Quentin QUADRAT
Login : quadra_q, UID : 17115, Promo : 2007

Travaux pratiques sur les Métaheuristiques pour l'optimisation difficile

15 novembre 2006

Résolution d'un problème de placement de composants électroniques en des sites prédéterminés

École Pour l'Informatique et les Techniques Avancées

1 Rappel du sujet

Le sujet du TP cherche à résoudre un problème de placement de composants électroniques en des sites prédéterminés d'un circuit.

Un circuit électronique est constitué :

- de composants numérotés de 1 à n, reliés entre eux par des arcs et placés aléatoirement sur le circuit ;
- de places disponibles prédéterminées et numérotées de 1 à n.

Il faut placer les composants dans les espaces disponibles tout en minimisant la longueur de leurs laisons. Le seul mouvement autorisé est la permutation de deux composants. Voici l'état initial du circuit électronique :

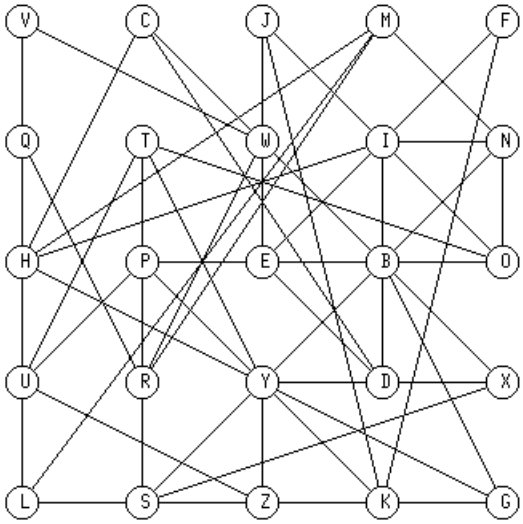

Fig. 1: Etat initial {`init`}

Voici l'état final du circuit électronique que l'on voudrait obtenir :

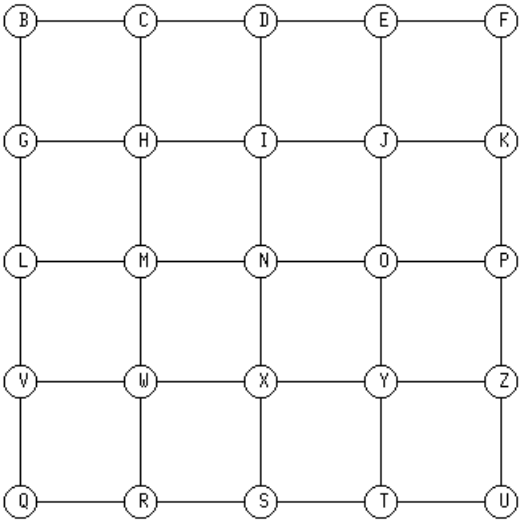

Fig. 2: Etat final {`final`}

Ces images ont été obtenues grâce à un programme codé en OCaml dont nous expliquerons

son code dans la section (3). Nous allons dábord introduire le principe du recuit simulé.

2 L'algorithme du recuit simulé

L'algorithme du recuit simulé inventé dans les années 80 a été inspirée d'un processus utilisé en métallurgie. Ce processus alterne des cycles de refroidissement lent et de réchauffage (recuit) qui tendent à minimiser l'énergie du matériau. L'algorithme du recuit simulé se déroule en plusieurs étapes comme le montre le schéma suivant (3).

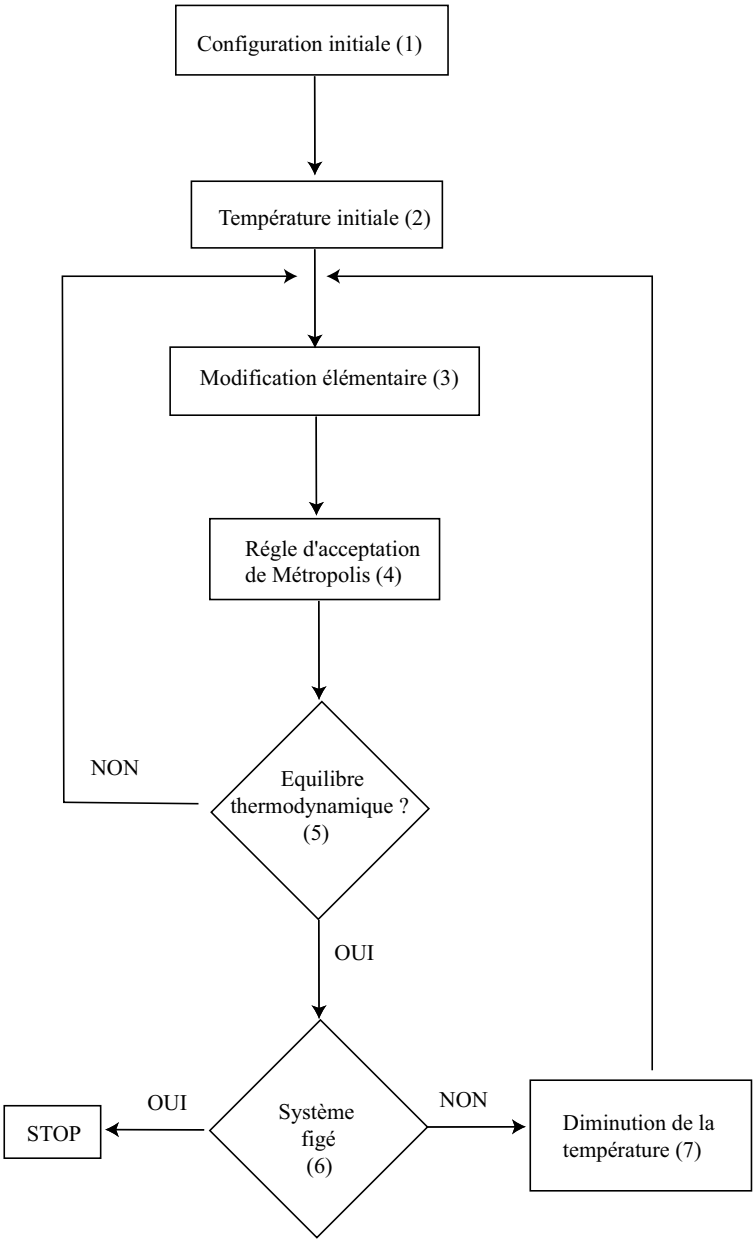

FIG. 3: Algorithme du recuit simulé. {recuit}

2.1 Configuration initiale

L'énergie E du système, dans le cas du problème de placement de composant, est la longueur des arcs qui les relient les uns aux autres.

La temprature T initiale sera prise élevée. Plus la température sera chaude plus une perturbation d'énergie sera accepetée. Ce choix est alors totalement arbitraire et va dépendre de la loi de décroissance utilisée.

2.2 Modification élementaire

On prend deux composants au hasard, on échange leur position. On calcule la variation d'énergie (différence de l'ancienne énergie avec la nouvelle). On obtient ΔE.

2.3 Régle d'acceptation de Métropolis

La modification élémentaire est retenue avec la probabilité suivante :

$$P = \begin{cases} 1 & \text{si } \Delta E < 0 \\ \exp^{\frac{-\Delta E}{T}} & \text{sinon} \end{cases}$$

2.4 Equilibre thermodynamique

L'équilibre thermodynamique est atteint lors du palier de température dès qu'une des deux conditions est remplie :

- $12n$ perturbations acceptées ;
- $100n$ perturbations tentées.

où n correspond au nombre de degré de libertés du problème. Dans notre cas nous prendrons le nombre total de composants :

2.5 Système figé

Le système est figé si aucun changement n'est accepté lors du palier de température ou que la température atteint est quasi nulle :

2.6 Diminution de la température

Il existe plusieurs méthodes pour diminuer la température :

- `décroissance géométrique` $T_{k+1} = 0.9T_k$
- `décroissance adaptative` $T_{k+1} = min(0.9, \frac{E_k}{M_k})T_k$ où E_k est l'énergie des configurations acceptées au palier k et M_k est l'énergie moyenne des configurations acceptées au cours du palier k.

3 Implémentation en OCaml

{code}

Nous allons définir manuellement le circuit sous la forme d'un graphe. Il s'agit d'une liste de composants (lecture verticale) qui connaissent leurs voisins (lecture horizontale). Le graphe représente une grille de 5×5 composants.

```
let taille = 5 ;;
let circuit =
  [
```

```
    (1, ref 1, [2;6]);
    (2, ref 2, [1;3;7]);
    (3, ref 3, [2;4;8]);
    (4, ref 4, [3;9;5]);
    (5, ref 5, [4;10]);
    (6, ref 6, [1;7;11]);
    (7, ref 7, [2;6;8;12]);
    (8, ref 8, [3;7;9;13]);
    (9, ref 9, [4;8;10;14]);
    (10, ref 10, [5;9;15]);
    (11, ref 11, [6;12;16]);
    (12, ref 12, [7;11;13;17]);
    (13, ref 13, [8;12;14;18]);
    (14, ref 14, [9;13;15;19]);
    (15, ref 15, [10;14;20]);
    (16, ref 16, [11;17;21]);
    (17, ref 17, [12;16;18;22]);
    (18, ref 18, [13;17;19;23]);
    (19, ref 19, [14;18;20;24]);
    (20, ref 20, [15;19;25]);
    (21, ref 21, [16;22]);
    (22, ref 22, [17;21;23]);
    (23, ref 23, [18;22;24]);
    (24, ref 24, [19;23;25]);
    (25, ref 25, [20;24])
  ] ;;
```

Le premier champ indique l'identifiant du composant qui servira lors de l'affichage, le deuxième champ indique le numéro de position où il se trouve (le mot clef `ref` indique que ce numéro est modifiable) et enfin, le troisième champ est une liste statique des composants voisins avec lesquels il est en contact.

Nous allons définir des accesseurs qui nous serons utiles pour l'écriture des autres fonctions. La première fonction permet d'accéder au `n`-ième composant du circuit :

```
let get_composant circuit n = List.nth circuit (n - 1) ;;
```

Nous voulons pouvoir modifier le nouveau numéro de position du composant `n` grâce à la fonction :

```
let set_position circuit n posi =
  let (_,p,_) = get_composant circuit n in p := posi ;;
```

et connaître facilement le numéro de position du composant `n` :

```
let get_position circuit n =
  let (_,p,_) = get_composant circuit n in !p ;;
```

Parce que connaitre le numéro de position n'est pas une information utile pour calculer l'énergie du circuit, nous devons créer une fonction retournant la position cartésienne en fonctionnant d'un numéro de position `n`.

```
let posi2coord n = (1 + (n - 1) mod taille, 1 + (n - 1) / taille) ;;
```

Pour calculer la distance entre deux composants (`c1` et `c2`), on utilise distance Manhattan qui est la somme des distances sur chaques axes, entre deux composants. Elle est également connue sous le nom de longueur en L :

```
let long_manhattan circuit c1 c2 =
  let (x1,y1) = posi2coord (get_position circuit c1) in
  let (x2,y2) = posi2coord (get_position circuit c2) in
    abs (x1 - x2) + abs (y1 - y2) ;;
```

Il est alors très facile de calculer l'energie locale d'un composant `n` : c'est la somme des longueurs Manhattan avec ses voisins.

```
let energie_locale circuit n =
  let (_,_,voisins) = get_composant circuit n in
  let liste = List.map (long_manhattan circuit n) voisins in
    List.fold_left (+) 0 liste ;;
```

L'énergie totale du circuit est la somme des énergies locales de tous les composants divisée par 2 :

```
let energie_totale circuit =
  let l = List.map (fun (x,_,_) -> energie_locale circuit x) circuit in
  let res = List.fold_left (+) 0 l in res ;;
```

OCaml permet de dessiner très simplement. Il suffit de charger le module graphique `graphics`, d'ouvrir une fenêtre pour dessiner, d'écrire une fonction qui dessine un arc entre deux composants et d'écrire une fonction qui dessine un composant. Dessiner le circuit entier se fait très facilement, grâce à un itérateur sur la liste des composants qui affichera les arcs et les noeuds :

```
#load "graphics.cma" ;;
Graphics.open_graph "" ;;

let dessine_arc circuit c1 c2 =
  let (x1,y1) = posi2coord (get_position circuit c1) in
  let (x2,y2) = posi2coord (get_position circuit c2) in
    Graphics.moveto (x1 * 75 + Graphics.size_x () / 10) (y1 * 75 + Graphics.size_y () / 10)
    Graphics.lineto (x2 * 75 + Graphics.size_x () / 10) (y2 * 75 + Graphics.size_y () / 10)

let dessine_noeud circuit c =
  let (x,y) = posi2coord (get_position circuit c) in
    Graphics.set_color Graphics.background;
    Graphics.fill_circle (x * 75 + Graphics.size_x () / 10) (y * 75 + Graphics.size_y () / 1
    Graphics.set_color Graphics.foreground;
    Graphics.draw_circle (x * 75 + Graphics.size_x () / 10) (y * 75 + Graphics.size_y () / 1
    Graphics.moveto (x * 75 + Graphics.size_x () / 10) (y * 75 + Graphics.size_y () / 10 - 5
    Graphics.draw_char (char_of_int (int_of_char 'A' + c - 1)) ;;

let dessine_circuit circuit =
  Graphics.clear_graph ();
  List.iter (fun (id,_,v) -> List.iter (dessine_arc circuit id) v) circuit;
  List.iter (fun (id,_,v) -> dessine_noeud circuit id) circuit ;;
```

Les fonctions de dessin et d'informations générales sur le circuit étant terminées, il ne reste plus qu'à programmer les fonctions les principales du recuit simulé.

L'équilibre thermodynamique est atteint lors du palier de température dés qu'une des deux conditions suivantes est remplie :

- $12n$ perturbations acceptées ;
- $100n$ perturbations tentées.

où n correspond au nombre de degré de libertés du problème. Dans notre cas nous prendrons le nombre total de composants :

```
let equilibre_thermo nb_permut_acceptee nb_permu_tentee =
  let max_pa = 12 * taille * taille in
  let max_pt = 100 * taille * taille in
    (nb_permut_acceptee >= max_pa) || (nb_permu_tentee >= max_pt) ;;
```

Le système est figé si aucun changement n'est accepté lors du palier de température ou que la température atteint est quasi nulle :

```
let syst_fige iteration energie =
  let iteration_max = 10000 in
  let energie_min = 0.001 in
    (iteration >= iteration_max) || (energie <= energie_min) ;;
```

Nous devons pouvoir permutter deux composants au hazard. La fonction commute joue se rôle et retourne également les numéros des composants permutés ainsi que leur position respective :

```
Random.self_init () ;;

let modif_elementaire circuit =
  let c1 = Random.int (taille * taille) + 1 in
  let p1 = get_position circuit c1 in
  let c2 = Random.int (taille * taille) + 1 in
  let p2 = get_position circuit c2
  in
    set_position circuit c1 p2;
    set_position circuit c2 p1;
    (c1,p1,c2,p2) ;;
```

La fonction d'acceptation est donc très simple à écrire. Elle retourne un triplet : le nombre de permutation acceptée, le nombre de permutation tentée et la nouvelle énergie. La seule entorse que l'on s'est sautorisée à faire est que cette fonction appelle `modif_elementaire` alors que ce n'est pas à elle de le faire. Ceci permet de simplifier l'écriture du programme :

```
let permut_et_accept circuit temperature energie nb_permut_acceptee nb_permu_tentee =
  let (c1,p1,c2,p2) = modif_elementaire circuit in
  let en = float_of_int (energie_totale circuit) in
  let delta_energie = en -. energie in
    match delta_energie <= 0.0 with
      | true  -> (nb_permut_acceptee + 1, nb_permu_tentee, en);
      | false ->
          let p = exp (-1.0 *. delta_energie /. temperature) in
          let r = Random.float 1.0 in
```

```
            match r <= p with
              | true  -> (nb_permut_acceptee + 1, nb_permu_tentee, en);
              | false -> (* restauration du circuit *)
                  set_position circuit c1 p1;
                  set_position circuit c2 p2;
                  (nb_permut_acceptee, nb_permu_tentee + 1, energie) ;;
```

Notre dernière fonction consite à définir la fonction principale du recuit simulé. Cette fonction est récursive. Notre température initiale est très chaude (666666) et décroie de façon géométrique :

```
let placement_de_composants circuit =
  let rec recuit_simule circuit temperature energie iter pa pt =
    let (ppa,ppt,nrj)  = permut_et_accept circuit temperature energie pa pt in
      match equilibre_thermo ppa ppt with
        | false -> recuit_simule circuit temperature nrj (iter + 1) ppa ppt
        | true  ->
             (match syst_fige iter nrj with
                | true  -> circuit (* Fin algo *)
                | false -> recuit_simule circuit (temperature *. 0.9) nrj
                                                 (iter + 1) ppa ppt)
  in
    recuit_simule circuit 666666.0 (float_of_int (energie_totale circuit)) 0 0 0 ;;
```

Pour exécuter cette fonction, l'interpreteur interactif OCaml peut être lance soit dans un console, soit avec Emacs. Après avoir mélangé préablement tous les composants un certain nombre de fois grâce à la fonction `commute circuit ; ;`, l'algorithme se lance simplement :

```
dessine_circuit (placement_de_composants circuit);;
```

Comme la fonction `placement_de_composants` modifie le circuit et que les 10000 itérations ne suffisent pas pour trouver la configuration optimale recherchés, on peut appeler plusieurs fois de suite cette fonction.

4 Résultat graphique

Voici un exemple de déroulement de notre programme OCaml de placement de composants. Toute la difficulte du recuit simulé consite à trouver la bonne diminution de l'énergie et de posséder un très bon générateur de nombre aléatoire. Sans être optimal, notre algorithme marche bien ; notre exemple s'est exécuté en 50000 itérations, d'autres essais ont permis d'obtenir en moins de 20000 itérations.

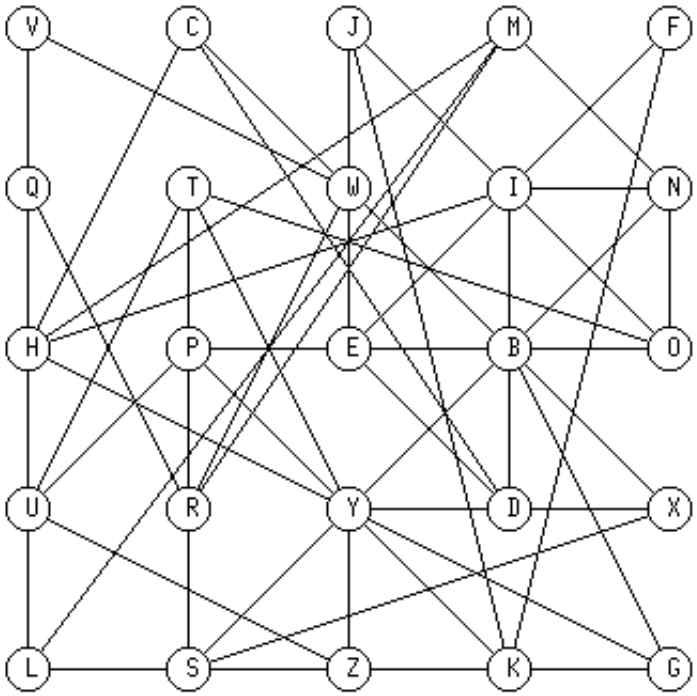

Fig. 4: Etat initial mélangé. {i}

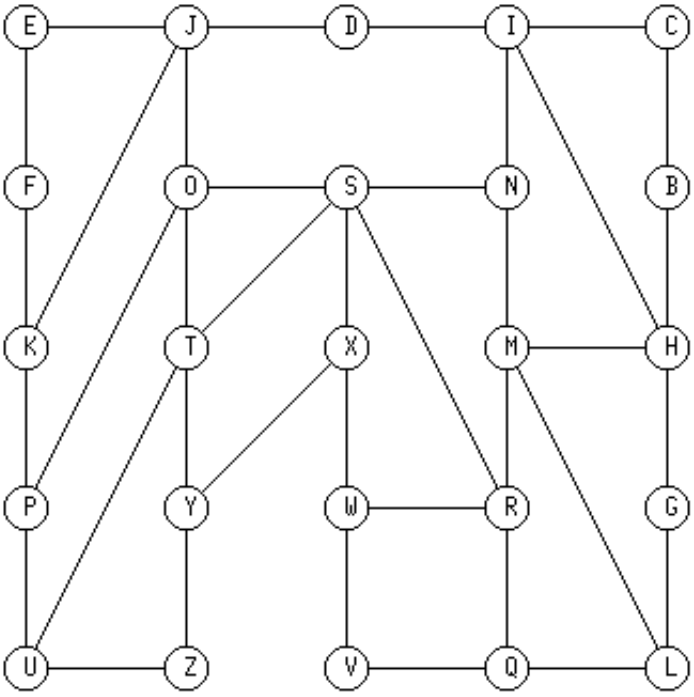

Fig. 5: Etat 1 (aprés 100000 itérations). {e1}

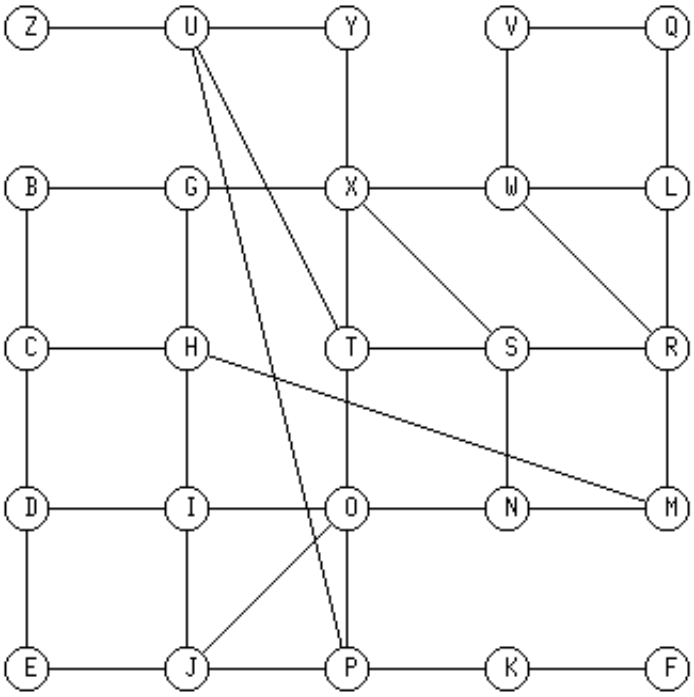

FIG. 6: Etat 2 (aprés 200000 itérations). {e2}

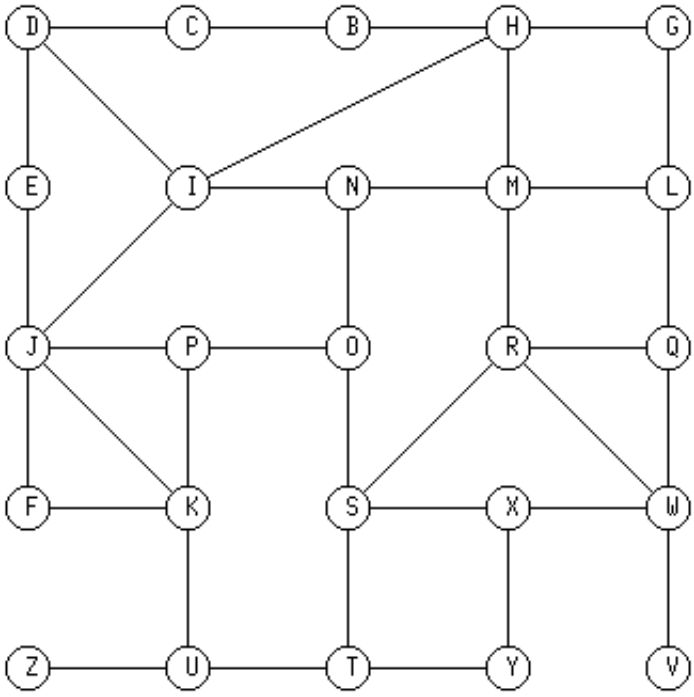

FIG. 7: Etat 3 (aprés 300000 itérations). {e3}

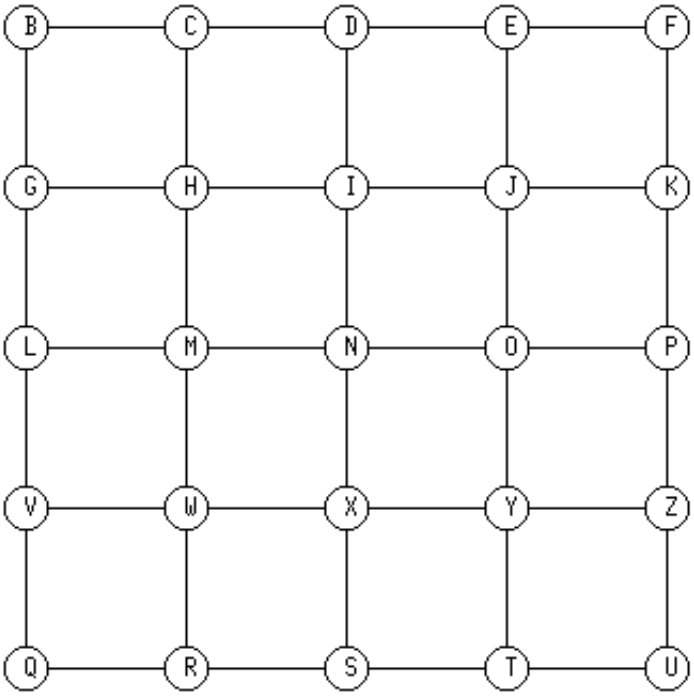

Fig. 8: Etat final (aprés 500000 itérations). {f}