Le problème du voyageur de commerce

Table des matières

Dans ce chapitre, nous allons examiner le problème bien connu du voyageur de commerce, appelé aussi "problème du commis voyageur" — en anglais, *Traveling Salesman Problem* (TSP). L'étude de ce problème nous permettra d'introduire de nouvelles approches de conception d'algorithmes — par exemple, algorithmes avec marche arrière (*backtracking*), algorithmes basés sur la *méthode de séparation et d'évaluation progressive* (*branch-and-bound*), algorithmes d'approximation, heuristiques et métaheuristiques, algorithmes probabilistes — tout en revoyant certaines des approches vues précédemment (diviser-pour-régner, programmation dynamique, décomposition pour algorithmes parallèles).

0 Présentation du problème

Soit un graphe orienté valué $G = (V, E)$ où les sommets V représentent des villes et les arcs[1] indiquent les coûts pour aller d'une ville à une autre — les arcs sont orientés et il est possible que le coût pour aller de v_i à v_j ($i \neq j$, car le coût est nul si $i = j$) ne soit pas le même que pour aller de v_j à v_i.[2]

Le problème du voyageur de commerce — qui a de nombreuses applications, par exemple, en fabrication de circuits VLSI ou en cristallographie aux rayons X — consiste à déterminer un chemin (un circuit) dans le graphe qui permettra de visiter chacune des villes une seule et unique fois en retournant ensuite à la ville de départ (le circuit est donc un *cycle*), et ce à un coût *minimum*.

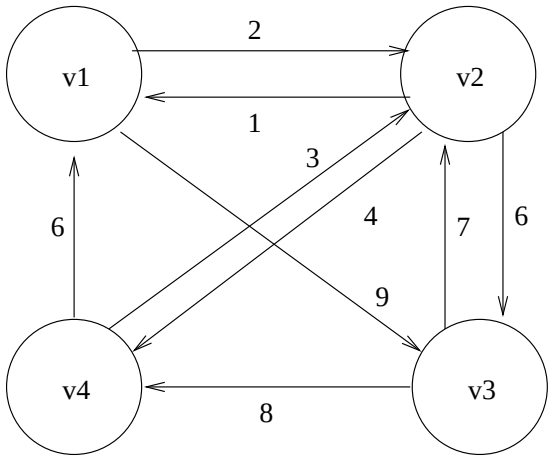

Figure 1: Un exemple de graphe pour le problème du voyageur de commerce

Par exemple, soit le graphe présenté à la figure 1. Des exemples de circuits possibles ayant v_1 comme point de départ seraient les suivants, le dernier étant celui de coût optimal (minimum) :

- $[v_1, v_2, v_3, v_4, v_1]$: coût 22.
- $[v_1, v_3, v_2, v_4, v_1]$: coût 26.
- $[v_1, v_3, v_4, v_2, v_1]$: coût 21.

De façon générale, on appelle une *tournée* — ou un cycle hamiltonien — d'un graphe orienté à n sommets, un chemin (une suite d'arcs) qui part d'un sommet v_1, passe exactement

[1] Rappelons la différence entre une arête et un arc: un arc est *orienté* alors qu'une arête ne l'est pas.

[2] Lorsque les coûts sont toujours les mêmes dans les deux directions, on parle alors du *symmetric TSP*.

une fois par chacun des sommets v_i puis retourne au sommet de départ v_1. Le problème du voyageur de commerce consiste donc à trouver une tournée de coût optimal sur un tel graphe.

Une approche naïve pour résoudre ce problème consiste à générer l'ensemble des chemins possibles et à calculer leurs coûts. Avant d'examiner cette solution, nous examinerons tout d'abord, à la prochaine section, un algorithme pour générer l'ensemble de tous les chemins possibles, indépendamment de leur coût. On verra ensuite (section 3) comment obtenir un chemin de coût minimal en générant l'ensemble des chemins possibles, puis ensuite comment obtenir un tel chemin sans nécessairement examiner tous les chemins. La section 4, quant à elle, présentera deux versions parallèles pour l'approche *branch-and-bound*. À la section 5, nous introduirons ensuite un algorithme de programmation dynamique pour résoudre le problème du voyageur de commerce. Comme nous le verrons, toutes ces solutions ont une complexité très élevée, rendant leur utilisation impraticable pour des problèmes comptant plus d'une centaine de villes — le record, en 1997, était pour une instance du problème avec 7397 villes, mais ce résultat avait alors demandé 3–4 *années* de temps CPU sur un réseau de stations Sun ; en 2002, le record était pour une instance avec 15 112 villes et a demandé l'utilisation d'un réseau de 110 processeurs pour un temps total équivalent à 22 *années* de temps CPU [DS04]. Nous verrons ensite, aux sections 6–10, diverses approximations et heuristiques qui nous permettront d'obtenir, en temps raisonnable, des solutions qui, bien que non optimales, *ne seront pas trop mauvaises*.

1 Génération des circuits hamiltoniens avec marche arrière (*backtracking*)

Avant d'examiner comment résoudre le problème du voyageur de commerce, nous allons illustrer la notion d'algorithme avec *marche arrière* (*backtracking*). Pour ce faire, nous allons tout d'abord examiner un problème "plus simple", à savoir générer *l'ensemble* des circuits hamiltoniens d'un graphe, et ce en ignorant les coûts de ces divers circuits.

La stratégie de marche arrière permet d'explorer un espace d'états, soit pour trouver un état qui satisfait un critère, soit pour générer l'ensemble des états qui satisfont ce critère. Le Grand dictionnaire terminologique[3], qui utilise plutôt le terme de procédure de "*retour arrière*", donne la définition suivante du *backtracking* : "Procédure de recherche qui consiste, lorsqu'un choix fait à un certain noeud conduit à un résultat inacceptable, à revenir au noeud d'origine pour faire un autre choix".

La propriété importante d'un algorithme avec marche arrière est de reconnaître certains culs-de-sac (*dead ends*), c'est-à-dire, certains états qui sont certains de ne pas conduire à une solution, ce qui évite dans certains cas de faire une recherche exhaustive de l'ensemble des états.

Dans de nombreux problèmes pour lesquels le *backtracking* est une stratégie appropriée, la sélection d'une solution se fait en effectuant une série de décisions, chaque décision se faisant sur la base de l'examen d'un ensemble "d'alternatives possibles". Dans de nombreux cas, l'espace des états peut être représenté (de façon effective ou virtuelle) par un arbre, où chaque noeud interne représente un état intermédiaire et où les feuilles représentent un état final, feuilles pouvant représenter ou non une solution valide. Dans ce contexte, un algorithme avec marche arrière consiste alors à explorer (en profondeur) l'arbre (réel ou virtuel) et, lorsqu'un noeud est rencontré qui ne satisfait pas un critère approprié de sélection (identification d'un cul-de-sac), à retourner (*backtrack*) au noeud parent pour explorer les autres alternatives, c'est-à-dire les autres enfants de ce parent.

L'algorithme 0 (en MPD) est un algorithme récursif pour générer l'ensemble des circuits hamiltoniens sur un graphe. On suppose que la matrice `W` des coûts des arcs est définie par une

[3] `http://www.granddictionnaire.com`

```
procedure arcExiste( int i, int j ) returns bool r
# ROLE: Determine si un arc existe du sommet i vers le sommet j dans le graphe.
# PRECONDITION:  i et j denote des numeros de sommets valides pour la matrice W.
# POSTCONDITION: r <=> W[i][j] != INFINI

procedure dejaVisite( Sequence chemin, int sommet ) returns bool r
# ROLE: Determine si le sommet a deja ete visite (appartient au chemin indique).
# POSTCONDITION: r <=> estElement(chemin, sommmet)

procedure genererCircuits( Sequence chemin, int n )
# ROLE: Genere les circuits en tentant d'etendre la sequence de sommets deja visites.
{
  if ( longueur(chemin) == n ) {
    # Tous les sommets ont ete visites et sont inclus dans le chemin.

    if ( arcExiste(queue(chemin), tete(chemin)) ) {
      # Un arc existe du dernier sommet vers le premier, on a donc
      # un cycle vers le sommet de depart couvrant tous les sommets.
      imprimer( chemin ); write();
    } else { # On ne fait rien (pas de cycle vers le sommet de depart).
    }

  } else {
    # Certains sommets n'ont pas encore ete visites. Pour la position suivante,
    # on explore, parmi les sommets pas visites, ceux pour lesquels un arc existe.
    for [s = 1 to n st ~dejaVisite(chemin, s)] {
      if ( arcExiste(queue(chemin), s) ) {
        # Un arc existe: on l'ajoute au chemin, on explore le nouveau chemin,
        # puis au revient au chemin initial pour explorer un autre sommet.
        Sequence ch = cloner(chemin);
        ajouterEnQueue(ch, s);
        genererCircuits(ch, n);
      } else { # Aucun arc => cul-de-sac.
      }
    }
  }
}

procedure genererTousLesCircuits( int sommetDepart, int n )
# ROLE: Genere l'ensemble des circuits ayant comme point de depart sommetDepart.
#       (n indique le nombre maximum de sommets possibles).
{
  Sequence chemin = creer();
  ajouterEnQueue(chemin, sommetDepart);
  genererCircuits(chemin, n);
}
```

Algorithme 0: Algorithme récursif (avec marche arrière) pour générer l'ensemble des circuits hamiltoniens sur un graphe

variable globale et qu'elle est accédée directement par la fonction `arcExiste`, laquelle retourne simplement `true` ou `false` selon qu'un arc existe ou non entre les deux sommets (si aucun arc n'existe, alors le cout associé est $+\infty$). La variable `chemin` représente la séquence des sommets *déjà visités*, c'est-à-dire le chemin parcouru jusqu'à présent. Une solution impossible (un cul-de-sac) est détectée lorsque le sommet qu'on voudrait visiter parce qu'il fait partie des sommets pas encore explorés (variable `s` dans la boucle `for`) ne possède aucun arc le reliant au dernier sommet du chemin courant — en d'autres mots, le chemin courant ne peut pas être étendu par ce nouveau sommet `s`. L'exploration du chemin courant se termine donc et on tente plutôt d'explorer un autre sommet (par l'intermédiaire des différents valeurs `s` n'ayant pas déjà été visités de la boucle `for`). Notons que le chemin courant est cloné (`ch = cloner(chemin)`), c'est-à-dire, copié (en profondeur) pour permettre que ce (préfixe de) chemin puisse être réutilisé pour explorer les divers autres sommets (`ajouterEnQueue(ch, s)`).[4]

Lorsque tous les sommets possibles à explorer pour le chemin courant auront été visités et les chemins associés explorés, on retournera alors à la procédure appelante (marche arrière) pour examiner les autres sommets du préfixe du chemin. En termes d'exploration de l'arbre des états, l'exploration avec marche arrière effectue donc un parcours *en profondeur* de l'arbre (*depth-first search*).

Le type abstrait de données 1 présente les opérations de manipulation de séquences (entêtes seulement, spécifiés à l'aide de pré/post-conditions) utilisées dans l'algorithme d'exploration du graphe et de génération des circuits. Notons que, dans une postcondition, une référence à `s'` (avec l'accent) dénote la valeur de `s` *avant* l'appel, c'est-à-dire avant le changement d'état effectué par l'opération, alors que la valeur `s` (sans accent) dénote l'état après le changement d'état.

Analyse de l'algorithme

Une analyse de cet algorithme peut se faire en définissant des équations de récurrence appropriées pour modéliser le temps d'exécution de la procédure `genererCircuits`. Notons par l la longueur du `chemin` (`longueur(chemin)`) reçu en argument par cette procédure. Notons alors par m le nombre de sommets du graphe qui n'ont pas encore été visités. En tout temps, on aura toujours que $m = n - l$, où n est le nombre de sommets du graphe.

Dans un premier temps, le temps pour la procédure `genererTousLesCircuits` pourra être défini comme suit (avec $n = m + l$), puisque l'appel initial à `genererCircuits` inclut déjà un sommet dans `chemin` :

$$T_{gtlc}(n) = T_{gc}(n-1) = T_{gc}(m)$$

On peut donc caractériser le temps T_{gc} de la procédure `genererCircuits` en fonction de m le nombre de sommets qui n'ont pas encore été visités. On peut supposer que le temps pour déterminer si un sommet a déjà été visité est $\Theta(1)$ (par exemple, une mise en oeuvre à l'aide d'un dictionnaire ou d'un tableau de bits). On aura alors les équations de récurrence suivantes pour `genererCircuits` (avec $n = m + l$) :

- $T_{gc}(m) = m\,[T_{gc}(m-1) + \Theta(l)]$
- $T_{gc}(0) = \Theta(n)$

Dans l'équation pour $T_{gc}(m)$, le facteur multiplicatif m apparaît car, dans le pire des cas — le pire cas est un graphe complet, c'est-à-dire un graphe où il existe un arc reliant n'importe quelle paire de sommets —, il faut visiter *chacun* des autres sommets (m itérations

[4] Une solution sans clonage aurait pû être définie, mais cela aurait rendu le code un peu plus complexe et, surtout, aurait rendu plus difficile la parallélisation de l'algorithme que nous verrons à la section 4. Une autre solution possible aurait aussi été d'utiliser un type abstrait définissant une *collection de valeurs* (entités immuables) plutôt qu'une *classe d'objets* (entités mutables).

```
type Sequence = ptr ...;

procedure creer() returns Sequence s
# POSTCONDITION
#   s = []

procedure longueur( Sequence s ) returns int l
# POSTCONDITION
#   l = length(s)

procedure cloner( Sequence s1 ) returns Sequence s2
# POSTCONDITION
#   s2 est un nouvel objet qui est une copie (profonde) de s1.

procedure ajouterEnQueue( Sequence s, int elem )
# POSTCONDITION
#   s = s' || [elem]

procedure tete( Sequence s ) returns int t
# PRECONDITION
#   s != []
# POSTCONDITION
#   t = s[1]

procedure queue( Sequence s ) returns int q
# PRECONDITION
#   s != []
# POSTCONDITION
#   q = s[length(s)]

procedure element( Sequence s, int i ) returns int e
# PRECONDITION
#   i IN domain(s)
# POSTCONDITION
#   e = s[i]

procedure estElement( Sequence s, int e ) returns bool r
# POSTCONDITION
#   r <=> e IN s

procedure imprimer( Sequence s )
# POSTCONDITION
#   Chacun des elements de s a ete imprime sur stdout.
```

Type abstrait de données 1: Types et procédures pour la spécification d'une *classe d'objets* pour des séquences (mise en oeuvre omise)

$$
\begin{aligned}
T_{gtlc}(n) &= T_{gc}(n-1) \\
&= (n-1)[T_{gc}(n-2) + c \times 1] \\
&= (n-1)T_{gc}(n-2) + c(n-1) \\
&= (n-1)(n-2)[T_{gc}(n-3) + c \times 2] + c(n-1) \\
&= (n-1)(n-2)T_{gc}(n-3) + 2c(n-1)(n-2) + c(n-1) \\
&= (n-1)(n-2)(n-3)[T_{gc}(n-4) + c \times 3] + 2c(n-1)(n-2) + c(n-1) \\
&= (n-1)(n-2)(n-3)T_{gc}(n-4) + 3c(n-1)(n-2)(n-3) + 2c(n-1)(n-2) + c(n-1) \\
&= \frac{(n-1)!}{(n-4)!}T_{gc}(n-4) + 3c\frac{(n-1)!}{(n-4)!} + 2c\frac{(n-1)!}{(n-3)!} + 1c\frac{(n-1)!}{(n-2)!} \\
&= \frac{(n-1)!}{(n-4)!}T_{gc}(n-4) + c(n-1)!\left[\frac{3}{(n-4)!} + \frac{2}{(n-3)!} + \frac{1}{(n-2)!}\right] \\
&= \frac{(n-1)!}{(n-4)!}T_{gc}(n-4) + c(n-1)!\sum_{i=1}^{3}\frac{i}{(n-(i+1))!} \\
&= \ldots \\
&= \frac{(n-1)!}{0!}T_{gc}(0) + c(n-1)!\sum_{i=1}^{n-1}\frac{i}{(n-(i+1))!} \\
&= (n-1)! \times c'n + c(n-1)!\sum_{i=1}^{n-1}\frac{i}{(n-(i+1))!} \\
&= (n-1)!\left[c'n + c\sum_{i=1}^{n-1}\frac{i}{(n-(i+1))!}\right] \\
&\in (n-1)! \times \Theta(n) \\
&\in \Theta(n!)
\end{aligned}
$$

Figure 2: Solution (par substitution) des équations de récurrence associées à `genererCircuits`

de la boucle `for`). Quant au terme $\Theta(l)$, il apparaît à cause de l'opération de clonage du chemin (toujours dans la boucle `for`). Finalement, dans le cas de base $T(0)$, le terme $\Theta(n)$ apparaît parce que dans le cas où un cycle est détecté, la procédure doit `imprimer` le chemin identifié.

La solution des équations pour T_{gc} peut alors être obtenue tel que décrit à la figure 2, où on suppose un facteur multiplicatif c pour le terme $\Theta(l)$, un facteur multiplicatif c' pour le terme $\Theta(n)$ dans $T_{gc}(0)$, et en se rappelant que $n = m+l$, avec un appel initial à `genererCircuits` tel que $m = n-1$, donc $l = 1$. On en conclut donc que cet algorithme est de complexité factorielle $\Theta(n!)$ pour un graphe avec n sommets. Notons que si on n'avait pas tenu compte de l'impression, on aurait alors obtenu une complexité asymptotique $\Theta((n-1)!)$. Bien que *moins pire* que $\Theta(n!)$ (puisque $\Theta((n-1)!) \in o(\Theta(n!))$), il s'agit néammoins d'une complexité élevée, en fait, pire qu'exponentielle $\Theta(2^n)$.

2 Solution au problème du voyageur de commerce avec un algorithme avec marche arrière

Les algorithmes avec marche arrière sont utiles principalement pour les problèmes *de décision* ou *d'énumération*, pour lesquels il suffit d'identifier les solutions qui satisfont un critère *de faisabilité*. Toutefois, l'exploration avec marche arrière peut aussi être adaptée pour des problèmes *d'optimisation*, c'est-à-dire des problèmes pour lequels l'objectif est d'identifier une (ou plusieurs) solution qui satisfait un critère *d'optimalité*, c'est-à-dire, une solution qui minimise (ou maximise) une fonction de coût appropriée.

L'algorithme 1 présente une telle adaption d'un algorithme de marche arrière pour résoudre le problème du voyageur de commerce. Tout d'abord, des variables globales sont introduites pour prendre note du chemin de coût minimum ayant été rencontré (`cheminMin`) et du coût qui lui est associé (`coutCheminMin`). Lorsqu'un nouveau cycle hamiltonien est rencontré (un chemin de longueur `n` est identifié et un arc vers le sommet de départ existe), on vérifie alors le coût du circuit résultant et, si nécessaire, on met à jour les variables globales. Le calcul du coût d'un chemin se fait à l'aide d'une procédure `coutChemin` que nous définirons ultérieurement ; rappelons simplement que les coûts des divers arcs sont donnés par une matrice `W` des coûts, `W[vi][vj]` indiquant le coût de l'arc reliant le sommet `vi` au sommet `vj` (coût $+\infty$ si aucun arc n'existe). L'impression (ave `imprimer`) du chemin minimal rencontré et du coût associé ne se fait alors qu'à la toute fin (procédure `trouverCircuitMin`), c'est-à-dire uniquement lorsque tous les circuits possibles ont été explorés.

3 Solution au problème du voyageur de commerce avec un algorithme *branch-and-bound*

Peut-on éviter d'explorer des chemins *non prometteurs*?

Le principal désavantage de la solution présentée à l'algorithme 1 est le suivant : tous les circuits potentiels *sont explorés jusqu'au bout*, et ce même si le coût du circuit en cours d'exploration risque d'être supérieur au coût du circuit minimum déjà identifié. Par exemple, il est clair qu'il est inutile d'explorer un chemin dont le coût (partiel) est déjà supérieur au coût du chemin minimum déjà identifié.

Une stratégie plus intéressante consisterait donc à introduire un mécanisme d'évaluation et de sélection des différents chemins pouvant être explorés, des différentes alternatives. L'introduction d'une telle stratégie conduit alors à un algorithme de type *branch-and-bound*, algorithme dit, selon le Grand dictionnaire terminologique[5], *de séparation et d'évaluation progressive*. Une telle méthode, toujours selon le Grand dictionnaire terminologique, sert "*à définir des solutions optimales et qui sont basées sur l'énumération et l'évaluation de solutions possibles.*

Plus précisément, dans l'exploration de type *branch-and-bound*, avant d'explorer une nouvelle alternative, on utilise une fonction d'évaluation qui permet de déterminer, à l'aide d'un *estimé* du coût prévu, si cette alternative pourrait permettre de conduire à un résultat meilleur que le meilleur résultat obtenu jusqu'à présent. Si c'est le cas, l'alternative est alors explorée, sinon elle est simplement ignorée — en d'autres mots, la fonction d'évaluation permet d'identifier les culs-de-sac potentiels, c'est-à-dire *d'élaguer* l'arbre de recherche (en anglais, on parle de *pruning*). Cette fonction d'évaluation peut aussi être utilisée, lorsque plusieurs alternatives sont possibles, pour sélectionner celle qui semble *la plus prometteuse* — dans ce cas, on parle alors d'une stratégie de type *best-first search*.

[5] `http://www.granddictionnaire.com`

```
Sequence cheminMin;
int coutCheminMin = INFINI;

procedure explorerCircuits( Sequence chemin, int n )
{
  if (longueur(chemin) == n) {

    if ( arcExiste(queue(chemin), tete(chemin)) ) {
      int cout = coutChemin(chemin) + W[queue(chemin)][tete(chemin)];
      if (cout < coutCheminMin) {
        # On vient de trouver un circuit ayant un cout inferieur.
        cheminMin = chemin;
        coutCheminMin = cout;
      }
    }

  } else {
    # Certains sommets n'ont pas encore ete visites.

    # Pour la position suivante, on explore tous les sommets n'ayant
    # pas encore ete visites et pour lesquels un arc existe.
    for [s = 1 to n st ~dejaVisite(chemin, s)] {
      if ( arcExiste(queue(chemin), s)  ) {
        Sequence ch = cloner(chemin);
        ajouterEnQueue(ch, s);
        explorerCircuits(ch, n);
      }
    }
  }
}

procedure trouverCircuitMin( int sommetDepart, int n )
{
  Sequence chemin = creer();

  # On explore l'ensemble des circuits.
  ajouterEnQueue(chemin, sommetDepart);
  explorerCircuits(chemin, n);

  # On imprime le resultat.
  imprimer( cheminMin ); write( " =>", coutCheminMin );
}
```

Algorithme 1: Algorithme avec marche arrière pour le problème du voyageur de commerce

Dans ses grandes lignes, l'approche *branch-and-bound* (sans utilisation de l'approche *best-first search*) utilise donc la stratégie suivante pour décider d'explorer ou non un nouveau sommet pour étendre le chemin courant :

```
// Extension d'un chemin (partiel) par un nouveau sommet.
DEBUT
  Soit ch le chemin (partiel) déjà exploré
  Soit s le prochain sommet à examiner
  estime <- Coût estimé pour le sommet s ajouté au chemin courant ch
  SI estime < Coût de la meilleure solution obtenue jusqu'à présent ALORS
    On explore le chemin courant augmenté du sommet s
       (il pourrait conduire à un meilleur résultat)
  SINON
    On ignore le sommet s
       (son utilisation conduit nécessairement à un moins bon résultat)
  FIN
FIN
```

Pour résoudre le problème du voyageur de commerce à l'aide d'un algorithme *branch-and-bound*, on peut alors s'inspirer de l'algorithme 1 d'exploration du graphe avec marche arrière. Il faut toutefois définir une fonction d'évaluation appropriée. Dans le contexte du problème du voyageur de commerce, on cherche à minimiser le coût d'un circuit hamiltonien. Notre fonction d'évaluation doit donc produire une *borne inférieure* du coût d'exploration — en d'autres mots, la fonction doit retourner une valeur telle que le véritable coût sera nécessairement supérieur. Si ce coût estimé, qui est une borne inférieure au véritable coût pour ce chemin potentiel, est supérieur au coût minimum des solutions déjà rencontrées, alors ce chemin pourra être considéré comme une alternative non prometteuse, alternative qu'il est donc inutile d'explorer : son vrai coût ne pourra qu'être supérieur à l'estimé, lequel est déjà supérieur au plus petit coût déjà rencontré. Dans la prochaine section, nous allons donc examiner de quelle façon obtenir un estimé approprié pour un chemin.

3.1 Fonction d'estimation des coûts pour le problème du voyageur de commerce

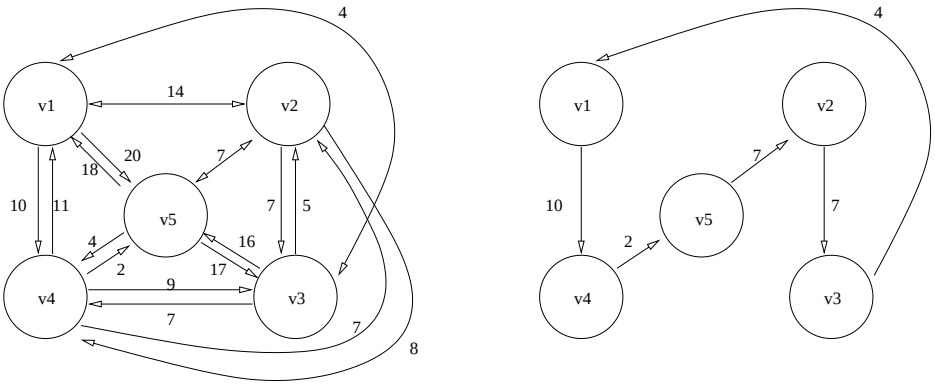

Figure 3: Un exemple de graphe complet à cinq (5) sommets pour le problème du voyageur de commerce et un circuit optimal

Soit le graphe complet présenté dans la partie gauche de la figure 3. La matrice des coûts

des divers arcs pour ce graphe est la suivante :

$$\begin{pmatrix} 0 & 14 & 4 & 10 & 20 \\ 14 & 0 & 7 & 8 & 7 \\ 4 & 5 & 0 & 7 & 16 \\ 11 & 7 & 9 & 0 & 2 \\ 18 & 7 & 17 & 4 & 0 \end{pmatrix}$$

Un circuit de coût minimum (30) est aussi présenté dans la partie droite de la figure 3.

Examinons à l'aide de cet exemple comment évaluer et estimer le coût pour un chemin partiel déjà parcouru qu'on désire étendre par un nouveau sommet. Pour un chemin donné, on doit trouver une borne inférieure pour un circuit qui débute par ce chemin, lequel chemin sera complété par un certain nombre d'arcs (vers des sommets non visités) jusqu'à ce qu'un cycle vers le sommet de départ soit obtenu.

Le coût minimum pour quitter un sommet est le minimum des coûts des divers arcs qui quittent ce sommet. En termes de la matrice W des coûts des arcs, le coût pour quitter un sommet s a donc, en gros, l'allure suivante (en supposant que le sommet 1 indique le point de départ) :

$$minArcsSortants(j) = \min_{j \neq s \wedge (j \text{ pas visité } \vee\ j=1)} (W[s][j])$$

Parce qu'un circuit doit nécessairement quitter à l'aide d'un arc chacun des sommets du graphe (un circuit est un cycle), le coût d'un circuit obtenu à partir d'un chemin `ch` est alors la somme des coûts des arcs de `ch`, auquel on ajoute la somme des minimums des coûts pour des arcs sortant d'un sommet n'ayant pas encore été visité. Toutefois, il faut exclure les arcs qui vont vers des sommets déjà visités, sauf évidemment le sommet de départ (puisqu'on doit ultimement créer un cycle).

Par exemple, soit le graphe de la figure 3. Supposons qu'on ait un chemin `ch = [v1, v2]` de sommets déjà visités. On veut déterminer le coût minimum pouvant être obtenu en étendant ce chemin par le sommet `v3` et, ensuite, par d'autres sommets (jusqu'à ce qu'on retourne au sommet de départ). Le coût minimum sera alors le suivant :

$$\begin{aligned} \texttt{coutEstime(ch || [v3])} &= \texttt{coutChemin(ch)} + coutArc(\texttt{v2}, \texttt{v3}) + \sum_{j=3}^{5} minArcsSortants(j) \\ &= \texttt{W[v1][v2]} + \texttt{W[v2][v3]} + min(7, 16) + min(11, 2) + min(18, 4) \\ &= 14 + 7 + min(7, 16) + min(11, 2) + min(18, 4) \\ &= 34 \end{aligned}$$

Notons que dans les arcs sortant d'un sommet j, il faut éviter d'aller vers un sommet déjà visité, *à moins que ce sommet ne soit le sommet de départ.* Ainsi, pour $j = 4$, il ne faut pas visiter aucun des sommets `v2`, `v3` (ni, évidemment, `v4`), d'où l'expression $min(11, 2)$. De plus, pour $j = 3$, qui correspond au sommet qu'on tente d'ajouter et pour lequel on sait que ce ne sera pas le dernier sommet (voir l'algorithme), il faut aussi éviter de retourner aux divers sommets visités, y compris le sommet de départ car on sait que ce n'est pas encore le moment de créer le cycle final (d'où l'expression $min(7, 16)$).

Dans les sections qui suivent, c'est une fonction d'évaluation basée sur cette stratégie (minimum des coûts des arcs qui quittent les divers sommets) qui sera utilisée. Notons toutefois qu'il existe d'autres fonctions d'évaluation pouvant être utilisées pour le problème du voyageur de commerce. Par exemple, on pourrait utiliser le minimum des coûts des arcs entrants (dans un circuit, on doit aller vers chacun des sommets), ou encore utiliser une fonction qui tient compte autant des arcs qui entrent que celles qui sortent (pondération moitié/moitié).

3.2 Algorithme *branch-and-bound* : première version (sans *best-first search*)

L'algorithme 2 effectue une exploration de type *branch-and-bound*. Cet algorithme ressemble à l'algorithme 1 avec marche arrière. La différence importante est que l'objectif est maintenant de déterminer *le circuit de coût minimal* tout en évitant d'explorer des chemins qui ne sont pas prometteurs. Seul le circuit de coût minimum est identifié et imprimé (dans la procédure, non récursive, de niveau supérieur `trouverCircuitMin`). Pour éviter l'exploration de certains chemins (élagage), la fonction `coutEstime` d'estimation du coût minimum d'un chemin est utilisée : voir fonctions auxiliaires 1. Quant au chemin de coût minimum rencontré jusqu'à un instant donné ainsi que le coût associé à ce chemin, ils sont définis à l'aide de variables globales lesquelles sont mises à jour à l'aide de l'opération `majCheminMin` : voir fonctions auxiliaires 2. Signalons que la constante `INFINI` peut être simplement définie, en MPD, par la déclaration suivante :

```
const int INFINI = high(int);
```

3.3 Algorithme *branch-and-bound* : deuxième version (avec *best-first search*)

Pour le graphe complet à cinq (5) sommets présentés plus haut, l'algorithme 2 va explorer six (6) circuits complets de façon à identifier le circuit optimal. Dans cet algorithme, l'exploration des nouveaux sommets s'effectue simplement dans l'ordre dans lequel les sommets sont générés (c'est-à-dire, dans ce cas-ci, en ordre croissant de leur numéro). Puisqu'on prend la peine de calculer un estimé du coût des chemins à explorer, une stratégie qui serait probablement plus intéressante consisterait à explorer les nouveaux sommets en fonction des coûts estimés associés. Dans ce cas, puisqu'on cherche à trouver un circuit de coût minimal, il s'agirait donc de sélectionner, comme prochain sommet à explorer, celui dont l'estimé obtenu est minimum.

L'algorithme 3 utilise cette stratégie (appelée *best-first search* en anglais). Une *file de priorité* est utilisée pour, à partir d'un chemin donné, traiter ensuite les différents sommets possibles en sélectionnant celui dont le coût estimé (le niveau de priorité) est le plus faible. Le type abstrait 2 présente les en-têtes des procédures de manipulation des files de priorité. Un fait intéressant à signaler est qu'un sommet peut avoir été ajouté à la file de priorité (`ajouter(aExplorer, estime, s)`) mais sans que ce sommet soit ensuite exploré (si `estime` est supérieur ou égal à `coutCheminMin` dans le `if` à l'intérieur de la boucle) parce que, entre le moment où il est ajouté dans la file de priorité et le moment où il en est retiré pour exploration, un chemin de coût inférieur pourrait avoir été identifié.

Dans le cas du graphe à cinq (5) sommets présentés plus haut, l'utilisation de l'algorithme 3 (avec stratégie *best-first search*) a pour effet de réduire à seulement deux (2) le nombre de circuits complets explorés.

3.4 Analyse et comparaisons empiriques des algorithmes

Empiriquement, si l'on compare les trois algorithmes présentés précédemment — avec marche arrière simple, *branch-and-bound* sans et avec file de priorité —, on constate que l'utilisation de l'algorithme *branch-and-bound* avec utilisation d'une file de priorité (donc *branch-and-bound* combinée à une recherche *best-first*) semble effectivement permettre de réduire le nombre de chemins complets qui sont explorés pour trouver la solution optimale. Les deux variantes *branch-and-bound* conduisent aussi à une réduction importante du nombre de chemins explorés comparativement à l'algorithme d'exploration exhaustive. Ceci est illustré, sur un exemple concret à savoir un graphe complet avec huit (8) sommets, à la figure 4. Toutefois,

```
procedure explorerCircuits( Sequence chemin, int n )
{
  if (longueur(chemin) == n) {

    if ( arcExiste(queue(chemin), tete(chemin)) ) {
      int cout = coutChemin(chemin) + W[queue(chemin)][tete(chemin)];
      # On vient de trouver un circuit ayant un cout inferieur.
      majCheminMin( chemin, cout );
    }

  } else {
    # Certains sommets n'ont pas encore ete visites.

    # Pour la position suivante, on explore, parmi tous les sommets n'ayant
    # pas encore ete visites, uniquement ceux pour lesquels un arc
    # existe et qui semblent prometteurs.
    for [s = 1 to n st ~dejaVisite(chemin, s)] {
      if ( arcExiste(queue(chemin), s)
         & coutEstime(chemin, s, n) < coutCheminMin ) {
        # Un arc existe et semble prometteur : on l'explore.
        Sequence ch = cloner(chemin);
        ajouterEnQueue( ch, s );
        explorerCircuits( ch, n );
      }
    }
  }
}

procedure trouverCircuitMin( int sommetDepart, int n )
{
  Sequence chemin = creer();

  # On explore l'ensemble des circuits.
  ajouterEnQueue( chemin, sommetDepart );
  explorerCircuits( chemin, n );

  # On imprime le resultat.
  imprimer( cheminMin ); write( " =>", coutCheminMin );
}
```

Algorithme 2: Algorithme *branch-and-bound* pour le problème du voyageur de commerce (sans file de priorité)

```
procedure coutChemin( Sequence chemin ) returns int cout
# POSTCONDITION
#   cout = sommme des couts des arcs reliant les sommets du chemin indique.
{
  cout = 0;
  for [i = 1 to longueur(chemin)-1] {
    cout += W[element(chemin, i)][element(chemin, i+1)];
  }
}

procedure coutMin( int couts[*], Sequence chemin, int aExclure ) returns int cout
# POSTCONDITION
#   cout = minimum des couts indiques, mais en ignorant le sommet aExclure
#          de meme que les sommets deja visites de chemin (mais a l'exception
#          du sommet de depart, pour creer le cycle).
{
  cout = high(int);
  for [i = 1 to ub(couts) st couts[i] > 0 & i != aExclure
                          & (~dejaVisite(chemin, i) | element(chemin, 1) == i) ]
  { cout = min(couts[i], cout); }
}

procedure coutEstime( Sequence chemin, int sommet, int n ) returns int cout
# ROLE
#   Determiner un estime (une borne inferieure) pour le cout d'un circuit
#   obtenu en ajoutant le sommet indique au chemin existant.
# POSTCONDITION
#   cout = somme des couts du chemin + somme de l'arc reliant le dernier sommet
#          du chemin a sommet + somme des couts minimum pour
#          aller vers les sommets pas visites puis vers le sommet de depart.
#
{
  cout = coutChemin(chemin);
  cout += W[queue(chemin)][sommet];
  cout += coutMin( W[sommet][*], chemin, tete(chemin) );
  for [s = 1 to n st s != sommet & ~dejaVisite(chemin, s)] {
    cout += coutMin( W[s][*], chemin, sommet );
  }
}
```

Fonctions auxiliaires 1: Fonctions auxiliaires pour l'estimation des coûts pour l'algorithme *branch-and-bound* (voyageur de commerce)

```
Sequence cheminMin;
int coutCheminMin = INFINI;

procedure majCheminMin( Sequence chemin, int cout )
{
  if (cout < coutCheminMin) {
    cheminMin = chemin;
    coutCheminMin = cout;
  }
}
```

Fonctions auxiliaires 2: Fonction auxiliaires pour mise-à-jour du chemin minimum et du coût associé dans l'algorithme *branch-and-bound*

```
procedure explorerCircuits( Sequence chemin, int n )
{
  if (longueur(chemin) == n) {

    if ( arcExiste(queue(chemin), tete(chemin)) ) {
      int cout = coutChemin(chemin) + W[queue(chemin)][tete(chemin)];
      # On vient de trouver un circuit ayant un cout inferieur.
      majCheminMin( chemin, cout );
    }

  } else {
    # Certains sommets n'ont pas encore ete visites.

    FilePriorite aExplorer = creerFP();
    # Pour la position suivante, on explore, parmi tous les sommets n'ayant
    # pas encore ete visites, uniquement ceux pour lesquels un arc existe.
    for [s = 1 to n st ~dejaVisite(chemin, s)] {
      int estime = coutEstime(chemin, s, n);
      if ( arcExiste(queue(chemin), s) & estime < coutCheminMin ) {
        # Un arc existe: on l'ajoute dans la file de priorite des noeuds a examiner.
        Sequence ch = cloner(chemin); ajouterEnQueue(ch, s);
        ajouter( aExplorer, estime, ch );
      }
    }
    # On explore les sommets en ordre croissant de couts estimes.
    while (~estVide(aExplorer)) {
      int estime;
      Sequence chemin;
      obtenirMin( aExplorer, estime, chemin );
      if ( estime < coutCheminMin ) {
        # On explore uniquement si l'estime est encore interessant.
        explorerCircuits( chemin, n );
      }
    }
  }
}

procedure trouverCircuitMin( int sommetDepart, int n )
{
  Sequence chemin = creer();

  # On explore l'ensemble des circuits.
  ajouterEnQueue( chemin, sommetDepart );
  explorerCircuits( chemin, n );

  # On imprime le resultat.
  imprimer( cheminMin ); write( " =>", coutCheminMin );
}
```

Algorithme 3: Algorithme *branch-and-bound* pour le problème du voyageur de commerce avec file de priorité (stratégie *best-first search*)

Trace d'exécution de l'algorithme récursif (avec marche arrière) pour 8 sommets

```
[1, 2, 3, 4, 5, 6, 7, 8] => 394
[1, 2, 3, 4, 5, 6, 8, 7] => 278
[1, 2, 3, 4, 5, 6, 8, 7] => 278
[1, 2, 3, 4, 5, 6, 8, 7] => 278
[1, 2, 3, 4, 5, 6, 8, 7] => 278
[1, 2, 3, 4, 5, 8, 7, 6] => 269
[1, 2, 3, 4, 5, 8, 7, 6] => 269
[1, 2, 3, 4, 5, 8, 7, 6] => 269
[1, 2, 3, 4, 5, 8, 7, 6] => 269
[1, 2, 3, 4, 5, 8, 7, 6] => 269
[1, 2, 3, 4, 5, 8, 7, 6] => 269
[1, 2, 3, 4, 6, 8, 7, 5] => 261
[1, 2, 3, 4, 6, 8, 7, 5] => 261
[1, 2, 3, 4, 6, 8, 7, 5] => 261
[1, 2, 3, 4, 6, 8, 7, 5] => 261
[1, 2, 3, 4, 6, 8, 7, 5] => 261
[1, 2, 3, 4, 6, 8, 7, 5] => 261
[1, 2, 3, 4, 6, 8, 7, 5] => 261
[1, 2, 3, 4, 6, 8, 7, 5] => 261
[1, 2, 3, 4, 6, 8, 7, 5] => 261
[1, 2, 3, 4, 6, 8, 7, 5] => 261
[1, 2, 3, 4, 6, 8, 7, 5] => 261
[1, 2, 3, 4, 8, 7, 5, 6] => 259
[1, 2, 3, 4, 8, 7, 6, 5] => 258
[1, 2, 3, 4, 8, 7, 6, 5] => 258
[1, 2, 3, 4, 8, 7, 6, 5] => 258
[1, 2, 3, 4, 8, 7, 6, 5] => 258
[1, 2, 3, 4, 8, 7, 6, 5] => 258
[1, 2, 3, 4, 8, 7, 6, 5] => 258
[1, 2, 3, 5, 4, 8, 7, 6] => 232
...
...
...
[1, 6, 4, 5, 2, 3, 8, 7] => 151
[1, 6, 4, 5, 2, 3, 8, 7] => 151
[1, 6, 4, 5, 2, 3, 8, 7] => 151
[1, 6, 4, 5, 2, 3, 8, 7] => 151
[1, 6, 4, 5, 2, 3, 8, 7] => 151
[1, 6, 4, 5, 2, 3, 8, 7] => 151
[1, 6, 4, 5, 2, 3, 8, 7] => 151
[1, 6, 4, 5, 2, 3, 8, 7] => 151
[1, 6, 4, 5, 2, 3, 8, 7] => 151
[1, 6, 4, 5, 2, 3, 8, 7] => 151
[1, 6, 4, 5, 2, 3, 8, 7] => 151
[1, 6, 4, 5, 2, 3, 8, 7] => 151
[1, 6, 4, 5, 2, 3, 8, 7] => 151
[1, 6, 4, 5, 2, 3, 8, 7] => 151
[1, 6, 4, 5, 2, 3, 8, 7] => 151
[1, 6, 4, 5, 2, 3, 8, 7] => 151
[1, 6, 4, 5, 2, 3, 8, 7] => 151
[1, 6, 4, 5, 2, 3, 8, 7] => 151
[1, 6, 4, 5, 2, 3, 8, 7] => 151
[1, 6, 4, 5, 2, 3, 8, 7] => 151
[1, 6, 4, 5, 2, 3, 8, 7] => 151
[1, 6, 4, 5, 2, 3, 8, 7] => 151
[1, 6, 4, 5, 2, 3, 8, 7] => 151

5040  circuits complets ont ete examines
```

Trace d'exécution de l'algorithme *branch-and-bound* pour 8 sommets

```
[1, 2, 3, 4, 5, 6, 7, 8] => 394
[1, 2, 3, 4, 5, 6, 8, 7] => 278
[1, 2, 3, 4, 5, 8, 7, 6] => 269
[1, 2, 3, 4, 6, 8, 7, 5] => 261
[1, 2, 3, 4, 8, 7, 5, 6] => 259
[1, 2, 3, 4, 8, 7, 6, 5] => 258
[1, 2, 3, 5, 4, 8, 7, 6] => 232
[1, 2, 3, 6, 4, 8, 7, 5] => 197
[1, 2, 3, 6, 5, 4, 8, 7] => 178
[1, 2, 5, 4, 6, 3, 8, 7] => 172
[1, 2, 5, 4, 8, 7, 3, 6] => 165
[1, 3, 8, 7, 4, 5, 2, 6] => 162
[1, 6, 3, 2, 5, 4, 8, 7] => 160
[1, 6, 3, 8, 7, 2, 5, 4] => 154
[1, 6, 4, 5, 2, 3, 8, 7] => 151

15  circuits complets ont ete examines
```

Trace d'exécution de l'algorithme *branch-and-bound* avec *best-first* pour 8 sommets

```
[1, 6, 3, 8, 7, 4, 5, 2] => 170
[1, 6, 3, 8, 7, 2, 5, 4] => 154
[1, 6, 4, 5, 2, 3, 8, 7] => 151

3  circuits complets ont ete examines
```

Figure 4: Comparaisons empiriques du nombre de circuits qui sont explorés au complet pour les trois algorithmes (exhaustif, *branch-and-bound* et *best-first*) dans le cas d'un graphe aléatoire avec huit (8) sommets

```
type FilePriorite = ptr ...;

procedure creerFP() returns FilePriorite fp
# POSTCONDITION
#   fp = {}

procedure ajouter( FilePriorite fp, int prio, Sequence elem )
# POSTCONDITION
#  fp = fp' U {(prio, elem)}

procedure estVide( FilePriorite fp ) returns bool r
# POSTCONDITION
#   r <=> fp = {}

procedure obtenirMin( FilePriorite fp, ref int prio, ref Sequence elem )
# PRECONDITION
#   ~estVide(fp)
# POSTCONDITION
#   (prio, elem) IN fp' & prio = MINIMUM( (p, e) IN fp' :: p )
#   fp = fp' - {(prio, elem)}
```

Type abstrait de données 2: Type et procédures pour la spécification d'une *classe d'objets* pour des files de priorité (mise oeuvre omise)

bien que le temps d'exécution semble diminué, la différence ne semble pas aussi significative qu'espéré, ce qui est probablement dû aux surcoûts introduits par la manipulation des files de priorité.[6]

En fait, pour vérifier que le nombre de chemins explorés est toujours réduit, il faudrait faire une analyse plus détaillée et formelle (que nous ne ferons pas ici) pour s'assurer qu'il n'existe pas de *contre-exemples* à cette propriété, c'est-à-dire, vérifier qu'il n'existe pas de graphes tels que le choix de ce qui semble *a priori* plus prometteur (*best-first*) conduise effectivement à effectuer de moins bons choix.

En fait, la question de l'existence de tels graphes s'applique aussi à l'analyse de la complexité asymptotique de ces deux algorithmes *branch-and-bound*, dans la mesure où l'identification du pire cas n'est plus aussi simple que lors de l'énumération des cycles hamiltoniens : un graphe complet n'est pas, en soit, le pire cas, car le comportement de chacun des algorithmes *branch-and-bound* va dépendre aussi des valeurs qui sont associées aux différents arcs du graphe, valeurs qui vont faire en sorte que certains chemins seront ou non explorés. En fait, il existe des exemples de graphe où l'exploration d'un chemin qui semble plus prometteur *ne conduit pas* au chemin optimal, ce qui a donc pour effet d'explorer la majeure partie de l'espace des solutions.

Dans le cas de l'algorithme *branch-and-bound* sans file de priorité, si on suppose le pire du pire (surestimation), à savoir que les estimés sont toujours inférieurs à `coutCheminMin` — mais ce qui n'est pas clair est de savoir quel graphe conduirait à une telle situation —, on obtient alors une complexité semblable à celle pour la recherche des chemins hamiltoniens (sans impression de chacun des chemins), à savoir $O((n-1)!)$.

Pour ce qui est de la version avec file de priorité, l'analyse asymptotique du pire cas est encore moins claire. Ce qui est certain, c'est qu'on peut constater de façon empirique, et ce pour les deux algorithmes *branch-and-bound*, que le temps d'exécution devient rapidement très long lorsqu'on augmente le nombre de sommets du graphe. En fait, sur la machine

[6] Dans le programme tests, les files de priorité sont mises en oeuvre de façon naïve, à l'aide d'un tableau de taille fixe. L'espace supplémentaire utilisé ne l'est pas de façon optimale non plus. De plus, le clonage des séquence est aussi coûteux et pourrait être amélioré.

`arabica` ou sur un PC avec Linux, le nombre maximum de sommets pouvant être traités est de l'ordre de moins d'une vingtaine d'éléments (17 ou 18), ce qui semble effectivement indiquer une complexité exponentielle ou factorielle.

Pour mieux comprendre les différences et similitudes entre algorithmes avec marche arrière et algorithmes *branch-and-bound*, il faut consulter le chapitre sur ces algorithmes, qui présente la forme générique d'un algorithme avec marche arrière et celle d'un algorithme *branch-and-bound*, illustrant le fait qu'un algorithme avec marche arrière, sous certaines conditions, peut être vu simplement comme un cas spécial d'algorithme *branch-and-bound*.

4 Versions parallèles des solutions au problème du voyageur de commerce

4.1 Algorithme pour une machine idéale (sans limite de ressources)

Une première façon simple, et naïve, de paralléliser la recherche d'une solution pour le problème du voyageur de commerce peut être obtenue en effectuant, en parallèle, l'exploration des diverses alternatives. Une telle solution est présentée à l'algorithme 4. Cette solution correspond à une version parallèle de l'algorithme 2, c'est-à-dire l'algorithme *branch-and-bound*, mais sans ordre particulier d'exploration des alternatives (pas la stratégie *best-first search*).

Notons que dans l'algorithme 4, tout comme dans l'algorithme 2, le chemin de coût minimum et le coût qui lui est associé sont définis à l'aide de variables *globales* (fonctions auxiliaires 3). Toutefois, ces variables doivent maintenant être mises à jour en étant *protégées* par un sémaphore (`semMajMin`), et ce de façon à permettre aux divers processus de partager et modifier correctement ces variables.

Pour une machine parallèle idéale et sans limite de ressources, cette solution semble intéressante au niveau *du temps d'exécution* puisqu'elle va générer un grand nombre de processus qui effectueront *en parallèle* l'exploration des divers chemins possibles. En d'autres mots, les diverses alternatives (les divers chemins) pourront donc toutes être explorées de façon concurrente.

À première vue, on pourrait donc croire que le temps d'exécution sera linéaire ($\Theta(n)$), c'est-à-dire le temps requis pour générer, en parallèle, les divers circuits qui correspondent à des chemins de longueur n. Toutefois, lorsqu'un chemin est exploré et génère un circuit approprié vers le sommet de départ, il faut aussi effectuer, si nécessaire (nouveau minimum), la mise-à-jour des variables `coutMin` et `cheminCoutMin`. Pour ce faire, il faut obtenir l'accès au sémaphore `semMajMin`, tel que cela est indiqué dans le code pour la procédure `majCheminMin` (fonctions auxiliaires 3). Or, puisque plusieurs processus vont avoir besoin d'accéder à ce sémaphore de façon concurrente, le sémaphore deviendra alors un *goulot d'étranglement*, réduisant de façon importante le parallélisme réel — on verra plus loin comment minimiser le nombre d'accès à ce sémaphore — conduisant à la limite à une séquentialisation des mises à jour.

Nonobstant la question du goulot d'étranglement pour le sémaphore, en termes de coût, cette solution parallèle n'est pas intéressante car elle génère un *très grand* nombre de processus. On peut se convaincre facilement que le nombre de feuilles de l'arbre, donc le nombre maximum de processeurs actifs en même temps, sera $\Theta((n-1)!)$. Ainsi, tel que cela est illustré à la figure 5 (arbre des appels récursifs), le nombre de processus au niveau k sera décrit par l'équation de récurrence suivante, en supposant que la racine est au niveau 0 et qu'elle ne compte pas pour un processus :

- $P_k(1) = n - 1$
- $P_k(k) = (n-k)P_k(k-1)$ (pour $k > 1$)

```
# Operation auxiliaire pour activation parallele dans le co.
op visiterSommet( Sequence chemin, int n, int s )

procedure explorerCircuits( Sequence chemin, int n )
{
  if (longueur(chemin) == n) {
    if ( arcExiste(queue(chemin), tete(chemin)) ) {
      int cout = coutChemin(chemin) + W[queue(chemin)][tete(chemin)];
      majCheminMin( chemin, cout ); # On a trouve un circuit de cout inferieur.
    }
  } else {
    # On explore (en parallele), parmi tous les sommets n'ayant pas encore ete
    # visites, ceux pour lesquels un arc existe et qui semblent prometteurs.
    co [s = 1 to n st ~dejaVisite(chemin, s)
                    & arcExiste(queue(chemin), s)
                    & coutEstime(chemin, s, n) < coutCheminMin]
      visiterSommet( chemin, n, s );
    oc
  }
}

proc visiterSommet( chemin, n, s )
{
  # On sait qu'une arc existe et que ca semble prometteur: on explore.
  Sequence ch = cloner(chemin);
  ajouterEnQueue( ch, s );
  explorerCircuits( ch, n );
}

procedure trouverCircuitMin( int sommetDepart, int n )
{
  Sequence chemin = creer();
  ajouterEnQueue(chemin, sommetDepart);

  # On explore l'ensemble des circuits atteignables du sommet de depart.
  explorerCircuits(chemin, n);

  # On imprime le resultat.
  imprimer( cheminMin ); write( " =>", coutCheminMin );
}
```

Algorithme 4: Algorithme *branch-and-bound parallèle* pour le voyageur de commerce (première version, sans *best-first search*)

```
Sequence cheminMin;
int coutCheminMin = INFINI;
sem semMajMin     = 1;

procedure majCheminMin( Sequence chemin, int cout )
{
  P(semMajMin);
  if (cout < coutCheminMin) {
    cheminMin = chemin;
    coutCheminMin = cout;
  }
  V(semMajMin);
}
```

Fonctions auxiliaires 3: Variables globales et procédure pour la mise à jour du chemin de coût minimum pour l'algorithme *branch-and-bound* parallèle

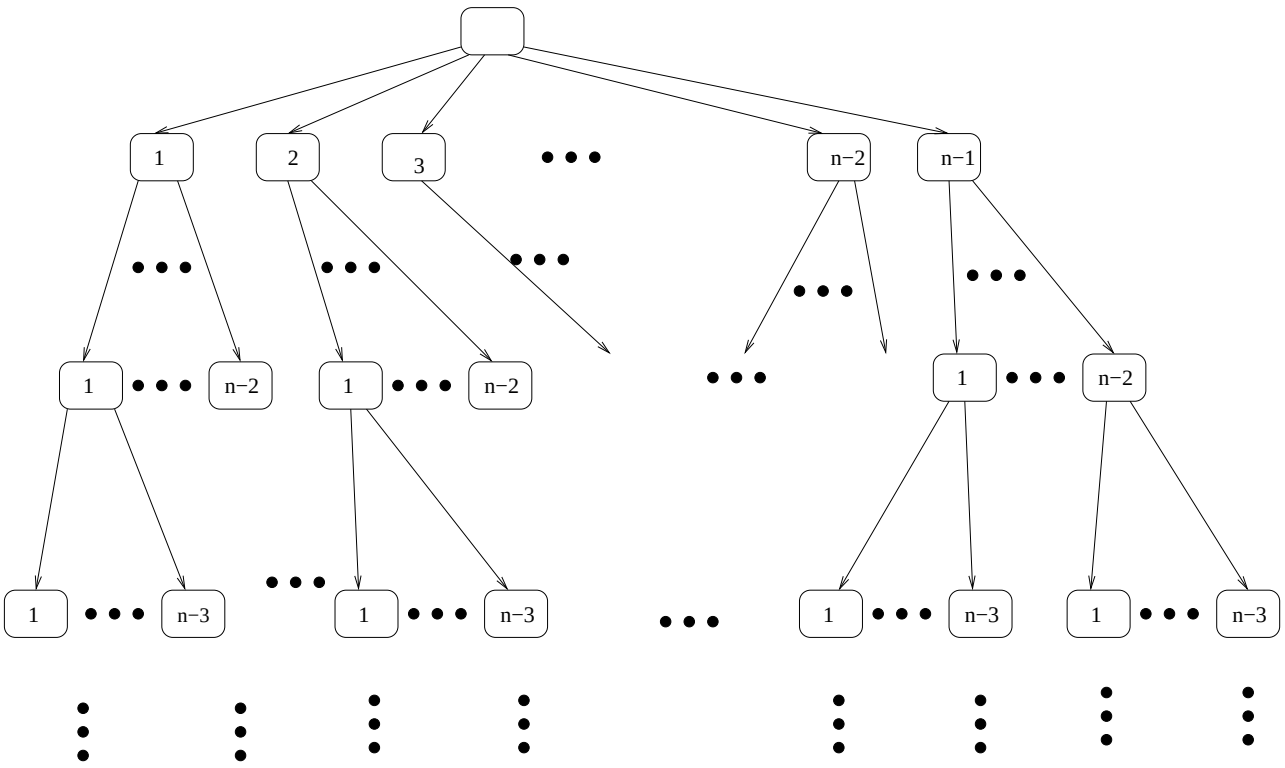

Figure 5: Arbre des appels récursifs parallèles pour l'exploration des chemins d'un graphe à n sommets

Le nombre de processus de niveau $n-1$ (le niveau des feuilles de l'arbre, puisqu'on suppose la racine de niveau 0) sera donc le suivant :

$$\begin{aligned} P_k(n-1) &= (n-(n-1))P_k(n-2) \\ &= 1 \times 2P_k(n-3) \\ &= \ldots \\ &= 1 \times 2 \times \ldots \times (n-2)P(1) \\ &= 1 \times 2 \times \ldots \times (n-2) \times (n-1) \\ &= (n-1)! \end{aligned}$$

Évidemment, le nombre *total* de processus créés par le programme sera encore plus grand :

$$\begin{aligned} \sum_{i=1}^{n-1} P_k(i) &= \sum_{i=1}^{n-1} i! \\ &= 1! + 2! + 3! + \ldots + (n-2)! + (n-1)! \\ &\in \Omega((n-1)!) \\ &\in O(n!) \end{aligned}$$

Comme on l'a fait précédemment, en supposant qu'un processus parent est bloqué pendant que ses enfants travaillent (donc aucun processeur n'a besoin d'être associé au processus parent quand ses enfants sont actifs), on peut conclure que le nombre de processeurs est $\Theta((n-1)!)$, processeurs utilisés pour vérifier en parallèle les divers chemins complets. En ignorant le goulot d'étranglement pour l'accès au sémaphore, donc en supposant un temps $\Theta(n)$, le coût de cet algorithme parallèle est donc $\Theta(n!)$, ce qui est évidemment un coût inacceptable pour de grandes valeurs de n.

Il est clair que si on exécute cet algorithme sur une machine possédant un nombre limité de processeurs, le comportement de l'algorithme résultant ne sera pas très intéressant à cause du grand nombre de processus qui sont générés. De plus, comme dans le cas séquentiel, il serait aussi préférable de pouvoir explorer en premier les alternatives qui semblent les plus prometteuses (*best-first search*) plutôt que de simplement explorer toutes les alternatives en parallèle. Une telle solution est présentée à la prochaine section.

Réduction du nombre de conflits d'accès au sémaphore

Avant d'explorer une autre solution parallèle basée sur l'approche *branch-and-bound*, soulignons qu'il est possible de *réduire* le nombre d'accès au sémaphore `semMajMin` en y accédant uniquement lorsque cela semble vraiment nécessaire, c'est-à-dire, lorsque le coût du circuit *semble* effectivement inférieur au coût minimum courant. On appelle cette façon de procéder la technique du "*double check*" et, dans ce cas, elle conduirait à modifier la procédure `majCheminMin`

comme suit :

```
procedure majCheminMin( Sequence chemin, int cout )
{
  if (cout < coutCheminMin) {
    # Le cout semble inferieur : on accede de facon exclusive au semaphore
    P(semMajMin);
    if (cout < coutCheminMin) {
      # On verifie si le cout est encore inferieur
      cheminMin = chemin;
      coutCheminMin = cout;
    }
    V(semMajMin);
  }
}
```

En d'autres mots, le processus ne tente d'accéder au sémaphore que si le `cout` calculé est actuellement inférieur à `coutCheminMin`. Dans ce cas, le processus tente d'obtenir le sémaphore. Dans ce cas, il est possible que, entre la première comparaison (non protégée) des coûts et le moment où le sémaphore a été demandé et obtenu, un autre processus ait modifié la variable `coutCheminMin`, d'où la nécessité d'une nouvelle vérification après l'obtention du sémaphore — notons qu'une mise-à-jour ne peut avoir pour effet que de diminuer la valeur de cette variable, pas de l'augmenter.

4.2 Algorithme pour une machine avec un nombre limité de processeurs (approche de style "sac de tâches")

Pour une machine possédant un nombre limité de processeurs, il est souvent préférable d'utiliser un algorithme parallèle qui génère un nombre *limité* de processus, nombre de processus qui correspond, à tout le moins en termes d'ordre de grandeur, au nombre de processeurs. De plus, dans le cas du problème du voyageur de commerce avec l'algorithme *branch-and-bound*, comme on désire assurer que les alternatives les plus prometteuses soient explorées en premier, l'utilisation d'une file de priorité, comme dans l'exemple de l'algorithme 3, semble donc appropriée. Ces deux aspects conduisent tout naturellement à l'utilisation d'une approche de style "sac de tâches", tel que cela est illustré par l'algorithme 5.

Cette solution parallèle possède les caractéristiques suivantes :

- Un certain nombre (`nbProcs`) de processus d'exploration (`explorerCheminsDeSac`) sont créés, nombre spécifié au moment de l'appel du programme : algorithme 5 ($1^{ère}$ partie).
- Les divers processus partagent un sac de tâches. Ce sac, réalisé par une file de priorité, contient les divers chemins possibles à explorer, chemins *qui semblaient* prometteurs au moment où ils ont initialement été rencontrés et ajoutés dans le sac. Les ajouts dans le sac sont effectués dans la procédure `explorerCircuits`, à l'aide d'un appel à la procédure `ajouterCheminAExplorer` (définie dans les fonctions auxiliaires 4).
- Lorsqu'un tâche est retirée du sac (processus `explorerCheminsDeSac`), il est possible qu'un chemin de coût inférieur ait été identifié depuis son ajout. Un test (`if (estime < coutCheminMin)`) assure alors que le chemin à explorer semble toujours prometteur avant d'amorcer son exploration (avec un appel à `explorerCircuits`).
- Le sac de tâches doit évidemment être manipulé de façon atomique. Un sémaphore (`semFP`) est donc utilisé pour protéger les accès à la file de priorité associée, par exemple, avant l'appel à `obtenirMin` (dans le corps du processus) ou avant l'appel à `ajouter` (dans la procédure `ajouterCheminAExplorer`).

```
# Lecture et verification du nombre de processus a creer.
int nbProcs; getarg(2, nbProcs);
if (nbProcs <= 0) { write("*** Erreur: nbProcs <= 0" ); stop(1); }

procedure explorerCircuits( Sequence chemin, int n )
{
  if (longueur(chemin) == n) {
    if ( arcExiste(queue(chemin), tete(chemin)) ) {
      int cout = coutChemin(chemin) + W[queue(chemin)][tete(chemin)];
      majCheminMin( chemin, cout );
    }
  } else {
    # On ajoute dans le sac des sommets a visiter, ceux pas encore visites et pour
    # lesquels un arc existe et qui, de plus, semblent prometteurs.
    for [s = 1 to n st ~dejaVisite(chemin, s) & arcExiste(queue(chemin), s)] {
      int estime = coutEstime(chemin, s, n);
      if ( coutEstime(chemin, s, n) < coutCheminMin ) {
        Sequence ch = cloner(chemin);
        ajouterEnQueue( ch, s );
        ajouterCheminAExplorer( ch, estime );
      }
    }
  }
}

process explorerCheminsDeSac[numProc = 1 to nbProcs]
{
  while (true) {
    P(nbCheminsAjoutes);
    P(semFP)
    if (estVide(cheminsAExplorer)) {
      V(semFP);
      return;
    }
    # Il reste un chemin a explorer dans la file de priorite.
    int estime;
    Sequence chemin;
    obtenirMin( cheminsAExplorer, estime, chemin )
    V(semFP);
    if ( estime < coutCheminMin ) {
      # On explore uniquement si l'estime est encore interessant
      explorerCircuits( chemin, n );
    }
  }
}

procedure trouverCircuitMin( int sommetDepart, int n )
{
  Sequence chemin = creer();
  ajouterEnQueue( chemin, sommetDepart );
  explorerCircuits( chemin, n );
}
```

Algorithme 5: Algorithme *branch-and-bound parallèle* avec nombre limité de processus pour le voyageur de commerce (version avec sac de tâches) ($1^{ère}$ partie)

```
#
# Programme principal.
#

trouverCircuitMin( 1, n );

final {
  imprimer( cheminMin ); write( " =>", coutCheminMin );

  # On debloque les processus qui attendent qu'un nouvel element soit
  # ajoute dans le sac des chemins a explorer, pour assurer une terminaison
  # "propre" du programme.
  for [i = 1 to nbProcs] {
    V(nbCheminsAjoutes);
  }
}
```

Algorithme 5: Programme principal, avec réactivation des processus pour terminaison correcte du programme ($2^{ième}$ partie)

```
# Sac des chemins a explorer et son semaphore d'acces exclusif.
FilePriorite cheminsAExplorer = creerFP();
sem semFP = 1;

# Semaphore de signalisation et de comptage pour (re-)activation des processus:
# sa valeur indique toujours le nombre d'elements presents dans le
# sac des chemins a explorer.
sem nbCheminsAjoutes = 0;

procedure ajouterCheminAExplorer( Sequence ch, int cout )
# Ajout (atomique) d'un nouveau chemin et de son cout dans le sac
# des chemins a explorer.
{
  P(semFP);
  ajouter(cheminsAExplorer, cout, ch);
  V(nbCheminsAjoutes); # Signal qu'un nouvel element vient d'etre ajoute.
  V(semFP);
}
```

Fonctions auxiliaires 4: Variables et procédure de manipulation du sac de tâches (file de priorité)

- Finalement, un sémaphore de signalisation et de comptage (*counting semaphore*) (et non un sémaphore d'exclusion mutuelle) est aussi utilisé pour assurer qu'un processus ne se termine de façon permanente (avec `return`) que si le sac est vide *et que si on est certain* qu'aucun *autre chemin n'y sera ajouté* par la suite — le sac pourrait être vide temporairement, parce que tous les éléments ont été retirés sans qu'aucun autre élément n'ait encore eu le temps d'être ajouté.

  Le sémaphore `nbCheminsAjoutes` joue ce rôle de signalisation et a la propriété que sa valeur indique toujours le nombre d'éléments présents dans le sac de tâche. La synchronisation "`P(nbCheminsAjoutes)`" au début de la boucle `while` du processus assure que celui-ci ne tentera d'obtenir un élément du sac que si un tel élément existe. Plus précisément, le test pour déterminer si un élément est présent ou non dans le sac ne se fera que lorsqu'un tel élément aura été ajouté. La seule exception sera à la fin du programme : lorsque tous les processus seront bloqués, en attente de l'ajout d'un élément, la clause `final` du programme principal (voir algorithme 5 ($2^{ième}$ partie)) sera exécutée. Son exécution aura pour effet de réactiver temporairement les divers processus (appel à `V` dans la boucle `for`), lesquels détecteront alors que le sac est vide (`estVide(cheminsAExplorer)`) et termineront correctement leur exécution (`return`).

4.3 Comparaisons empiriques entre les deux solutions parallèles

Les deux solutions parallèles ont été testées sur des graphes complets (pire cas) générés aléatoirement. Comme on peut s'en douter, la première solution, avec un grand nombre de processus, n'est pas du tout intéressante lorsqu'on l'exécute sur la machine `arabica` (avec un ou plusieurs processeurs) ou sur une machine Linux (un processeur). En fait, pour un graphe comportant neuf (9) sommets, le temps d'exécution devient rapidement très long (aucune réponse après quelques minutes). Par contre, la deuxième solution semble pouvoir traiter sans problème des graphes allant jusqu'à environ 17 sommets. Au-delà, le nombre d'éléments présents dans le sac de tâches devient très grand (supérieur à 100 000) et, même lorsqu'on augmente la taille du sac, aucun résultat n'est produit après plusieurs minutes.

5 Solution au problème du voyageur de commerce avec programmation dynamique

Pour que la programmation dynamique soit utile et appropriée pour résoudre un problème, il faut tout d'abord que la solution du problème fasse apparaître une *sous-structure optimale*. Intuitivement, ceci signifie qu'une solution optimale au problème global comporte des solutions optimales aux sous-problèmes. De plus, il faut que des *problèmes superposés* apparaissent, c'est-à-dire qu'un même sous-problème ait à être résolu plusieurs fois si on utilise une approche diviser-pour-régner simple.

Examinons tout d'abord la question de la sous-structure optimale. Soit $G = (V, E)$ un graphe avec n sommets. Soit v_1 le sommet de départ. Soit alors un circuit optimal O ayant v_1 comme point de départ et parcourant chacun des n sommets. Le circuit O aura donc la structure suivante : $O = [v_1, v_{i_1}, \ldots, v_{i_{n-1}}]$. Le chemin allant de v_{i_1} à $v_{i_{n-1}}$ puis retournant à v_1 passe par l'ensemble des autres villes ($V - \{v_1\}$, c'est-à-dire toutes les villes à l'exclusion de v_1). Ce chemin est nécessairement optimal pour ce sous-ensemble de villes. Si ce n'était pas le cas, on pourrait alors trouver un autre chemin passant par ces villes, retournant à v_1 et de moindre coût. Cet autre chemin pourrait alors être utilisé à la place de la partie de O débutant après v_1 pour obtenir un chemin hamiltonien sur l'ensemble des villes qui serait de moindre coût, contredisant le fait que le chemin global O était optimal.

Quant à la question des problème superposés, elle sera examinée un peu plus loin, lorsqu'on examinera une solution purement récursive basée sur une approche diviser-pour-régner.

Version simplifiée du problème

Dans ce qui suit, nous allons examiner le problème de trouver simplement *le coût* d'un circuit hamiltonien optimal. En d'autres mots, nous ne chercherons pas à identifier le circuit optimal lui-même mais uniquement son coût, et ce dans le but de simplifier la présentation des diverses solutions nous conduisant à la solution de programmation dynamique classique.

Nous allons supposer, comme dans les autres sections, que le graphe possède n sommets et que les coûts des divers arcs sont représentés par une matrice W. Comme ces valeurs restent inchangées dans les diverses versions de l'algorithme et qu'elles ne sont jamais modifiées dans les sous-procédures, elles seront définies comme variables globales plutôt que d'être passées explicitement en paramètres.

Type `Ensemble` et opérations associées

```
type Ensemble = ...;

procedure creerVide( int n ) returns Ensemble s
# POSTCONDITION
#   s = {}

procedure creerPlein( int n ) returns Ensemble s
# POSTCONDITION
#   s = {1, 2, ..., n}

procedure estVide( Ensemble s ) returns bool r
# POSTCONDITION
#   r <=> s == {}

procedure cardinalite( Ensemble s ) returns int c
# POSTCONDITION
#   c = |s|

procedure estElement( Ensemble s, int e ) returns bool r
# POSTCONDITION
#   r <=> e IN s

procedure ajouter( Ensemble s, int e ) returns Ensemble s2
# POSTCONDITION
#   s2 = s U {e}

procedure supprimer( Ensemble s, int e ) returns Ensemble s2
# POSTCONDITION
#   s2 = s - {e}

procedure imprimer( Ensemble s )
# POSTCONDITION
#   Chacun des elements de s a ete imprime sur stdout.
```

Type abstrait de données 3: Type et opérations pour la spécification d'une *collection de valeurs* pour des ensembles de sommets

Dans les algorithmes qui suivent, nous aurons besoin de manipuler des ensembles de sommets. Le type abstrait 3 présente l'interface des opérations de manipulation d'ensembles que nous utiliserons. Quelques points importants à souligner concernant ce type abstrait sont les suivants :

- Contrairement aux séquences de la partie précédente, ce type et les opérations associées ne définissent pas des objets (classe d'objets), mais bien des valeurs (on parle parfois, en Java entre autres, d'"objets immuables", ce qui me semble toutefois une contradiction dans les termes, c'est-à-dire, un *oxymoron*). En d'autres mots, les opérations telles `ajouter` et `retirer` sont des *fonctions* (au sens pur et mathématique du terme) qui reçoivent un ensemble et un élément et retournent un résultat qui est un nouvel ensemble — l'ensemble reçu en argument demeure donc inchangé.

- Les éléments manipulés par ces ensembles sont des petits entiers, compris entre 1 et `n`. En d'autres mots, ces ensembles définissent et réalisent une forme de `bitsets`. Ceci est possible et acceptable dans le contexte de notre problème puisqu'on veut manipuler des ensembles de sommets (villes) et qu'on suppose (sans perte de généralité) que les sommets sont identifiés simplement par des nombres compris entre 1 et n (n, de par la nature du problème, étant relativement petit, disons inférieur ou égal à 32).

- La restriction sur les éléments des ensembles (petits entiers compris entre 1 et n) fait en sorte qu'un `Ensemble` peut être représenté efficacement par un simple `int` (la mise en oeuvre détaillée est omise). Toutes les opérations indiquées dans la spécification du type abstrait 3 (à l'exception de `creerPlein` et, évidemment, de `imprimer`) peuvent donc être exécutées en temps $\Theta(1)$.

5.1 Algorithme diviser-pour-régner purement récursif

La première étape pour bien comprendre un problème et, si approprié, définir un algorithme de programmation dynamique, est souvent de produire un algorithme purement récursif basé sur une stratégie diviser-pour-régner. La première difficulté est alors, pour certains problèmes plus complexes, de bien comprendre et identifier/formuler la notion appropriée de "problème" : rappelons que pour qu'un algorithme purement récursif soit utilisable, il faut que les sous-problèmes générés par l'étape de décomposition soient *similaires* au problème initial. Il faut donc parfois, dans un premier temps, trouver une formulation légèrement modifiée du problème initial qui permette une telle décomposition en sous-problèmes similaires.

Dans notre cas, reformulons le problème initial comme suit : on désire trouver un "*sous-circuit*" de coût minimal qui sera une extension (une continuation) d'un chemin de sommets déjà visités qui débute au sommet `v1` et dont le dernier sommet est `vi`, sous-circuit qui passera par chacun des sommets d'un ensemble `a`. Les sommets s appartenant à l'ensemble V de départ mais pas à `a` ($s \in V - \{\texttt{a}\}$) seront donc des sommets qui feront déjà partie d'une portion initiale de chemin qu'on désire étendre en passant par les sommets de l'ensemble `a` pour ultimement retourner à `v1`, et ce tout en reliant ce sous-circuit au sommet `vi`.

La figure 6 illustre un problème possible obtenu suite à la décomposition du problème initial, illustrant le rôle de `v1`, `vi` et de l'ensemble `a`. La figure 7, quant à elle, illustre ensuite les trois (3) sous-problèmes qui peuvent être générés pour le problème de la figure 6.

L'en-tête d'une procédure solutionnant ce problème est donc le suivant, sa mise en oeuvre détaillée étant présentée à l'algorithme 6 :

```
procedure coutSousCircuitMinRec( int v1, int vi, Ensemble a ) returns int c
```

Examinons la façon dont la stratégie diviser-pour-régner peut être utilisée :

- Cas de base = `a` est vide, c'est-à-dire, toutes les villes ont été visitées : comme le dernier sommet visité (du chemin des villes déjà visitées) est `vi` et qu'on veut retourner au sommet `v1`, le coût associé au sous-circuit est alors simplement le coût de l'arc reliant `vi` à `v1` (si un tel arc n'existe pas, le coût est simplement $+\infty$ — `INFINI` dans le programme MPD).

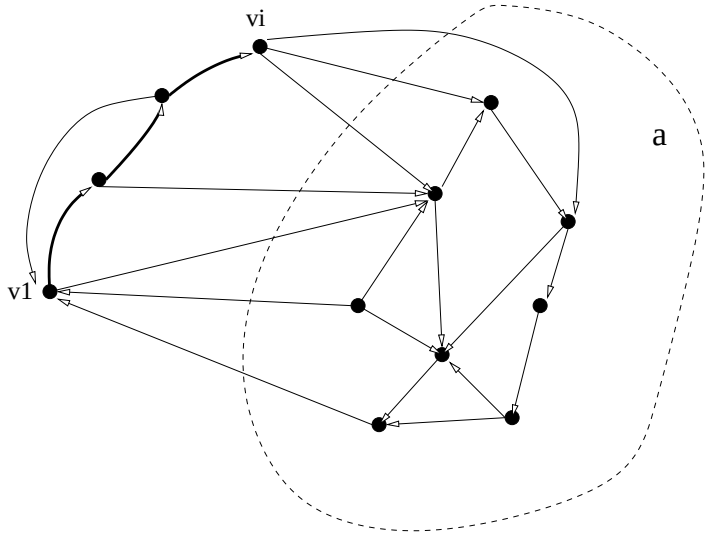

Figure 6: Un exemple de problème généré suite à la décomposition du problème initial

```
procedure coutSousCircuitMinRec( int v1, int vi, Ensemble a ) returns int c
#
# Cette procedure determine le cout minimum d'un sous-circuit ayant les
# proprietes suivantes:
#   - Ce sous-circuit sera la continuation d'un chemin de sommets
#     deja visites dont le dernier sommet est vi;
#   - Le sous-circuit se termine (cycle) au sommet de depart (v1);
#   - Ce sous-circuit visite une fois chacun des sommets de l'ensemble a.
#
# PRECONDITION
#   ~estElement(v1, a) & ~estElement(vi, a)
{
  if (estVide(a)) {
    c = W[vi][v1]
  } else {
    c = INFINI;
    for [vj = 1 to n st estElement(a, vj) & arcExiste(vi, vj)] {
      int cout = W[vi][vj] + coutSousCircuitMinRec(v1, vj, supprimer(a, vj));
      c = min(c, cout);
    }
  }
}


procedure coutCircuitMin( int v1 ) returns int c
{
  Ensemble v = creerPlein(n);

  c = coutSousCircuitMinRec( v1, v1, supprimer(v, v1) );
}
```

Algorithme 6: Algorithme purement récursif (diviser-pour-régner) pour trouver le coût d'un circuit hamiltonien minimum

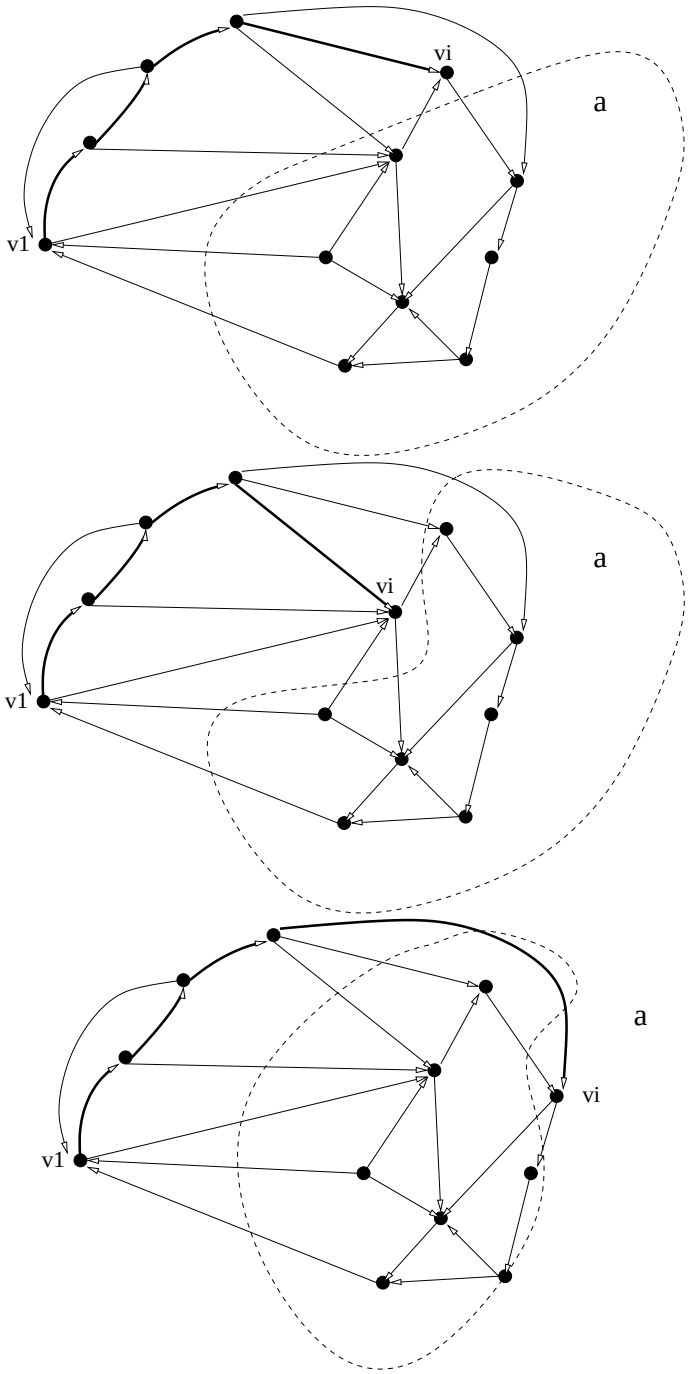

Figure 7: Les trois sous-problèmes générés pour le problème de la figure 6

- Cas récursif : un sous-problème consiste à trouver un sous-circuit de coût minimum sur l'ensemble `a - {vj}`, et ce pour un `vj` arbitraire. Comme on désire obtenir un minimum global, il faut alors identifier les coûts de tels sous-circuits pour tous les `vj` possibles appartenant à `a`. Pour un `vj` donné, le coût est donc le suivant :

  $$\texttt{W[vi][vj]} + \texttt{coutSousCircuitMinRec}(\texttt{v1}, \texttt{vj}, \texttt{supprimer}(\texttt{a}, \texttt{vj}))$$

  La combinaison des solutions de ces divers sous-problèmes s'effectue ensuite en calculant le minimum parmi les résultats obtenus, et ce à l'aide de la boucle `for` dans laquelle on effectue des appels à `min`.

  Notons que dans la boucle `for` du cas récursif, la condition `arcExiste(vi, vj)` assure qu'on effectuera le calcul et la mise à jour du minimum pour le sommet `vj` uniquement si un arc existe bien de `vi` vers ce `vj`. Notons toutefois que cette condition aurait aussi pû être omise sans aucun effet sur le résultat produit : si on supprime la condition du quantificateur, le calcul sera effectué même si aucun arc n'existe, mais comme `W[vi][vj]` sera alors $+\infty$ et que cette valeur est un *élément neutre* de l'opérateur `min`, le résultat sera le même que si on avait omis de traiter ce sommet. Notons aussi que, dans le pire des cas, à savoir un graphe complet, `arcExiste(vi, vj)` retournera `true` pour n'importe quel sommet `vi` et `vj`.

Soit `v` l'ensemble complet des sommets du graphe et soit `v1` le sommet de départ. La solution au problème initial de trouver un circuit hamiltonien de coût minimum pourra alors être obtenue simplement par l'appel suivant :

```
coutSousCircuitMinRec( v1, v1, supprimer(v, v1) );
```

Sous-problèmes superposés

La présence de sous-problèmes superposés commence à apparaître lorsque le graphe contient cinq (5) sommets ou plus. Un appel à `coutCircuitMin(1)` pour `n=5` générera les appels récursifs indiqués à la figure 8, où le niveau d'indentation représente le niveau de l'appel, ce qui permet de déterminer, pour chacun des appels, les appels récursifs effectués pour traiter cet appel.

Pour un graphe à cinq sommets, deux appels de la procédure sont effectués avec les séries d'arguments `(1, 3, {2})`, `(1, 4, {2})` et `(1, 5, {2})`. Pour un graphe à six sommets, de tels appels (avec un ensemble singleton) apparaissent six fois chacun (pour $3, 4, 5, 6$), alors que des appels de la forme `(1, 4, {2 3})` (i.e., avec une composante ensemble dont la cardinalité est de 2) apparaissent deux fois chacun, etc. On a donc bien la présence de problèmes superposés, puisqu'on doit résoudre plusieurs fois le même sous-problème (appels avec la même série d'arguments).

Analyse de l'algorithme

Il est clair que la présence de problèmes superposés va conduire à un algorithme dont le temps d'exécution ne sera pas intéressant. Dans le cas présent, le temps d'exécution de la procédure `coutSousCircuitMinRec` peut être décrit par les équations de récurrence suivantes, où m dénote la taille de l'ensemble de villes à parcourir (c'est-à-dire, m = taille de l'ensemble `a`) — le facteur m apparaît devant $T(m-1)$ car le nombre de `vj` tel que `estElement(a, vj)` $= m =$ taille de `a` et, dans le pire cas (graphe complet), il existe un arc reliant `vj` à `vi` :

$$\begin{aligned} T(m) &= mT(m-1) + \Theta(1) \\ T(0) &= \Theta(1) \end{aligned}$$

La solution (par substitution) de cette équation de récurrence nous permet alors de conclure que le temps d'exécution de `coutSousCircuitMinRec` est $T(m) \in \Theta(m!)$. Or, l'appel

```
coutSousCircuitMinRec( 1, 1, { 2 3 4 5 } )
>> coutSousCircuitMinRec( 1, 2, { 3 4 5 } )
>> >> coutSousCircuitMinRec( 1, 3, { 4 5 } )
>> >> >> coutSousCircuitMinRec( 1, 4, { 5 } )
>> >> >> >> coutSousCircuitMinRec( 1, 5, { } )
>> >> >> coutSousCircuitMinRec( 1, 5, { 4 } )
>> >> >> >> coutSousCircuitMinRec( 1, 4, { } )
>> >> coutSousCircuitMinRec( 1, 4, { 3 5 } )
>> >> >> coutSousCircuitMinRec( 1, 3, { 5 } )
>> >> >> >> coutSousCircuitMinRec( 1, 5, { } )
>> >> >> coutSousCircuitMinRec( 1, 5, { 3 } )
>> >> >> >> coutSousCircuitMinRec( 1, 3, { } )
>> >> coutSousCircuitMinRec( 1, 5, { 3 4 } )
>> >> >> coutSousCircuitMinRec( 1, 3, { 4 } )
>> >> >> >> coutSousCircuitMinRec( 1, 4, { } )
>> >> >> coutSousCircuitMinRec( 1, 4, { 3 } )
>> >> >> >> coutSousCircuitMinRec( 1, 3, { } )
>> coutSousCircuitMinRec( 1, 3, { 2 4 5 } )
>> >> coutSousCircuitMinRec( 1, 2, { 4 5 } )
>> >> >> coutSousCircuitMinRec( 1, 4, { 5 } )
>> >> >> >> coutSousCircuitMinRec( 1, 5, { } )
>> >> >> coutSousCircuitMinRec( 1, 5, { 4 } )
>> >> >> >> coutSousCircuitMinRec( 1, 4, { } )
>> >> coutSousCircuitMinRec( 1, 4, { 2 5 } )
>> >> >> coutSousCircuitMinRec( 1, 2, { 5 } )
>> >> >> >> coutSousCircuitMinRec( 1, 5, { } )
>> >> >> coutSousCircuitMinRec( 1, 5, { 2 } )
>> >> >> >> coutSousCircuitMinRec( 1, 2, { } )
>> >> coutSousCircuitMinRec( 1, 5, { 2 4 } )
>> >> >> coutSousCircuitMinRec( 1, 2, { 4 } )
>> >> >> >> coutSousCircuitMinRec( 1, 4, { } )
>> >> >> coutSousCircuitMinRec( 1, 4, { 2 } )
>> >> >> >> coutSousCircuitMinRec( 1, 2, { } )
>> coutSousCircuitMinRec( 1, 4, { 2 3 5 } )
>> >> coutSousCircuitMinRec( 1, 2, { 3 5 } )
>> >> >> coutSousCircuitMinRec( 1, 3, { 5 } )
>> >> >> >> coutSousCircuitMinRec( 1, 5, { } )
>> >> >> coutSousCircuitMinRec( 1, 5, { 3 } )
>> >> >> >> coutSousCircuitMinRec( 1, 3, { } )
>> >> coutSousCircuitMinRec( 1, 3, { 2 5 } )
>> >> >> coutSousCircuitMinRec( 1, 2, { 5 } )
>> >> >> >> coutSousCircuitMinRec( 1, 5, { } )
>> >> >> coutSousCircuitMinRec( 1, 5, { 2 } )
>> >> >> >> coutSousCircuitMinRec( 1, 2, { } )
>> >> coutSousCircuitMinRec( 1, 5, { 2 3 } )
>> >> >> coutSousCircuitMinRec( 1, 2, { 3 } )
>> >> >> >> coutSousCircuitMinRec( 1, 3, { } )
>> >> >> coutSousCircuitMinRec( 1, 3, { 2 } )
>> >> >> >> coutSousCircuitMinRec( 1, 2, { } )
>> coutSousCircuitMinRec( 1, 5, { 2 3 4 } )
>> >> coutSousCircuitMinRec( 1, 2, { 3 4 } )
>> >> >> coutSousCircuitMinRec( 1, 3, { 4 } )
>> >> >> >> coutSousCircuitMinRec( 1, 4, { } )
>> >> >> coutSousCircuitMinRec( 1, 4, { 3 } )
>> >> >> >> coutSousCircuitMinRec( 1, 3, { } )
>> >> coutSousCircuitMinRec( 1, 3, { 2 4 } )
>> >> >> coutSousCircuitMinRec( 1, 2, { 4 } )
>> >> >> >> coutSousCircuitMinRec( 1, 4, { } )
>> >> >> coutSousCircuitMinRec( 1, 4, { 2 } )
>> >> >> >> coutSousCircuitMinRec( 1, 2, { } )
>> >> coutSousCircuitMinRec( 1, 4, { 2 3 } )
>> >> >> coutSousCircuitMinRec( 1, 2, { 3 } )
>> >> >> >> coutSousCircuitMinRec( 1, 3, { } )
>> >> >> coutSousCircuitMinRec( 1, 3, { 2 } )
>> >> >> >> coutSousCircuitMinRec( 1, 2, { } )
```

Figure 8: Les divers appels récursifs effectués pour un graphe à cinq sommets

initial dans la procédure `coutCircuitMin` se faisant avec un ensemble `a` de taille $n-1$ (`cardinalite(supprimer(v, v1))` $= \mid V \mid -1 = n-1$), la complexité de l'algorithme dans son ensemble (pour un sommet donné `v1`) est donc $\Theta((n-1)!)$.

5.2 Version récursive avec mémorisation

```
procedure coutSousCircuitMinMemo( int v1, int vi, Ensemble a, ref int D[*][0:*] )
          returns int c
{
  # On verifie si la combinaison des arguments vi/a a deja ete rencontree.
  if (D[vi][a] != PAS_DEFINI) {
    c = D[vi][a];
    return;
  }

  # Premiere occurence de la combinaison de vi et a.
  if (estVide(a)) {
    c = W[vi][v1]
  } else {
    c = INFINI;
    for [vj = 1 to n st estElement(a, vj) & arcExiste(vi, vj)] {
      int cout = W[vi][vj] + coutSousCircuitMinMemo(v1, vj, supprimer(a, vj), D);
      c = min(c, cout);
    }
  }
  D[vi][a] = c;
}


procedure coutCircuitMin( int v1 ) returns int c
{
  Ensemble v = creerPlein(n);
  int D[n][0:2**n-1] = ([n] ([2**n] PAS_DEFINI));

  c = coutSousCircuitMinMemo( v1, v1, supprimer(v, v1), D );
}
```

Algorithme 7: Algorithme récursif (diviser-pour-régner) avec mémorisation pour trouver le coût d'un circuit hamiltonien minimum

Pour éviter de résoudre plusieurs fois un même problème, on peut, tout en conservant la stratégie récursive, utiliser une approche avec *mémorisation*. Pour ce faire, on peut ajouter un argument à la procédure, argument qui conservera les résultats déjà calculés, et ce en fonction des arguments `vi` et `a` transmis dans la procédure originale. Une solution basée sur cette stratégie est présenté à l'algorithme 7. La procédure `coutSousCircuitMinMemo` se distingue de `coutSousCircuitMinRec` comme suit, tout le reste du code étant identique :

- En-tête de la procédure : un argument `D` a été ajouté, de type `int D[*][0:*]`.

  Plus précisément, le tableau transmis dans l'appel initial à la procédure (dans le corps de `coutCircuitMin`) est déclaré avec les bornes et valeurs initiales suivantes :

  ```
  int D[n][0:2**n-1] = ([n] ([2**n] PAS_DEFINI));
  ```

  Les valeurs possibles pour les arguments `vi` et `a` dans un appel à `coutSousCircuitMinRec` sont les suivantes :

```
procedure imprimer( Ensemble s )
{
  writes("{ ");
  int i = 1;
  while ( s != 0 ) {
    if (s % 2 == 1) { writes( i, " " ); }
    s /= 2;
    i += 1;
  }
  writes( "}" );
}
```

Fonctions auxiliaires 5: Fonction pour imprimer un ensemble

  - `vi` indique un sommet, donc une valeur comprise inclusivement entre `1` et `n`.
  - `a` indique un ensemble de sommets provenant des sommets du graphe $G = (V, E)$, c'est-à-dire, `a` $\subseteq V$. Comme on a `n` éléments dans l'ensemble initial de sommets, le nombre de sous-ensembles possibles est donc `2**n`. Or, la représentation (*bitset*) utilisée pour le type `Ensemble` (type abstrait 3) est telle qu'une correspondance biunivoque (bijection) existe entre les sous-ensembles (d'éléments compris entre 1 et n) et les entiers compris entre 0 et 2^{n-1}. Cette correspondance ne sera pas définie formellement (en termes mathématiques) mais peut être illustrée par la procédure servant à imprimer un ensemble (fonctions auxiliaires 5).

    En d'autres mots, la présence d'un élément `i` dans un ensemble `s` est dénotée par le fait que le `i`ème bit de `s` (un entier) est à 1.

- Début de la procédure : on vérifie si la paire d'arguments `vi` et `a` a déjà été rencontrée, c'est-à-dire, si le sous-problème correspondant a déjà été solutionné :

  ```
  if (D[vi][a] != PAS_DEFINI) {
    c = D[vi][a];
    return;
  }
  ```

  Si c'est le cas, le tableau `D` nous donne le solution déjà calculée pour ce sous-problème qu'on utilise directement pour produire le résultat `c` : par défaut, la valeur associée à une position de `D` est `PAS_DEFINI`, tel que spécifié au moment de la déclaration et initialisation du tableau `D`.

- Fin de la procédure : on met à jour la position appropriée du tableau `D`, de façon à prendre en note le résultat obtenu pour le sous-problème qui vient d'être calculé :

  ```
  D[vi][a] = c;
  ```

Analyse de l'algorithme

Pour une analyse précise et détaillée de la complexité asymptotique de cet algorithme, on devrait employer la méthode utilisée précédemment pour d'autres algorithmes de ce type. Plus précisément, il faudrait dénombrer, de façon exacte, le nombre de noeuds de l'arbre des appels récursifs ainsi que le travail effectué dans chacun des noeuds. Évidemment, pour ce faire, il faudrait alors tenir compte du fait que certains noeuds de l'arbre *ne sont pas développés à cause de la mémorisation*, c'est-à-dire que certains des sous-arbres sont tronqués. Une fois qu'on aurait obtenu le nombre exact de noeuds, on pourrait alors produire un estimé Θ approprié.

```
coutSousCircuitMinMemo( 1, 1, { 2 3 })
  coutSousCircuitMinMemo( 1, 2, { 3 })
    coutSousCircuitMinMemo( 1, 3, { })
  coutSousCircuitMinMemo( 1, 3, { 2 })
    coutSousCircuitMinMemo( 1, 2, { })
```

Figure 9: Trace d'exécution de l'algorithme avec mémorisation pour $n = 3$

```
coutSousCircuitMinMemo( 1, 1, { 2 3 4 })
  coutSousCircuitMinMemo( 1, 2, { 3 4 })
    coutSousCircuitMinMemo( 1, 3, { 4 })
      coutSousCircuitMinMemo( 1, 4, { })
    coutSousCircuitMinMemo( 1, 4, { 3 })
      coutSousCircuitMinMemo( 1, 3, { })
  coutSousCircuitMinMemo( 1, 3, { 2 4 })
    coutSousCircuitMinMemo( 1, 2, { 4 })
      coutSousCircuitMinMemo( 1, 4, { })
    coutSousCircuitMinMemo( 1, 4, { 2 })
      coutSousCircuitMinMemo( 1, 2, { })
  coutSousCircuitMinMemo( 1, 4, { 2 3 })
    coutSousCircuitMinMemo( 1, 2, { 3 })
      coutSousCircuitMinMemo( 1, 3, { })
    coutSousCircuitMinMemo( 1, 3, { 2 })
      coutSousCircuitMinMemo( 1, 2, { })
```

Figure 10: Trace d'exécution de l'algorithme avec mémorisation pour $n = 4$

Pour faire une telle analyse, il faudrait donc développer la structure de l'arbre des appels et réussir à identifier une formule appropriée. Toutefois, pour ce problème, la structure de l'arbre des appels devient rapidement très complexe, principalement à cause de l'argument de type ensemble : un ensemble de départ de n éléments va générer jusqu'à 2^n sous-ensembles, donc jusqu'à 2^n valeurs différentes possibles pour les divers arguments.

Pour ce problème, plutôt que de faire une analyse générale détaillée, nous allons tout d'abord examiner un certain nombre de cas concrets. Plus précisément, les arbres des appels pour des valeurs spécifiques de n, valeurs relativement petites ($n = 3, 4, 5$), sont présentés aux figures 9–11. Ces "*arbres d'appels*" sont simplement des traces de l'exécution de l'algorithme pour $n = 3, 4$ et 5, traces obtenues en imprimant, au début de la procédure principale (`coutSousCircuitMinMemo`), la valeur des différents arguments, tel qu'indiqué par l'extrait de code MPD 1 : le dernier argument est un ensemble pour lequel on utilise la procédure `imprimer`, alors que la variable `niveau` permet simplement d'indenter les appels en fonction du niveau de récursion, ce qui permet de bien voir les relations appelants–appelés.

```
if (DEBUG) {
  for [i = 1 to niveau] { printf("  "); }
  printf( "coutSousCircuitMinMemo( %d, %d, ", v1, vi );
  imprimer(a);
  printf( ")\n" );
  niveau++;
}
```

Extrait de code MPD 1: Code MPD pour produire la trace des appels récursifs

Pour différentes valeurs de n, on constate que le nombre d'appels croît très rapidement (les

```
coutSousCircuitMinMemo( 1, 1, { 2 3 4 5 })
  coutSousCircuitMinMemo( 1, 2, { 3 4 5 })
    coutSousCircuitMinMemo( 1, 3, { 4 5 })
      coutSousCircuitMinMemo( 1, 4, { 5 })
        coutSousCircuitMinMemo( 1, 5, { })
      coutSousCircuitMinMemo( 1, 5, { 4 })
        coutSousCircuitMinMemo( 1, 4, { })
    coutSousCircuitMinMemo( 1, 4, { 3 5 })
      coutSousCircuitMinMemo( 1, 3, { 5 })
        coutSousCircuitMinMemo( 1, 5, { })
      coutSousCircuitMinMemo( 1, 5, { 3 })
        coutSousCircuitMinMemo( 1, 3, { })
    coutSousCircuitMinMemo( 1, 5, { 3 4 })
      coutSousCircuitMinMemo( 1, 3, { 4 })
        coutSousCircuitMinMemo( 1, 4, { })
      coutSousCircuitMinMemo( 1, 4, { 3 })
        coutSousCircuitMinMemo( 1, 3, { })
  coutSousCircuitMinMemo( 1, 3, { 2 4 5 })
    coutSousCircuitMinMemo( 1, 2, { 4 5 })
      coutSousCircuitMinMemo( 1, 4, { 5 })
      coutSousCircuitMinMemo( 1, 5, { 4 })
    coutSousCircuitMinMemo( 1, 4, { 2 5 })
      coutSousCircuitMinMemo( 1, 2, { 5 })
        coutSousCircuitMinMemo( 1, 5, { })
      coutSousCircuitMinMemo( 1, 5, { 2 })
        coutSousCircuitMinMemo( 1, 2, { })
    coutSousCircuitMinMemo( 1, 5, { 2 4 })
      coutSousCircuitMinMemo( 1, 2, { 4 })
        coutSousCircuitMinMemo( 1, 4, { })
      coutSousCircuitMinMemo( 1, 4, { 2 })
        coutSousCircuitMinMemo( 1, 2, { })
  coutSousCircuitMinMemo( 1, 4, { 2 3 5 })
    coutSousCircuitMinMemo( 1, 2, { 3 5 })
      coutSousCircuitMinMemo( 1, 3, { 5 })
      coutSousCircuitMinMemo( 1, 5, { 3 })
    coutSousCircuitMinMemo( 1, 3, { 2 5 })
      coutSousCircuitMinMemo( 1, 2, { 5 })
      coutSousCircuitMinMemo( 1, 5, { 2 })
    coutSousCircuitMinMemo( 1, 5, { 2 3 })
      coutSousCircuitMinMemo( 1, 2, { 3 })
        coutSousCircuitMinMemo( 1, 3, { })
      coutSousCircuitMinMemo( 1, 3, { 2 })
        coutSousCircuitMinMemo( 1, 2, { })
  coutSousCircuitMinMemo( 1, 5, { 2 3 4 })
    coutSousCircuitMinMemo( 1, 2, { 3 4 })
      coutSousCircuitMinMemo( 1, 3, { 4 })
      coutSousCircuitMinMemo( 1, 4, { 3 })
    coutSousCircuitMinMemo( 1, 3, { 2 4 })
      coutSousCircuitMinMemo( 1, 2, { 4 })
      coutSousCircuitMinMemo( 1, 4, { 2 })
    coutSousCircuitMinMemo( 1, 4, { 2 3 })
      coutSousCircuitMinMemo( 1, 2, { 3 })
      coutSousCircuitMinMemo( 1, 3, { 2 })
```

Figure 11: Trace d'exécution de l'algorithme avec mémorisation pour $n = 5$

n	nb. d'appels
3	5
4	16
5	53
6	166
7	487
8	1352
9	3593
10	9226
11	23051
12	56332
13	135181
14	319502
15	745487

Tableau 1: Nombre d'appels (de noeuds) pour différentes valeurs de n

n	2^n	$(n-1)2^{n-1}$	$n2^n$	nb. appels	$n^2 2^n$
3	8	8	24	5	72
4	16	24	64	16	256
5	32	64	160	53	800
6	64	160	384	166	2304
7	128	384	896	487	6272
8	256	896	2048	1352	16384
9	512	2048	4608	3593	41472
10	1024	4608	10240	9226	102400
11	2048	10240	22528	23051	247808
12	4096	22528	49152	56332	589824
13	8192	49152	106496	135181	1384448
14	16384	106496	229376	319502	3211264
15	32768	229376	491520	745487	7372800

Tableau 2: Nombre d'appels (de noeuds) vs. différentes fonctions de n

différents appels pour $n > 5$ ne sont pas présentés dans les figures parce que trop nombreux), tel qu'indiqué dans la table 1.

Une analyse, tout à fait empirique, permet de constater que pour n suffisamment grand ($n \geq 11$), ces différentes valeurs semblent bornées inférieurement par $n \times 2^n$ et supérieurement par $n^2 \times 2^n$, tel qu'illustré dans la table 2.

La borne inférieure de $(n-1) \times 2^{n-1}$ est évidente et s'explique par le fait que l'ensemble de départ est de taille $n-1$, et ce à cause de l'appel initial à `supprimer(v, v1)`. Or, pour trouver la solution optimale, tous les sous-ensembles de cet ensemble de départ doivent être explorés, et ce pour les différents sommets ($n-1$ sommets, puisqu'on exclut le sommet de départ). Le tableau D dans son ensemble est de taille $n2^n$. Toutefois, on constate en examinant la trace des appels que certaines combinaisons d'arguments *n'apparaissent pas*, en d'autres mots, ce ne sont pas tous les éléments du tableau `D` qui sont définis. Par contre, on constate aussi que certains appels, bien qu'ils ne soient pas *élaborés* plusieurs fois à cause de la mémorisation, sont malgré tout rencontrés à plusieurs reprises (évidemment, après le premier appel, le résultats est déjà calculé et on retourne simplement le résultat déjà calculé). Pour de grandes valeurs de n, le nombre de ces appels répétitifs semble donc croître de façon

importante, d'où la borne inférieure $n2^n$ pour le nombre d'appels.

Le travail effectué dans chaque appel récursif est $\Theta(1)$: la procédure contient une boucle `for`, mais le travail effectué par cette boucle (qui consiste à faire des appels et à combiner ensuite, en temps $\Theta(1)$, le résultat retourné par l'appel) *est déjà compté* dans les différents appels récursifs, puisque l'arbre complet des appels correspond en fait à un *dépliage* de l'arbre. On peut donc en conclure que la complexité de l'algorithme est au moins $\Omega(n2^n)$. Par contre, par les résultats présentés plus haut, sa complexité semble aussi bornée supérieurement par $O(n^2 2^n)$.

Il est clair qu'un algorithme d'une telle complexité ne peut évidemment pas être considéré comme *efficace*, puisqu'il devient rapidement inutilisable lorsque `n` augmente. Par contre, on peut évidemment constater que l'utilisation de la stratégie de mémorisation a un impact positif sur l'efficacité (relative ;) du programme résultant. Ainsi, sur une machine PC/Linux, la version récursive pure semble atteindre sa limite pour un graphe (complet) comportant 13 sommets (aucune réponse après plusieurs minutes). La version avec mémorisation, quant à elle, peut traiter sans problème des graphes complets comptant 19 sommets. Par contre, à partir de 20 sommets, la taille de la matrice semble devenir trop grande, ce qui génère alors du *trashing* intensif (manque d'espace en mémoire virtuelle impliquant de nombreux accès disques).

5.3 Version avec tableau construit de façon ascendante

Nous avons vu, dans l'algorithme précédent, le rôle du tableau `int D[n][2**n-1]`, qui est défini de telle façon que `D[vi][a]` nous donne le coût minimum d'un sous-circuit se terminant au sommet de départ `v1` et passant exactement une fois par chacun des sommets de l'ensemble `a`, sous-circuit qui est une extension d'un chemin dont le dernier sommet est `vi`.

Il est possible de définir la matrice `D` à l'aide des équations récursives suivantes — notons que la condition $v_j \in A$ aurait aussi pû être renforcée en indiquant plutôt, comme on l'a expliqué précédemment, $v_j \in A \wedge W[v_i][v_j] \neq +\infty$:

$$\begin{aligned} D[v_i][\emptyset] &= W[v_i][v_1] \\ D[v_i][A] &= mininum_{v_j \in A}(W[v_i][v_j] + D[v_j][A - \{v_j\}]) \text{ si } A \neq \emptyset \end{aligned}$$

Cette définition correspond à celle utilisée pour calculer `c` et ensuite définir la position correspondante de la matrice `D` dans l'algorithme 7. Pour obtenir ensuite un algorithme de programmation dynamique, de forme *classique*, il nous faut alors définir un algorithme qui construit de façon *ascendante* et itérative, donc à l'aide de boucles, la matrice `D` (plutôt que de la construire par l'intermédiaire d'appels récursifs). En d'autres mots, il faut s'assurer de calculer les diverses entrées de la matrice dans un ordre qui sera tel que tout calcul nécessitant l'utilisation d'éléments de `D` fera nécessairement référence à des éléments *ayant déjà été calculés.*

La figure 12 présente une trace de l'exécution de l'algorithme récursif avec mémorisation, trace qui indique l'ordre dans lequel les différents éléments de `D` sont définis pour un graphe à cinq sommets. Le point important à noter est que pour calculer `D[i][a]`, les valeurs `D[j][a']` pour $a' \subseteq a$ doivent avoir déjà été calculées. L'algorithme de programmation dynamique classique devra donc construire `D` pour des ensembles `a` *de cardinalité croissante.*

L'algorithme 8 donne l'algorithme résultant. Tel que mentionné précédemment, l'algorithme ne fait que retourner le coût du circuit optimal, pas le circuit lui-même — on peut consulter le livre de Neapolitan et Naimipour des détails additionnels (bien qu'incomplets) sur les éléments d'algorithme qu'il faudrait ajouter pour identifier le circuit optimal.

Pour simplifier la présentation de l'algorithme, un tableau auxiliaire `PV` est utilisé, tableau dont le contenu est défini à l'aide de la procédure `genererSousEnsembles` (dont le code a été omis). Ce tableau a été nommé `PV` pour rappeller qu'il contient des sous-ensembles de

```
>> >> >> >> D[5][{ }) =  40
>> >> >> D[4][{ 5 }) =  72
>> >> >> >> D[4][{ }) =  24
>> >> >> D[5][{ 4 }) =  39
>> >> D[3][{ 4 5 }) =  74
>> >> >> D[3][{ 5 }) =  75
>> >> >> >> D[3][{ }) =  25
>> >> >> D[5][{ 3 }) =  33
>> >> D[4][{ 3 5 }) =  65
>> >> >> D[3][{ 4 }) =  42
>> >> >> D[4][{ 3 }) =  44
>> >> D[5][{ 3 4 }) =  50
>> D[2][{ 3 4 5 }) =  94
>> >> D[2][{ 4 5 }) =  83
>> >> >> D[2][{ 5 }) =  84
>> >> >> >> D[2][{ }) =  5
>> >> >> D[5][{ 2 }) =  48
>> >> D[4][{ 2 5 }) =  80
>> >> >> D[2][{ 4 }) =  62
>> >> >> D[4][{ 2 }) =  52
>> >> D[5][{ 2 4 }) =  67
>> D[3][{ 2 4 5 }) =  98
>> >> D[2][{ 3 5 }) =  77
>> >> D[3][{ 2 5 }) =  83
>> >> >> D[2][{ 3 }) =  63
>> >> >> D[3][{ 2 }) =  47
>> >> D[5][{ 2 3 }) =  55
>> D[4][{ 2 3 5 }) =  87
>> >> D[2][{ 3 4 }) =  80
>> >> D[3][{ 2 4 }) =  70
>> >> D[4][{ 2 3 }) =  66
>> D[5][{ 2 3 4 }) =  78
D[1][{ 2 3 4 5 }) =  112
```

Figure 12: Ordre de définition du contenu de `D`

```
procedure coutSousCircuitMinPD( int v1 ) returns int c
{
  int D[n][0:2**n-1] = ([n] ([2**n] PAS_DEFINI));
  int PV[0:n-1][2**(n-1)];

  # On genere tous les sous-ensembles de V qui ne contiennent pas v1.
  genererSousEnsembles( PV, v1 );

  # Definition des cas de base.
  Ensemble vide = creerVide(n);
  for [vi = 1 to n] {
    D[vi][vide] = W[vi][v1];
  }

  # Construction ascendante des autres cas.
  for [k = 1 to n-1] {
    for [na = 1 to nbSousEnsembles(k, n-1)] {
      Ensemble a = PV[k][na];
      for [vi = 1 to n st ~estElement(a, vi)] {
        D[vi][a] = INFINI;
        for [vj = 1 to n st estElement(a, vj) & arcExiste(vi, vj)] {
          D[vi][a] = min( D[vi][a], W[vi][vj] + D[vj][supprimer(a, vj)] );
        }
      }
    }
  }

  Ensemble v = creerPlein(n);
  c = D[v1][supprimer(v, v1)];
}

procedure coutCircuitMin( int v1 ) returns int c
{
  c = coutSousCircuitMinPD( v1 );
}
```

Algorithme 8: Algorithme avec programmation dynamique classique (ascendante) pour trouver le coût d'un circuit hamiltonien minimum

sommets (i.e., $\mathcal{P}(V) = 2^V$ = l'ensemble des sous-ensembles de V). `PV` est défini de façon telle que la ligne `PV[k]` contient la *collection*, dans un ordre arbitraire, de tous les sous-ensembles de V de taille `k`. Plus précisément :

$$\texttt{PV[k]} = \{A \mid A \subseteq V - \{\texttt{v1}\} \wedge |A| = k\}$$

En d'autres mots, `PV[k]` contient tous les sous-ensembles de V qui ne contiennent pas `v1`, puisque le sommet de départ ne fait jamais partie de `a` et ne doit être visité que pour finaliser le cycle du circuit. Le nombre de ces sous-ensembles est donné par la fonction `nbSousEnsembles(k, n-1)`. Pour énumérer chacun des sous-ensembles de $V - \{\texttt{v1}\}$ possédant exactement `k` éléments, on utilise alors simplement la boucle `for` suivante :

```
for [na = 1 to nbSousEnsembles(k, n-1)] {
  Ensemble a = PV[k][na];
  ...
}
```

À son tour, cette boucle doit être exécutée pour toutes les valeurs possibles de `k`, c'est-à-dire, pour `k` allant de 1 à `n-1`.

Notons que la boucle `for [vi = ...]` de l'algorithme 8 contient le quantificateur "`st ~estElement(a, vi)`" pour refléter l'invariant de la solution récursive, où on utilisait la condition `~estElement(a, vi)` — la suppression de ce quantificateur n'a aucun effet sur le résultat produit (on est certain que l'entrée correspondante de la matrice *ne sera pas utilisée*).

Analyse de l'algorithme

Une analyse de cet algorithme nous montre que, comme dans le cas précédent (algorithme récursif avec mémorisation), il est de complexité *plus* qu'exponentielle :

- Le temps pour générer le tableau `PV` des sous-ensembles de V ne contenant pas `v1` est $\Theta(2^{n-1})$ (le nombre de sous-ensembles).
- L'initialisation pour les cas de base est $\Theta(n)$.
- La combinaison des deux boucles internes (avec `vi` et `vj` comme variables d'itérations) est $\Theta(n^2)$. Or, ces deux boucles doivent être exécutées pour chacun des 2^{n-1} sous-ensembles de $V - \{v1\}$. Les trois boucles imbriquées seront donc de complexité $n^2 2^{n-1}$.

Dans l'ensemble, l'algorithme sera donc $\Theta(n^2 2^{n-1})$, ce qui peut s'exprimer plus simplement par $\Theta(n^2 2^n)$ — puisque $2^{n-1} = \frac{1}{2} \times 2^n \in \Theta(2^n)$.

---

6 Approximations et heuristiques pour le problème du voyageur de commerce

Note : Dans les sections qui suivent, nous chercherons à identifier un circuit englobant toutes les villes, indépendamment de la ville de départ. Le choix de la ville de départ est secondaire, puisqu'un circuit $[c_1, \ldots, c_{i-1}, c_i, c_{i+1}, \ldots, c_n]$ peut toujours être transformé en un circuit de coût équivalent débutant à la ville c_i : $[c_i, c_{i+1}, \ldots, c_n, c_1, \ldots, c_{i-1}]$

Problèmes NP-difficiles

Nous avons vu, dans les sections précédentes, divers algorithmes séquentiels et parallèles pour résoudre le problème du voyageur de commerce. Ces algorithmes permettent d'obtenir *une* solution *optimale* à une instance donnée du problème du voyageur commerce. Malheureusement, on a aussi vu que la complexité asymptotique de ces algorithmes était très élevée — complexité exponentielle $\Theta(2^n)$ ou factorielle $\Theta(n!)$ — ce qui les rend impraticable pour de grandes valeurs de n. Ainsi, pour les programmes MPD présentés précédemment, le temps d'exécution devient *très* long dès lors que n se situe quelque part entre 13 et 20, selon l'algorithme.

Or, en pratique, les instances réelles du problème du voyageur de commerce pour lesquelles on désire obtenir des solutions peuvent souvent atteindre plusieurs centaines, sinon plusieurs milliers, de villes. La recherche d'une solution exacte optimale à l'aide des algorithmes présentés précédemment est donc impensable.

Il existe de nombreux problèmes, semblables à celui du voyageur de commerce, pour lesquels *il n'existe aucun algorithme connu* qui soit efficace (de temps polynomial) et qui permette d'obtenir une solution exacte et optimale. La théorie des problèmes NP-difficiles,[7] que nous n'aborderons pas dans le cadre de ce cours, porte sur cette classe de problèmes. Tous ces problèmes ont la particularité que si jamais une solution polynomiale efficace était trouvée *pour un de ces problèmes*, alors on obtiendrait aussitôt une solution efficace *pour tous les autres problèmes de cette classe.*

Bien que cela n'ait encore jamais été prouvé formellement, la conjecture généralement acceptée veut qu'il sera toujours *impossible* d'obtenir un algorithme efficace pour ces problèmes. Malheureusement, de tels problèmes NP-complets — voyageur de commerce, sac à dos 0–1, coloriage de graphes, recherche de cliques — se retrouvent souvent au coeur même de problèmes réels auxquels les informaticiens sont confrontés : production d'itinéraires et d'horaires de transport, placement de composants dans un circuit micro-électronique, etc.

Utilisation d'approximations et d'heuristiques

Pour des instances de grande taille de tels problèmes, on sait qu'il est impossible d'obtenir une solution exacte optimale. Pour pouvoir, malgré tout, résoudre de tels problèmes, on se rabat donc souvent sur l'utilisation d'approximations ou d'*heuristiques.* Le grand dictionnaire terminologique[8] définit une heuristique comme une "méthode de recherche empirique ayant recours aux essais et erreurs pour la résolution de problèmes."

Ainsi, une heuristique — en anglais, on parle aussi de *rules of thumb* — définit une stratégie de solution qui, face à un problème complexe, permet d'obtenir, généralement, une solution *qui n'est pas trop mauvaise*, et ce en temps *raisonnable.* En d'autres mots, on fait un compromis entre temps d'exécution et exactitude de la solution : on accepte que la solution produite par l'heuristique ne soit pas *une solution exacte optimale*, en autant qu'elle ne soit pas trop mauvaise (pas "trop loin" de la *vraie* solution) et qu'elle soit produite plus rapidement (temps polynomial).

Il existe de nombreuses heuristiques possibles pour le problème du voyageur de commerce. Dans les sections qui suivent, nous examinerons divers types d'heuristiques pour une version spécialisée du problème du voyageur de commerce où les distances entre les villes satisfont certaines conditions spéciales, mais quand même naturelles, à savoir le problème du voyageur de commerce *géométrique* — lorsque la distance entre les villes correspond à la distance euclidienne classique (distance usuelle dans le plan à deux dimensions).

Tout d'abord, à la section 8, nous examinerons un algorithme qui produit une 2-approximation. Ensuite, à la section 9, nous examinerons deux métaheuristiques de base spécialisées

[7] On parle de problèmes NP-complets dans le cas de *problèmes de décision.* Un problème d'optimisation qui correspond à un problème de décision NP-complet est dit NP-difficile (*NP-hard*).

[8] `http://www.granddictionnaire.com`

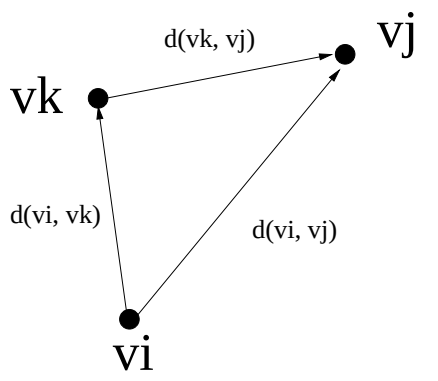

Figure 13: Inégalité du triangle

pour le voyageur de commerce. Finalement, à la section 10, nous examinerons un type particulier d'algorithme *probabiliste* pour le problème du voyageur de commerce, algorithme basé sur la métaheuristique du *recuit simulé*. Mais tout d'abord, à la section 7, deux heuristiques simples qui peuvent être utilisées pour la version générale du problème.

7 Deux heuristiques pour le problème du voyageur de commerce général

7.1 Heuristique `AL`

Voir exercices.

7.2 Heuristique `NN`

Voir exercices.

8 Approximation pour le problème du voyageur de commerce géométrique

Soit un graphe pour lequel la matrice des distances respecte l'*inégalité du triangle*, c'est-à-dire que la condition suivante est satisfaite :

$$\forall i, j, k \bullet d(v_i, v_j) \leq d(v_i, v_k) + d(v_k, v_j)$$

Graphiquement, cette condition est celle illustrée à la figure 13.

Supposons aussi, comme on le fera à la section 10 pour l'algorithme de recuit simulé, que les distances sont symétriques, c'est-à-dire que la condition suivante est aussi satisfaite :

$$\forall i, j \bullet d(v_i, v_j) = d(v_j, v_i)$$

En d'autres mots, cette dernière condition conduit à considérer le graphe comme un graphe non orienté. Le problème du voyageur de commerce restreint à des graphes de ce type est appelé le problème du *voyageur de commerce géométrique* (*geometric TSP*) ou problème

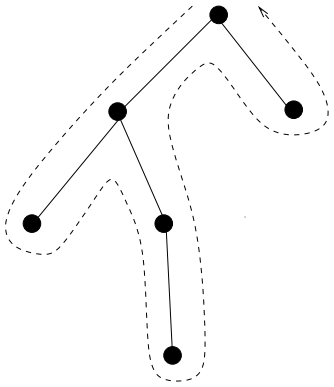

Figure 14: Un circuit généré à partir d'un arbre de recouvrement minimum

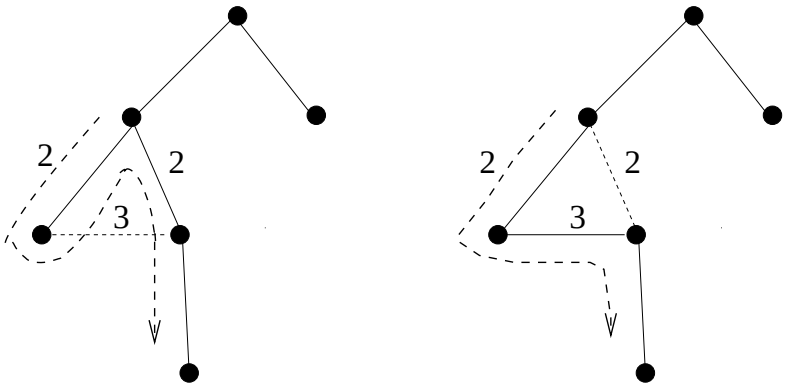

Figure 15: Un raccourci pour générer un circuit Hamiltonien à partir d'un circuit obtenu par double parcours d'un arbre de recouvrement minimum

voyageur de commerce *euclidien* ; on dit aussi problème du voyageur de commerce *avec inégalité du triangle*.

Nous allons montrer que pour des graphes de ce type, il est possible d'obtenir une solution qui est une bonne approximation de la solution optimale, plus précisément, une approximation qui produit un circuit dont le coût est au plus deux fois le coût du circuit optimal. La technique utilisée consiste à produire un circuit à partit d'un arbre de recouvrement minimal.

Soit un circuit de coût optimal pour ce graphe. Si on retire un arc quelconque de ce circuit, on obtient une arbre de recouvrement de ce graphe. Le coût d'un arbre de recouvrement minimal doit nécessairement être inférieur au coût d'un circuit optimal. Comme on l'a vu précédemment (chapitre sur les algorithmes voraces), il existe des algorithmes qui permettent de produire un arbre de recouvrement minimum en temps polynomial.

Soit alors un circuit (non Hamiltonien) obtenu en parcourant deux (2) fois chacune des arêtes de l'arbre de recouvrement minimum (figure 14). Ce circuit permet de visiter l'ensemble des sommets du graphe. Toutefois, il est possible qu'ainsi un sommet soit visité plus d'une fois, tel que cela est illustré à la figure 14, donc il ne s'agit pas d'un circuit Hamiltonien. Lorsque cette situation survient, on peut alors utiliser un *raccourci*, en choisissant de visiter un sommet une seule fois, et ce par l'intermédiaire de l'arc dont le coût est minimum, tel que cela est illustré à la figure 15 (les arcs en pointillés font partie du circuit). En répétant ce processus pour chacune des villes visitées plus d'une fois, on obtiendra finalement un circuit Hamiltonien.

En résumé, l'heuristique est la suivante :

1. Identifier un arbre de recouvrement minimum.
2. Créer un circuit qui visite chacune des villes en parcourant deux fois chacune des arêtes de l'arbre.
3. Créer un circuit (Hamiltonien) qui ne visite jamais deux fois aucun sommet en utilisant des raccourcis appropriés. Plus précisément, soit L la liste des sommets visités par un parcours préfixe de l'arbre de recouvrement minimal. Le circuit produit correspond alors au cycle hamiltonien qui visite les sommets dans l'ordre L.

Soit alors $cout_{arbreMin}$ le coût d'un arbre de recouvrement minimum. Soit $cout_{circuitMin}$ le coût d'un circuit minimum. Finalement, soit $cout_{heuristique}$ le coût d'un circuit obtenu par cette heuristique. On aura nécessairement les inégalités suivantes :

$$cout_{arbreMin} < cout_{heuristique} \leq 2 \times cout_{arbreMin} < 2 \times cout_{circuitMin}$$

En d'autres mots, dans le pire cas, l'heuristique produira un circuit qui sera au plus deux fois le coût du circuit minimum.

9 Deux métaheuristiques de base appliquée au problème du voyageur de commerce géométrique

Dans cette section, nous présentons quelques heuristiques qui ne sont pas des approximations pour le problème du voyageur de commerce :

1. Deux variantes de recherche *locale* (section 9.2).
2. Une heuristique de recherche avec tabous (section 9.3).

Ces heuristiques sont en fait des spécialisations des deux métaheuristiques de base correspondantes.

La section 9.1, quant à elle, introduit tout d'abord un certain nombre de procédures auxiliaires communes aux diverses heuristiques.

9.1 Procédures auxiliaires communes aux algorithmes de recherche (locale et avec tabous)

L'extrait de code MPD 2 présente diverses procédures qui seront utilisées dans les sous-sections qui suivent.

Génération des variantes pour l'exploration du voisinage L'idée centrale des heuristiques de recherche locale ou avec tabous est d'explorer le *voisinage* d'une *solution* existante. Contrairement à d'autres heuristiques (par exemple, l'heuristique du voisin le plus près (*nearest neighbor*) vue dans les exercices), il s'agit ici d'explorer le voisinage *d'une solution "complète"*, mais qui n'est pas nécessairement optimale. Ainsi, dans le contexte du problème du voyageur de commerce, on suppose qu'on a déjà sous la main un circuit hamiltonien pour l'ensemble des villes (et non simplement un circuit partiel qu'on veut étendre). On va alors explorer le voisinage de cette solution — i.e., générer d'autres circuits hamiltoniens à partir de celui qu'on a sous la main — pour tenter d'identifier d'autres solutions intéressantes, évidemment dans le but d'identifier la *meilleure* des solutions.

La stratégie utilisée pour générer le voisinage d'une solution est la même que celle utilisée dans le contexte du recuit simulé, mais sans le caractère aléatoire associé au choix des villes

```
procedure modificationDeCoutDuVoisin( Sequence s, int si, int j ) returns real modifCout
# ROLE
#  Determiner le changement de cout associe a une possible sous-sequence.
# PRECONDITION
#   s denote un circuit valide.
# POSTCONDITION
#   modifCout = cout qui resulterait de l'inversion de s[si:j].
{
  int n = longueur(s);
  int i = (si+n-2) % n + 1;    # Predecesseur de si.
  int sj = j % n + 1;          # Successeur de j.

  if ( i != j ) {
    real cij   = W[element(s,  i)][element(s,  j)];
    real cisi  = W[element(s,  i)][element(s, si)];
    real cjsj  = W[element(s,  j)][element(s, sj)];
    real csisj = W[element(s, si)][element(s, sj)];
    modifCout = real( cij + csisj - cisi - cjsj );
  } else {
    modifCout = 0.0;
  }
}

procedure modifierCircuit( Sequence s, int i, int j )
# POSTCONDITION
#   Les elements de la sous-sequence de s allant de i a j sont inverses.
# EXEMPLE
#   s = [10, 20, 30, 40, 50], i = 2, j = 5 => s' = [10, 50, 40, 30, 20]
{
  int nij = (j - i + n) % n + 1;
  for [k = 1 to nij/2] {
    echanger( s, (i+k-2) % n + 1, (j-k+n) % n + 1 );
  }
}

# Un chemin direct existe en n'importe quelle villes, alors on prend simplement
# la sequence ordonnee des differentes villes.
procedure genererCheminInitial( int n ) returns Sequence s
{
  s = creer();
  for [i = 1 to n] { ajouterEnQueue(s, i); }
}
```

Extrait de code MPD 2: Procédures communes aux divers algorithmes

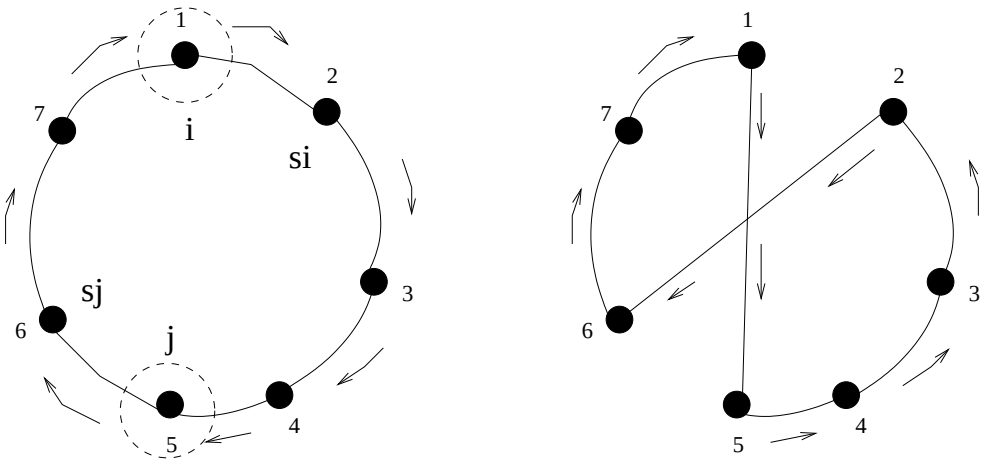

Figure 16: Illustration graphique de la génération d'une variante

identifiant le sous-segment à inverser.[9] Soit alors un circuit c défini par la séquence de villes suivantes :

$$c_1, c_2, \ldots, c_{n-1}, c_n$$

On veut produire une variante de c basée sur deux villes c_i et c_j ; notons alors par c_{si} et c_{sj} les successeurs de ces villes (modulo n, si nécessaire) :

$$c_1, \ldots, c_i, c_{si}, c_{si+1} \ldots, c_{j-1}, c_j, c_{sj}, \ldots, c_n$$

Le nouveau circuit, la variante, est alors obtenu en *renversant* la sous-séquence formée des villes allant de c_{si} à c_j, générant ainsi le circuit suivant :

$$c_1, \ldots, c_i, c_j, c_{j-1}, \ldots, c_{si+1}, c_{si}, c_{sj}, \ldots, c_n$$

Ceci est illustré graphiquement à la figure 16, pour $\mathtt{n} = 7$, $i = 1$ et $j = 5$. C'est cette tâche qu'effectue la procédure `modifierCircuit`. Quant à la fonction `modificationDeCoutDuVoisin`, elle permet de calculer le coût de la nouvelle variante ainsi générée.

Génération d'une solution initiale Comme un chemin direct existe entre n'importe quelle deux villes, alors on peut obtenir un circuit hamiltonien initial en prenant simplement la séquence ordonnée des différentes villes. C'est ce qui est fait par la fonction `genererCheminInitial`.

9.2 Deux variantes de recherche locale

9.2.1 Première version : sélection du premier voisin acceptable

L'algorithme 9 présente une première version de recherche locale. Dans cette version, une exploration du voisinage se termine, et donc retourne un nouveau voisin, aussitôt qu'un voisin ayant un coût inférieur à la solution courante est trouvé. Si aucun voisin n'est meilleur que la solution courante, alors c'est qu'un minimum local a été atteint, donc l'algorithme se termine (`minimumLocal = true`).

[9] Notons que deux conditions importantes doivent être satisfaites pour que cette stratégie soit utilisable et permette un calcul efficace du coût d'une variante, conditions qui s'appliquent dans le cas du problème du voyageur de commerce géométrique : (i) la matrice des distances doit être symétrique ; (ii) il doit exister un chemin direct entre n'importe quelles deux villes.

```
procedure explorerVoisinage( Sequence s, ref bool minimumLocal )
# POSTCONDITION
#   minimumLocal = false si on a trouve un renversement qui ameliore la solution
#                  courante, auquel cas on effectue le premier renversement
#                  rencontre qui ameliore la solution courante.
#   minimumLocal = true sinon
{
  int n = longueur(s);
  # Boucle sur tous les couples (u, v) de sommets du circuit
  for [ u = 1 to n, v = u+1 to n ] {
    if ( modificationDeCoutDuVoisin(s, u, v) < 0.0 ) {
      # On a trouve un voisin qui diminue le cout : on termine
      modifierCircuit(s, u, v);
      minimumLocal = false;
      return;
    }
  }
  # Rien trouve :(
  minimumLocal = true;
}

procedure explorerEspace( Sequence s, int nbIterationsMax, ref int nbIterations )
{
  for [ i = 1 to nbIterationsMax ] {
    bool minimumLocal;
    explorerVoisinage( s, minimumLocal );
    if ( minimumLocal ) {
      nbIterations = i;
      return;
    }
  }
  nbIterations = 0;
}

Sequence chemin = genererCheminInitial( n );
explorerEspace( chemin, iterations, nbIterations );

if ( nbIterations > 0 ) {
  write("Optimum local atteint apres ", nbIterations, "iterations.");
}
writes("Chemin final "); imprimer( chemin );  write( " =>", coutChemin(chemin) );
```

Algorithme 9: Exploration locale du voisinage avec sélection du premier voisin intéressant

9.2.2 Deuxième version : sélection du meilleur voisin

L'algorithme 10 présente une deuxième version de recherche locale. Dans cette version, une exploration du voisinage immédiat de la solution courante se termine uniquement lorsque *tous les voisins immédiats ont été examinés*, l'exploration du voisinage retournant alors le *meilleur* de ces voisins immédiats. Comme dans la version précédente, l'algorithme se termine si aucun meilleur voisin n'existe.

9.3 Recherche avec tabous

L'algorithme 11 présente une heuristique de *recherche avec tabous*. Une telle heuristique maintient une liste des voisins qui sont *tabous*, i.e., qui ne doivent pas être explorés, et ce pour éviter de "tourner en rond" (donc pour assurer d'explorer des voisinages divers).

Dans cette forme de recherche locale, une exploration du voisinage immédiat de la solution courante se termine lorsque *tous les voisins immédiats ont été examinés*, l'exploration du voisinage retournant alors le *meilleur* des voisins immédiats *qui n'est pas dans la liste des tabous*. L'algorithme se termine si aucun meilleur voisin n'existe ou si tous les voisins sont tabous.

10 La métaheuristique du *recuit simulé* appliquée au problème du voyageur de commerce géométrique

10.1 Définition générale des processus de *recuit* et de *recuit simulé*

Le terme anglais "*annealing*" — du verbe "*to anneal*", qu'on peut traduire en français par durcir, recuire[10] ou tempérer — désigne le processus utilisé pour rendre un produit — par ex., le verre, le métal — plus résistant, moins fragile (*less brittle*). Ce processus est réalisé en liquéfiant tout d'abord le produit, à très haute température, puis en le refroidissant *très graduellement* pour l'amener dans un état solide. Des considérations de physique et de statistiques mécaniques, que nous n'explorerons pas en détail ici, permettent alors d'assurer que si le refroidissement se fait de façon suffisamment graduelle, alors des cristaux réguliers (donc formant des structures d'énergie minimale) seront créés, assurant ainsi une plus grande résistance du produit.

En d'autres mots, des cristaux réguliers sont associés à des niveaux minimaux d'énergie des atomes formant ces cristaux. À l'état liquide, le niveau d'énergie des atomes est très élevé et ceux-ci bougent rapidement, avec un niveau global d'énergie élevé. Si le refroidissement est très rapide, il est alors possible que les atomes, malgré leur haut niveau d'énergie, forment des liens rigides (solides), ce qui rend alors le solide plus fragile. Par contre, si le refroidissement est très graduel, alors au fur et à mesure où la température diminue, il devient de moins en moins probable d'arriver à de tels liens rigides entre atomes possédant de hauts niveaux d'énergie, ce qui permet alors d'obtenir un réseau de cristaux plus résistant, moins fragile.

Une particularité importante de ce processus de refroidissement graduel est que des hauts niveaux d'énergie, plus instables, sont possibles et acceptables lorsque la température est élevée, mais deviennent de moins en moins *probables* lorsque la température diminue. C'est principalement cette particularité que les algorithmes de recuit simulé cherchent à reproduire.

Un algorithme de recuit simulé est une forme d'algorithme probabiliste. De tels algorithmes existent pour différents types de problèmes. Un dictionnaire des termes informatiques[11] définit le processus de recuit simulé de la façon suivante :

[10] Selon le grand dictionnaire terminologique : "Recuire (métallurgie) = chauffer et maintenir un temps donné à une température déterminée, puis refroidir en général lentement."

[11] `http://dictionary.reference.com`

```
procedure explorerVoisinage( Sequence s, ref bool minimumLocal )
# POSTCONDITION
#   minimumLocal = false si on a trouve un renversement qui ameliore la solution
#                  courante, auquel cas on effectue le premier renversement
#                  rencontre qui ameliore la solution courante.
#   minimumLocal = true sinon
{
  int n = longueur(s);
  int  meilleur_u, meilleur_v; # Meilleur renversement rencontre
  real modifCoutMax = 0.0;     # Meilleure amelioration rencontree

  # Boucle sur tous les couples (u, v) de sommets du circuit
  for [ u = 1 to n, v = u+1 to n ] {
    real modifCout = modificationDeCoutDuVoisin(s, u, v);
    if ( modifCout < modifCoutMax ) {
      meilleur_u = u; meilleur_v = v; modifCoutMax = modifCout;
    }
  }

  if ( modifCoutMax >= 0.0 ) {
    # On est tombe dans un optimum local
    minimumLocal = true;
  } else  {
    # On a trouve un voisin qui ameliore le cout.
    modifierCircuit(s, meilleur_u, meilleur_v);
    minimumLocal = false;
  }
}

procedure explorerEspace( Sequence s, int nbIterationsMax, ref int nbIterations )
{
  for [ i = 1 to nbIterationsMax ] {
    bool minimumLocal;
    explorerVoisinage( s, minimumLocal );
    if ( minimumLocal ) {
      nbIterations = i;
      return;
    }
  }
  nbIterations = 0;
}

Sequence chemin = genererCheminInitial( n );
explorerEspace( chemin, iterations, nbIterations );

if ( nbIterations > 0 ) {
  write("Optimum local atteint apres ", nbIterations, "iterations.");
}
writes("Chemin final "); imprimer( chemin );  write( " =>", coutChemin(chemin) );
```

Algorithme 10: Exploration locale du voisinage avec sélection du meilleur voisin intéressant

```
# Initialisation de la sequence tabou, avec des sequences vides.
Sequence Tabou[L];
for [ i = 1 to L ] {
  Tabou[i] = creer();
}
# Indique a quel endroit ecrire dans la liste Tabou (position dernier ecriture).
int curseurTabou = 0;

procedure explorerVoisinage( Sequence s )
# POSTCONDITION
#   On a effectue le meilleur renversement de sequence ne conduisant
#   pas a un circuit present dans la liste Tabou.
#   Si un tel renversement n'existe pas, rien ne change
{
  int n = longueur(s);

  real modifCoutMax = INFINI;  # Meilleure amelioration rencontree
  int  meilleur_u, meilleur_v; # Meilleur renversement rencontre

  bool voisinInteressantTrouve = false; # Indique si on a trouve un renversement.

  # Boucle sur tous les couples d'aretes
  for [ u = 1 to n, v = u+1 to n ] {
    real modifCout = modificationDeCoutDuVoisin(s, u, v);

    if (modifCout != 0.0) {
      # On examine les sous-sequences dont le renversement diminue le cout.

      # On teste si le renversement courant mene a un chemin tabou
      bool estTabou = false;
      modifierCircuit(s, u, v);
      for [ j = 1 to min(curseurTabou, L) ] {
        estTabou |= egalite(s, Tabou[j]);
      }
      modifierCircuit(s, u, v);
      if ( ~estTabou & modifCout < modifCoutMax ) {
        meilleur_u = u; meilleur_v = v;
        modifCoutMax = modifCout;
        voisinInteressantTrouve = true;
      }
    }
  }

  if ( voisinInteressantTrouve ) {
    modifierCircuit( s, meilleur_u, meilleur_v );
  }

  # Mise a jour de la liste Tabou
  Tabou[curseurTabou % L + 1] = cloner(s);
  curseurTabou++;
}
```

Algorithme 11: Exploration avec tabous (1ère partie)

```
Sequence cheminMin;     # Chemin optimal
real     coutMin;       # Cout optimal

procedure explorerEspace( Sequence s, int nbIterationsMax )
{
  for [ i = 1 to nbIterationsMax ] {
    explorerVoisinage( s );
    real cout = coutChemin(s);
    if ( cout < coutMin ) {
      cheminMin = cloner(s);
      coutMin = cout;
    }
  }
}

Sequence chemin = genererCheminInitial( n );
cheminMin = cloner(chemin);
coutMin = coutChemin(cheminMin);

explorerEspace( chemin, iterations );

writes("Chemin final "); imprimer( cheminMin );  write( " =>", coutChemin(cheminMin) );
```

Algorithme 11: Exploration avec tabous (2ième partie)

> *A technique which can be applied to any minimisation or learning process based on successive update steps (either random or deterministic) where the update step length is proportional to an arbitrarily set parameter which can play the role of a temperature. Then, in analogy with the annealing of metals, the temperature is made high in the early stages of the process for faster minimisation or learning, then is reduced for greater stability.*

Une autre définition des termes *simulated annealing* est celle présentée le *National Institute of Standards and Technology*[12] :

> *A technique to find a good solution to an optimization problem by trying random variations of the current solution. A worse variation is accepted as the new solution with a probability that decreases as the computation proceeds. The slower the cooling schedule, or rate of decrease, the more likely the algorithm is to find an optimal or near-optimal solution*

10.2 Les grandes lignes du processus de recuit simulé appliqué au problème du voyageur de commerce

Les grandes lignes d'un algorithme réalisant un processus de recuit simulé pour le problème du voyageur de commerce sont les suivantes :

[12] `http://www.nist.gov/dads/HTML/simulatedAnnealing.html`

```
// Procédure générale de recuit simulé
DEBUT
  c <- Un circuit initial valide (possiblement choisi de façon aléatoire)
  t <- Température initiale appropriée (suffisamment élevée)
  POUR un certain nombre d'itérations FAIRE
    Explorer les variantes de c (à la température t)
    Diminuer un peu la température t
  FIN
FIN
```

L'exploration des variantes de `c` (le circuit "minimum" courant) à une température `t` donnée se fait alors, en gros, comme suit :

```
// Procédure pour explorer des variantes
DEBUT
  TANTQUE un certain nombre d'essais n'ont pas été effectués
        ET un certain nombre de changements n'ont pas eu lieu FAIRE
    c' <- Sélectionner une variante possible de c
    SI la variante sélectionnée c' est acceptable pour la température t ALORS
      c <- c'
    FIN
  FIN
FIN
```

Soulignons tout d'abord que les grandes lignes de l'algorithme décrit plus haut (processus général et exploration des variantes) ne sont pas spécifiques au problème du voyageur de commerce. Elles représentent plutôt, si on remplace le terme "circuit" par le terme plus général de "candidat" ou de "solution potentielle", les grandes lignes de n'importe quel algorithme de recuit simulé. La notion de génération d'une variante du candidat courant que nous expliquerons dans les lignes qui suivent s'applique évidemment au cas spécifique du problème du voyageur de commerce, de même que la notion de changement de niveau d'énergie associé.

Les deux questions clés associées à la procédure d'exploration à une température donnée, pour le cas spécifique du problème du voyageur de commerce, sont les suivantes :

1. Comment générer, à un coût raisonnable, une variante du circuit courant `c` qui soit aussi un circuit intéressant?
2. Comment déterminer si la variante ainsi obtenue est acceptable?

Génération des variantes

La génération de variantes d'un circuit existant est basée sur une technique initialement introduite par Lin (en 1965) et décrite plus en détail par Brich Hansen. Notons tout d'abord que deux conditions importantes doivent être satisfaites pour que cette stratégie soit utilisable et permette un calcul efficace du coût d'une variante :

1. La matrice des distances doit être symétrique, c'est-à-dire, la distance pour aller de c_i à c_j doit être la même que celle pour aller de c_j à c_i. Bien qu'on puisse imaginer des situations où les distances ne sont pas symétriques, la symétrie correspond plutôt au cas usuel.
2. Il doit exister un chemin direct entre n'importe quelles deux villes, c'est-à-dire que la distance entre c_i et c_j ne doit jamais être infinie. En fait, cette condition n'est pas absolument nécessaire mais facilite les calculs et évite de traiter de façon spéciale les cas de circuits impossibles, donc de coûts infinis, qui de toute façon seraient ultimement rejetés.

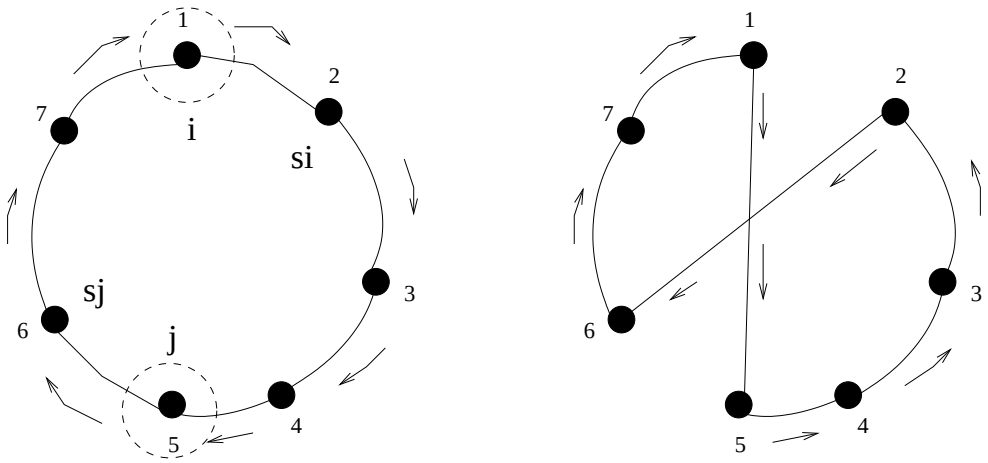

Figure 17: Illustration graphique de la génération d'une variante

Soit alors un circuit c défini par la séquence de villes suivantes :

$$c_1, c_2, \ldots, c_{n-1}, c_n$$

Pour produire une variante de c, il s'agit alors de choisir *aléatoirement* deux villes c_i et c_j ; notons alors par c_{si} et c_{sj} les successeurs de ces villes (modulo n, si nécessaire) :

$$c_1, \ldots, c_i, c_{si}, \ldots, c_j, c_{sj}, \ldots, c_n$$

Le nouveau circuit, la variante, est alors obtenu en *renversant* la sous-séquence formée des villes allant de c_{si} à c_j, générant ainsi le circuit suivant :

$$c_1, \ldots, c_i, c_j, \ldots, c_{si}, c_{sj}, \ldots, c_n$$

Ceci est illustré graphiquement à la figure 17, pour $\mathtt{n} = 7$, $i = 1$ et $j = 5$.

Acceptation vs. rejet d'une variante

Pour comprendre le processus de sélection ou rejet d'une variante, il faut tout d'abord voir à quoi correspond la notion de niveau d'énergie d'une solution, d'un circuit. L'objectif du processsus de recuit simulé est d'obtenir une solution avec un niveau d'énergie minimum. Dans le contexte du problème du voyageur de commerce, le niveau d'énergie d'un circuit correspond donc au coût de ce circuit, puisqu'on cherche à obtenir un circuit de coût minimum.

Le caractéristique clé du recuit simulé concernant la sélection ou le rejet d'une variante est le fait que cette décision repose uniquement sur la *différence* entre le niveau d'énergie associé à la nouvelle variante par rapport au niveau d'énergie de la variante de départ. Si le niveau d'énergie (le coût) du nouveau circuit (obtenu aléatoirement) est inférieur au coût du circuit de départ, alors ce nouveau circuit est *nécessairement accepté*. Par contre, même si le coût résultant est supérieur à celui du circuit de départ, alors ce nouveau circuit pourra quand même être accepté, *mais seulement avec une certaine probabilité qui dépend de la température globale* : plus la température est élevée, plus une augmentation de niveau d'énergie risque d'être acceptée ; inversement, plus la température globale baisse, plus une augmentation du niveau d'énergie risque d'être rejetée.

Les conditions sur les distances décrites précédemment (symétrie et existence d'un arc) font en sorte qu'il est possible de déterminer *en temps constant* ($\Theta(1)$) si le coût du nouveau circuit est inférieur ou supérieur au coût du circuit de départ, donc sans avoir besoin de

calculer le coût complet de ce nouveau circuit (ce qui se ferait alors en temps $\Theta(n)$). Notons par $d(c_{i_1}, c_{i_2})$ la distance entre deux villes c_{i_1} et c_{i_2}. La différence de niveau d'énergie (de coût) entre le circuit initial

$$c_1, \ldots, c_i, c_{si}, \ldots, c_j, c_{sj}, \ldots, c_n$$

et le nouveau circuit généré par le choix aléatoire de deux villes c_i et c_j et le renversement de la sous-séquence de villes intermédiaires

$$c_1, \ldots, c_i, c_j, \ldots, c_{si}, c_{sj}, \ldots, c_n$$

sera alors donnée par

$$dE = d(c_i, c_j) + d(c_{si}, c_{sj}) - d(c_i, c_{si}) - d(c_j, c_{sj})$$

Ici, c'est la condition de symétrie de la fonction de distance qui assure que le renversement des villes intermédiaires n'a aucun effet sur la modification du coût du circuit, alors que l'existence d'un lien entre n'importe quelle deux villes assure que le circuit résultant est bien valide (coût non infini).

```
FONCTION changementAccepte( dE: real; T: real ): Boolean
DEBUT
  SI dE <= 0.0 ALORS
    // Diminution du coût => on accepte
     RETOURNER( VRAI )
  FIN
  // Augmentation  de coût => on accepte avec une certaine probabilité
  nbAleatoire <- un nombre choisi aléatoirement (entre 0.0 et 1.0)
  RETOURNER( e^(-dE/T) > nbAleatoire )
FIN
```

Algorithme 12: Fonction pour accepter ou non une variante selon le changement de niveau d'énergie associé

Des considérations de physique statistique, dans lesquelles nous n'entrerons pas, font en sorte que la condition d'acceptation ou rejet d'un changement de coût peut être définie par la fonction `changementAccepte` présentée à l'algorithme 12, fonction qui dépend, tel qu'expliqué précédemment, de la température globale ambiante T.

On peut noter que plus la température T est élevée, plus $e^{-dE/T}$ sera grand, donc plus grande sera la probabilité que l'augmentation de niveau d'énergie soit acceptée, et inversement lorsque la température diminue.

10.3 Le programme MPD réalisant l'algorithme de recuit simulé

Un programme MPD de recuit simulé est présenté dans l'algorithme 13 ($1^{ère}$, $2^{ième}$ et $3^{ième}$ parties). Ce programme est une adaptation du programme (écrit en SuperPascal) présenté par Brich Hansen. La structure générale du programme est donnée par le diagramme hiérarchique présenté à la figure 18.

L'algorithme 13 ($1^{ière}$ partie) présente les procédures et fonction suivantes :

- `selectionnerSousSeq` : procédure qui permet de sélectionner aléatoirement les deux villes pour lesquelles la sous-séquence de villes intermédiaires sera inversée si la variante est acceptée.

```
procedure selectionnerSousSeq( Sequence c, res int si, res int j, res real modifCout )
# ROLE
#  Selectionner une sous-sequence de c a inverser et determiner le changement de cout associe.
# PRECONDITION
#   c denote un circuit valide.
# POSTCONDITION
#   si et j : bornes inf. et sup. de la sous-sequence a inverser.
#   modifCout : changement de cout resultant de l'inversion de la sous-seq.
{
  int i, sj;

  i = int(random(0, n)) + 1;  # Note : u <= random(u, v) < v.
  j = int(random(0, n)) + 1;
  si = i % n + 1;
  sj = j % n + 1;
  if ( i != j ) {
    int cij   = W[element(c,  i)][element(c,  j)];
    int cisi  = W[element(c,  i)][element(c, si)];
    int cjsj  = W[element(c,  j)][element(c, sj)];
    int csisj = W[element(c, si)][element(c, sj)];
    modifCout = real( cij + csisj - cisi - cjsj );
  } else {
    modifCout = 0.0;
  }
}

procedure changementAccepte( real modifCout, real temp ) returns bool r
# ROLE
#  Determiner si le changement de cout est accepte ou non.
{
  if (modifCout <= 0.0) {
    r = true;
  } else {
    # Si le cout augmente, on l'accepte avec une certaine probabilite.
    # Cette probabilite diminue lorsque la temperature diminue.
    r = exp(-modifCout/temp) > random();
  }
}

procedure modifierCircuit( Sequence c, int i, int j )
# POSTCONDITION
#   Les elements de la sous-sequence de c allant de i a j sont inverses.
#   (Voir diagramme dans les notes de cours.)
# EXEMPLE
#   c = [10, 20, 30, 40, 50], i = 2, j = 5 => c' = [10, 50, 40, 30, 20]
{
  int nij = (j - i + n) % n + 1;
  for [k = 1 to nij/2] {
    echanger( c, (i+k-2) % n + 1, (j-k+n) % n + 1 );
  }
}
```

Algorithme 13: Programme MPD pour recuit simulé ($1^{ière}$ partie)

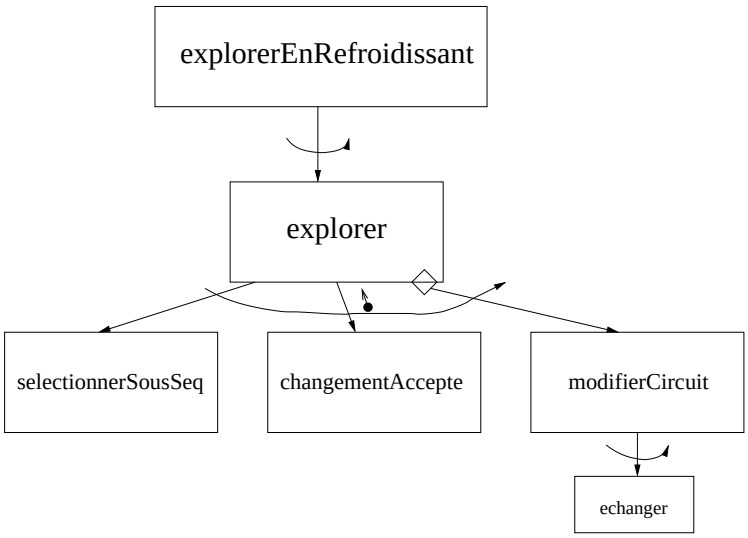

Figure 18: Diagramme hiérarchique illustrant la structure générale du programme MPD pour l'algorithme de recuit simulé appliqué au problème du voyageur de commerce

- `changementAccepte` : fonction qui détermine si un changement de niveau d'énergie est acceptable ou non en fonction de la température ambiante (version MPD de la fonction présentée à l'algorithme 12).
- `modifierCircuit` : procédure qui inverse la sous-séquence de villes entres les deux villes choisies aléatoirement pour générer le nouveau circuit, tel qu'illustré graphiquement à la figure 17.

  Notons que cette procédure utilise l'opération `echanger`, définie sur les séquences (type abstrait 1) et dont le code est omis, qui échange les deux éléments de la séquence qui sont indiqués par les index reçus en argument.

L'algorithme 13 ($2^{ième}$ partie) présente les deux procédures suivantes :

- `explorer` : procédure qui génère diverses variantes de `c` à la température indiquée (`temp`). Les arguments `nbEssais` et `nbChangements`, tel qu'indiqué précédemment, permettent de borner le nombre de variantes qui seront générées et examinées. La procédure d'exploration se termine, à une température donnée, soit lorsqu'un certain nombre d'essais ont été effectués, soit lorsqu'un certain nombre de changements ont été effectués (nombre de variantes acceptées).
- `explorerEnRefroidissant` : procédure, du niveau supérieur, qui effectue le processus d'exploration pour un nombre donné d'itérations, réduisant la température à chaque itération.

  Le paramètre `alpha`, choisi tel que $0.0 <$ `alpha` < 1.0, dénote le facteur de refroidissement. Évidemment, pour que le refroidissement soit graduel, `alpha` doit être plus près de 1.0 que de 0.0 — on verra plus loin l'effet de ce facteur sur les résultats produits.

Quant à l'algorithme 13 ($3^{ième}$ partie), il présente le corps du programme principal, qui génère un circuit initial à l'aide de la procédure `genererCheminInitial` — notons que ce circuit n'est pas généré aléatoirement, la séquence `c` $= [v_1, \ldots, v_n]$ étant simplement utilisée comme circuit initial — et qui amorce ensuite le processus d'exploration par refroidissement. Les paramètres indiqués ont été sélectionnés en se basant sur diverses considérations expliquées par Brich Hansen. Ces paramètres définissent ce qu'on peut appeler l'*échéancier*

```
procedure explorer( Sequence c, real temp,
                    int nbEssais, int nbChangements )
# ROLE
#   Effectuer un certain nombre de changements aleatoires du circuit c
#   pour la temperature indiquee (temp).
#
#   Les parametres nbEssais et nbChangements determinent les conditions
#   pour terminer la recherche a cette temperature : lorsque le nombre
#   d'essais indique aura ete effectue ou lorsque nbChangements auront
#   ete effectues.
{

  int ne = 0;
  int nc = 0;
  while (ne < nbEssais & nc < nbChangements) {
    int debut, fin;  # Bornes de la sous-sequence a inverser.
    real modifCout;  # Modification de cout (changement de niveau d'energie).

    selectionnerSousSeq( c, debut, fin, modifCout );
    if ( changementAccepte(modifCout, temp) ) {
      modifierCircuit( c, debut, fin );
      nc += 1;
    }
    ne += 1;
  }
}


procedure explorerEnRefroidissant(
                  Sequence c,
                  real tempMax, real alpha,
                  int nbIterations, int nbEssais, int nbChangements )
{
  real temp = tempMax;

  for [k = 1 to nbIterations] {
    explorer( c, temp, nbEssais, nbChangements );
    temp = alpha * temp;
  }
}
```

Algorithme 13: Programme MPD pour recuit simulé ($2^{ième}$ partie)

```
# Puisqu'on suppose qu'un chemin direct existe en n'importe quelle deux villes
# (aucune entree de la matrice n'est INFINI), alors le chemin initial
# est simplement la sequence ordonnee des differentes villes.
procedure genererCheminInitial( int n ) returns Sequence s
{
  s = creer();
  for [i = 1 to n] { ajouterEnQueue(s, i); }
}

# Mise en marche du processus d'exploration (annealing).
#
# Le choix des parametres est "explique" dans les notes de cours.
# Note : tempsMax IN O(n) pcq. W[i][j] <= n

Sequence c = genererCheminInitial(n);
explorerEnRefroidissant( c, n, 0.95, int(50*log(n)), 100*n, 10*n );

# Impression du cout resultant
write( coutChemin(c) );
```

Algorithme 13: Programme MPD pour recuit simulé ($3^{ième}$ et dernière partie)

de refroidissement (*cooling schedule*), puisqu'ils déterminent la durée du processus de recuit simulé :

- Température initiale : `tempMax = n`.

  Cette température doit être du même ordre de grandeur que la distance maximale entre les villes. Dans la procédure de génération de la matrice des distances entre les villes (code non présenté), cette distance, pour `n` villes, est toujours comprise entre 1 et `n`.

- Facteur de refroidissement : `alpha = 0.95`.

  On verra un peu plus loin pourquoi ce facteur a été choisi : des valeurs inférieures (refroidissement trop rapide) ou supérieures (trop lent) semblent donner de moins bons résultats.

- Nombre d'itérations : `nbIterations` $= 50\ log_e$ `n`.

  Le nombre d'itérations, en relation avec le facteur de refroidissement, doit conduire à une température finale qui soit d'ordre $\Theta(1)$, pour faire en sorte que les changements qui conduisent à des augmentation de niveau d'énergie soient rejetés.

- Nombre d'essais : `nbEssais = 100*n`.

- Nombre de changements : `nbChangements = 10*n`.

Notons que les règles pour le choix de ces différents paramètres sont plutôt *empiriques* (heuristiques). En fait, comme l'indique Brich Hansen, ces paramètres sont déterminés en se basant sur certaines des considérations mentionnées plus haut et sur "*the folklore of simulated annealing*".

10.4 Trace d'exécution et résultats d'exécution

Pour illustrer plus concrètement le fonctionnement du processus de recuit simulé, nous présentons, à la figure 19, une trace partielle d'exécution pour `n = 5`, trace générée à partir du chemin initial `c = [1, 2, 3, 4, 5]` et qui indique les modifications effectuées au circuit `c`. L'objectif de cette courte trace est d'illustrer la caractéristique fondamentale du recuit

```
[1, 2, 3, 4, 5] (15) => [1, 2, 4, 3, 5] (16)
[1, 2, 4, 3, 5] (16) => [2, 1, 4, 3, 5] (16)
[2, 1, 4, 3, 5] (16) => [4, 1, 2, 5, 3] (16)
[4, 1, 2, 5, 3] (16) => [4, 1, 2, 5, 3] (16)
[4, 1, 2, 5, 3] (16) => [2, 1, 4, 3, 5] (16)
[2, 1, 4, 3, 5] (16) => [2, 5, 3, 4, 1] (16)
[2, 5, 3, 4, 1] (16) => [2, 5, 3, 4, 1] (16)
[2, 5, 3, 4, 1] (16) => [2, 5, 3, 4, 1] (16)
[2, 5, 3, 4, 1] (16) => [5, 2, 1, 4, 3] (16)
[5, 2, 1, 4, 3] (16) => [5, 1, 2, 4, 3] (16)
[5, 1, 2, 4, 3] (16) => [3, 1, 2, 4, 5] (14)
[3, 1, 2, 4, 5] (14) => [1, 3, 2, 4, 5] (15)
[1, 3, 2, 4, 5] (15) => [1, 3, 2, 4, 5] (15)
[1, 3, 2, 4, 5] (15) => [4, 3, 2, 1, 5] (15)
[4, 3, 2, 1, 5] (15) => [4, 5, 1, 2, 3] (15)
[4, 5, 1, 2, 3] (15) => [2, 1, 5, 4, 3] (15)
[2, 1, 5, 4, 3] (15) => [3, 4, 5, 1, 2] (15)
[3, 4, 5, 1, 2] (15) => [5, 4, 3, 2, 1] (15)
[5, 4, 3, 2, 1] (15) => [2, 4, 3, 5, 1] (16)
...                     ...
```

Figure 19: Trace (partielle) d'exécution pour `n = 5`

simulé, à savoir qu'un changement peut être accepté même s'il produit un moins bon résultat. Toutefois, comme nous le verrons un peu plus loin, à long terme, ceci a généralement un effet positif sur la solution finale obtenue. L'exemple de la figure 19 indique les 20 premiers circuits générés et acceptés pour `n = 5` — le nombre entre parenthèses indique le coût du circuit, alors que chaque ligne indique que la séquence de gauche est transformée en celle de droite.

On remarque donc que, bien que dans la procédure `selectionnerSousSeq` `i` et `j` sont toujours choisis tels que $\texttt{i} \neq \texttt{j}$, il peut quand même arriver que la modification effectuée par `modifierCircuit` *n'ait aucun effet*. Ceci survient lorsque la sous-séquence de villes intermédiaires est vide, c'est-à-dire, lorsque $\texttt{j} = \texttt{i}+1 = \texttt{si}$. Ce cas spécial pourrait évidemment être évité en modifiant légèrement le code.

Pour la trace complète de cette exécution, on peut calculer les statistiques suivantes (les autres appels correspondent à des changements du circuit, mais qui n'ont aucun effet sur le coût) :

Nombre total d'appels à `modifierCircuit`	4000
Nombre d'appels qui correspondent à des diminutions de coût	230
Nombre d'appels qui correspondent à des augmentations de coût	220
Nombre d'appels qui n'ont aucun effet sur le circuit courant	1126

On peut aussi déterminer que les divers circuits générés sont de coût 13 à 18 inclusivement. Pour chacun de ces coûts, les nombres d'occurences des circuits générés et acceptés sont les suivants :

Coût	Nombre d'occurences
13	2362
14	359
15	786
16	396
17	71
18	26

Notons que pour `n = 5`, le résultat final produit par l'algorithme de recuit simulé est bien le circuit de coût minimum (13).

n	Prog. dyn.	*Sim. ann.*
2	2	2
3	3	3
4	6	6
5	8	8
6	10	10
7	13	14
8	13	13
9	19	19
10	25	25
11	30	30
12	29	29
13	33	33

Tableau 3: Comparaisons des résultats produits par l'algorithme de programmation dynamique vs. le recuit simulé pour des petites valeurs de `n`

Une illustration graphique du comportement du recuit simulé est présentée à l'annexe A. Les figures présentées dans cette annexe représentent les différents circuits générés par une exécution du programme de recuit simulé sur une instance du problème comptant 32 villes réparties aléatoirement sur une grille 100×100 (variante géométrique, ou euclidienne, du problème).

Pour des petites valeurs de `n` ($n \leq 13$), valeurs pour lesquelles l'algorithme exact basé sur la programmation dynamique (voir section 5) peut être utilisé, on peut voir, tel que cela est illustré dans le tableau 3, que les résultats produits par l'algorithme de recuit simulé sont presque toujours identiques aux résultats optimaux. Dans cette série, la seule valeur différente est pour `n = 7`. En fait, sur cinq séries d'exécution différentes (c'est-à-dire, avec des germes différents utilisés pour initialiser le générateur de nombres aléatoires), cette série est la seule où les deux algorithmes ont produit des résultats qui n'étaient pas identiques ; dans tous les autres cas, le résultat produit par l'algorithme de recuit simulé était donc toujours le résultat optimal.

Par contre, les deux algorithmes diffèrent de façon importante au niveau des performances. Si on examine le temps d'exécution pour ces deux programmes — tous deux exécutés sur un PC roulant sous Linux —, on obtient les temps d'exécution suivants pour quelques valeurs de `n` :

`n`	Prog. dyn.	*Sim. ann.*
9	0.11	0.04 s
13	29.40 s	0.10 s
99	_ _	3.80 s
199	_ _	10.00 s

Pour `n` supérieur à 13, l'algorithme de programmation dynamique ne produit aucune réponse après plusieurs minutes d'attente. Par contre, l'algorithme de recuit simulé produit une réponse *en moins de 10 secondes* pour `n = 199`! Évidemment, il n'est pas possible de déterminer si la solution ainsi produite est d'un coût qui est *vraiment* près du minimum optimal.

Pour une comparaison plus détaillée des différentes stratégies de solution, voir l'annexe B.

10.5 Choix (empirique) des paramètres

Le choix des paramètres de l'échéancier de refroidissement, comme on l'a indiqué précédemment, se fait généralement de façon relativement informelle et empirique. Nous allons maintenant examiner l'effet de deux des paramètres.

Nombre total d'itérations

N	`n` = 11	`n` = 33	`n` = 66	`n` = 99
1	66	480	1732	3818
10	48	204	597	1093
20	37	78	191	305
50	28	78	183	302
100	28	78	183	302

Tableau 4: Effets du nombre d'itérations (N) sur le résultat produit pour différents nombres de villes (`n`)

Dans ce qui suit, fixons tout d'abord le taux de refroidissment comme suit (on verra pourquoi plus bas) : `alpha` = 0.95. Le nombre total d'itérations doit être $\Theta(log\ \texttt{n})$, pour arriver à une température finale qui soit $\Theta(1)$. Soit N la constante multiplicative utilisée dans la définition du nombre total d'itérations : `nbIterations` $= N * log(\texttt{n})$.

On peut voir au tableau 4, pour différents `n` (nombre de villes), l'effet sur le résultat obtenu (coût du circuit trouvé par l'algorithme) pour $N = 1, 10, 20, 50, 100$.

Le passage de $N = 20$ à $N = 50$ semble produire une légère amélioration du résultat, ce qui ne semble pas être le cas lorsque N est augmenté à 100 — rappelons que ce facteur influence directement le temps d'exécution, et que passer de $N = 50$ à $N = 100$ double le temps d'exécution (nombre total d'itérations de refroidissement). C'est pour cette raison que $N = 50$ a été choisi précédemment dans l'algorithme 13.

Taux de refroidissment

`alpha`	`n` = 11	`n` = 33	`n` = 66	`n` = 99	`n` = 199
0.1	30	83	218	394	953
0.5	28	80	186	361	994
0.8	28	82	179	324	869
0.9	28	83	179	326	853
0.95	28	83	193	297	823
0.97	35	115	280	463	1188
0.99	44	289	1207	3106	7731

Tableau 5: Effets du taux de refroidissement (`alpha`) sur le résultat produit pour différents nombres de villes (`n`)

Pour un N fixé (à 50), examinons ensuite l'effet de la modification du facteur `alpha`, pour `alpha` $\in \{0.1, 0.5, 0.8, 0.9, 0.95, 0.97, 0.99\}$, et ce pour divers nombres de villes. Les résultats sont présentés au tableau 5.

On remarque donc que, pour chacun des `n`, le facteur de refroidissement ne doit conduire ni à refroidissement trop rapide (par ex., `alpha = 0.5`, où les solutions avec un coût élevé n'ont pas le temps d'être rejetées), ni à un refroidissement trop lent (par ex., `alpha = 0.99`, où de

nombreuses solutions avec coûts élevées sont acceptées). La valeur `alpha = 0.95` semble un bon compromis.

10.6 Comparaison avec un algorithme probabiliste utilisant une recherche strictement locale

La particularité principale de l'algorithme de recuit simulé est qu'il est possible que des solutions qui augmentent le coût soient acceptées. À première vue, il peut sembler bizarre, pour ne pas dire déraisonnable, d'accepter une solution dont le coût associé est supérieur. Or, ceci se comprend mieux lorsqu'on réalise qu'une fonction peut posséder de nombreux minimums locaux, lesquels peuvent être relativement éloignés du minimum global, tel que cela est illustré à la figure 20.

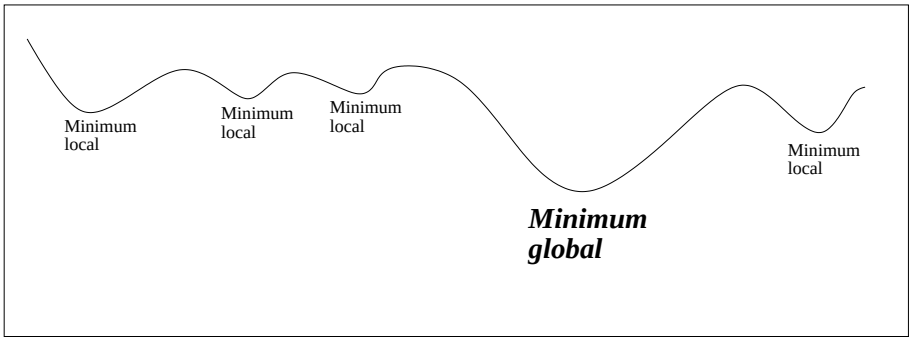

Figure 20: Minimums locaux vs. minimum global

L'algorithme de recuit simulé permet, en acceptant des augmentations du niveau d'énergie, de sortir d'une région contenant un minimum local non optimal possiblement pour aller vers une région qui conduira au minimum global. Ainsi, supposons que l'on modifie l'algorithme pour que toute augmentation de niveau d'énergie (de coût du circuit) soit *refusée*. Dans le code MPD présenté précédemment, ceci peut se faire simplement en modifiant la fonction `changementAccepte` (algorithme 13) comme suit :[13]

```
procedure changementAccepte( real modifCout, real temp ) returns bool r
{
  r = (modifCout <= 0.0);
}
```

Avec cette modification de la fonction `changementAccepte`, on constate (les résultats détaillés ont été omis) que pour des petites valeurs de `n` (4 et 7), il arrive que parfois le résultat obtenu soit légèrement meilleur lorsqu'on refuse toute augmentation du niveau d'énergie. Par contre, pour `n` = 10, 11 ou 12, il arrive plus fréquemment que le résultat obtenu *soit moins intéressant*.

En fait, cette détérioration des résultats devient encore plus marquée pour de grandes valeurs de `n` (mais pour lesquelles, évidemment, on ne connaît pas le coût optimal). Par exemple, les résultats (coût du circuit minimum) obtenus pour des exécutions de l'algorithme de recuit simulé comparés à l'exécution de l'algorithme de recherche locale (où toute augmentation de coût est refusée) pour diverses valeurs de `n` sont présentés au tableau 6 (les résultats sont ceux obtenus pour un germe initial de 1, les résultats pour d'autres germes étant tout

[13] En fait, dans le vrai code utilisé pour les tests, l'appel à `random()` a été conservé, mais son résultat n'est pas utilisé. Essentiellement, ceci a pour but de ne pas perturber et modifier la suite de nombres aléatoires qui est générée au cours de l'exécution du programme.

n	Recuit simulé	Recherche locale
40	109	131
80	225	280
120	418	483
160	599	763
199	803	1026

Tableau 6: Comparaisons des résultats produits par le *simulated annealing* vs. une recherche strictement locale

à fait similaires) : on voit que l'algorithme de recherche locale semble produire des résultats moins intéressants.

A Illustration *graphique* du comportement du recuit simulé

Cette section présente une illustration graphique du comportement du recuit simulé. Les figures présentées dans cette annexe représentent un certain nombre de circuits parmi tous ceux générés par une exécution du programme de recuit simulé sur une instance du problème comptant 32 villes. Ces villes sont réparties aléatoirement sur une grille 100×100 ; il s'agit donc ici d'une variante géométrique, ou euclidienne, du problème.

La figure 21 présente le circuit de départ, qui consiste simplement à aller de la ville i à la ville $i + 1$ en partant de la ville 1. La figure 32 présente le circuit final, c'est-à-dire la solution finale produite par l'algorithme. Quant aux autres figures, elles donnent l'état du circuit après divers nombres d'itérations.

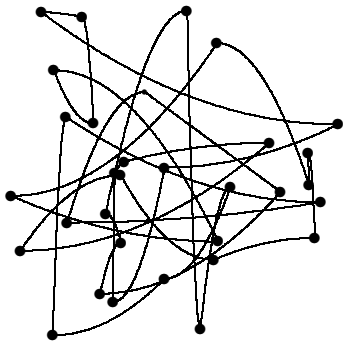

Figure 21: État initial (température = 32.00)

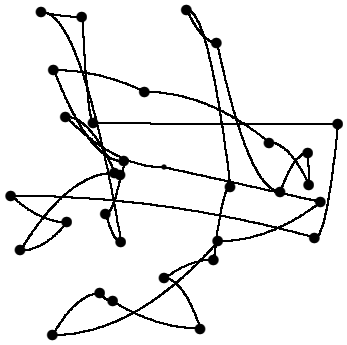

Figure 22: État après 5 itérations (température = 24.76)

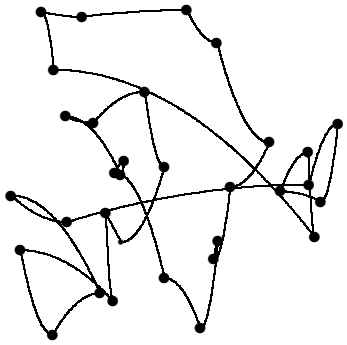

Figure 23: État après 10 itérations (température = 19.16)

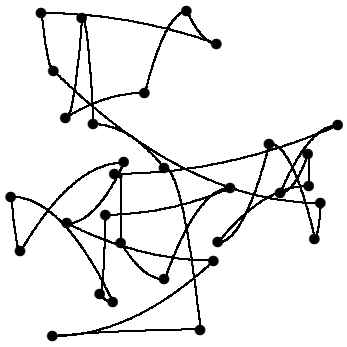

Figure 24: État après 15 itérations (température = 14.83)

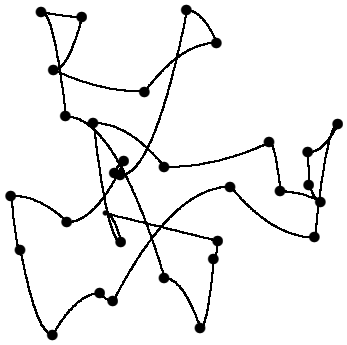

Figure 25: État après 20 itérations (température = 11.47)

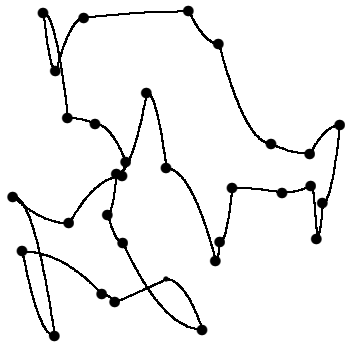

Figure 26: État après 25 itérations (température = 8.88)

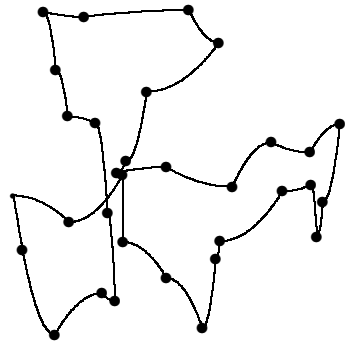

Figure 27: État après 30 itérations (température = 6.87)

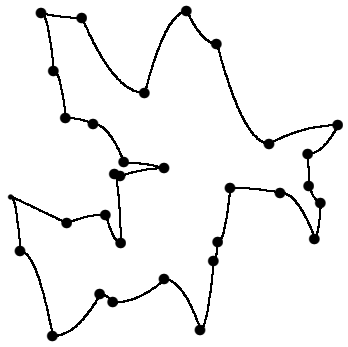

Figure 28: État après 40 itérations (température = 4.11)

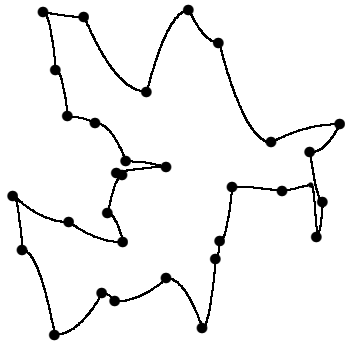

Figure 29: État après 50 itérations (température = 2.46)

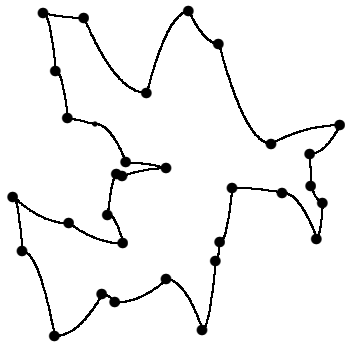

Figure 30: État après 100 itérations (température = 0.19)

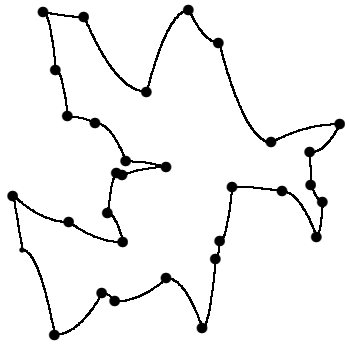

Figure 31: État après 150 itérations (température = 0.01)

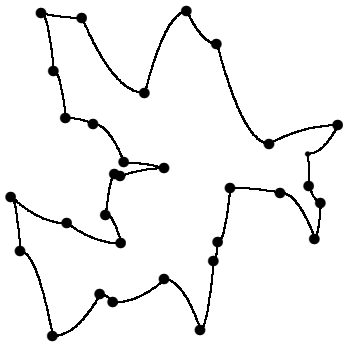

Figure 32: État final, après 173 itérations (température = 0.00)

B Comparaisons des résultats et des temps d'exécution des diverses heuristiques pour le problème du voyageur de commerce

Le tableau 7 présente les résultats (longueur du plus court chemin trouvé) et les temps d'exécution (en secondes, obtenus avec la commande Unix `time`) pour six heuristiques pour le problème du voyageur de commerce. Quatre tailles différentes de problème (n = 10, 50, 100 et 1000) ont été utilisées, générant dans chaque cas, de façon aléatoire, des graphes *géométriques complets* avec le nombre indiqué de villes (sommets), toutes les villes étant situées dans un carré de taille 10×10 — les distances sont donc des nombres réels (points-flottants), ce qui, dans le cas du recuit simulé, conduit à des temps d'exécution légèrement différents de ceux présentés à la section 10, où les distances étaient des entiers naturels.

Chaque algorithme a été exécuté quatre (4) fois, avec des germes différents pour initialiser le générateur de nombres aléatoires — tous les algorithmes, pour un germe donné, traitent donc exactement la même matrice. Les heuristiques comparées sont les suivantes, avec les paramètres indiqués — l'ordre de présentation correspond, en gros, à l'ordre croissant de la qualité des résultats produits :

- AL : Algorithme d'exploration aléatoire du graphe.
- HC1 : Exploration locale du voisinage avec sélection du premier voisin intéressant (*Hill Climbing First*), durant 50 itérations.
- RT : Recherche avec tabous, durant 20 itérations et avec une liste tabou de taille 20 (un nombre d'itérations égal à 50 rendait le temps d'exécution trop long). Notons qu'aucun résultat n'est indiqué pour 1000 villes, le temps d'exécution étant trop long.
- HCB : Exploration locale du voisinage avec sélection du meilleur voisin (*Hill Climbing Best*), durant 50 itérations.
- NN : Algorithme déterministe glouton allant toujours au voisin le plus près.
- RS : Recuit simulé, avec les paramètres spécifiés dans les notes de cours (50 itérations, facteur de refroidissement de 0.95, etc.)

Comme on le constate, le recuit simulé semble produire les meilleurs résultats, mais avec toutefios des temps d'exécution plus élevés. Par contre, le temps reste quand même raisonnable, surtout si on compare avec les algorithmes exacts qui ne peuvent traiter des instances de problèmes comptant un grand nombre de villes, à l'exception de l'heuristique

du meilleur voisin qui produit des résultats intéressants à coût faible. Notons que ce bon comportement de l'heuristique du voisin le plus près se produit parce qu'il s'agit de graphes aléatoires ; dans le pire des cas, on peut toujours construire un graphe pour lequel le voisin le plus près produira une très mauvaise solution.

$n = 10$			$n = 50$		
	Résultat	*Temps*		*Résultat*	*Temps*
AL	42.94	0.02	AL	274.92	0.01
AL	62.85	0.01	AL	257.57	0.01
AL	57.65	0.02	AL	250.78	0.02
AL	40.24	0.00	AL	248.85	0.00
HC1	24.26	0.01	HC1	165.15	0.01
HC1	30.65	0.01	HC1	155.51	0.02
HC1	28.02	0.00	HC1	148.15	0.03
HC1	26.68	0.01	HC1	199.07	0.03
RT	24.00	0.16	RT	94.40	17.64
RT	30.29	0.15	RT	85.86	17.71
RT	28.02	0.16	RT	85.91	17.84
RT	26.68	0.16	RT	95.11	17.82
HCB	24.00	0.02	HCB	61.29	0.30
HCB	30.29	0.02	HCB	61.59	0.27
HCB	28.02	0.02	HCB	58.59	0.31
HCB	26.68	0.00	HCB	64.21	0.29
NN	26.38	0.00	NN	70.40	0.02
NN	32.78	0.01	NN	69.21	0.02
NN	32.21	0.00	NN	74.95	0.03
NN	29.78	0.01	NN	76.44	0.01
RS	24.00	0.25	RS	58.17	6.10
RS	30.29	0.25	RS	58.91	6.02
RS	28.02	0.25	RS	56.26	5.99
RS	26.68	0.24	RS	61.60	5.99

$n = 100$			$n = 1000$		
	Résultat	*Temps*		*Résultat*	*Temps*
AL	587.27	0.02	AL	5263.53	1.78
AL	518.44	0.04	AL	5191.38	1.51
AL	488.57	0.04	AL	5173.77	1.55
AL	519.14	0.02	AL	5229.54	1.59
HC1	430.28	0.02	HC1	5153.75	0.98
HC1	423.25	0.02	HC1	5054.45	0.96
HC1	402.43	0.02	HC1	5134.18	0.95
HC1	443.41	0.03	HC1	4968.28	0.97
RT	274.20	141.69	RT	?	?
RT	279.71	143.46	RT	?	?
RT	253.58	142.10	RT	?	?
RT	289.23	144.04	RT	?	?
HCB	135.82	1.19	HCB	4392.05	124.05
HCB	125.88	1.19	HCB	4262.20	124.02
HCB	125.58	1.18	HCB	4352.38	123.97
HCB	139.11	1.20	HCB	4227.59	123.93
NN	92.25	0.07	NN	293.54	55.64
NN	95.06	0.08	NN	289.34	55.47
NN	100.47	0.08	NN	286.23	55.46
NN	98.96	0.10	NN	298.07	55.49
RS	80.85	15.81	RS	248.52	548.71
RS	79.58	15.64	RS	245.50	549.40
RS	82.22	15.65	RS	248.76	548.48
RS	79.61	15.63	RS	245.72	551.28

Tableau 7: Tableau comparatif des résultats et temps d'exécution pour diverses heuristiques pour le voyageur de commerce

Références

[AK89] E. Aarts and J. Korst. *Simulated Annealing and Boltzmann Machines.* John Wiley & Sons, Chichester, UK, 1989. [QA402.5A27].

[AO93] G.R. Andrews and R.A. Olsson. *The SR Programming Language.* Benjamin/Cummings Publishing, 1993. [QA76.73S68A54].

[DS04] M. Dorigo and T. Stutzle. *Ant Colony Optimization.* The MIT Press, 2004.

[Han95] P.B. Hansen. *Studies in Computational Science—Parallel Programming Paradigms.* Prentice-Hall, Englewood Cliffs, NJ, 1995.

[JM97] D.S. Johnson and L.A. McGeoch. The traveling salesman problem: A case study in local optimization. In E.H.L. Aarts and J.K Lenstra, editors, *Local Search in Combinatorial Optimization*, pages 215–310. John Wiley & Sons, 1997.

[NN04] R. Neapolitan and K. Naimipour. *Foundations of Algorithms Using C++ Pseudocode (Third edition).* Jones and Bartlett Publishers, 2004.

Cette partie des notes de cours est basée principalement sur les références suivantes :

- [NN04, Sections 3.6, 5.6, 6.2 et 9.5]
- [AO93, Chapitre 16]
- [Han95, Chapitre 11]
- [AK89, Chapitres 1 et 2]
- [JM97]
- Transparents produits par Cédric Chauve, à l'automne 2004, dans le cadre du cours INF7440 :

  `http://www.lacim.uqam.ca/~chauve/Enseignement/INF7440/planning.html`