limites Finalement, l'un des avantages du problème de la tomographie discrète en $2D$ est qu'il permet facilement d'expérimenter les limites d'une recherche exhaustive, en augmentant légèrement la taille de l'image à reconstruire. Imaginons par exemple que le test d'un candidat solution prenne une microseconde (10^{-6} seconde) :

- Image $5 \times 5 \rightsquigarrow 25$ pixels $\rightsquigarrow 2^{25} \sim 10^7$ candidats à tester $\rightsquigarrow 10$ secondes
- Image $6 \times 6 \rightsquigarrow 36$ pixels $\rightsquigarrow 2^{36} \sim 10^{10}$ candidats à tester $\rightsquigarrow 2,7$ heures
- Image $7 \times 7 \rightsquigarrow 49$ pixels $\rightsquigarrow 2^{49} \sim 10^{14}$ candidats à tester $\rightsquigarrow 3,2$ ans
- Dans la réalité, une image de petite taille comprend plusieurs centaines de pixels en largeur comme en hauteur, et une image avec une bonne définition, plusieurs milliers...

4 Preuves d'algorithmes

Face à un algorithme, trois questions se posent en cascade :

- Est-ce-qu'il donne un résultat ou bien est-ce-qu'il ne s'arrête jamais ?
  C'est la question de la **terminaison** de l'algorithme. En effet, une condition nécessaire pour qu'un algorithme réponde à un problème est que, pour n'importe quelle instance du problème, le calcul se termine en un temps fini et donne une réponse.
- Est-ce-qu'il donne le résultat attendu ou bien est-ce-qu'il calcule n'importe quoi ?
  On parle ici de la **correction** de l'algorithme. Il est très souhaitable de pouvoir donner une preuve que l'algorithme proposé répond bien au problème posé. Pour dépasser une « preuve par l'exemple », qui n'en est évidemment pas une, il y a bien souvent une véritable difficulté.
- Est-ce-qu'il donne le résultat en un temps raisonnable ou bien est-ce-qu'il faut attendre plusieurs siècles ?
  Répondre à cette question, c'est analyser la **complexité** de l'algorithme. C'est une question importante, mais je ne l'aborderai pas ici.

Dans ce paragraphe, je montre deux exemples de preuves d'algorithmes. D'abord, le cas ultra-classique de l'algorithme d'Euclide permet d'introduire la notion de convergent et celle d'invariant de boucle. Ce sont deux outils qui permettent dans certains cas de prouver respectivement la terminaison et la correction d'un algorithme. Ensuite, je présente l'algorithme de Prim, qui permet de déterminer un arbre recouvrant de poids minimal pour un graphe valué connexe. La démonstration de la terminaison et de la correction de cet algorithme amènent une illustration un peu plus évoluée de l'utilisation d'un convergent et d'un invariant de boucle.

4.1 Algorithme d'Euclide

L'algorithme d'Euclide ci-dessous permet de calculer le pgcd de deux entiers a et b donnés en entrée :

début
  Donner à x la valeur a
  Donner à y la valeur b
  tant que $y \neq 0$ **faire**
    Donner à *temp* la valeur y
    Donner à y la valeur $x \mod y$
    Donner à x la valeur *temp*
  fin
  retourner : x
fin

On présente rapidement ci-dessous les preuves de terminaison et de correction de l'algorithme d'Euclide.

Cet algorithme se termine si la condition d'arrêt de la boucle se réalise, c'est-à-dire si y s'annule. Dans la boucle, y est remplacé par un reste $\mod y$. Ainsi, à chaque passage, l'entier y décroît strictement, tout en restant ≥ 0. Par conséquent, y finit par atteindre 0, on sort de la boucle et l'algorithme se termine.

Cet algorithme est supposé calculer $\text{pgcd}(a, b)$. Au départ, on a $x = a$ et $y = b$. Dans la boucle, on remplace (x, y) par $(y, x \mod y)$. Or on prouve que $\text{pgcd}(x, y) = \text{pgcd}(y, x \mod y)$. Donc à chaque étape, on a $\text{pgcd}(x, y) = \text{pgcd}(a, b)$. En sortie de boucle, on a $y = 0$, on renvoie x, et on sait que $\text{pgcd}(x, 0) = x$. Par conséquent, l'algorithme calcule bien $\text{pgcd}(a, b)$.

Outil pour la terminaison

Définition 1. On appelle convergent une quantité qui prend ses valeurs dans un ensemble bien fondé et qui diminue strictement à chaque passage dans une boucle.

Remarques.
- *Un ensemble bien fondé est un ensemble totalement ordonné dans lequel il n'existe pas de suite infinie strictement décroissante.*
- *En particulier, $\mathbb{N}$, ou $\mathbb{N}^k$ muni de l'ordre lexicographique, sont des ensembles bien fondés.*
- *On a $(a, b) < (c, d)$ pour l'ordre lexicographique lorsque $a < c$ ou ($a = c$ et $b < d$).*

Propriété. *L'existence d'un convergent pour une boucle garantit que lors de l'exécution de l'algorithme, on finit par sortir de cette boucle.*

Dans le cas de l'algorithme d'Euclide, nous avons vu ci-dessus que y est un convergent.

Outil pour la correction

Définition 2. On appelle invariant de boucle une propriété qui, si elle est vraie avant l'entrée dans une boucle, reste vraie après chaque passage dans cette boucle, et donc est vraie aussi à la sortie de cette boucle.

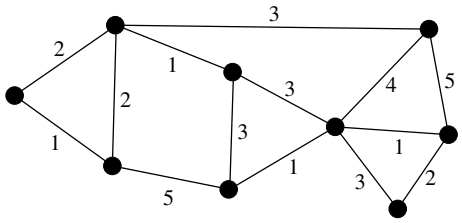
Remarque. Analogie évidente avec une preuve par récurrence :
- Entrée de boucle $\longrightarrow$ initialisation
- Passage dans la boucle $\longrightarrow$ hérédité
- Sortie de boucle $\longrightarrow$ conclusion

Propriété. *La mise en évidence d'un invariant de boucle adapté permet de prouver la correction d'un algorithme.*

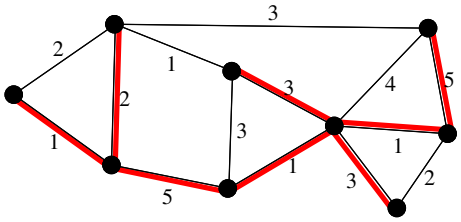
Dans le cas de l'algorithme d'Euclide, nous avons vu ci-dessus que $pgcd(x, y) = pgcd(a, b)$ est un invariant de boucle.

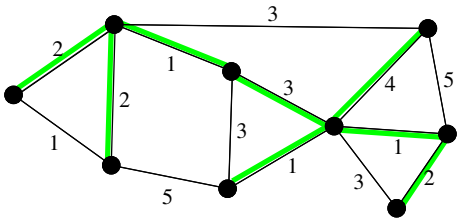
4.2 Algorithme de Prim

À titre d'illustration, je propose maintenant de démontrer la terminaison et la correction d'un algorithme un peu plus compliqué, appelé algorithme de Prim. Il s'agit d'un algorithme classique de théorie des graphes, utilisé pour calculer un arbre recouvrant de poids minimal d'un graphe valué connexe donné en entrée, comme celui ci-dessous.



Un graphe est connexe lorsque deux sommets quelconques sont toujours reliés par au moins une chaîne d'arêtes. Un cycle est une chaîne d'arêtes fermée. Un graphe est valué lorsque chaque arête possède un poids réel, qui peut être interprété comme un temps de parcours, un distance, un coût, etc. selon les applications envisagées.
Un arbre recouvrant d'un graphe connexe est un graphe connexe, sans cycle, dont les sommets sont tous ceux du graphe initial et dont les arêtes sont certaines de celles du graphe initial. Le poids d'un arbre recouvrant est la somme des poids des arêtes qui le composent. Par exemple, voici ci-dessous à gauche en rouge un arbre recouvrant de poids 21 et à droite en vert un arbre recouvrant de poids 16 de notre graphe.





Le problème qui nous intéresse dans ce paragraphe est le suivant :

Problème
Instance : un graphe valué connexe
Question : un arbre recouvrant de poids total le plus petit possible

Pour notre graphe, nous connaissons un arbre recouvrant de poids 21 et un arbre recouvrant de poids 16, la question est donc : peut-on faire mieux ? Bien entendu, ce qui est attendu comme réponse au problème, c'est un algorithme générique qui permet de calculer un arbre recouvrant de poids minimal pour n'importe quel graphe valué donné en entrée. Il en existe plusieurs. Les deux principaux qui sont enseignés aux étudiants en algorithmique sont l'algorithme de Kruskal et l'algorithme de Prim. Nous nous intéressons à ce dernier.

L'idée générale de l'algorithme de Prim est la suivante :
- On part d'un arbre initial réduit à un seul sommet du graphe.
- À chaque itération, on agrandit l'arbre en lui ajoutant un sommet accessible à l'aide d'une arête de poids le plus petit possible.
- On stoppe quand l'arbre est recouvrant.

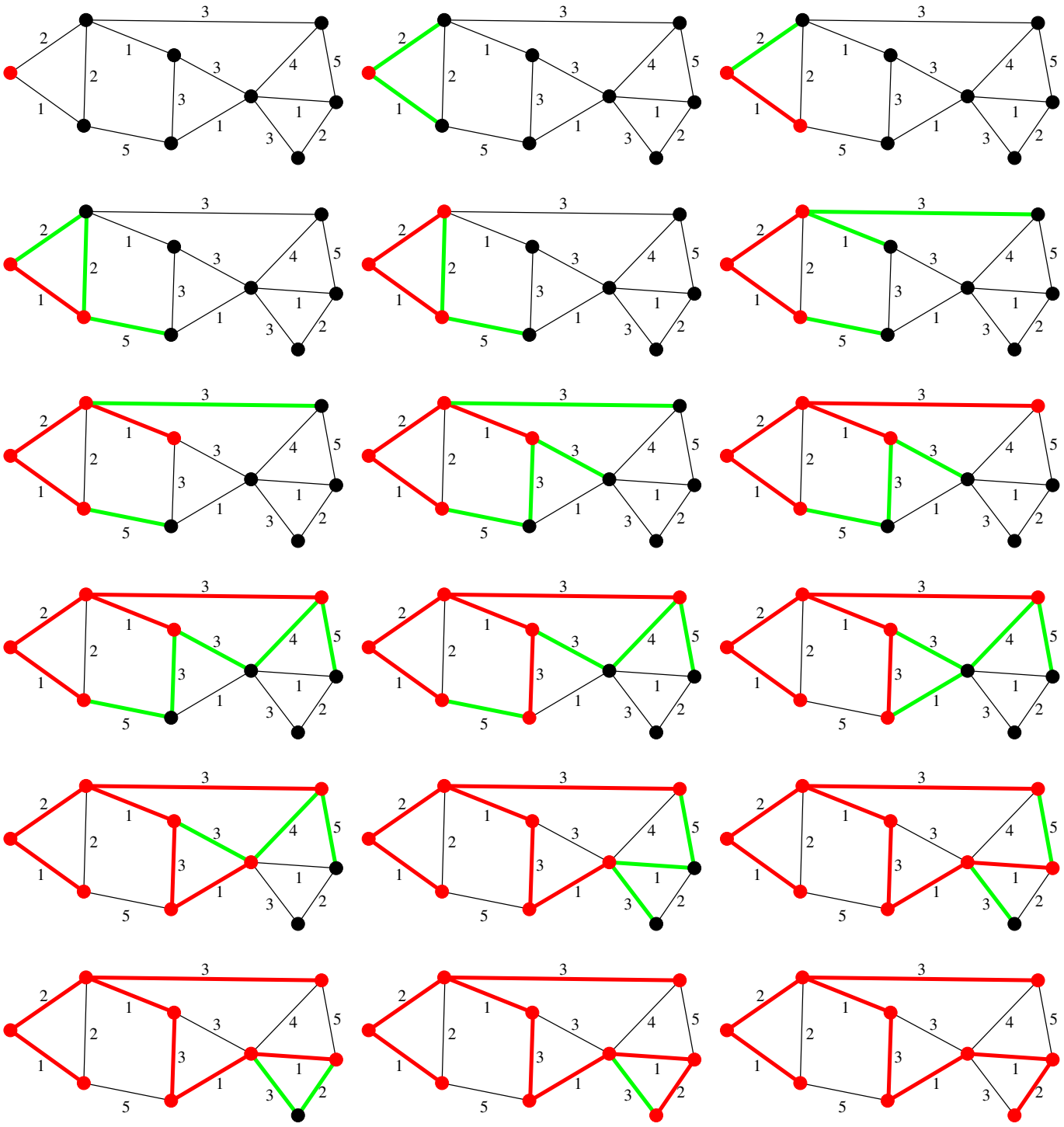
Voici une description un peu plus précise de cet algorithme. On appelle G le graphe initial et T l'arbre que construit peu à peu l'algorithme. Pour le programmer effectivement, par exemple dans le langage `Python`, il faudrait avant tout décider de la façon de représenter un graphe, puis préciser, en fonction de cela, comment effectuer chacune des opérations décrites ci-dessous. Pour plus de détails techniques, nous renvoyons le lecteur aux nombreuses implémentations qu'on peut trouver sur internet.

Algorithme de Prim

- Initialiser T avec $\begin{cases} \text{sommets : un sommet de } G \text{ qu'on choisit} \\ \text{arêtes : aucune} \end{cases}$
- Répéter :
  - Trouver toutes les arêtes de G qui relient un sommet de T et un sommet extérieur à T
  - Parmi celles-ci, choisir une arête de poids le plus petit possible
  - Ajouter à T cette arête et le sommet correspondant
- S'arrêter dès que tous les sommets de G sont dans T
- Retourner T

Il s'agit d'un algorithme de type « glouton », c'est-à-dire qu'il construit progressivement une solution optimale en choisissant à chaque instant une solution optimale localement. Dans tous les algorithmes de ce type, il est important de prouver que la solution finalement obtenue est globalement optimale.

Voici une exécution possible sur le graphe exemple, à lire de gauche à droite puis de haut en bas. L'arbre recouvrant construit est en rouge, les sommets accessibles sont ceux qui se trouvent à l'extrémité d'une arête verte. La façon de « choisir une arête » n'étant pas spécifiée, il y a souvent plusieurs exécutions posssibles, qui vont donner des arbres différents, mais de même poids.



On obtient un arbre recouvrant de poids total 14. Il faut maintenant une preuve pour démontrer que l'algorithme de Prim calcule toujours un arbre recouvrant de poids minimal.

Remarquons d'abord que **l'algorithme est bien défini** car si la condition d'arrêt «*Tous les sommets de G sont dans T*» n'est pas réalisée, alors il existe au moins une arête qui relie un sommet extérieur à T à un sommet de T (car G est connexe). Ce qui signifie qu'il sera possible d'ajouter une arête dans la boucle.

Passons maintenant à la preuve de terminaison. **L'algorithme se termine** car on ajoute à chaque passage un sommet et une arête, donc au bout d'un nombre fini d'itérations (égal au nombre de sommets du graphe initial), la condition d'arrêt «*Tous les sommets de G sont dans*

T» est réalisée. Le convergent utilisé ici est « *le nombre de sommets de G qui ne sont pas des sommets de T* ».

La **preuve de correction** se décompose en plusieurs points.

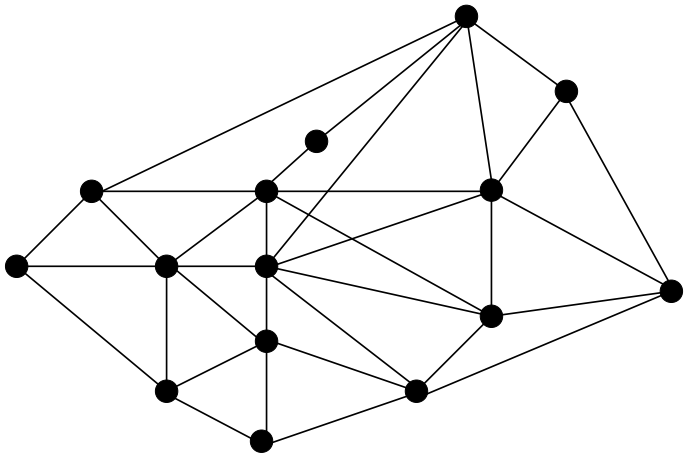
Pour commencer, on vérifie que **l'algorithme renvoie un arbre recouvrant**. En effet, au départ, T est réduit à un sommet, c'est donc un arbre. À chaque passage dans la boucle, on ajoute une arête et une feuille (une feuille d'un arbre est un sommet qui n'a qu'un seul sommet voisin) à l'arbre T, donc l'objet construit est encore connexe et sans boucle, c'est-à-dire qu'il demeure encore un arbre. Finalement, quand on sort de la boucle on retourne T qui est bien un arbre. Comme l'arbre T obtenu à l'issue de l'algorithme a pour sommets tous les sommets de G et pour arêtes certaines arêtes de G, c'est bien un arbre recouvrant de G.

Il reste à vérifier que **l'arbre recouvrant T est de poids minimal**, ce qui est la partie la plus délicate de la preuve. Pour cela, on va utiliser un invariant de boucle :

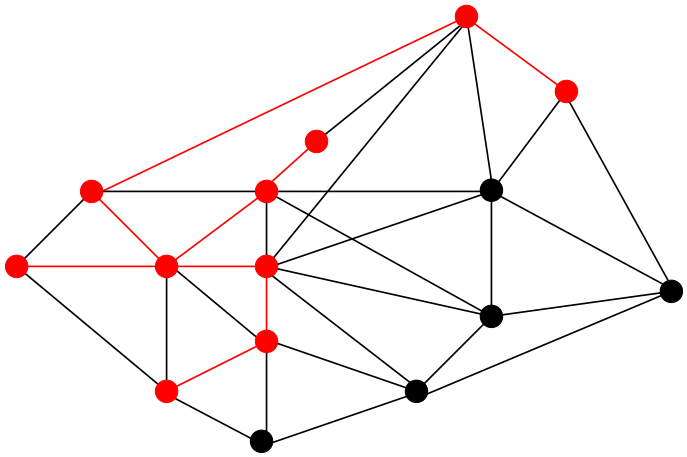
« *L'ensemble des arêtes choisies par l'algorithme de Prim est contenu dans un arbre recouvrant de G de poids minimal.* »

Initialement, l'ensemble des arêtes choisies par l'agorithme de Prim est vide, donc la propriété est trivialement vérifiée.

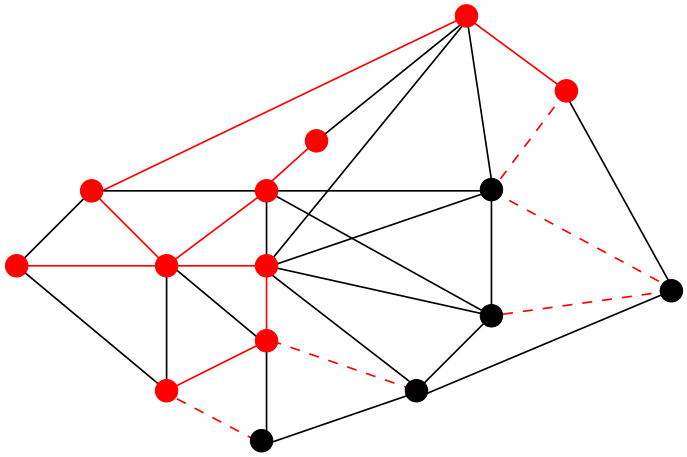
Nous allons maintenant vérifier l'hérédité de la propriété, en illustrant le raisonnement sur le graphe ci-contre. Pour simplifier, je n'ai pas fait figurer de poids sur les arêtes.



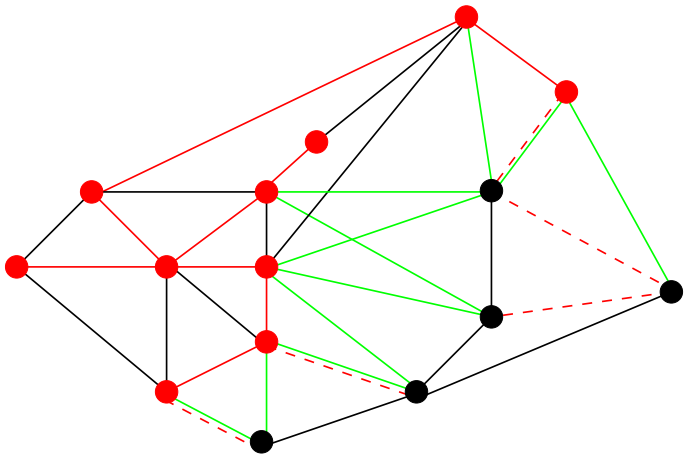
Voici l'arbre T construit jusqu'ici par l'algorithme de Prim (en rouge).



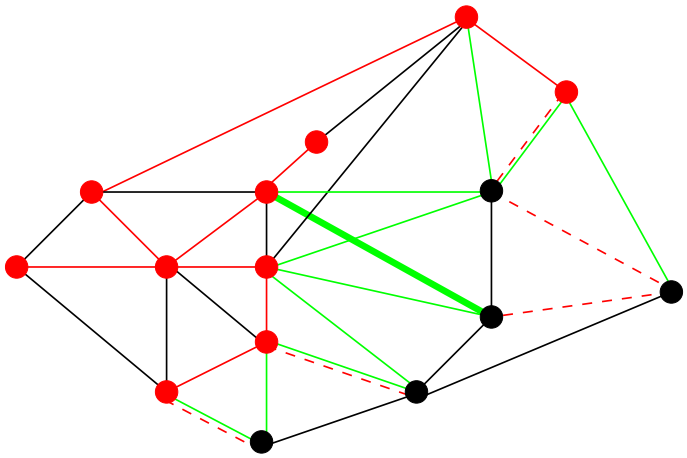
Par hypothèse, cet arbre T est contenu dans un arbre recouvrant de poids minimal de G, que nous appellerons T_0 et dont j'ai figuré les arêtes qui ne font pas partie de T en rouge pointillé. Ces arêtes permettent d'atteindre tous les sommets de G extérieurs à T (en noir), puisque T_0 est un arbre recouvrant de G.



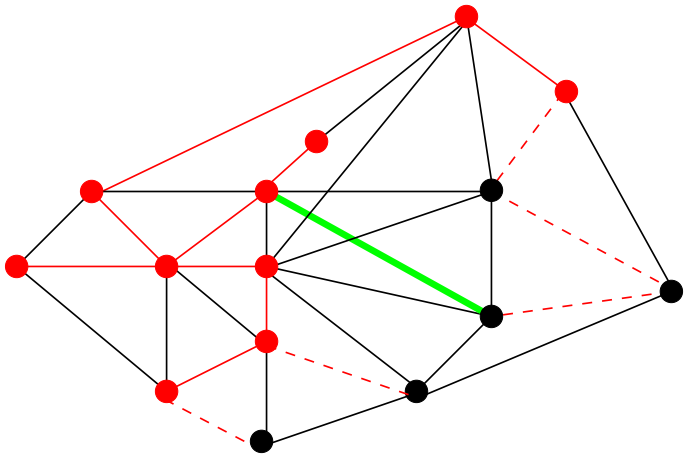
L'étape suivante de l'algorithme de Prim consiste à choisir une des arêtes reliant un sommet de T à un sommet extérieur à T. J'ai fait figurer toutes ces arêtes en vert sur le schéma suivant.



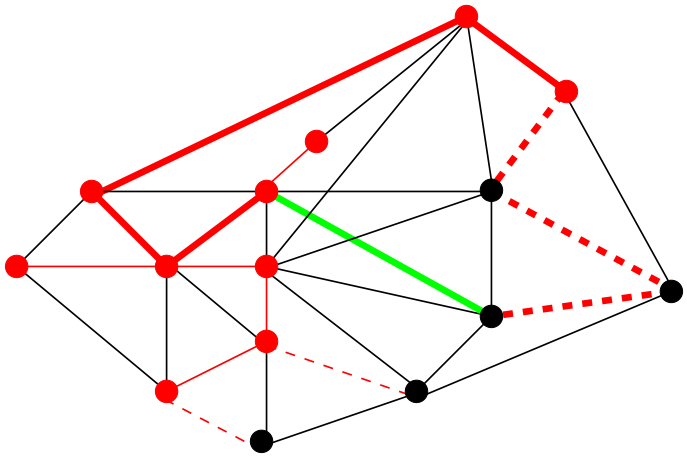
Et parmi les arêtes vertes, j'ai indiqué en gras celle que l'algorithme choisit effectivement, que nous appellerons e. Appelons T' l'arbre obtenu en ajoutant à T cette nouvelle arête e et le sommet noir à son extrémité. Il s'agit maintenant de montrer que T' est encore contenu dans un arbre recouvrant de poids minimal de G.



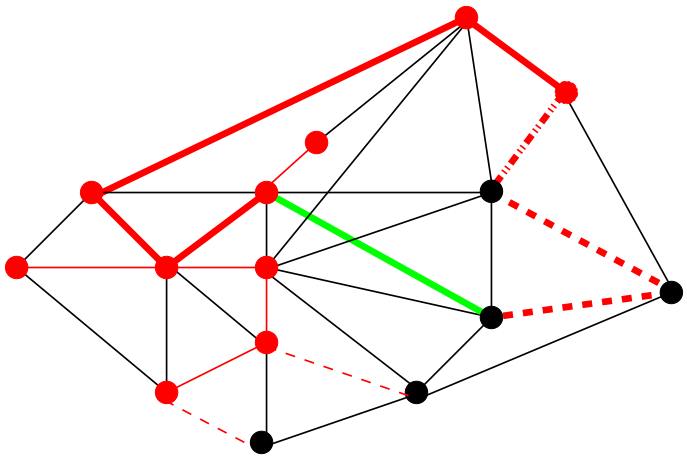
Si la nouvelle arête e est aussi une arête (rouge pointillée) de l'arbre recouvrant de poids minimal T_0 contenant T, la propriété est conservée : le nouvel arbre T' est bien contenu dans un arbre recouvrant de poids minimal de G. Dans le cas contraire, que j'ai illustré sur le schéma ci-contre, il reste à montrer que T' (arêtes rouges en traits pleins + arête e verte en gras) est contenu dans un autre arbre recouvrant de poids minimal de G.



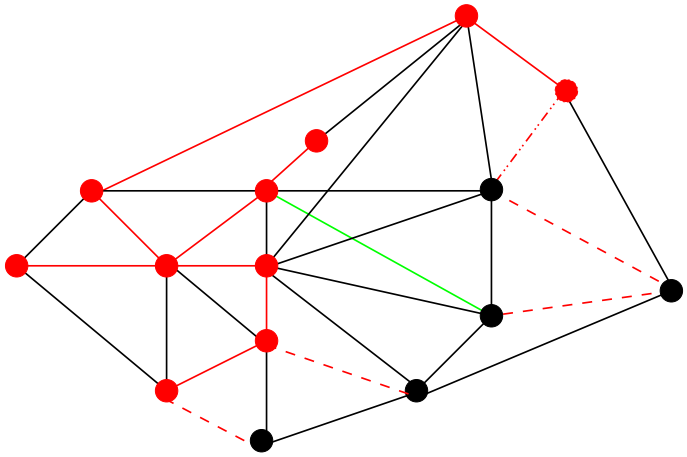
Pour cela, je considère les deux extrémités de la nouvelle arête e : l'un des sommets (en rouge) appartient à T et pas l'autre (en noir). Ces deux sommets sont aussi des sommets de T_0, et comme ce dernier est un arbre, il existe une unique chaîne d'arêtes de T_0 reliant ces deux sommets, que nous appellerons C. J'ai représenté cette chaîne d'arêtes C en gras sur le schéma ci-contre. Par hypothèse, la nouvelle arête e n'appartient pas à T_0 et donc ne fait pas partie de cette chaîne C.



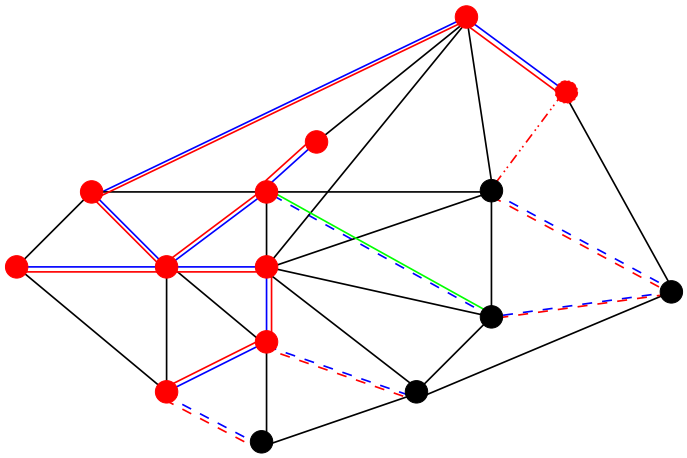
Les deux extrémités de la chaîne d'arêtes rouges C sont de couleurs différentes, puisque ce sont celles de e (l'une est dans T et l'autre pas). Les arêtes en traits pleins de C (i.e. celles qui appartiennent à T) forment une sous-chaîne de C car T est un sous-arbre de T_0. Il existe donc une unique arête de C qui a pour extrémités le dernier sommet de T et le premier sommet extérieur à T. Elle est représentée sur le schéma ci-contre avec des pointillés différents, et nous l'appellerons e'. Notons que $e' \neq e$, mais que, comme e, les extrémités de e' sont de couleurs différentes.



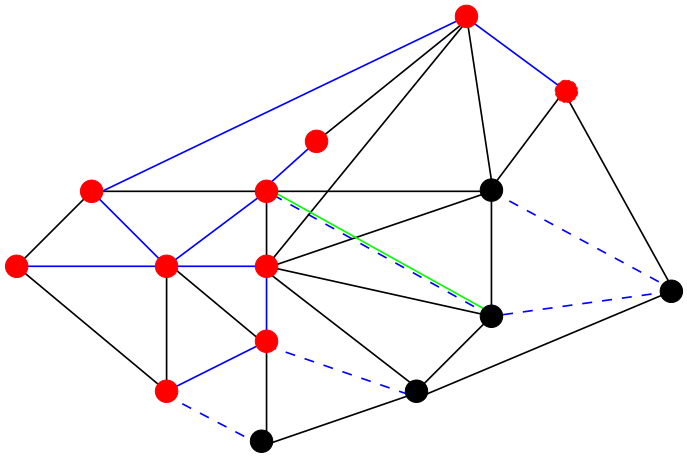
Nous avons maintenant tous les ingrédients nécessaires pour terminer la preuve : G, T, T_0, T', e et e'.



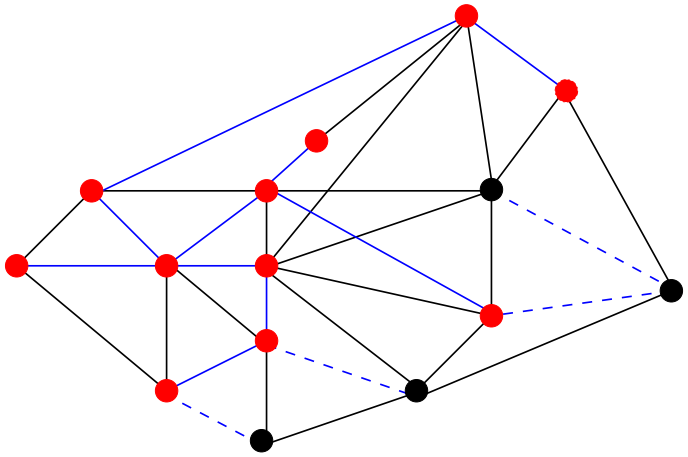
Appelons T_1 l'objet obtenu à partir de T_0 en supprimant l'arête e' et en ajoutant l'arête e, représenté en bleu sur le schéma ci-contre. Notons d'abord que T_1 est un arbre recouvrant de G. En effet, enlever l'arête e' à T_0 le déconnecte, mais ajouter ensuite l'arête e le reconnecte. Notons aussi que T_1 contient T'.



Vérifions maintenant que T_1 est de poids minimal. Pour cela, revenons au choix de l'arête e par l'algorithme de Prim. Les arêtes e et e' faisaient partie des candidates, puisqu'elles relient toutes les deux un sommet de T et un sommet extérieur à T. Si e a été préférée à e', c'est que $poids(e') \geq poids(e)$. Comme T_0 et T_1 ne diffèrent que par cette arête, on a $poids(T_0) \geq poids(T_1)$, et par minimalité du poids de T_0, on a en réalité $poids(T_0) = poids(T_1)$, et T_1 est bien un arbre recouvrant de poids minimal de G.



Finalement, notons que T' est contenu dans T_1. La propriété est donc aussi conservée dans ce cas.



Ceci nous assure qu'à la fin de l'exécution de l'algorithme, l'ensemble des arêtes composant T est contenu dans un arbre recouvrant de G de poids minimal. Comme T constitue lui-même un arbre recouvrant de G, il est bien de poids minimal, et la preuve est terminée.