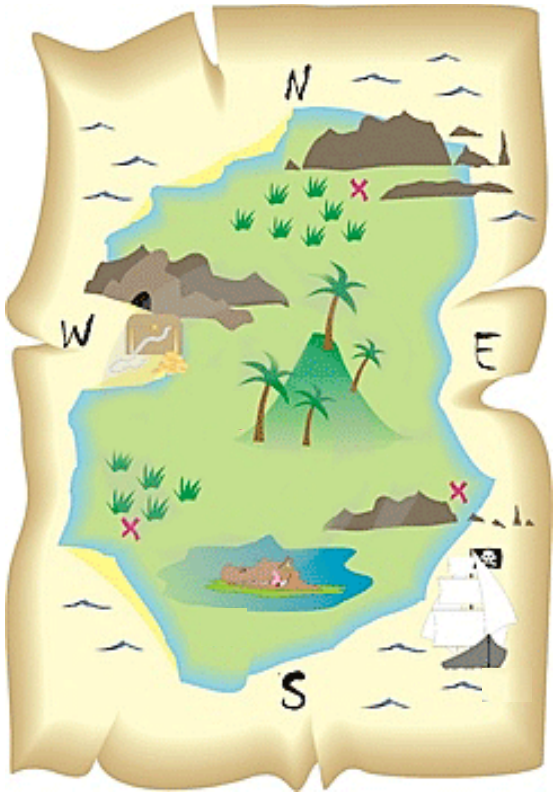
Algorithmes probabilistes

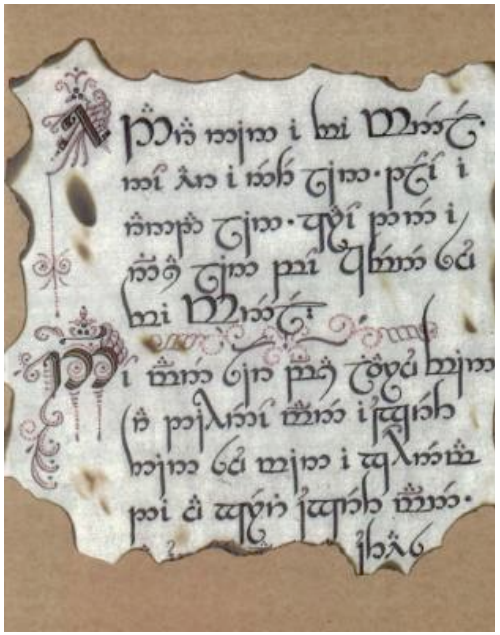
Références: **Fundamentals of Algortihms** de Gilles Brassard et Paul Bratley

Note de cours de Pierre McKenzie

Mise en contexte:



Indices:



Vous êtes à la recherche d'un trésor légendaire gardé par un dragon.

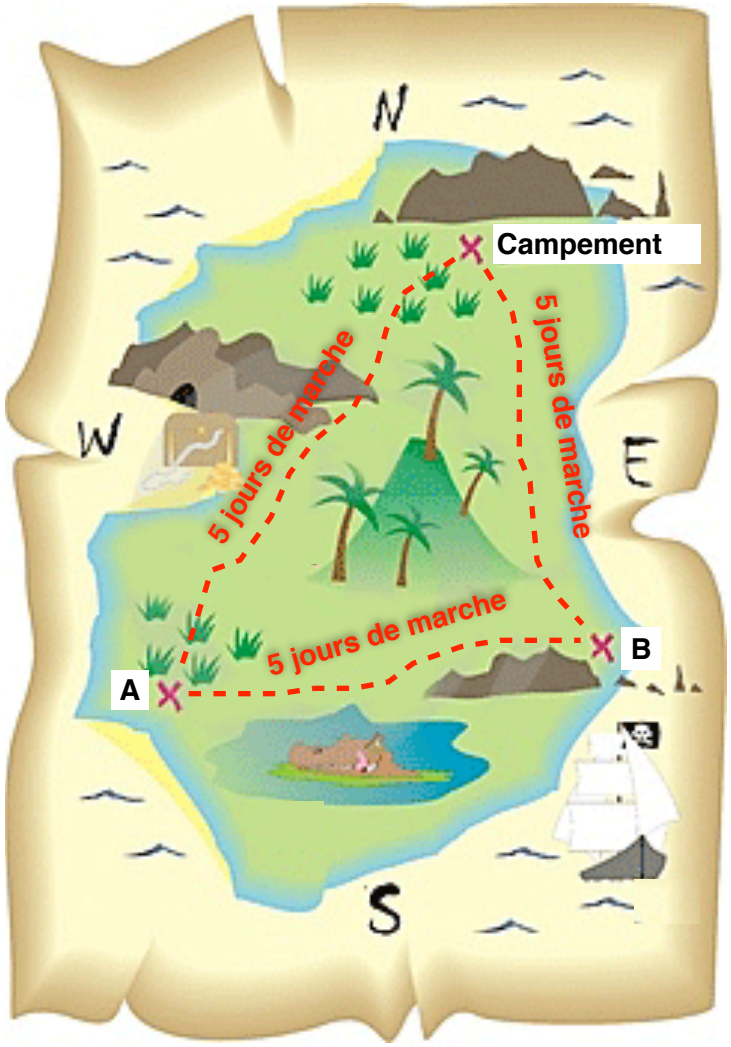
Vous disposez d'une carte au trésor avec indices compliqués en langage elfique.

Ce que vous savez:

Le trésor est soit en A, soit en B

Chaque nuit, le dragon apporte une portion y du trésor dans son antre

(On suppose que le montant du trésor x est tel que $x > 10y$)



Ce que vous voulez:

Rapporter la plus grande partie du trésor chez vous.

Vous pouvez:

1) Rester à votre campement (vous y avez l'électricité et votre ordinateur) pour compléter le décryptage de la carte. Ce décryptage devrait prendre 4 jours.

Profit espéré: $x - 9y$

2) Demander à un Elfe de déchiffrer les indices pour vous. Il peut vous donner la position du trésor en quelques minutes. Pour ses efforts, il vous demande un montant équivalent à ce que le dragon transporte en 3 nuits.

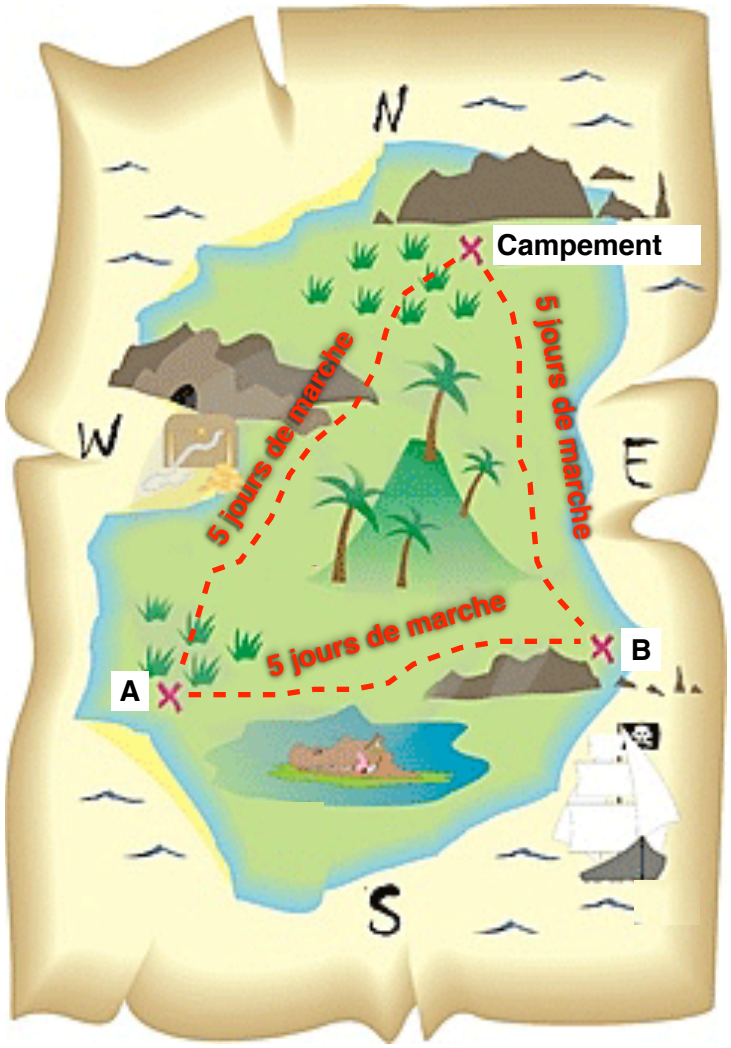
Profit espéré: $x - 8y$

3) Choisir au hasard d'aller à A ou B.

Si on fait le bon choix, le profit est de $x - 5y$

Sinon le profit est de $x - 10y$

Profit espéré: $x - 7.5y$



Traduction algorithmique:

Quand un algorithme est confronté à un choix, il est parfois préférable de choisir aléatoirement la prochaine étape, au lieu de travailler très fort pour trouver la meilleure alternative.

Cela arrive quand le temps requis pour trouver un choix optimal est très élevé comparé au temps gagné en moyenne par ce choix optimal

Caractéristique principale d'un algorithme probabiliste:

Se comporte différemment lorsqu'on le fait tourner deux fois sur les mêmes données:

- différents temps d'exécution
- le résultat peut varier considérablement à chaque exécution

Avantages de cette caractéristique p/r à un algo déterministe:

1) On peut permettre à un algorithme probabiliste de s'égarer (boucle infinie, division par zéro) sur une instance donnée en autant que la probabilité que cela arrive soit très faible.

2) On peut permettre à un algorithme probabiliste de retourner un résultat érroné en autant que cela ne se produise qu'avec une très faible probabilité.

3) Si on a plus d'une solution optimale à notre problème, à partir d'une instance donnée, on peut trouver plusieurs de ces solutions en faisant tourner l'algorithme plusieurs fois sur cette même instance.

Première surprise: le hasard peut être utile!

- Accélérer une recherche

- Réduire l'effet des mauvaises instances de départ

- Estimer π

- Vérifier la primalité d'un nombre

- . . .

Deuxième surprise: le hasard peut être précis

Exemple: Pile = succès et Face = échec

Prob[succès] = $\frac{1}{2}$

Prob[succès après 2 essais] = $\frac{3}{4}$

$\vdots$

Prob[succès après n essais] = $1 - (\frac{1}{2})^n$

$$n = 10 \quad \Longrightarrow \quad \text{succès} \sim 99.90\%$$

$$n = 13 \quad \Longrightarrow \quad \text{succès} \sim 99.99\%$$

Types d'algorithmes probabilistes:

1) Numérique: Retourne une solution approximative à un problème numérique

plus de temps = plus de précision

2) Monte Carlo: Retourne toujours une réponse mais peut se tromper.

plus de temps = plus grande probabilité que la réponse soit bonne

3) Las Vegas: Ne retourne jamais une réponse inexacte mais parfois ne trouve pas de réponse du tout

plus de temps = plus grande probabilité de succès sur chaque instance de départ

Temps espéré $\neq$ Temps moyen

TempsMoyen(n) = $$\frac{\sum_{|w|=n} \text{temps sur instance } w}{\#\{w:|w|=n\}}$$

Temps espéré: Défini sur chaque instance w

TempsEspéré(w) = C'est le temps moyen que prend l'algorithme probabiliste pour résoudre w un grand nombre de fois

TempsMoyenEspéré(n) = $$\frac{\sum_{|w|=n} \text{TempsEspéré}(w)}{\#\{w:|w|=n\}}$$

PireTempsEspéré(n) = $$\max_{|w|=n} \{ \text{TempsEspéré}(w) \}$$

Algorithmes numériques

Ce sont les premiers algorithmes à utiliser l'aléatoire

Retourne une solution approximative à un problème numérique

La solution retournée est toujours approximative mais sa précision augmente avec le temps disponible pour trouver une solution.

Habituellement, l'erreur est inversément proportionnelle à la racine carré de la quantité de travail effectuée.

Un des première utilisation des algorithmes probabilistes: Estimer π

L'aiguille de Buffon

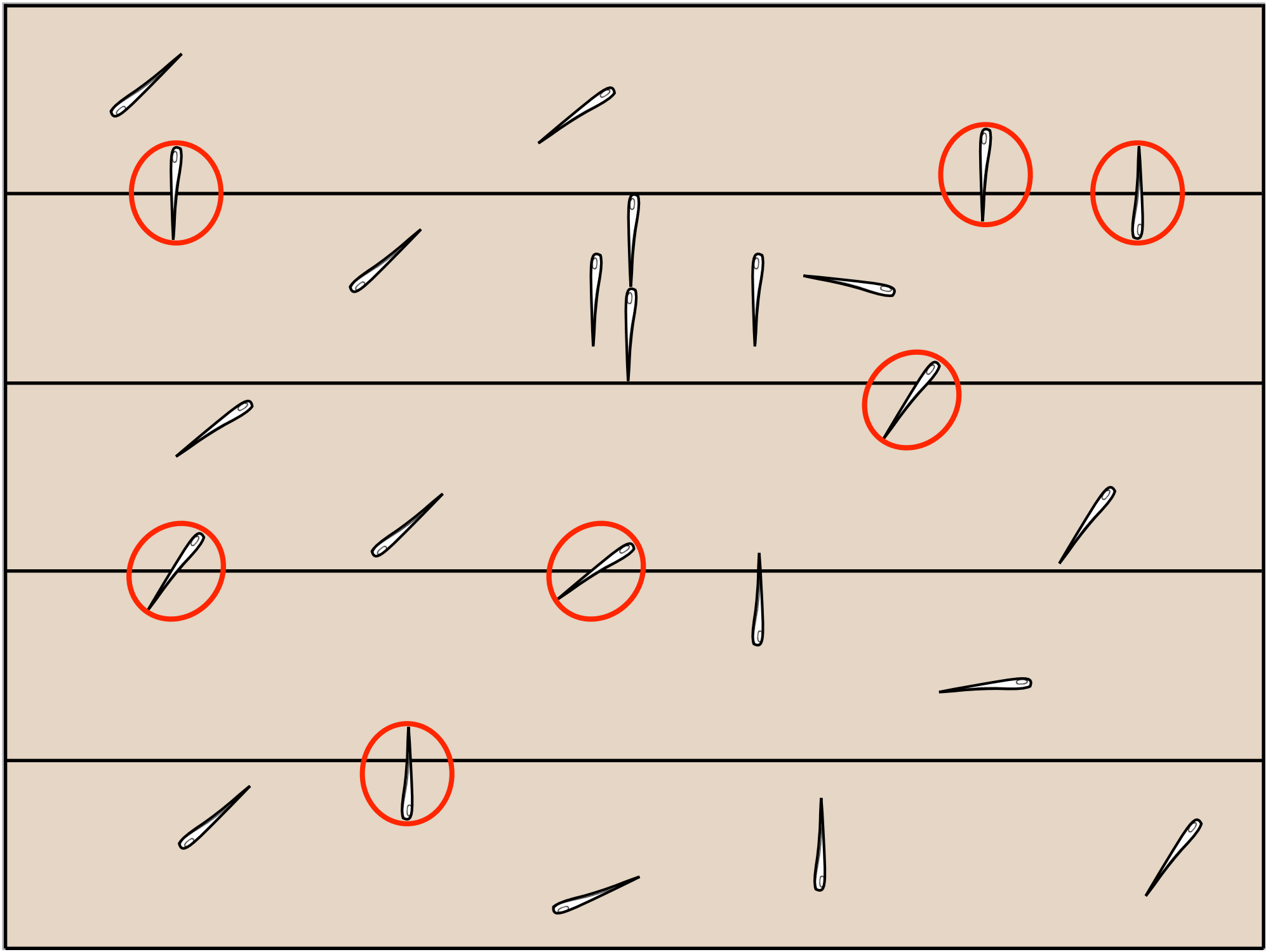
Au 18ième siècle, Georges Louis Leclerc, Comte de Buffon,
a démontré le théorème suivant:

Supposons qu'on lance une aiguille de façon aléatoire (dans une position aléatoire à un angle aléatoire avec une distribution uniforme) sur un plancher fait de lattes de bois de largeur constante, si

- l'aiguille est exactement la longueur d'une demi-largeur de latte

- la largeur des craques entre les lattes est 0

alors la probabilité que l'aiguille tombe au travers d'un craque est $\frac{1}{\pi}$



L'aiguille de Buffon - généralisarion

Si on n'assume plus que la longueur de l'aiguille = 1/2 largeur de latte

Théorème de Buffon généralisé:

Soient w , la largeur des lattes et λ la longueur d'une aiguille,alors

la probabilité qu'une aiguille lancée de façon aléatoire tombe au travers d'une craque est

$$p = \frac{2\lambda}{w\pi}$$

Algorithmes Monte Carlo

Peut se tromper (pas d'avertissement si c'est le cas)

Trouve une **solution correcte** avec une bonne probabilité et ce **quelle que soit** l'instance de départ

On dit qu'un algorithme Monte Carlo est p-correct s'il retourne une solution correcte avec probabilité p ou plus, $0 < p < 1$

Une propriété intéressante des algorithmes Monte Carlo est qu'il est souvent possible de réduire la probabilité d'erreur en augmentant le temps de calcul (i.e. faire tourner l'algorithme un grand nombre de fois sur chaque instance de départ)

Amplifier l'avantage stochastique

Coupe minimale d'un graphe

Soit G=(V,E) un graphe connexe, non-orienté, à arêtes multiples mais sans boucle.

Soit X, Y une partition des sommets de G i.e $X \cup Y = V$ et $X \cap Y = \emptyset$

Une coupe de G est l'ensemble C des arêtes de G qui ont une extrémité dans X et l'autre dans Y

Problème de la coupe minimale:

Étant donné un graphe G, trouver une coupe minimale i.e. une coupe qui contient le minimum d'arêtes.

Coupe minimale d'un graphe

Définition: Étant donné un graphe G et une arête e, la contraction de e est l'opération consistant à fusionner les 2 sommets extrémités de e et à retirer e. Le graphe obtenu est dénoté G/e.

(Dans notre cas, comme notre graphe de départ contient des arêtes multiples, nous considérons que l'opération de contraction retire aussi toutes les boucles formées lors de l'opération.)

Algorithme MinCut_Random(G)

Tant que G à plus de 2 sommets faire

- choisir une arête de G au hasard et de façon uniforme
- remplacer G par G/e

Fin Tant que

Retourner les arêtes de G

Attention: Ne retourne pas toujours une coupe minimale

Coupe minimale d'un graphe

Proposition: Pour tout graphe G à n sommets, la probabilité que MinCut_Random(G) retourne une coupe minimale est supérieure ou égale à $2/n^2$

Pas très bon mais si on fait tourner l'algorithme un nombre suffisant de fois, la probabilité de succès devient très bonne:

Proposition: Pour tout graphe G à n sommets et tout réel $0 < \epsilon < 1$, la probabilité que $\frac{n^2 ln(1/\epsilon)}{2}$ répétitions indépendantes de l'algorithme donnent la coupe minimale est $\geq 1 - \epsilon$

Algorithmes Las Vegas

Las Vegas de type 1: Utilise le hasard pour guider ses choix et arrive toujours à résoudre le problème

$$\text{mauvais choix} \Rightarrow \text{plus lent}$$

Exemple: tri

Las Vegas de type 2: Utilise le hasard pour tenter de résoudre un problème quitte à échouer

$$\text{mauvais choix} \Rightarrow \text{échec avoué}$$

Exemple: les 8 reines

Tri Rapide Randomisé

Algorithme TriRapideRandomisé (tri une liste L de n éléments)

si $n \leq 1$

retourner L

sinon

- choisir un élément de L, le pivot, au hasard et de façon uniforme

- pivoter les éléments de L autour du pivot choisi de sorte que L_1 contienne les éléments $\leq$ pivot et L_2 les autres éléments

- TriRapideRandomisé(L_1)

- TriRapideRandomisé(L_2)

Le temps pris par l'algorithme dépend des choix aléatoires fait par l'algorithme et donc varie à chaque appel de l'algorithme sur la même liste L

temps moyen espéré: $\Theta(n \log n)$

Conclusions Las Vegas de type 1

Particulièrement utile quand un algorithme déterministe existe pour le problème qui est:

- bon en moyenne
- mauvais au pire cas

Un algorithme Las Vegas pour ce problème pourra alors:

- éliminer les instances pire cas
- uniformiser les instances
- maintenir un bon temps espéré

Las Vegas de type 2

Rappel: un tel algorithme peut échoué mais détecte alors son échec.

Procédure LV (x, **var** y, **var** succès)

Si succès = vrai alors y est la solution du problème sur l'instance x

Si succès = faux alors "pas de chance"

$$p(x) = \text{ probabilité de succès}$$

$$(\forall \text{ instance } x)\,[p(x) > 0]$$

Mieux encore:

$$(\exists \delta > 0)(\forall \text{ instance } x)\,[p(x) > \delta]$$

Las Vegas de type 2 (suite)

On pourrait donc décider de répéter un algorithme de ce type jusqu'à un succès:

fonction *obstiné(x)*
  répéter
    LV(x, y, succès)
  jusqu'à *succès*
  retourner *y*

© Notes de cours Pierre McKenzie

On aura alors assurément une réponse mais quand ...

Las Vegas de type 2 (suite)

Soient

p: *probabilité de succès de LV*

s: *temps espéré de LV en cas de succès*

e: *temps espéré de LV en cas d'échec*

t: *temps espéré de obstiné*

Alors

$$t = ps + (1-p)(e+t)$$

d'où

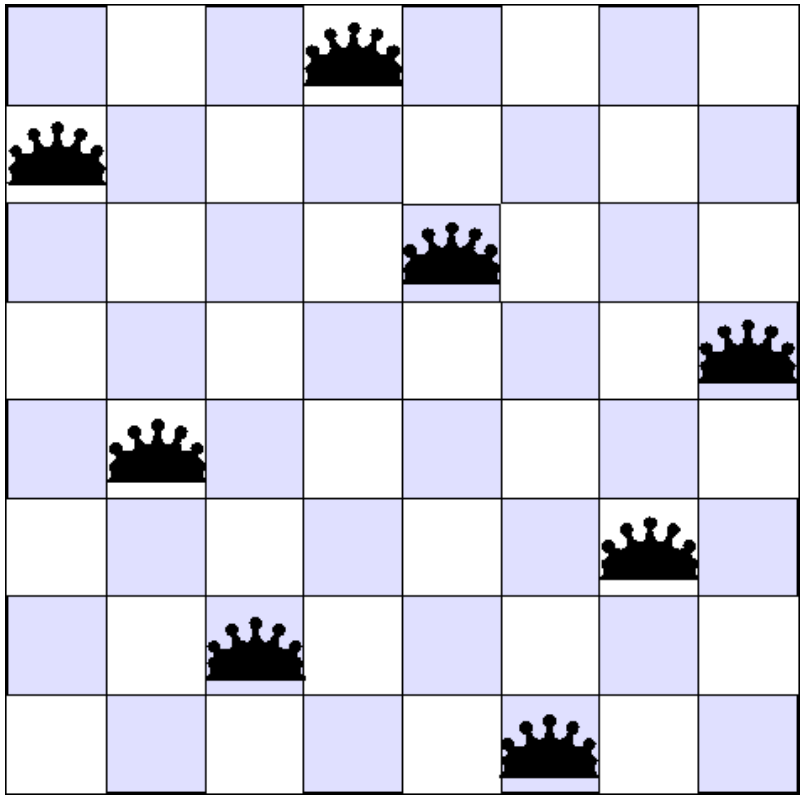
$$t = s + \frac{(1-p)}{p}e$$

Bref: $s \downarrow,\ e \downarrow,\ p \uparrow \quad \Rightarrow \quad t \downarrow$

© Notes de cours Pierre McKenzie

Rappel algorithme de placement des 8 reines

Problème: Placer 8 reines sur un échiquier de sorte qu'aucune reine n'en menace une autre. (Une reine menace toutes les pièces positionnées dans la même rangée, la même colonne ou la même diagonale).



Parallel Programming in C for the Transputer
© D. Thiébaut, 1995

Avantages de l'algorithme retour en arrière

- Beaucoup moins que 8! noeuds dans l'arbre:

  On peut calculer assez facilement avec un ordinateur qu'il y a 2057 noeuds

  En pratique, on ne doit visiter que 114 d'entre eux pour trouver une solution

Algorithme Las Vegas vorace:

Comme les prochaines positions des reines dans la construction des k-vecteurs semblent arbitraires, les choisir de façon aléatoire.

L'algorithme probabiliste, place donc la première reine en choisissant de façon uniforme l'une des 8 colonnes.

L'algorithme place ensuite les autres reines en choisissant aléatoirement une colonne non menacée par une reine déjà placée.

L'algorithme peut donc arriver dans un cul-de-sac et ne trouver aucune solution ou trouver une solution

Avantages de l'algorithme Las Vegas

- Conceptuellement plus simple

- Plus rapide en principe:

p = Prob[succès] = 0.1293 (#solutions/#total de positionnements, calculée par ordinateur)

s : temps espéré en cas de succès

= coût pour générer 9 vecteurs k-prometteurs, $0 \leq k \leq 8$

e : temps espéré en cas d'échec

= coût pour générer 6.971 vecteurs (calculé par ordinateur)

t : temps espéré de l'algo

$= s + \frac{(1-p)}{p} e$ = coût pour générer 55.93 vecteurs

Avantages de l'algorithme Las Vegas

En fait: Dans le cas des 8 reines, le coût de la génération des nombres aléatoires (pour choisir la prochaines reines) annule le gain en nombre de vecteurs générés

Peut-on faire mieux?

Oui, en ajustant s, e et p

Idée: générer les k premières reines aléatoirement et les $8-k$ dernières par retour en arrière

$k=2$: 3 fois plus rapide que retour en arrière

$k=3$: seulement 2 fois plus rapide même si moins de vecteurs générés

Avantages de l'algorithme Las Vegas

L'avantage de Las Vegas sur le retour en arrière croît énormément lorsque n augmente.

Exemple: $n = 39$

- pur retour en arrière:

10^{10} vecteurs explorés avant une solution

- pur Las Vegas:

un million de fois plus rapide

- hybride avec $k = 29$:

deux millions de fois plus rapide et
20 millions de moins de vecteurs générés