5 Back - Tracking

Algorithmique et Programmation C++ 2

Principe

Le but du problème est d'avancer d'états en états dans lesquels le problème est de plus en plus résolu. Un état sera symbolisé par un **nœud** de l'arbre.

- Un **succès** est un état final où le problème est entièrement résolu.
- Un **échec** est un état final pour lequel on sait que le problème n'est pas soluble.
- Une **transition** est une 'branche' de l'arbre. Elle est acceptable si elle n'amène pas à un échec.

Exemple : Jeu de Dame

On peut imaginer différentes transitions, qui influencent ensuite sur le filtrage (et éventuellement sur la rapidité d'exécution du code) :

Transition : pour toutes les cases
Acceptable : les cases noires, les cases devant, distance de 1 en diagonale

Transition : pour toutes les cases noires
Acceptable : cases devant, distance de 1 en diagonale

Transition : les deux case noires devant en diagonale
Acceptable : tout

Algorithme

Fonction principale

Voici l'algorithme générique d'une fonction procédant par essais/succès/échecs ou *back-tracking* :

```
essayer(etat)
{
      si est_une_solution(etat)
            traiter(etat);
      sinon
            pour toute les transitions ti
                  si ti est acceptable
                        modifier(etat,ti,etat')
                        essayer(etat')
                        demodifier(etat',ti,etat)
}
```

La fonction `traiter` permet de gérer plusieurs solutions, d'afficher la meilleure, ou de sortir du programme.

La *démodification* est seulement nécessaire si l'état est passé par référence, par adresse, ou en étant global. Elle n'est pas nécessaire lorsque l'état est passé par valeur, c'est-à-dire qu'il est copié à chaque appel récursif, et donc qu'une 'sauvegarde' est générée de manière implicite. Le choix de la méthode dépend de l'importance de l'objet manipulé, du coût de la recopie, et du coût de la modification/démodification.

Exemple

Etudions et résolvons le problème des *n* reines sur un échiquier *n*x*n*. L'objectif est de placer ces *n* reines de telle sorte qu'aucune reine ne soit mise en échec par les autres. La méthode de programmation par essais/succès/échecs convient particulièrement dans ce cas :

```
int abs(int v)
{return v<0?-v:v;}

int enEchec(int x, int y, int x2, int y2)
{
      // En ligne horizontalement ou verticalement :
      if(x1==x2) return 1;
      if(y1==y2) return 1;
      // Ou sur la diagonale :
      if(abs(x1-x2)==abs(y1-y2)) return 1;
      return 0;
}
```

```
void essayer(int * E, int & N, int n, int & cpt)
{
      int i;
      if(n==N) {afficher(E,N); cpt++;}
      else
            for(i=0; i<N; i++)
                  if(acceptable(n,i,E,N))
                  {
                        E[N]=i;
                        essayer(E,N,n+1,cpt);
                  }
}
```

```
int acceptable(int y, int x, int * E, int N)
{
      int i;
      for(i=O; i<y; i++)
            if(enEchec(x,y,E[i],i)) return 0;
      return 1;
}
```

```
int main()
{
      // Stocke dans E[i] la coordonnee 'colonne' d'une reine en ligne i
      int * E;
      int N;
      // Compteur :
      int cpt=0;

      cin >> N;

      E=new int[N];
      essayer(E,N,0,cpt);
      cout << cpt << "solutions !" << endl;
      return 0;
}
```

Et voici une des solutions pour *n*=8 :