# Chapitre 7 : branchements et coupures

Carole Porrier
carole.porrier@univ-paris13.fr

Département d'informatique
IUT de Villetaneuse

6 mai 2024

Paradigmes de développement universels

# Table des matières

# Table des matières

# Récursivité

- Lorsqu'on utilise la **récursivité**, on fait appel à deux **mécanismes** fondamentaux:
  - On effectue un **branchement**, c'est-à-dire qu'on **divise** le problème en **plus petits problèmes**;
  - On utilise également la **marche-arrière** (*backtracking*) lorsqu'on termine l'**appel récursif**.
- Dans la plupart des cas, ces mécanismes sont gérés à l'aide de la **pile système**.
- On peut représenter l'**espace** des appels à l'aide d'une structure **arborescente**.

# Le problème de la monnaie (1/3)

- Considérons un montant d'argent $x$;
- On souhaite former le montant $x$ en utilisant le **moins de pièces/billets** possible;
- En euros, les **devises possibles** sont

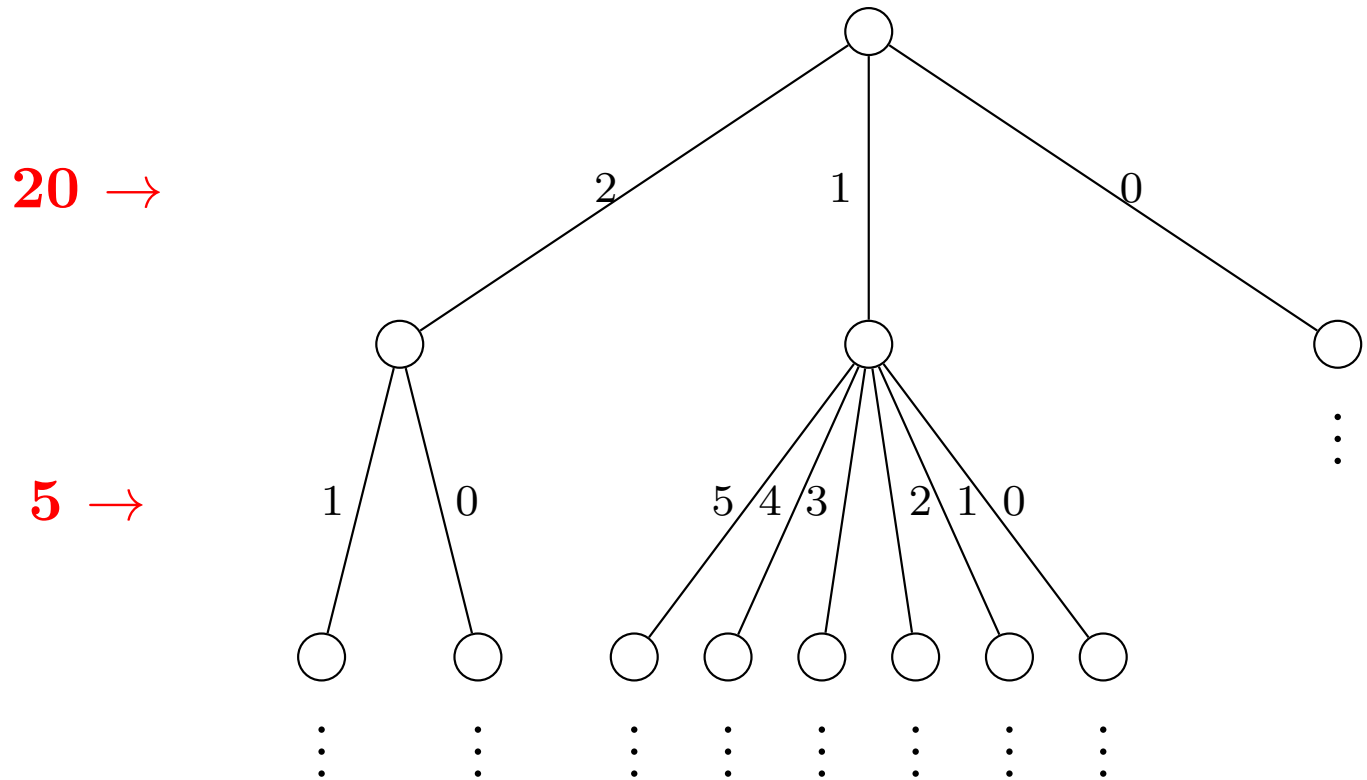
$$500.00 \quad 200.00 \quad 100.00 \quad 50.00 \quad 20.00 \quad 10.00 \quad 5.00$$
$$2.00 \quad 1.00 \quad 0.50 \quad 0.20 \quad 0.10 \quad 0.05 \quad 0.02 \quad 0.01$$

- **Exemple**: $x = 46.35$. Alors on prend

$$\mathbf{2} \times 20.00 + \mathbf{1} \times 5.00 + \mathbf{1} \times 1.00 + \mathbf{1} \times 0.20 + \mathbf{1} \times 0.10 + \mathbf{1} \times 0.05$$

- **Total**: 7 pièces/billets.
- Est-ce que c'est le **minimum**?

# Le problème de la monnaie (2/3)




- Chaque **chemin** dans l'arbre correspond à **une solution**;
- Le **nombre** de chemins croît **très rapidement**.

# Le problème de la monnaie (3/3)

- Le modèle s'implémente facilement en **Prolog**:

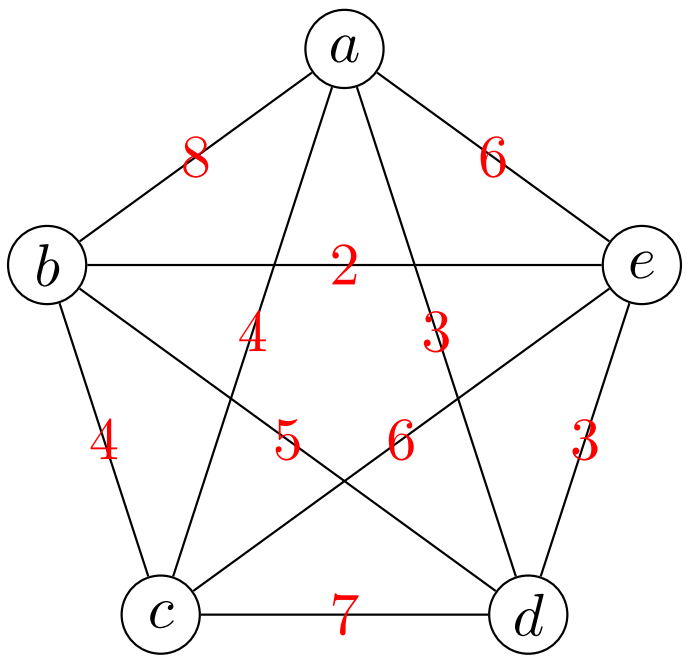
```
devises(E) :- E = [50000, 20000, 10000, 5000, 2000, 1000,
    500, 200, 100, 50, 20, 10, 5, 2, 1].

monnaie(0, [], []).

monnaie(M, [E|ES], [P|PS]) :-
    M >= 0,
    Pmax is M div E,
    between(0, Pmax, P1),
    P is Pmax - P1,
    M1 is M - E * P,
    monnaie(M1, ES, PS).
```

- Utile pour éviter de **tronquer les listes** à l'affichage:

```
set_prolog_flag(answer_write_options,[max_depth(0)]).
```

# Le voyageur de commerce (1/5)

- Un autre problème classique en **optimisation** et en **intelligence artificielle**;
- Un voyageur de commerce doit visiter $n$ **villes exactement** une fois;
- Quelle est la **distance** d'une tournée minimale?



# Le voyageur de commerce (2/5)

- Une **tournée** peut être représentée par une **permutation**;
- Par exemple, si les **villes** sont $a$, $b$, $c$, $d$ et $e$, alors les tournées possibles sont:

$$(a, b, c, d, e), (a, b, c, e, d), (a, b, d, c, e), (a, b, d, e, c), \ldots$$

- Au total, il y a $(n-1)!$ tournées possibles, si $n$ est le **nombre de villes**.
- On peut utiliser **Prolog** pour énumérer les tournées et **calculer** une tournée optimale.

# Le voyageur de commerce (3/5)

- On modélise d'abord le **graphe des villes**:

```
% Distance entre les villes
distance(a, b, 8).
distance(a, c, 4).
distance(a, d, 3).
distance(a, e, 6).
distance(b, c, 4).
distance(b, d, 5).
distance(b, e, 2).
distance(c, d, 7).
distance(c, e, 6).
distance(d, e, 3).

% Pour que la distance soit symétrique
dist(X, Y, D) :- distance(X, Y, D); distance(Y, X, D).
```

# Le voyageur de commerce (4/5)

- On peut calculer la distance d'une **tournée**:

```
% Distance d'une tournée
distanceTournee([V|VS], D) :-
    append([V|VS], [V], V1),
    distanceChemin(V1, D).

% Distance d'un chemin
distanceChemin([_], 0).
distanceChemin([V1,V2|VS], D) :-
    distanceChemin([V2|VS], D1),
    dist(V1, V2, D2),
    D is D1 + D2.
```

# Le voyageur de commerce (5/5)

- Puis on calcule une **tournée optimale**:

```
% Tournée
tournee(V, T, D) :-
    permutation(V, T),
    distanceTournee(T, D).

% Tournée minimale
distanceMinimale(V, Dmin) :-
    findall(D, tournee(V, _, D), DS),
    min_list(DS, Dmin).

% Tournée minimale
tourneeMinimale(V, Tmin) :-
    tournee(V, Tmin, D),
    distanceMinimale(V, D),
    distanceTournee(Tmin, D).
```

# `findall, bagof, setof`

- Lorsqu'on souhaite énumérer **toutes les solutions** d'un but;
- `findall(Template, Goal, Bag)`: retourne toutes les **instantiations** de `Template` qui vérifient `Goal` et place le résultat dans `Bag`;
- `bagof(Template, Goal, Bag)`: idem que `findall`, mais échoue s'il n'y a pas de solution;
- `setof(Template, Goal, Bag)`: idem que `bagof`, mais sans doublon.

# Table des matières

# Calculs inutiles?

- Il est possible en Prolog de trouver **toutes** les solutions à un problème;
- Mais parfois, on n'a besoin que d'en calculer **une**;
- Ou encore on souhaite simplement prouver l'**existence** d'une solution;
- Lorsque c'est le cas, on peut utiliser une **coupure**;
- Il s'agit essentiellement de dire à Prolog de ne pas faire de **marche-arrière** (*backtracking*).

# Utilité

- On distingue essentiellement **trois** raisons d'utiliser la coupure;
- **Cas 1**: On souhaite **confirmer** à Prolog qu'il a choisi la bonne règle pour satisfaire un but;
- **Cas 2**: On souhaite **indiquer** à Prolog qu'il ne satisfera jamais le but et qu'il cesse d'essayer;
- **Cas 3**: On souhaite éviter la génération de solutions **alternatives** afin d'économiser du temps de calcul.

# Cas 1: confirmation d'une coupure (1/2)

```
% Version 1
% Version de base
somme1(0, 0).
somme1(N, S) :-
    N1 is N - 1,
    somme1(N1, S1),
    S is S1 + N.

% Version 2
% Version avec coupure
somme2(0, 0) :- !.
somme2(N, S) :-
    N1 is N - 1,
    somme2(N1, S1),
    S is S1 + N.
```

# Cas 1: confirmation d'une coupure (2/2)

```
% Version 3
% Cas négatif avec coupure
somme3(N, 0) :- N =< 0, !.
somme3(N, S) :-
    N1 is N - 1,
    somme3(N1, S1),
    S is S1 + N.

% Version 4
% Cas négatif avec négation
somme4(N, 0) :- N =< 0.
somme4(N, S) :-
    \+(N = 0),
    N1 is N - 1,
    somme4(N1, S1),
    S is S1 + N.
```

# Cas 2: Forcer l'abandon

```
% Vérifier si deux listes sont disjointes
disjoint([], _) :- !.
disjoint(_, []) :- !.
disjoint([X|_], L) :- member(X, L), !, fail.
disjoint([_|XS], L) :- disjoint(XS, L).
```

- Dès qu'on a une **liste vide**, c'est disjoint et on arrête d'explorer;
- Dès qu'on trouve un élément (`X`) dans les deux listes, on **arrête** l'exploration avec un **échec**;
- Sinon, on **poursuit**.

# Lien entre coupure et négation

- Le prédicat de **négation** `\+` est défini comme suit:

```
\+ X :- X, !, fail.
\+ _.
```

- **Attention!** Pas équivalent au **non-logique**:

```
?- member(X, [a,b,c]).
X = a ;
X = b ;
X = c.

?- \+ (\+ member(X, [a,b,c])).
true.
```

# Cas 3: Améliorer l'efficacité (1/2)

- Considérons le prédicat suivant:

```
max(X, Y, Y) :- X =< Y.
max(X, Y, X) :- X > Y.
```

- Clairement, le **maximum** est unique;
- Si le **premier but** est satisfait, il est inutile d'explorer le **deuxième but**;
- On peut donc écrire

```
max(X, Y, Y) :- X =< Y, !.
max(X, Y, X) :- X > Y.
```

- Cette coupure est dite **verte**, car elle ne change pas la sémantique du prédicat.

# Cas 3: Améliorer l'efficacité (2/2)

- Considérons maintenant le prédicat suivant

```
max(X, Y, Y) :- X =< Y, !.
max(X, Y, X).
```

- Nous avons un comportement inattendu avec l'appel `max(2, 3, 2)`, qui retourne vrai!
- On peut rectifier

```
max(X, Y, Z) :- X =< Y, !, Y = Z.
max(X, Y, X).
```

- Ici, la coupure est **nécessaire** et on dit qu'elle est **rouge**.
- Le programme est **plus efficace**, mais **moins lisible**.

# Exemple complet

Jeu de tic-tac-toe en Prolog.