Chapitre 8 : Introduction aux méthodes heuristiques

J.-F. Scheid

v2.1

Plan du chapitre

I. Introduction

II. Quelques méthodes heuristiques

1. Algorithme A^*
2. Recuit simulé
3. Algorithmes génétiques
4. Algorithme de colonies de fourmis

I. Introduction

Pour les problèmes de grandes tailles :

- pas de temps de calculs "raisonnables" avec les méthodes **exactes**
- recherche de "bonnes" solutions **approchées**.

I. Introduction

Pour les problèmes de grandes tailles :

- pas de temps de calculs "raisonnables" avec les méthodes **exactes**
- recherche de "bonnes" solutions **approchées**.

Heuristiques : règles empiriques simples basées sur l'expérience (résultats déjà obtenus) et sur l'analogie. Généralement, on n'obtient pas la solution optimale mais une solution **approchée**.

I. Introduction

Pour les problèmes de grandes tailles :

- pas de temps de calculs "raisonnables" avec les méthodes **exactes**
- recherche de "bonnes" solutions **approchées**.

Heuristiques : règles empiriques simples basées sur l'expérience (résultats déjà obtenus) et sur l'analogie. Généralement, on n'obtient pas la solution optimale mais une solution **approchée**.

Méta-heuristiques : algorithmes d'optimisation (généralement de type stochastique) combinant plusieurs approches heuristiques.

Exemple d'heuristique : initialisation du "Branch-and-Bound" pour le problème du sac-à-dos (calcul de $\overline{F}$).

Exemple d'heuristique : initialisation du "Branch-and-Bound" pour le problème du sac-à-dos (calcul de $\overline{F}$).

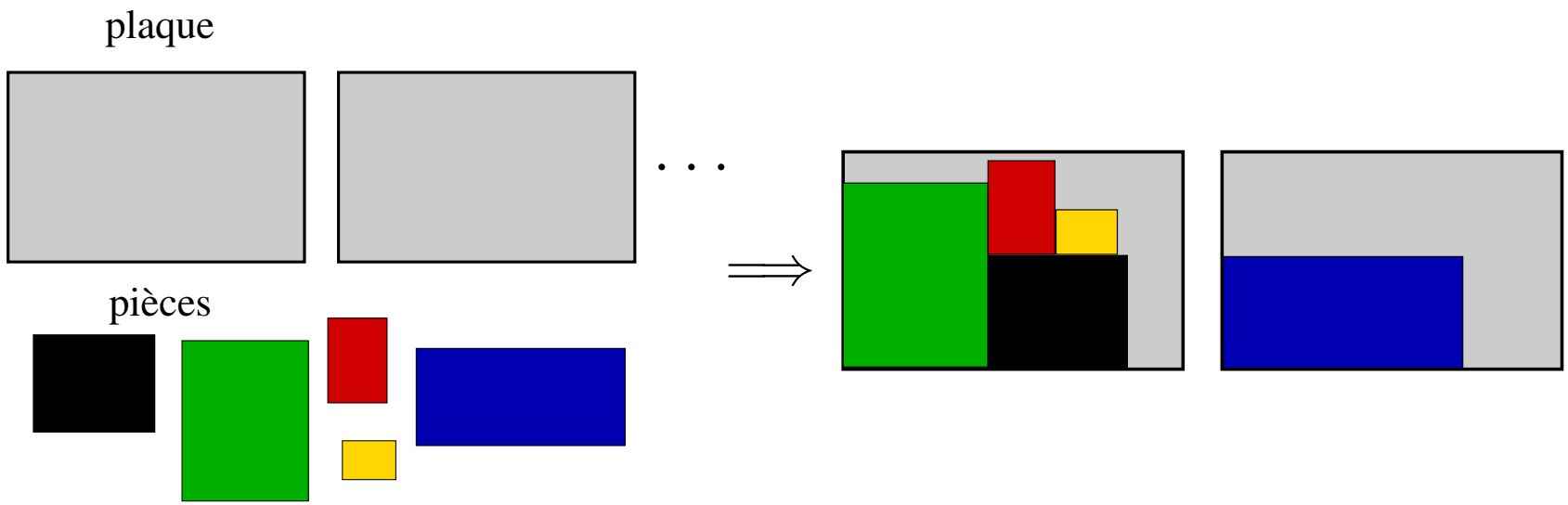
Algorithme glouton ("greedy") : construction d'une solution *réalisable* en se ramenant à une suite de décisions qu'on prend à chaque fois au mieux en fonction d'un **critère d'optimisation local** sans remettre en question les décisions déjà prises. Généralement, la solution obtenue est **approchée**.

Intérêt : algorithmes simples à implémenter.

Défaults : solutions approchées obtenues plus ou moins bonnes, critère local ("myopie").

Exemple : Placement optimal de pièces 2D (Bin Packing).

On dispose de plaques rectangulaires toutes identiques dans lesquelles on veut placer des pièces rectangulaires sans chevauchement. Les pièces à placer ont des dimensions différentes.



On veut trouver le placement pour minimiser le nombre de plaques utilisées.

Algorithme glouton : *trier les pièces en fonction de leur taille et placer d'abord les pièces les plus grandes.*

Quelques méthodes approchées

1. Méthode heuristique en programmation dynamique : Algorithme A^*

Quelques méthodes approchées

1. Méthode heuristique en programmation dynamique : Algorithme A^*
2. Recuit simulé
3. Algorithmes génétiques
4. Algorithme de colonies de fourmis, recherche Tabou, $\cdots$

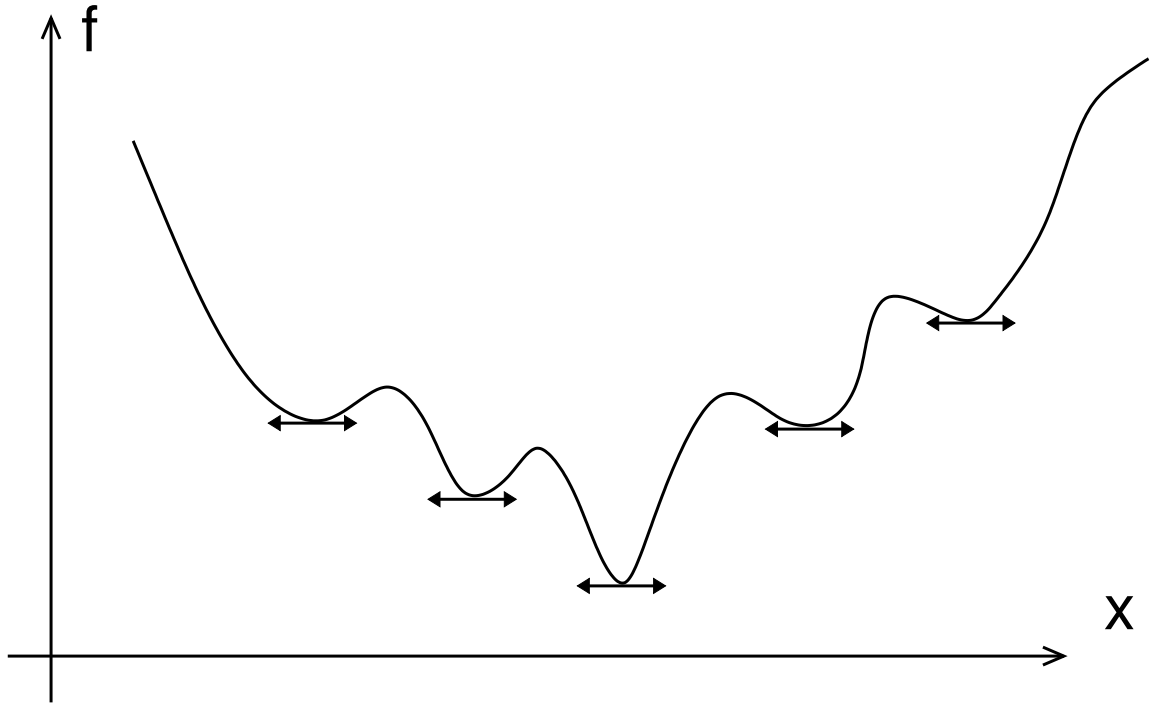
Intérêt et domaines d'applications des méthodes heuristiques

Problèmes d'optimisation de la forme

$$\min_{\mathbf{x} \in X} \mathbf{f}(\mathbf{x})$$
$$\begin{cases} \text{sous des contraintes} \\ \mathbf{g}(\mathbf{x}) \leq \mathbf{b} \end{cases}$$

La fonction $\mathbf{f}$ peut-être vectorielle (optimisation multi-objectifs) et les contraintes $\mathbf{g}$ sont généralement vectorielles.

De façon générale, la fonction objectif $\mathbf{f}$ et les contraintes $\mathbf{g}$ sont **nonlinéaires**.



Difficultés : Plusieurs **minima locaux** possibles $\Rightarrow$ les méthodes classiques d'optimisation non-linéaire (basées sur le calcul différentiel) sont parfois coûteuses et incapables de capturer la solution **globale**.

Méthodes méta-heuristiques : capacité à s'extraire d'un minimum *local* pour aller vers un minimum *global*

II. Quelques méthodes heuristiques

1) Algorithme A^*

Proposé par Hart,Nilsson,Raphael (1968). Il s'agit d'un algorithme *heuristique* de programmation dynamique qui fournit généralement une solution *approchée*. Algorithme très utilisé pour sa rapidité (jeux vidéo, ...)

II. Quelques méthodes heuristiques

1) Algorithme A^*

Proposé par Hart,Nilsson,Raphael (1968). Il s'agit d'un algorithme *heuristique* de programmation dynamique qui fournit généralement une solution *approchée*. Algorithme très utilisé pour sa rapidité (jeux vidéo, ...)

C'est un algorithme de recherche d'un plus court chemin dans un graphe allant de x_0 à x_F.

- Il est basé sur une **évaluation heuristique** à chaque sommet pour *estimer* le meilleur chemin qui y passe.
- Les sommets sont parcourus en fonction de cette évaluation heuristique : on retient le sommet où l'évaluation est la plus petite.

Dans l'algorithme de Dijkstra, on effectue une recherche exhaustive parmi tous les sommets. Dans l'algorithme A^*, on réduit l'ensemble des sommets à explorer.

Recherche d'un plus court chemin dans un graphe allant de x_0 à x_F.

$F(x)$: longueur du plus court chemin allant de x_0 au sommet x.
$L^*(x)$: longueur du plus court chemin allant du sommet x à x_F.

Pour tout sommet x, on suppose qu'on sait calculer une approximation minorante $L(x)$ de la longueur minimale $L^*(x)$ i.e. $L^*(x) \geq L(x)$.
L est une évaluation heuristique de L^*.

Recherche d'un plus court chemin dans un graphe allant de x_0 à x_F.

$F(x)$: longueur du plus court chemin allant de x_0 au sommet x.
$L^*(x)$: longueur du plus court chemin allant du sommet x à x_F.

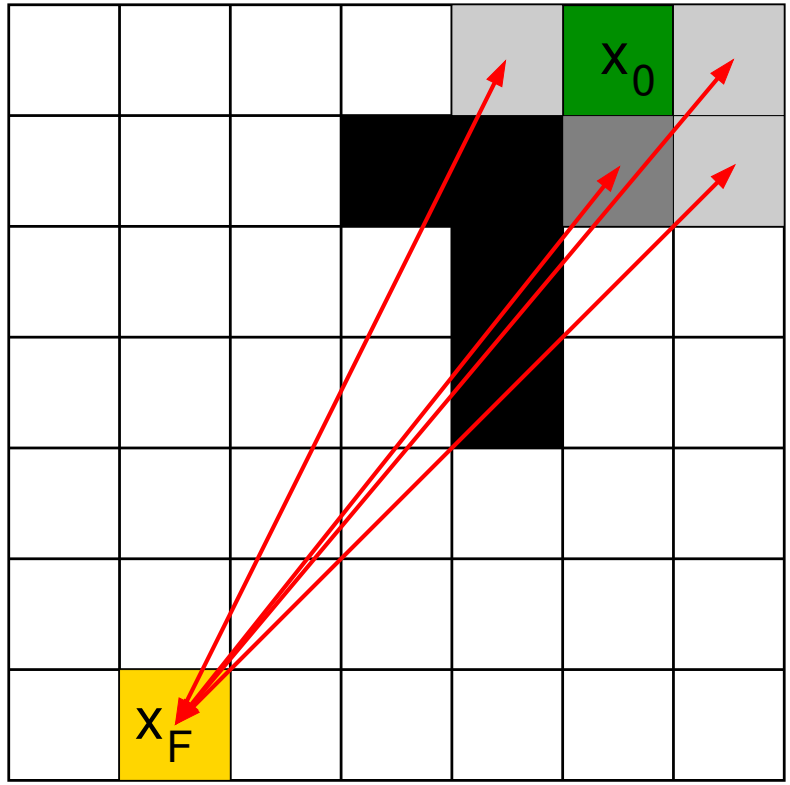
Pour tout sommet x, on suppose qu'on sait calculer une approximation minorante $L(x)$ de la longueur minimale $L^*(x)$ i.e. $L^*(x) \geq L(x)$.
L est une évaluation heuristique de L^*.

- On construit progressivement une liste de sommets S telle que pour tout $x' \in S$, la longueur $F(x')$ a déjà été calculée.
- A chaque étape, on ajoute à une autre liste $\overline{S}$ le sommet x tel que

$$\phi(x) = F(x) + L(x) = \min_{x' \in S} \left(F(x') + L(x') \right)$$

- On utilise les successeurs de x pour continuer
- On arrête dès que le sommet x_F est rencontré

Exemple : Plus court chemin dans une grille avec obstacle.



Exemple d'évaluation heuristique $L(x)$: distance (à vol d'oiseau) de x à x_F.

On construit deux listes :

S *liste ouverte* : contient les sommets (successeurs) **à examiner**.
$\overline{S}$ *liste fermée* : contient tous les sommets **déjà examinés**, qui appartiennent au chemin solution.

On commence par le sommet $x = x_0$

On construit deux listes :

S *liste ouverte* : contient les sommets (successeurs) **à examiner**.
$\overline{S}$ *liste fermée* : contient tous les sommets **déjà examinés**, qui appartiennent au chemin solution.

On commence par le sommet $x = x_0$

1. On regarde tous les successeurs (voisins) x' de x

On construit deux listes :

S *liste ouverte* : contient les sommets (successeurs) **à examiner**.
$\overline{S}$ *liste fermée* : contient tous les sommets **déjà examinés**, qui appartiennent au chemin solution.

On commence par le sommet $x = x_0$

1. On regarde tous les successeurs (voisins) x' de x
2. Si un sommet x' *n'a jamais été rencontré* (i.e. x' n'est ni dans la liste S ni dans la liste $\overline{S}$), alors on l'ajoute dans S.

On construit deux listes :

S *liste ouverte* : contient les sommets (successeurs) **à examiner**.
$\overline{S}$ *liste fermée* : contient tous les sommets **déjà examinés**, qui appartiennent au chemin solution.

On commence par le sommet $x = x_0$

1. On regarde tous les successeurs (voisins) x' de x
2. Si un sommet x' *n'a jamais été rencontré* (i.e. x' n'est ni dans la liste S ni dans la liste $\overline{S}$), alors on l'ajoute dans S.
3. Si un sommet x' *a déjà été rencontré* (i.e. x' est déjà dans S ou bien dans $\overline{S}$) et si x' a un nouveau coût ϕ plus petit, alors on met à jour le coût.
   De plus, si $x' \in \overline{S}$ alors on l'enlève de $\overline{S}$ et on l'ajoute à S (pour qu'il soit à nouveau examiné).

On construit deux listes :

S *liste ouverte* : contient les sommets (successeurs) **à examiner**.
$\overline{S}$ *liste fermée* : contient tous les sommets **déjà examinés**, qui appartiennent au chemin solution.

On commence par le sommet $x = x_0$

1. On regarde tous les successeurs (voisins) x' de x
2. Si un sommet x' *n'a jamais été rencontré* (i.e. x' n'est ni dans la liste S ni dans la liste $\overline{S}$), alors on l'ajoute dans S.
3. Si un sommet x' *a déjà été rencontré* (i.e. x' est déjà dans S ou bien dans $\overline{S}$) et si x' a un nouveau coût ϕ plus petit, alors on met à jour le coût.
   De plus, si $x' \in \overline{S}$ alors on l'enlève de $\overline{S}$ et on l'ajoute à S (pour qu'il soit à nouveau examiné).
4. On détermine le meilleur sommet x de toute la liste ouverte S (si $S = \emptyset$ alors arrêt, pas de solution).

On construit deux listes :

S *liste ouverte* : contient les sommets (successeurs) **à examiner**.
$\overline{S}$ *liste fermée* : contient tous les sommets **déjà examinés**, qui appartiennent au chemin solution.

On commence par le sommet $x = x_0$

1. On regarde tous les successeurs (voisins) x' de x
2. Si un sommet x' *n'a jamais été rencontré* (i.e. x' n'est ni dans la liste S ni dans la liste $\overline{S}$), alors on l'ajoute dans S.
3. Si un sommet x' *a déjà été rencontré* (i.e. x' est déjà dans S ou bien dans $\overline{S}$) et si x' a un nouveau coût ϕ plus petit, alors on met à jour le coût.
   De plus, si $x' \in \overline{S}$ alors on l'enlève de $\overline{S}$ et on l'ajoute à S (pour qu'il soit à nouveau examiné).
4. On détermine le meilleur sommet x de toute la liste ouverte S (si $S = \emptyset$ alors arrêt, pas de solution).
5. On met x dans la liste fermée $\overline{S}$ et on le supprime de la liste S.

On construit deux listes :

S *liste ouverte* : contient les sommets (successeurs) **à examiner**.
$\overline{S}$ *liste fermée* : contient tous les sommets **déjà examinés**, qui appartiennent au chemin solution.

On commence par le sommet $x = x_0$

1. On regarde tous les successeurs (voisins) x' de x
2. Si un sommet x' *n'a jamais été rencontré* (i.e. x' n'est ni dans la liste S ni dans la liste $\overline{S}$), alors on l'ajoute dans S.
3. Si un sommet x' *a déjà été rencontré* (i.e. x' est déjà dans S ou bien dans $\overline{S}$) et si x' a un nouveau coût ϕ plus petit, alors on met à jour le coût.
   De plus, si $x' \in \overline{S}$ alors on l'enlève de $\overline{S}$ et on l'ajoute à S (pour qu'il soit à nouveau examiné).
4. On détermine le meilleur sommet x de toute la liste ouverte S (si $S = \emptyset$ alors arrêt, pas de solution).
5. On met x dans la liste fermée $\overline{S}$ et on le supprime de la liste S.
6. On itère avec le sommet courant x (retour en 1)

Algorithme A^*

$S = \{x_0\}$; $\overline{S} = \emptyset$; $F(x_0) = 0$.

Tant que $S \neq \emptyset$

- Déterminer $x \in S$ tq $\phi(x) = F(x) + L(x) = \min_{x' \in S}(\phi(x'))$
- $S = S \setminus \{x\}$; $\overline{S} = \overline{S} \cup \{x\}$
- Si $x = x_F$ alors arrêt (solution trouvée x)

Pour tout $x' \in$ Successeur(x)

Si $x' \notin S \cup \overline{S}$

$S = S \cup \{x'\}$

$F(x') = F(x) + l(x, x')$; $\phi(x') = F(x') + L(x')$

Sinon Si $F(x) + l(x, x') + L(x') < \phi(x')$

$F(x') = F(x) + l(x, x')$; $\phi(x') = F(x') + L(x')$

Si $x' \in \overline{S}$ alors $S = S \cup \{x'\}$ et $\overline{S} = \overline{S} \setminus \{x'\}$

Fin si

Fin pour

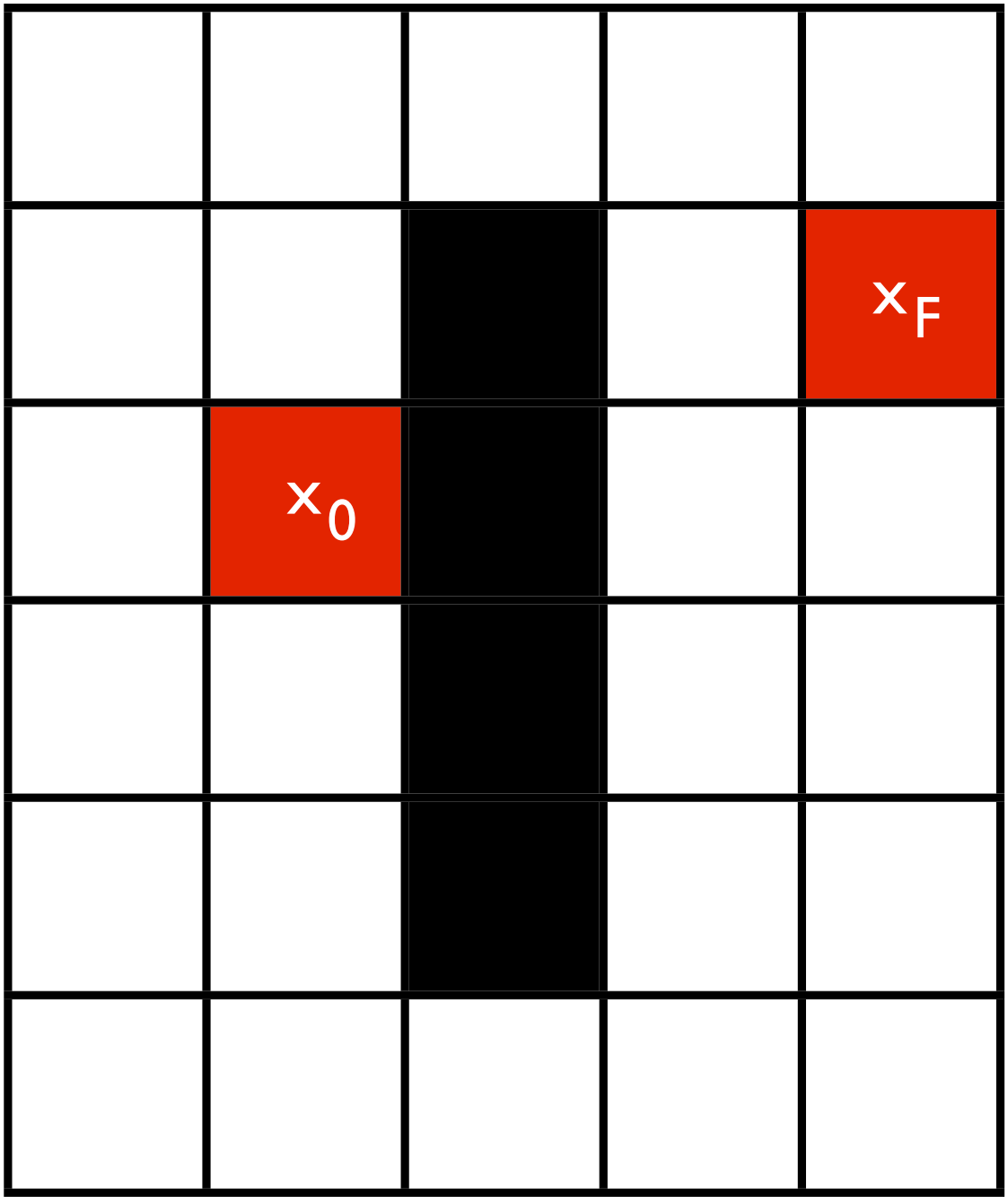
Fin tant que

Un exemple

Recherche d'un plus court chemin dans une grille avec obstacle. Les déplacements autorisés sont horizontaux et verticaux.

Choix de l'heuristique :

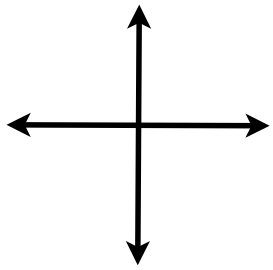
$$\begin{aligned} L(x) &= \text{distance de Manhattan du sommet } x \text{ au sommet d'arrivée } x_F. \\ &= \text{nombre de déplacements horizontaux et verticaux pour aller de } x \text{ à } x_F. \end{aligned}$$

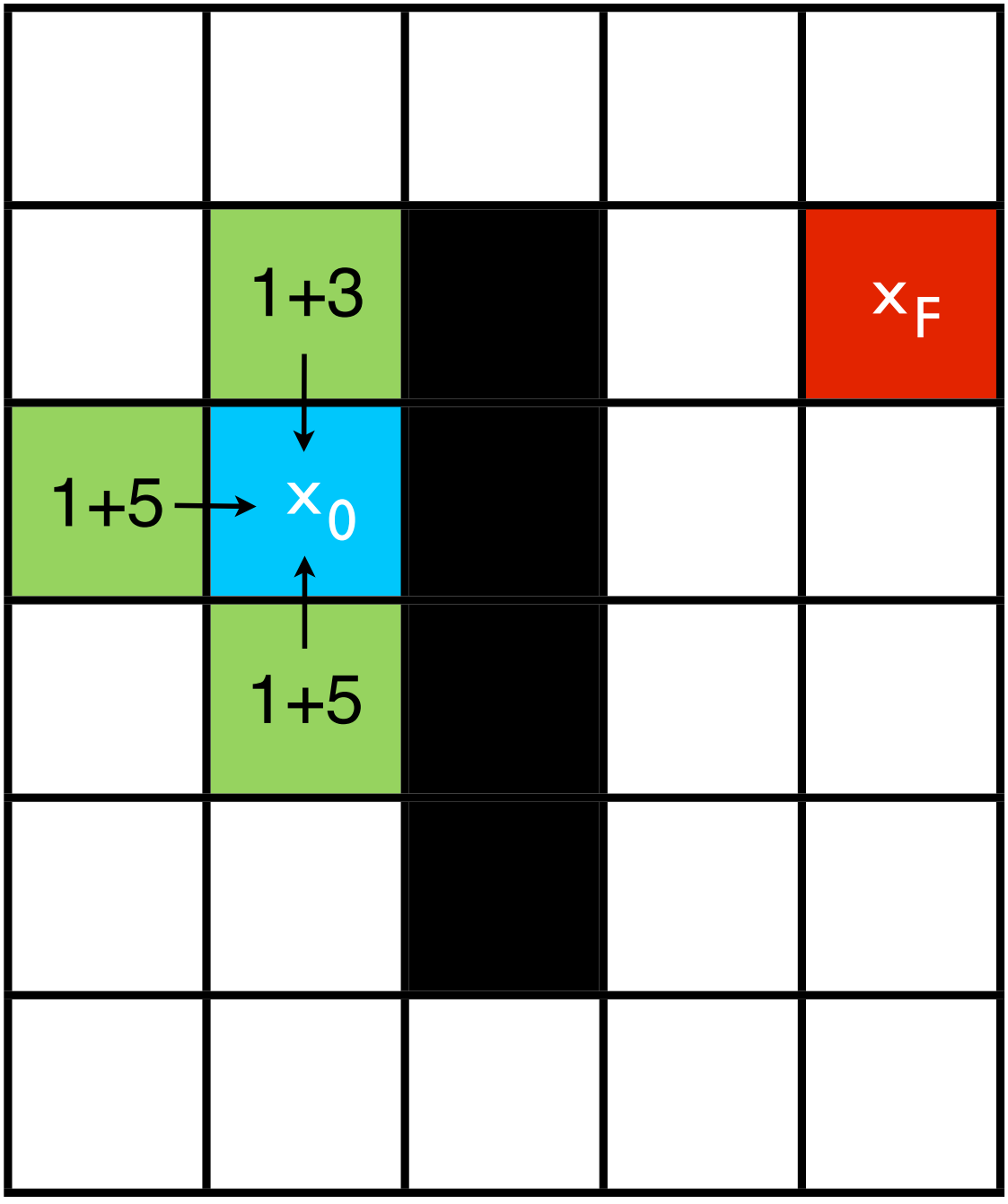


x_0 sommet de départ

x_F sommet d'arrivée

Déplacements horizontaux/verticaux





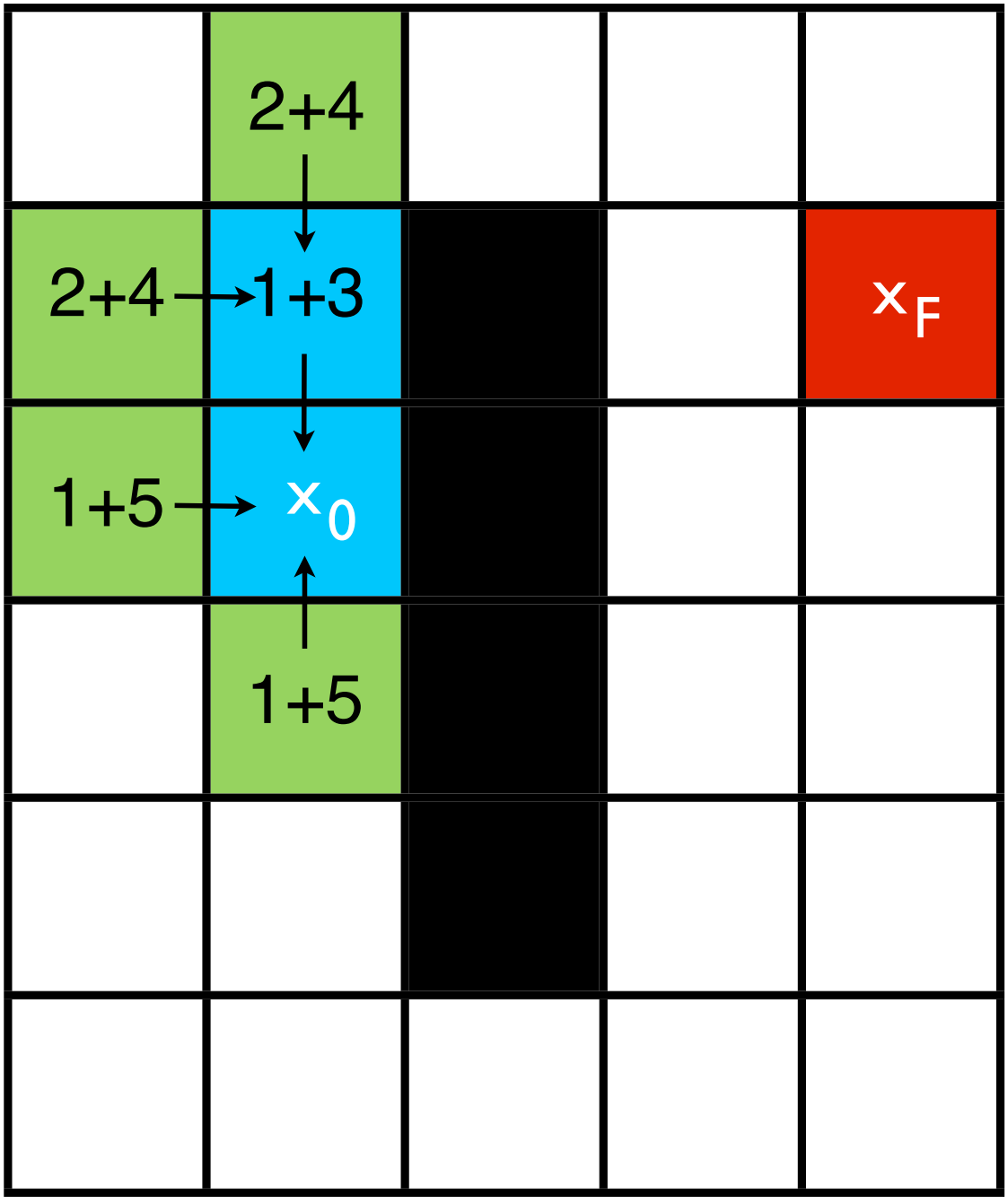
sommet déjà examiné ($\overline{S}$)

sommet à examiner (S)

$$\phi(x) = F(x) + L(x)$$

$F(x)$: distance de x_0 à x

$L(x)$: distance de x à x_F



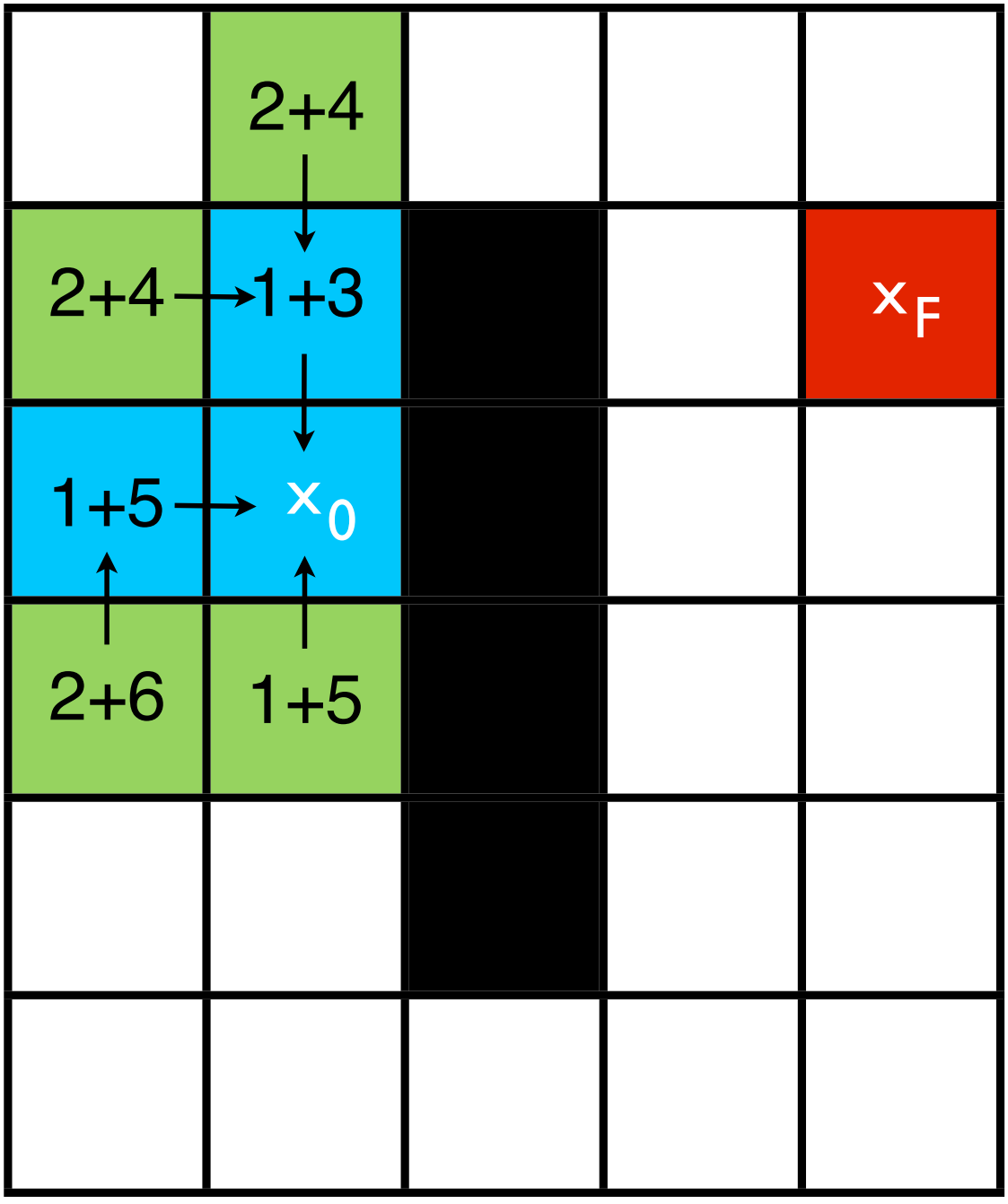
sommet déjà examiné ($\overline{S}$)

sommet à examiner (S)

$$\phi(x) = F(x) + L(x)$$

$F(x)$: distance de x_0 à x

$L(x)$: distance de x à x_F



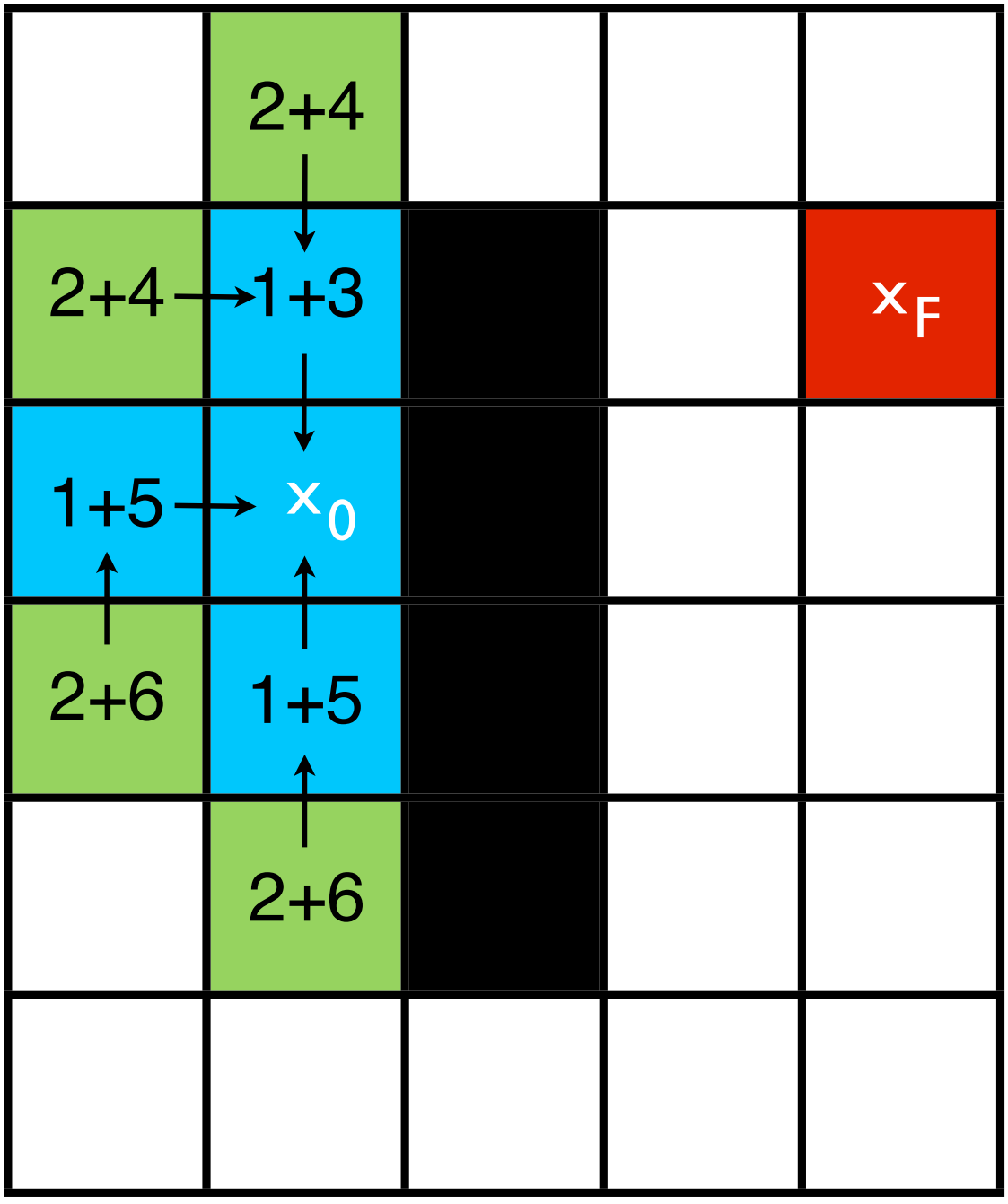
sommet déjà examiné ($\overline{S}$)

sommet à examiner (S)

$$\phi(x) = F(x) + L(x)$$

$F(x)$: distance de x_0 à x

$L(x)$: distance de x à x_F



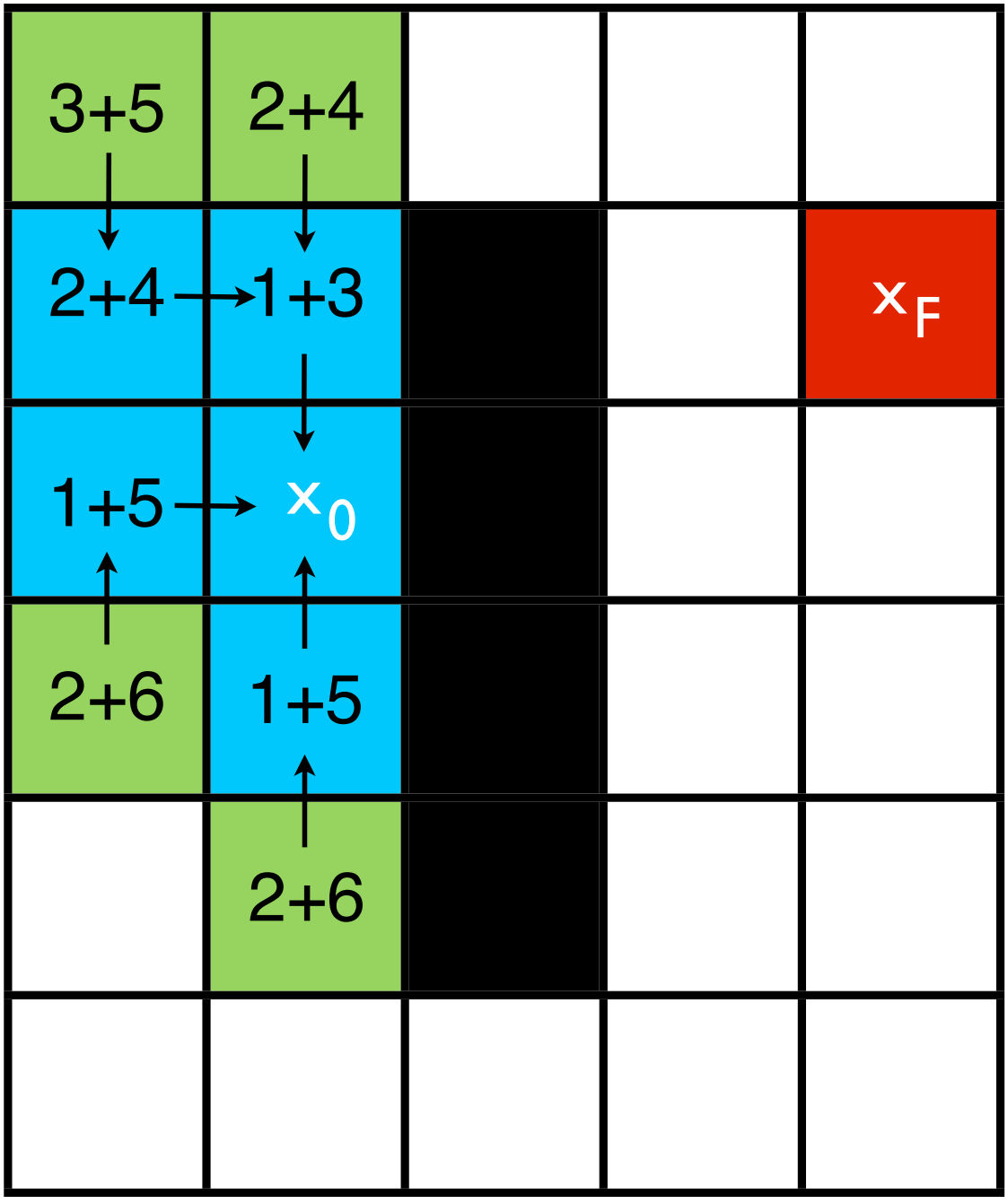
sommet déjà examiné ($\overline{S}$)

sommet à examiner (S)

$$\phi(x) = F(x) + L(x)$$

$F(x)$: distance de x_0 à x

$L(x)$: distance de x à x_F



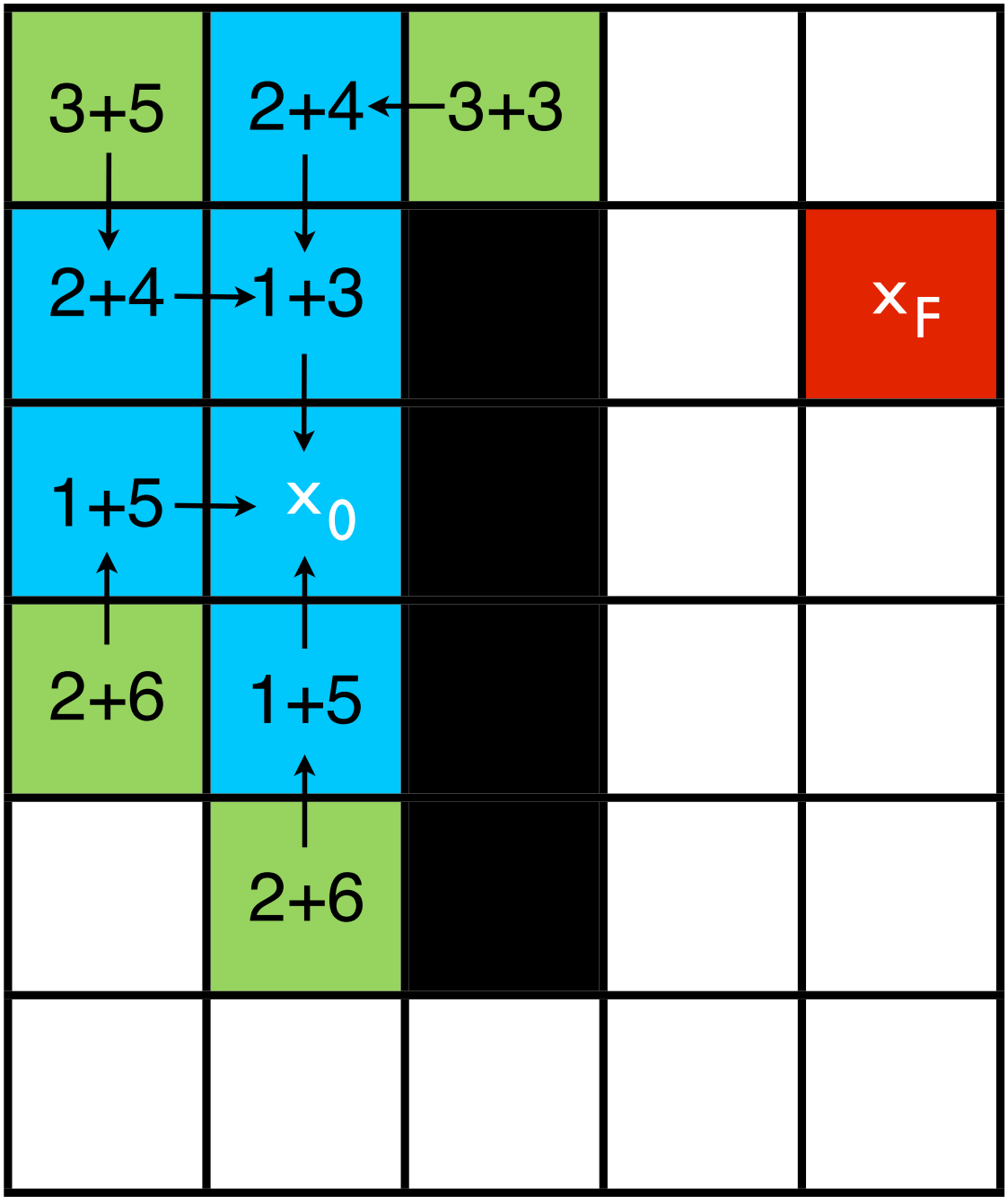
sommet déjà examiné ($\overline{S}$)

sommet à examiner (S)

$$\phi(x) = F(x) + L(x)$$

$F(x)$: distance de x_0 à x

$L(x)$: distance de x à x_F



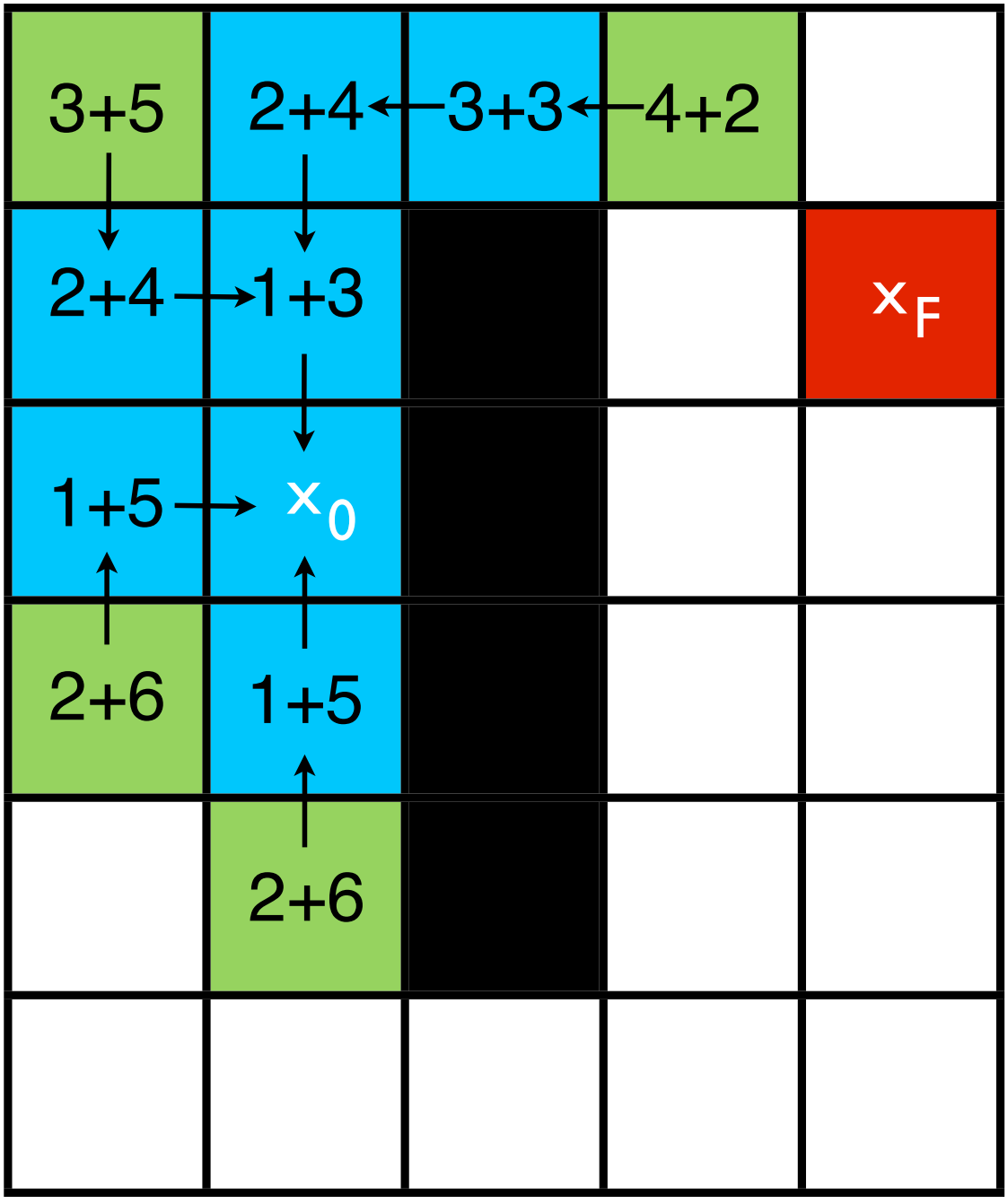
sommet déjà examiné ($\overline{S}$)

sommet à examiner (S)

$$\phi(x) = F(x) + L(x)$$

$F(x)$: distance de x_0 à x

$L(x)$: distance de x à x_F



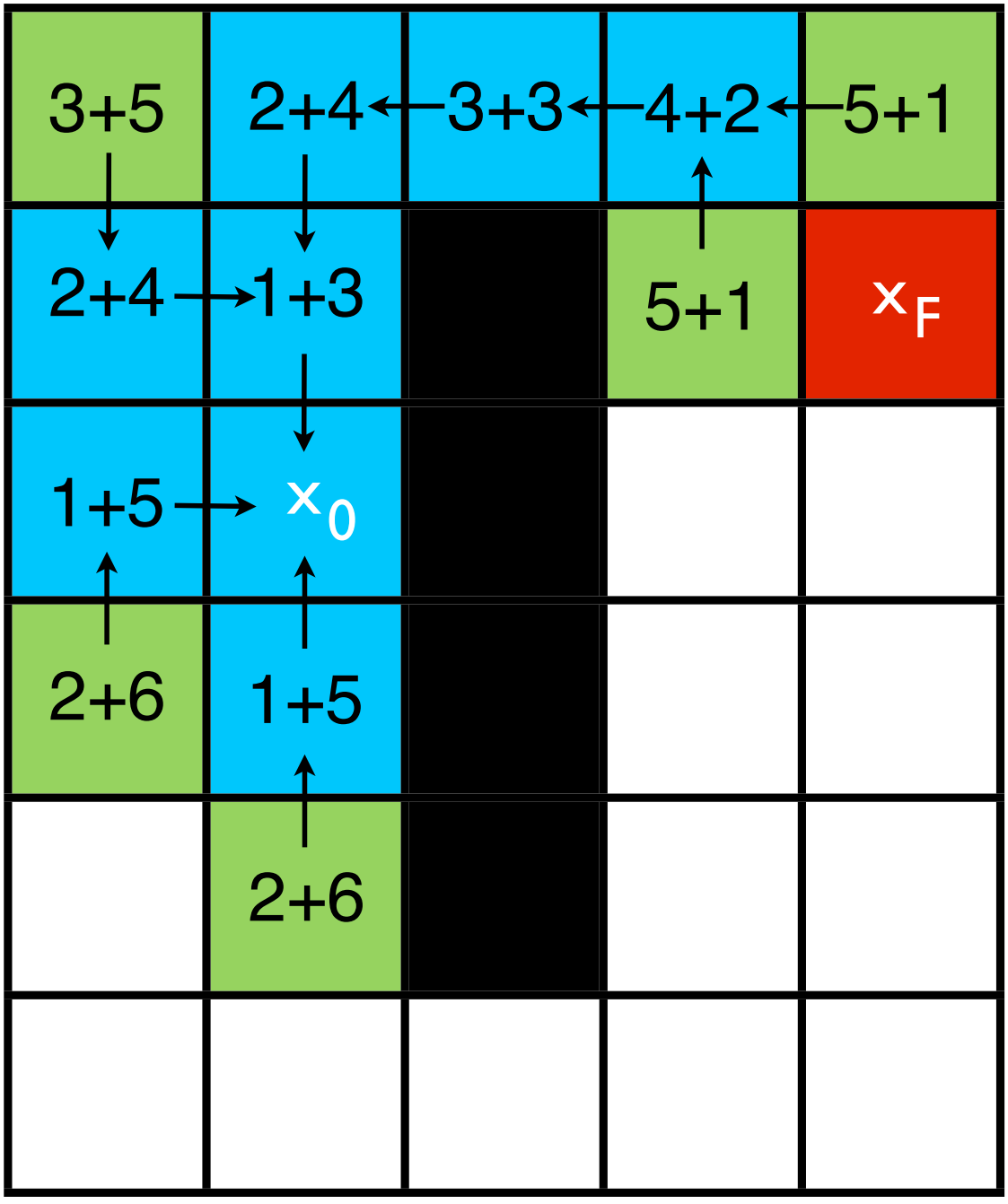
sommet déjà examiné ($\overline{S}$)

sommet à examiner (S)

$$\phi(x) = F(x) + L(x)$$

$F(x)$: distance de x_0 à x

$L(x)$: distance de x à x_F



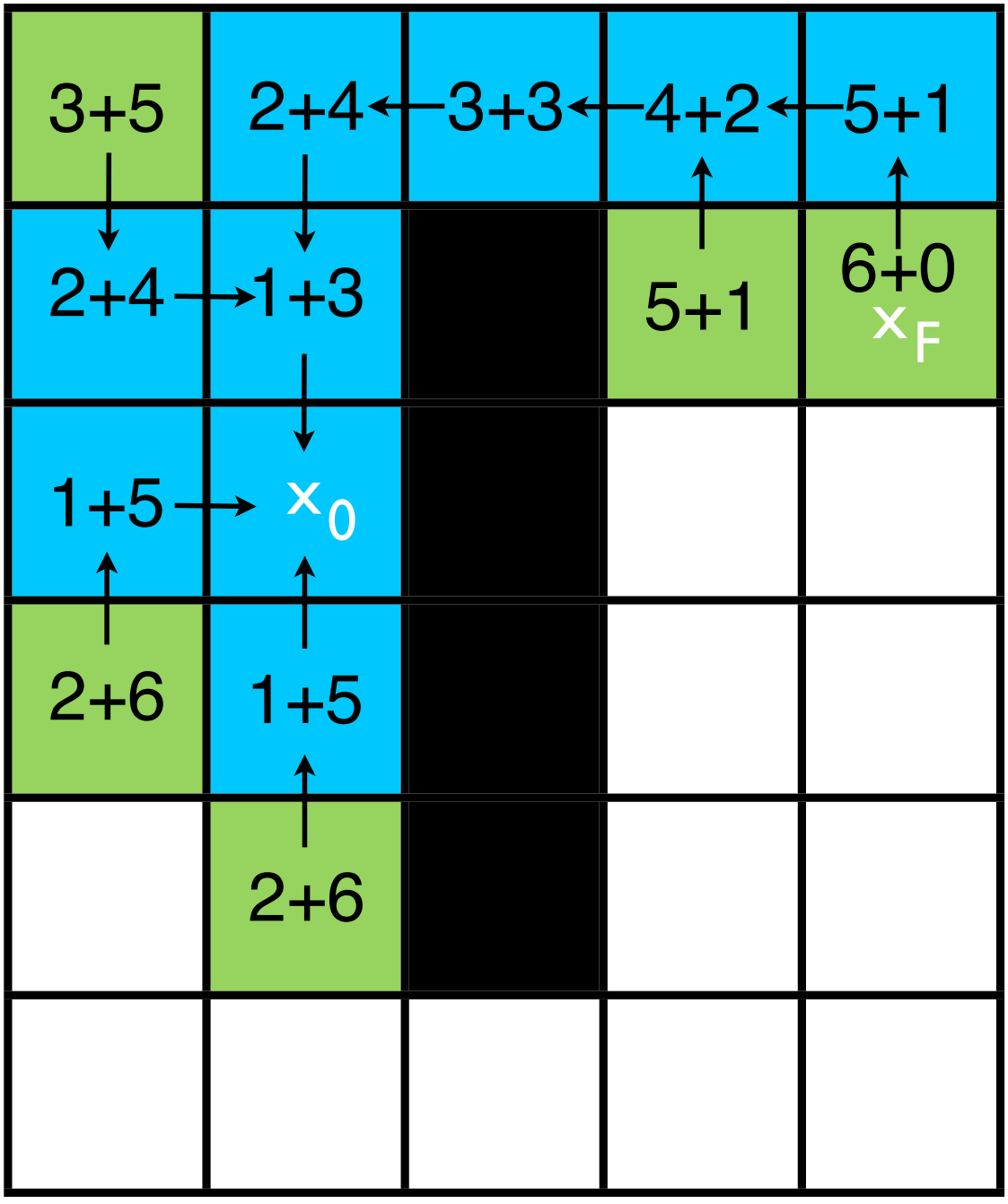
sommet déjà examiné ($\overline{S}$)

sommet à examiner (S)

$$\phi(x) = F(x) + L(x)$$

$F(x)$: distance de x_0 à x

$L(x)$: distance de x à x_F



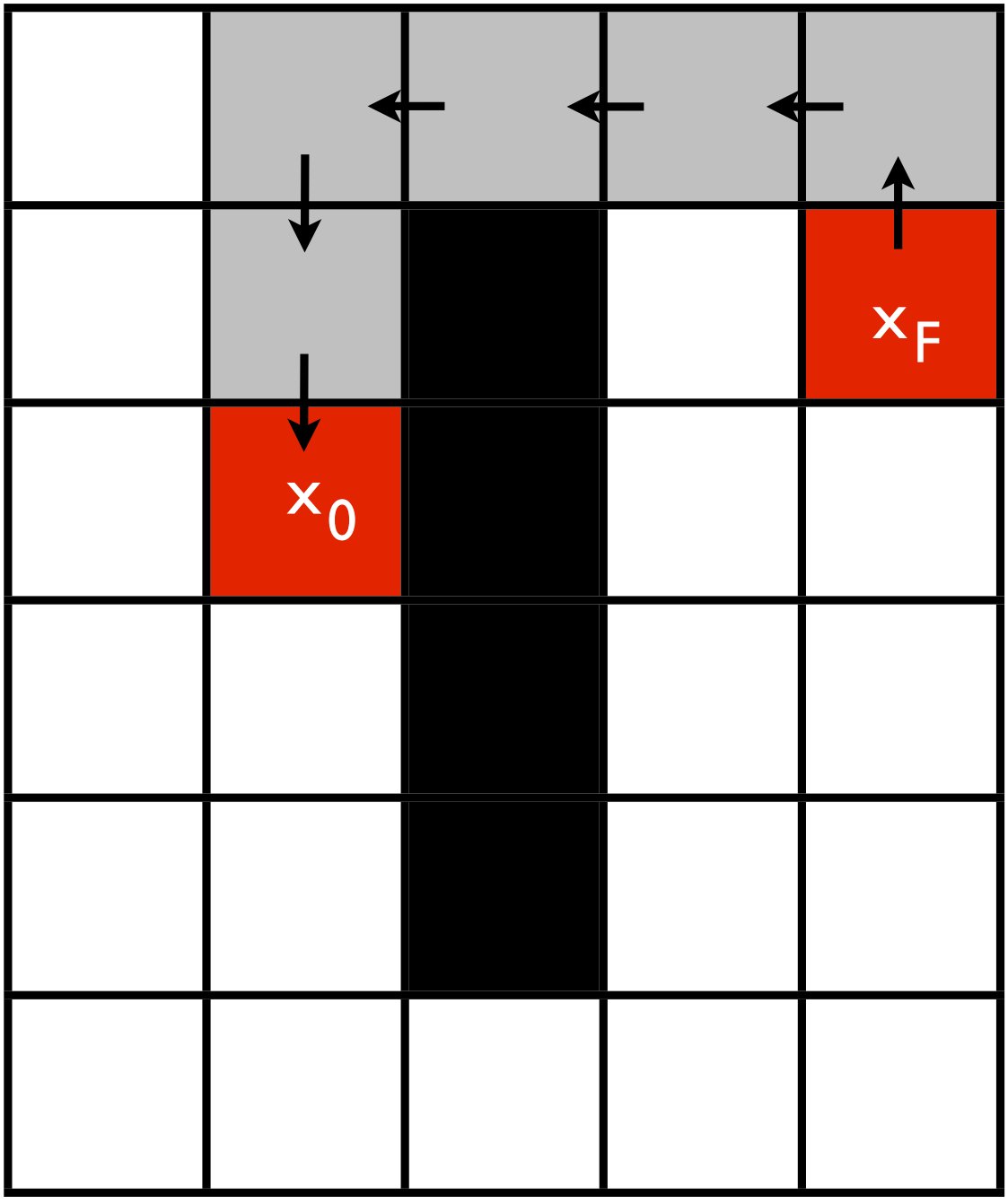
sommet déjà examiné ($\overline{S}$)

sommet à examiner (S)

$$\phi(x) = F(x) + L(x)$$

$F(x)$: distance de x_0 à x

$L(x)$: distance de x à x_F



sommet déjà examiné ($\overline{S}$)

sommet à examiner (S)

$$\phi(x) = F(x) + L(x)$$

$F(x)$: distance de x_0 à x

$L(x)$: distance de x à x_F

Propriétés de A^*

On note $l(x, y)$ la distance de x à son successeur y et $L^*(x)$ est la distance minimale de x à x_F.

- *Heuristique consistante* : L est une heuristique *consistante* si

$$L(x) \leq L(y) + l(x, y) \quad \forall x, \forall y \text{ successeur de } x.$$

- *Heuristique admissible* : L est une heuristique *admissible* si

$$L(x) \leq L^*(x) \quad \forall x.$$

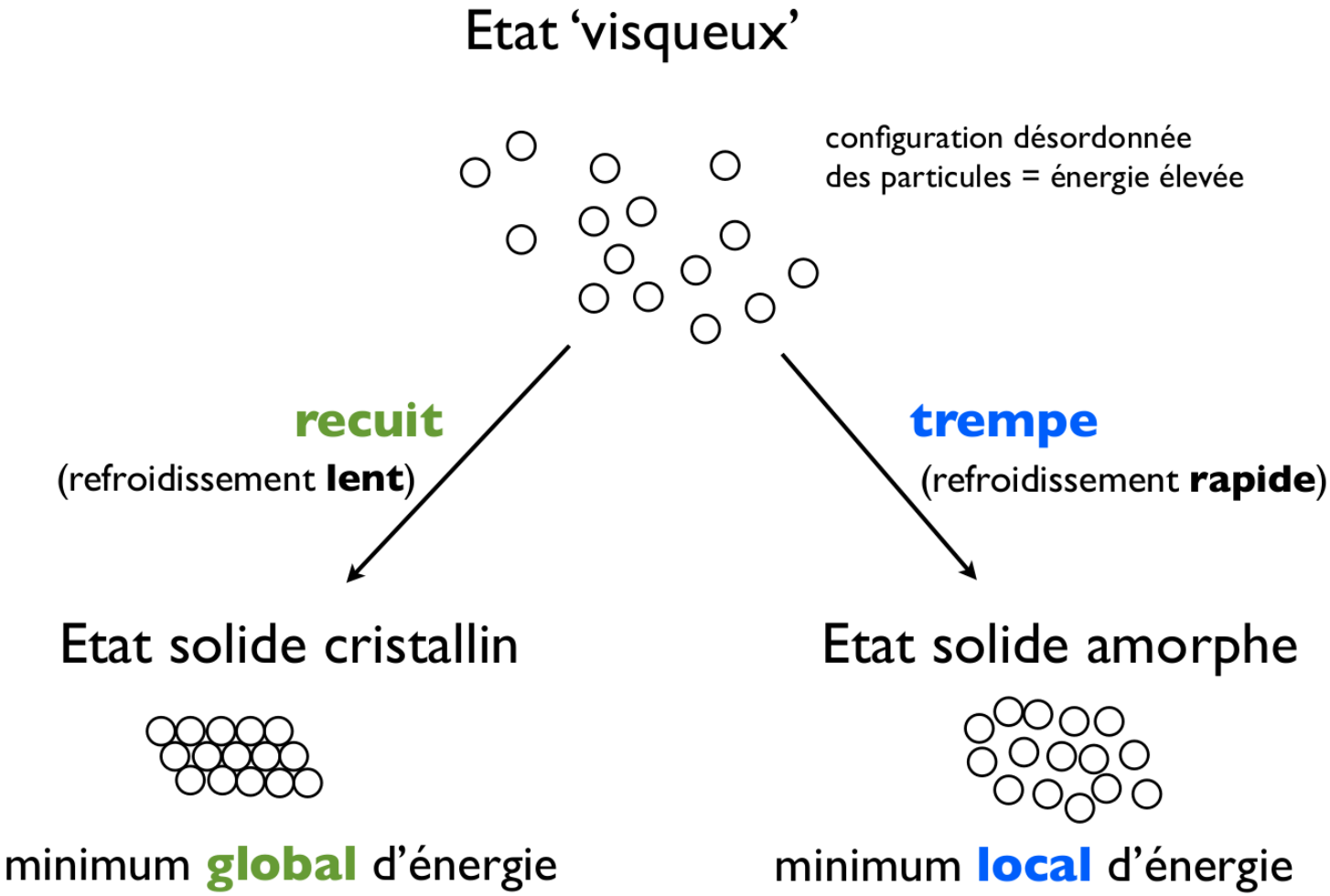
Propriétés

- Si L est admissible alors l'algorithme A^* trouvera toujours le chemin optimal.
- Si L est consistante alors
  - L est admissible
  - l'algorithme A^* a une complexité linéaire

2) Recuit simulé

Proposé par Kirkpatrick et co-auteurs (1983).

Méthode inspirée de la physique statistique (métallurgie). Analogie avec le processus d'alternance de refroisissement lent et de réchauffage (recuit) d'une pièce de métal afin d'obtenir un état d'énergie minimale.



Principe du recuit simulé : Une solution réalisable $\mathbf{x} \in X$ représente, par analogie, l'état d'un système avec une certaine distribution de probabilités p.

On effectue une petite variation de la solution qui entraine une variation $\Delta f(\mathbf{x})$ de l'énergie du système.

- Si $\Delta f(\mathbf{x}) < 0$ (l'énergie du système diminue) alors on accepte cette variation ($p = 1$).
- Sinon, elle est acceptée avec une probabilité

$$p = e^{-\frac{\Delta f(\mathbf{x})}{T}} \quad \text{(règle de Metropolis)}$$

Recuit simulé

x_0 solution initiale

$k = 0$; $x^* = x_0$ (meilleure solution rencontrée).

Choisir une suite décroissante de nombre $\theta_k > 0$ $(k = 1, 2, \cdots)$

Tant que (condition d'arrêt non vérifiée)

> Choisir aléatoirement y proche de x_k (voisins)
>
> Calculer $p = \min\left(1, e^{-\frac{f(y)-f(x_k)}{\theta_k}}\right)$
>
> Définition de x_{k+1} :
>
> $x_{k+1} = y$ avec probabilité p
>
> $x_{k+1} = x_k$ avec probabilité $1 - p$
>
> Si $f(x_{k+1}) < f(x^*)$ alors $x^* = x_{k+1}$
>
> k=k+1

Fin tant que

Remarques.

- La fonction f n'est pas nécessairement linéaire.
- Si $f(y) \leq f(x_k)$ alors $p = 1 \Rightarrow x_{k+1} = y$.
- En revanche, si $f(y) > f(x_k)$ alors y n'est pas nécessairement rejeté. p est d'autant plus faible que $f(y) - f(x_k)$ est grand.
- Choix courant de la température : $\theta_{k+1} = \lambda\theta_k$ avec $\lambda < 1$ ($\lambda = 0.99$).
- Si θ_k n'est pas trop faible alors la probabilité p peut être suffisamment grande pour s'affranchir de l'attraction d'un *minimum local*.

3) Algorithme génétique

Analogie avec la génétique et l'évolution naturelle par croisement, mutation et sélection (Golberg 1989).

Principe de sélection/croisement/mutation : faire évoluer une *population de solutions* en sélectionnant les meilleures solutions à chaque étape (nouvelle génération). Evolution par croisement des solutions sélectionnées avec des *mutations* possibles.

3) Algorithme génétique

Analogie avec la génétique et l'évolution naturelle par croisement, mutation et sélection (Golberg 1989).

Principe de sélection/croisement/mutation : faire évoluer une *population de solutions* en sélectionnant les meilleures solutions à chaque étape (nouvelle génération). Evolution par croisement des solutions sélectionnées avec des *mutations* possibles.

Croisement : Par exemple, 2 solutions x et y sous forme de codage binaire (x_i , $y_i \in \{0, 1\}$) :

$$x = (x_1, x_2, \cdots, x_n) \qquad y = (y_1, y_2, \cdots, y_n)$$

On choisit au hasard l'indice k et on engendre 2 nouvelles solutions

$$u = (\boxed{x_1, \cdots, x_{k-1}}, \boxed{y_k, \cdots, y_n}) \qquad v = (\boxed{y_1, \cdots, y_{k-1}}, \boxed{x_k, \cdots, x_n})$$

3) Algorithme génétique

Analogie avec la génétique et l'évolution naturelle par croisement, mutation et sélection (Golberg 1989).

Principe de sélection/croisement/mutation : faire évoluer une *population de solutions* en sélectionnant les meilleures solutions à chaque étape (nouvelle génération). Evolution par croisement des solutions sélectionnées avec des *mutations* possibles.

Croisement : Par exemple, 2 solutions x et y sous forme de codage binaire (x_i , $y_i \in \{0, 1\}$) :

$$x = (x_1, x_2, \cdots, x_n) \qquad y = (y_1, y_2, \cdots, y_n)$$

On choisit au hasard l'indice k et on engendre 2 nouvelles solutions

$$u = (\boxed{x_1, \cdots, x_{k-1}}, \boxed{y_k, \cdots, y_n}) \qquad v = (\boxed{y_1, \cdots, y_{k-1}}, \boxed{x_k, \cdots, x_n})$$

Mutation : Par exemple, changer un ou plusieurs bits au hasard dans u et v

Algorithme génétique (principe général)

- Population initiale composée de m solutions :
  $P_0 = \{x_1, x_2, \cdots, x_m\}$
- $k = 0$

Tant que (condition d'arrêt non vérifiée)

- Appliquer un opérateur de *sélection* pour obtenir des paires de solutions de la population P_k.
  Paires de solutions sélectionnées : $(y_1, z_1), \cdots, (y_p, z_p)$
- $P_{k+1} = \emptyset$.

Pour j de 1 à p

- Appliquer l'opérateur de *croisement* à (y_j, z_j) pour obtenir $\chi(y_j, z_j)$ l'ensemble des nouvelles solutions
- Pour chaque solution $x \in \chi(y_j, z_j)$, appliquer l'opérateur de *mutation* pour obtenir $\mu(x)$
- $P_{k+1} = P_{k+1} \cup \mu(x)$

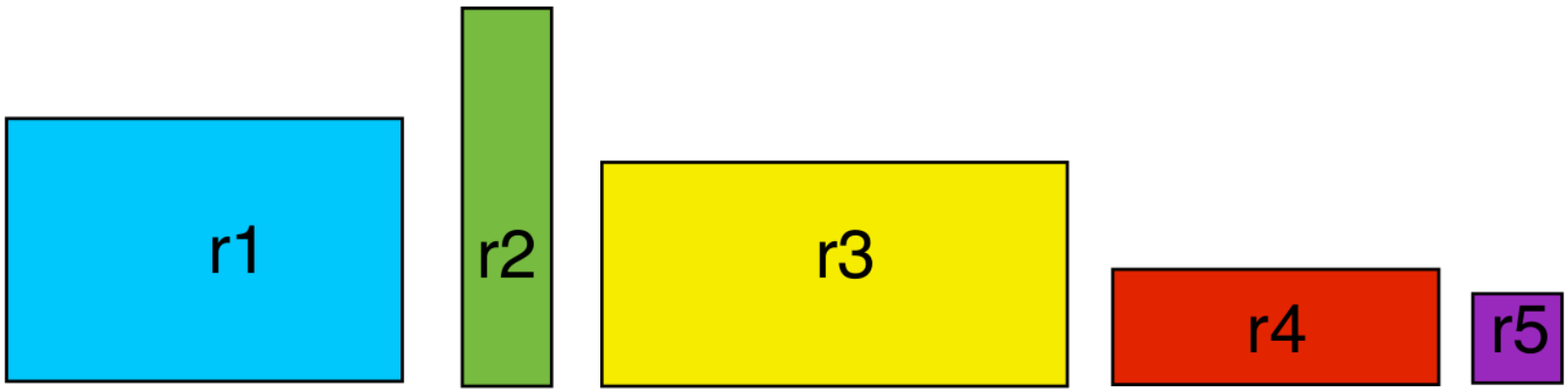
Fin pour

k=k+1

Fin tant que

Un exemple : Binpack 2D

Placement optimal de rectangles sur des plaques pour minimiser le nombre de plaques utilisées.

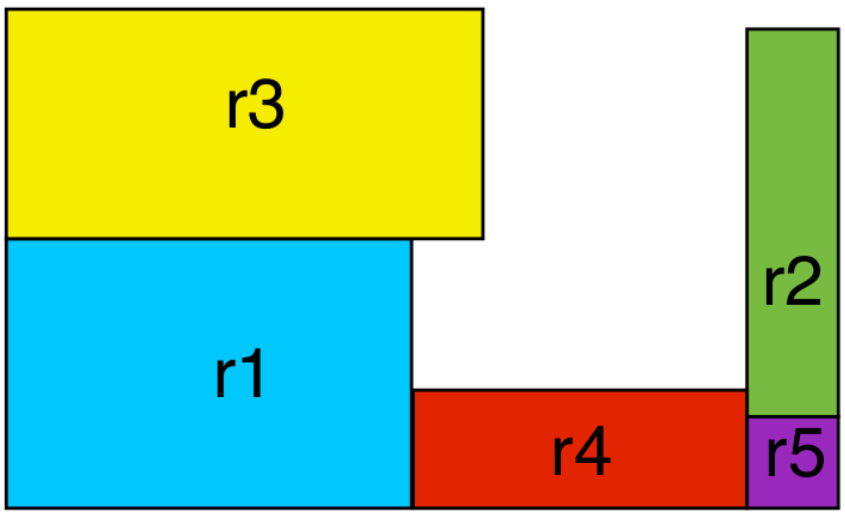


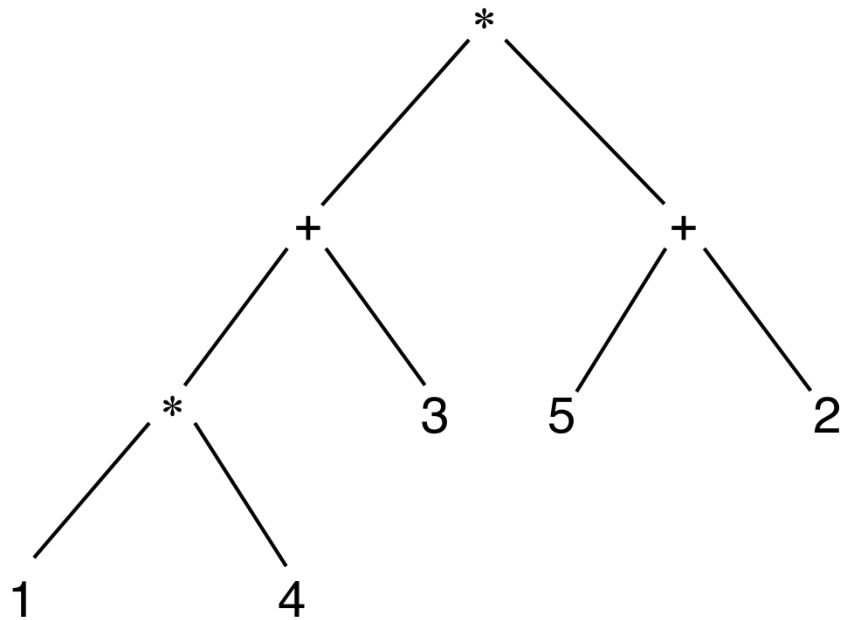
1 Codage

Opérateurs de placement

$*$: placement gauche/droite

$+$: placement haut/bas



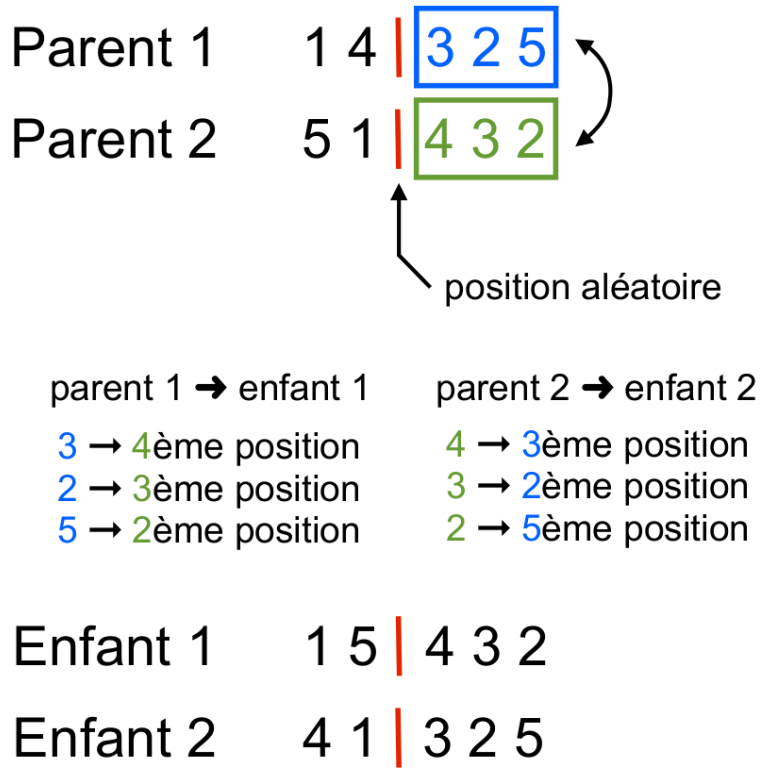


Notation post-fixée (polonaise inverse) : $14 * 3 + 52 + *$

❷ Croisement hybride

Parent 1 : $1432 + 5 * +*$
Parent 2 : $51 * 4 + 3 * 2+$

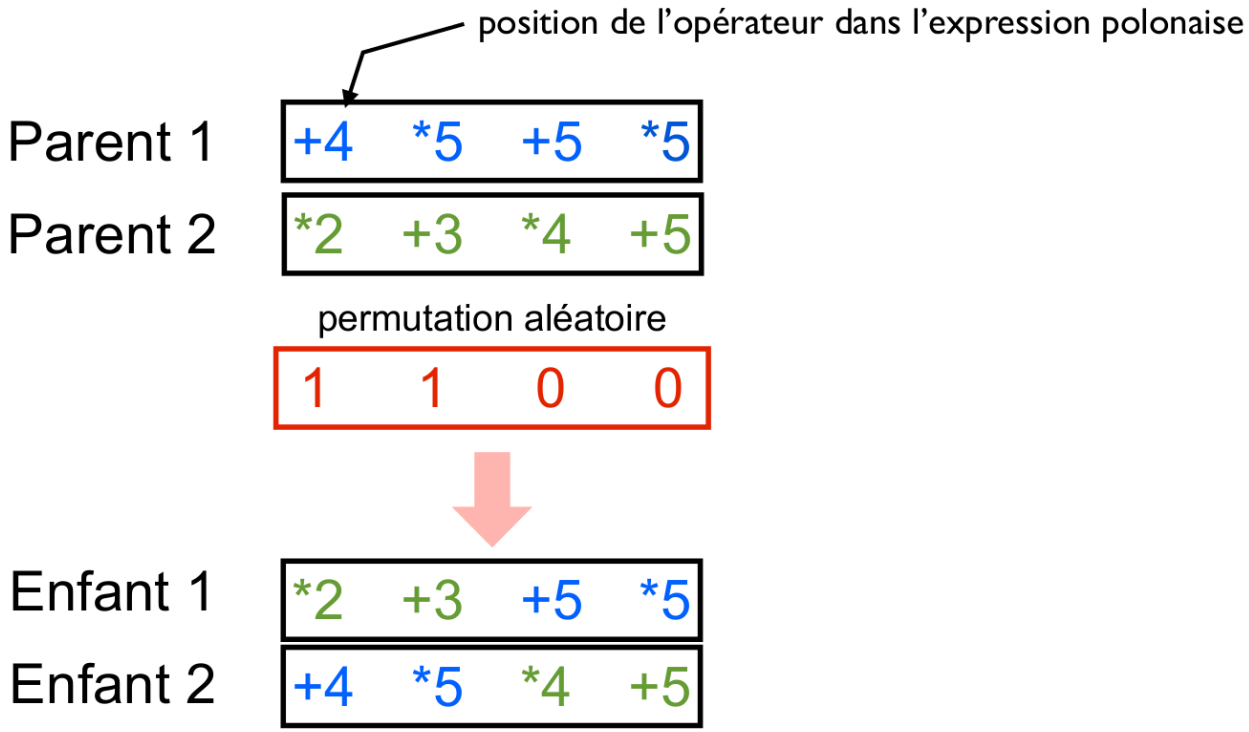
(a) Partie index



② Croisement hybride

Parent 1 : $1432+5*+*$
Parent 2 : $51*4+3*2+$

(b) Partie opérateur



Finalement, on obtient

Enfant 1 : $15*4+32+*$
Enfant 2 : $4132+*5*+$

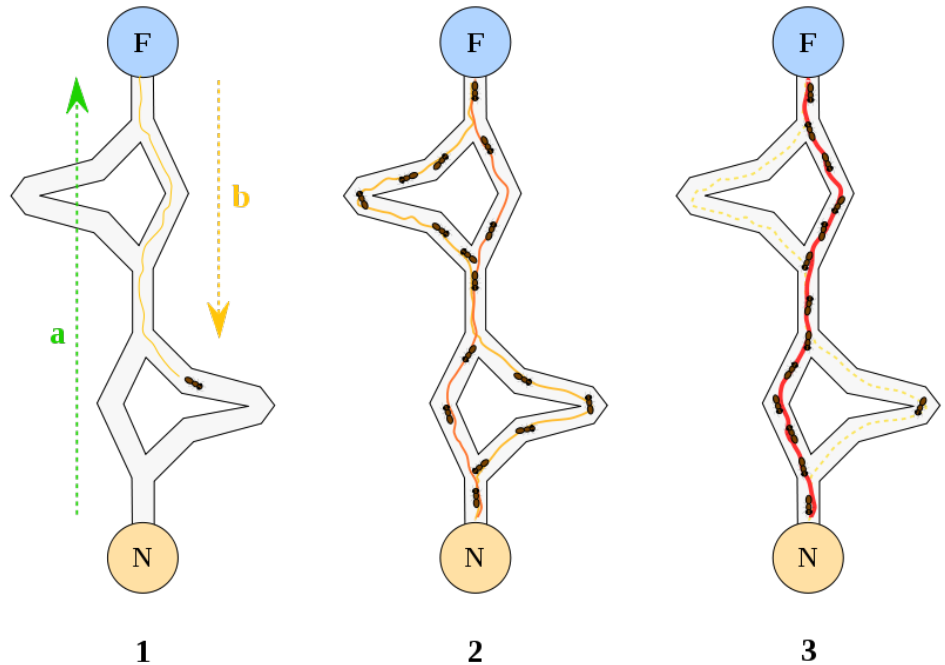
3. Mutations
   - rotation d'un rectangle
   - permutation de 2 rectangles
   - déplacement d'un opérateur

Remarques.

- La fonction objectif f peut être non-linéaire.
- Temps de calculs souvent importants.
- Difficultés de mise en oeuvre.
- Fournit généralement une solution approchée.
- Problèmes d'extréma locaux.

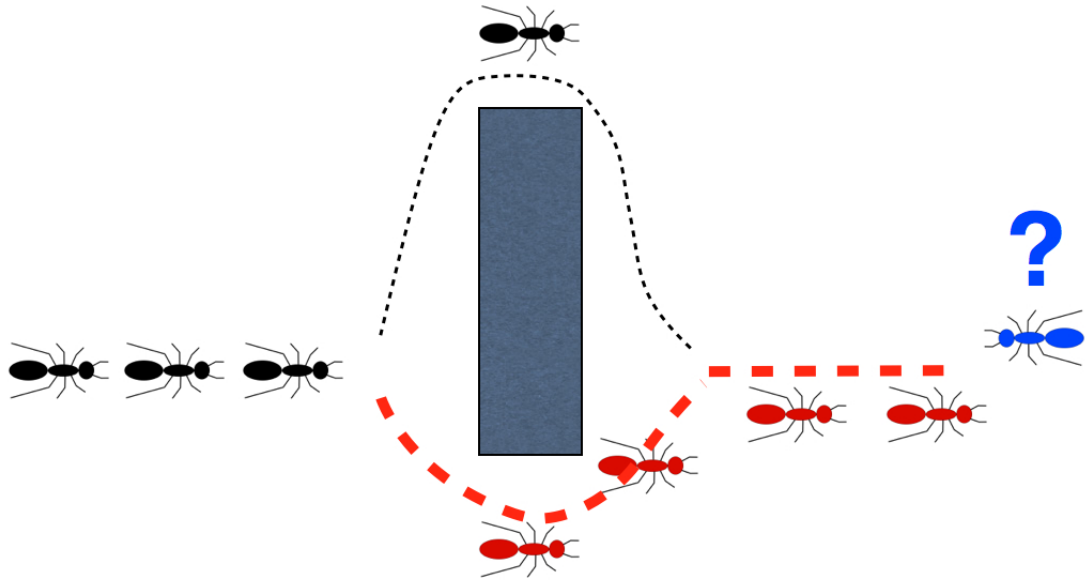
4) Algorithme de colonies de fourmis

Dû à M. Dorigo dans les années 90, s'inspire du comportement collectif des fourmis pour la recherche d'un plus court chemin de la fourmilière à un point de nourriture.



- Après avoir trouvé la nourriture, les fourmis rentrent à la fourmilière (par le même chemin) en déposant des *phéromones* au sol.
- Les phéromones étant attractives, les fourmis qui partent de la fourmilière seront guidées.

- Si 2 chemins sont possibles pour atteindre la nourriture, le chemin le + court sera parcouru par + de fourmis.



☞
- la piste courte sera renforcée et de + en + attractive.
- la piste longue finira par disparaître à cause de *l'évaporation* des phéromones

- A terme, le chemin le + court sera choisi par la majorité des fourmis.