Le jeu de go

Concours Centrale MPI Informatique 2024

Un Corrigé

I Introduction au jeu de go

Q 1. Il y a 2 libertés pour le groupe Noir : (1,0) et (3,0). Le score de Noir est 10 et celui de Blanc est 15, c'est donc Blanc qui gagne.

Q 2. Toutes les intersections sont contrôlées par l'un des deux joueurs en fin de partie d'après les règles. Comme d^2 est impair, il y a forcément un gagnant.

II Implémentation du jeu de go

Q 3. C'est une initialisation de structure :

```
goban initialisation(int d) {
  goban g;
  g.d = d;
  g.m = malloc(d * d * sizeof(int)); // allocation
  for (int i = 0; i < d * d; i++) g.m[i] = 0; // initialisation
  return g;
};
```

Q 4.

```
  ...
  int k = 0;
  if (i - 1 >= 0)  {v.t[k] = pos(i - 1, j); k = k + 1;}
  if (i + 1 < g.d) {v.t[k] = pos(i + 1, j); k = k + 1;}
  if (j - 1 >= 0)  {v.t[k] = pos(i, j - 1); k = k + 1;}
  if (j + 1 < g.d) {v.t[k] = pos(i, j + 1); k = k + 1;}
  ...
```

Q 5. Le goban étudié est ici appréhendé comme un graphe, une arête existe entre deux pierres connectées. Un groupe s'obtient par un parcours en profondeur : c'est exactement l'état final du tableau vu. Soit :

```
void parcours(goban graphe, goban vu, position p) {
    int couleur = lire(graphe, p); // supposée non nulle
    ecrire(vu, p, couleur); //marquage
    voisins v = pos_voisins(graphe, p);
    for (int i = 0; i < v.nb; i++)
    {
        position q = v.t[i];
        if (lire(graphe, q) == couleur && lire(vu, q) == 0) // connectée && pas vue
            parcours(graphe, vu, q);
    }
}
```

Puis :

```
goban groupe(goban g, position p) {
  // précondtion : on suppose qu'une pierre est en p.
  goban vu = initialisation(g.d);
  parcours(g, vu, p);
  return vu;
}
```

Q 6. Cette fonction possède les caractéristiques suivantes :

1. Elle teste correctement le voisinage d'une pierre.
2. Elle gère correctement les bords.
3. Une intersection voisine de plusieurs pierres d'un groupe est comptée plusieurs fois.
4. Elle ne vérifie pas qu'il y a une pierre en `p` : Dans la suite, dans ce cas le programme renverra -1.

Un test sera implémenté sous la forme `test(goban, position, nb_libertés_attendu)`.

On note `goban_abcd` le goban ayant les valeurs `a, b, c, d` respectivement aux positions (0,0), (0,1), (1,0), (1,1), et 0 ailleurs.

On propose alors les tests associés suivants :

1. `test(goban_0021, pos(1,1), 3)` réussit.
2. `test(goban_1000, pos(0,0), 2)` réussit.
3. `test(goban_1110, pos(0,0), 3)` échoue, car ce programme trouve 4 libertés.
4. `test(goban_0000, pos(1,1), -1)` échoue, car `cpt` $\geqslant 0$.

Q 7.

— Pour corriger le premier défaut, il faut créer un goban `vu` comme en question 5 , qu'on libère en fin de fonction. Puis remplacer `if (lire(g, v.t[k]) == 0) {cpt = cpt + 1;}` par :
`if (lire(g, v.t[k]) == 0 && lire(vu, v.t[k]) == 0) {cpt = cpt + 1; ecrire(vu, v.t[k], 1);}`.

— Pour corriger le second, il suffit d'insérer `if (couleur == 0) return -1;` après la définition de `couleur`.

Q 8. On refait un parcours en profondeur, la mise à 0 de l'intersection servant directement de marquage ici :

```
void retire_groupe(goban g, position p) {
  int couleur = lire(g, p); // supposée non nulle
  ecrire(g, p, 0); // prise de la pierre et marquage.
  voisins v = pos_voisins(g, p);
  for (int i = 0; i < v.nb; i++)
  {
    position q = v.t[i];
    if (lire(g, q) == couleur) retire_groupe(g, q);
  }
}
```

Q 9.

```
void joue(goban g, position p, int couleur) {
  assert (p.i >= 0 && p.i < g.d);
  assert (p.j >= 0 && p.j < g.d);
  assert(lire(g, p) == 0);  // place libre
  ecrire(g, p, couleur); // ajout de la pierre
  voisins v = pos_voisins(g, p);
  for (int i = 0; i < v.nb; i++) {
    if (lire(g, v.t[i]) != couleur && liberte(g, v.t[i]) == 0)
```

```
    // liberte <0 si couleur == 0.
      retire_groupe(g, v.t[i]);
  }
  assert(liberte(g, p) != 0); // coup illégal
};
```

III Prédiction et évaluation d'un prochain coup

III.A K plus proches voisins

Q 10. On en déduit une profondeur moyenne de $29\,10^6/16\,10^3 \simeq 184$ coups. Pour la largeur on à 19^2 choix au premier coup, $19^2 - 1$ au second, puis aucun en fin de partie. en constatant que $19^2 = 361 \simeq 2 \times 184$, on (sous-)estime grossièrement la profondeur à $361 \times 359 \times \ldots 3 \times 1 \simeq 10^{384}$. L'algorithme min-max sans heuristique nécessite une exploration exhaustive de cet arbre, ce qui n'est donc pas envisageable ici.

Q 11. On réalise k recherches de minimum successives en place , le début du tableau contient les distances recherchées :

```
double* k_plus_petits(double* distances, int n, int k) {
    for (int i = 0; i < k; i++) {
        for (int j = i + 1; j < n; j++) {
            if (distances[i] > distances[j]) {
                double tmp = distances[i];
                distances[i] = distances[j];
                distances[j] = tmp;
              }
          }
      }
    return distances;
  }
```

Pour chaque $i < k$, on fait moins de n comparaisons ce qui garantit une complexité $O(nk)$.

Pour la boucle externe, « les $i + 1$ premiers éléments sont les plus petits éléments en fin de boucle » est visiblement le bon invariant ; ce qui entraîne la correction.

Q 12. On peut améliorer la complexité avec une structure de tas. Chaque extraction de minimum se fait alors en $\mathcal{O}(\log_2 n)$. Construire le tas est de complexité linéaire, on obtient donc une complexité globale en $\mathcal{O}(n + k \log_2 n)$.

Q 13. Soit a et b les deux plus petites distances. Même si toutes les comparaisons ont été effectuées sauf celle entre a et b, alors la séquence triée des distances est soit $a, b, \ldots$, soit $b, a, \ldots$: seule la comparaison directe permet de conclure sur le plus petit élément.

Q 14. Pour obtenir la complexité demandée, if faut organiser les comparaisons sous forme de tournoi : À chaque tour, on ne conserve que les gagnants.

Ainsi, en $\left(\frac{1}{2^p}\right)^{\text{e}}$ de finale, on effectue 2^p confrontations, soit 2^p comparaison(s). Le tournoi nécessite donc au total $\frac{n}{2} + \frac{n}{4} + \ldots + 1 = n - 1$ comparaisons.

En prenant la précaution de conserver pour chaque gagnant la liste des distances associées à ses adversaires directs, la seconde plus petite distance sera d'après la question précédente dans la liste associée au vainqueur final. Elle sera la plus petite de cette liste et trouvée en $\log_2 n - 1$ comparaisons, Le gagnant ayant réalisé exactement $\log_2 n$ confrontations.

On peut donc bien trouver les deux plus petites distances en exactement $\underline{n + \log_2 n - 2}$ comparaisons.

III.B Hachage de Zobrist

Q 15. On dénombre $n_i = 4^{\frac{i(i+1)}{2}}$ intersections pour un motif de zoom i, et 4 valeurs possibles ($\in 0, 1, 2, 3$) par intersection, soit $4^{n_i} = 16^{i(i+1)}$ motifs de zoom i. Il y a donc $16^2 = 256$ motifs de zoom 1 différents.

Dans la base, on peut isoler ($m \simeq 29$ millions $\times d^2$) motifs. Comme $m \gg 256$, on peut supposer que chacun apparait plusieurs fois.

Q 16.

$$\begin{array}{lcll} A(g(\text{vide},(0,+1))) & : & 00001111 & \oplus \\ A(g(\text{vide},(-1,0))) & : & 01111011 & \oplus \\ A(g(\text{noir},(0,-1))) & : & 00011000 & \oplus \\ A(g(\text{blanc},(+1,0))) & : & 10001101 & \\ \hline h(Z_1(g,(7,6))) & = & 11100001 & \end{array}$$

Pour rajouter une pierre blanche en (6,6), on enlève d'abord « vide » :

$$h(Z_1(g,(6,6))) = h(Z_1(g,(7,6))) \oplus A(g(\text{vide},(-1,0))) \oplus A(g(\text{blanc},(-1,0))).\,[1]$$

Q 17.

```
uint64_t* init_t_hash(int d, int zoom) {
    uint64_t* table = malloc(d * d * sizeof(uint64_t));
    goban goban_vide = initialisation(d);
    for (int i = 0, k = 0; i < d; i++) {
      for (int j = 0; j < d; j++, k++) {
        table[k] = hachage(pos(i, j), goban_vide, zoom);
      }
    }
    return table;
  }
```

Q 18. On peut proposer le code suivant :

```
void maj_n(t_hachage table, uint64_t* tableau_hache, goban gN, position p) {
  int k = p.i * gN.d + p.j;
  int cle = tableau_hache[k] % table.nt;
  table.t[cle].n++;
}

void maj_v(t_hachage table, uint64_t* tableau_hache, goban gN, goban gB, position p) {
  int k = p.i * gN.d + p.j;
  int cle = tableau_hache[k] % table.nt;
  if (lire(gN, p) != lire(gB, p)) table.t[cle].v++; // jeu au centre du motif
}

void maj_h(uint64_t* tableau_hache, goban gN, goban gNs, int zoom, position p){
     int k = p.i * gN.d + p.j;
     tableau_hache[k] = maj_hachage(p, gN, gNs, zoom, tableau_hache[k]);
}
```

Q 19. Un wrapper avec les fonctions précédentes :

```
t_hachage lit_partie(partie* lp, int zoom, t_hachage table) {
  uint64_t* tableau_hache = init_t_hash(lp->g.d, zoom);
  while (true) {
```

1. Le sujet ne présente ici que l'aspect technique du hachage de Zobrist. Il est concu pour minimiser les collisions entre configurations, et permet de passer rapidement d'une configuration à une configuration voisine.

```
    goban gN = lp->g;
    partie* lB = lp->suivant;
    if (lB == NULL) return;
    goban gB = lB->g;
    maj_coup(table, tableau_hache, gN, gB, zoom);
    partie* lNs = lB->suivant;
    if (lNs == NULL) return;
    goban gNs = lNs->g;
    maj_t_hache(tableau_hache, gN, gNs, zoom);
    partie* lp = lNs;
  }
}
```

Q 20. D'après la question 16, 2 XOR permettent de poser une pierre sur une case, et on pose au plus c pierres sur le motif, soit au plus $2c$ XOR. Mais chacune de ces pierres peut être capturée, ce qui coutera $2c$ XOR supplémentaires, soit au total au plus $4c$ XOR par motif lors d'une partie.

Q 21.

```
bool appartient(position p, goban g, t_hachage tbl, int zoom) {
  int k = hachage(p, g, zoom) % tbl.nt;
  return tbl.t[k].n > 0;
}
```

Q 22.

```
float evalue_zoom(position p, goban g, t_hachage tbl, int zoom) {
  int k = hachage(p, g, zoom) % tbl.nt;
  observation obs = tbl.t[k];
  return obs.v / obs.n;
}
```

Q 23. Par construction, le niveau de zoom $i+1$ ne peut exister que si le niveau i existe. De plus d'après la suggestion de la question 15, le zoom 1 existe toujours. On peut alors avoir une approche dichotomique :

```
float evaluation(position p, goban g, tt_hachage tt) {
  int i = 1, j = tt.nz;
  while (j > i + 1) {
    // invariant : le niveau de zoom maximal est dans [i,j[.
    int k = (i + j ) / 2;
    if (appartient(p, g, tt.t[k], k)) { i = k; } else { j = k; }
  }
  return evalue_zoom(p, g, tt.t[i], i);
}
```

— la terminaison est garantie par la décroissance stricte de $j - i$;

— un invariant de boucle est que le niveau de zoom maximal est dans $[i, j[$. Ceci garantit qu'il vaille i en fin de boucle, lorque $j = i + 1$.

— $(j - i)$ est divisé par 2 à chaque fois, ce qui conduit à une complexité logarithmique.

IV Recherche arborescente de Monte-Carlo

Q 24.

```
let js = function Noir -> Blanc | Blanc -> Noir
```

IV.A Simulation de partie pour évaluer une configuration

Q 25.

```
let rec sim_defaut (g:goban) (j:joueur) =
  try
    let p = strategie_defaut g j in
    let g = joue g p j in
    sim_defaut g (js j)
  with Pas_de_coup_valide -> gagnant g
```

IV.B Valeur d'action

Q 26.

```
  let etiq (N (e,a)) = e
```

Q 27.

```
let valeurActionNoir (N (e, a)) =
  Mutex.lock e.t;
  let va =
    if e.n = 0 then
      e.prior
    else
      let n, v = (float_of_int e.v, float_of_int e.n) in
      v /. n +. e.prior /. (n +. 1.)
  in
  Mutex.unlock e.t;
  va
```

Q 28. Pour Blanc, puisque $p_{\text{Blanc gagne}} + p_{\text{Noir gagne}} = 1$, on définit naturellement la valeur d'action pour Blanc ainsi :

$$V_a^B = \begin{cases} \frac{n_a - v_a}{n_a} + \frac{1 - \text{prior}_a}{1+n_a} & \text{si } n_a > 0 \\ 1 - \text{prior}_a & \text{sinon} \end{cases}$$

Q 29.

```
let rec popargmax l f =
  match l with
  | [] -> raise Pas_de_coup_valide
  | [ a ] -> (a, [])
  | t :: q ->
    let u, v = popargmax q f in
    if f t > f u then (t, u :: v) else (u, t :: v)
```

IV.C Descente et remontée dans l'arbre

Q 30. On applique la recette fournie. On utilise une exception en arrivant sur une feuille.

```
let rec descente_remontee_arbre (g : goban) (j : joueur) (a : arbre) =
  match a with
  | N (e, l) ->
      try
```

```
        let a', _ =
          popargmax l (if j = Noir then valeurActionNoir else valeurActionBlanc)
        in
        Mutex.lock e.t;
        e.v <- (if j = Noir then e.v + 5 else e.v - 5);
        Mutex.unlock e.t;
        let j' = descente_remontee_arbre (joue g e.pos j) (js j) a' in
        Mutex.lock e.t;
        e.v <- (if j = Noir then e.v - 5 else e.v + 5);
        e.n <- e.n + 1;
        if j' = Noir then e.v <- e.v + 1;
        Mutex.unlock e.t;
        j'
      with Pas_de_coup_valide -> sim_defaut g j
```

IV.D Expansion de l'arbre

Q 31.

```
let expansion_feuille (g : goban) (j : joueur) =
  List.map
  (fun (pos, prior) ->
  N ({ n = 0; v = 0; prior; pos; t = Mutex.create () }, []))
  (liste_coup_prior g j)
```

Q 32.

```
let rec expansion (g : goban) (j : joueur) (N (e, l) : arbre) =
  try
    let a', l' =
    popargmax l (if j = Noir then valeurActionNoir else valeurActionBlanc)
    in
    N (e, expansion (joue g e.pos j) (js j) a' :: l')
  with Pas_de_coup_valide -> N (e, expansion_feuille g j)
```

IV.E Algorithme complet

Q 33. Classiquement : `let attendre = List.iter Thread.join`

Q 34.

```
let etape_complete nt g j a =
  attendre
    (List.map
       (Thread.init (descente_remontee_arbre g j))
       (List.init nt (fun _ -> a)));
  expansion g j a
```

Q 35. L'excusion mutuelle est ici assurée par des verrous libérés immédiatement , ce qui empêche l'interblocage. Si l'arbre est une feuille,`descente_remontee_arbre` termine car `sim_defaut` termine. Sinon on se ramène à à un sous arbre de profondeur strictement inférieur, ce qui garantit la terminaison.

Q 36.

```
let rec recherche_arborescente ns nt (g : goban) (j : joueur) (a : arbre) =
  if ns > 0 then
```

```
    recherche_arborescente (ns - 1) nt g j (etape_complete nt g j a)
  else
    match a with
    | N (_, l) ->
        let N (e', l'), _ = popargmax l (fun (N (e, _)) -> e.n) in
        (e'.pos, N (e', l'))
```

V Complexité du problème de la stratégie gagnante au go

Q 37.

— La classe **NP** est la classe des problèmes de décision possédant un vérificateur polynomial.

— Un problème de décision P_1 est dit **NP-dur** si tout problème P_2 de la classe **NP** peut être réduit en temps polynomial à P_1.

— Un problème **NP-complet** un un problème de **NP** qui est **NP-dur**.

Le problème de l'existence d'un chemin hamitonien dans un graphe est un problème NP-complet.

Q 38. le gadget (d) a deux "yeux". Noir ne peut jouer dans un de ces yeux, car l'autre œil empêche la capture du gadget : le coup est donc interdit d'après les règles. Si la réserve est liée à ce gadget, elle est donc protégée.

Q 39. Si Blanc joue en 1, Noir doit jouer en 2 pour éviter que Blanc ne le fasse, ce qui lui permettrai de sécuriser sa réserve avec le gadget (d) du bas, en capturant trois pierres noires au passage. En jouant alors en 5, Blanc « continue » son chemin vers la droite, et Noir doit alors jouer en 6 pour ne pas perdre au coup suivant. Symétriquement en démarrant en 2, il peut au pire continuer vers la gauche. Blanc peut donc choisir, et il garde la main.

On a vu que les coups de Noirs sont contraints. Ceux de Blanc aussi : si Blanc ne joue pas en 1 ou 2, en 3 ou 4 par exemple, Noir joura en 2 puis en 1 ou 5 et capturera ainsi la réserve. Il existe un raisonnement symétrique si Blanc joue en 5 ou 6. On peut s'aider d'un figuier par exemple pour trouver le raisonnement précédent.

Q 40. Il suffit de supprimer la pierre noire dans la colonne centrale dans le gadget (a), et d'inverser les couleurs des autres pierres. Blanc doit jouer là où la pierre a été supprimée, sinon Noir sauve sa réserve en jouant en cette position. On est ramené ainsi au cas inverse de la question précédente, et Noir choisit.

Q 41. Le gadget (b) permet à Blanc de continuer à faire vivre au choix sa réserve gauche en jouant en 1 ou sa réserve droite en jouant en 2.

Q 42. Il suffit pour Blanc de jouer en 1. Si Noir joue en 4, Blanc gagne en jouant ensuite en 4, connectant les yeux à la réserve. Sinon Blanc joue en 4 et gagne, si le tuyau transversal du haut a été préalablement relié à la réserve.

Q 43. D'après la question 39, il suffit a Noir de jouer en 1 (respectivement 2) si Blanc à joué en 2 (respectivement 1), pour que la connexion ne puisse se faire que par l'un des deux chemins.

Q 44. D'après la question 39, Blanc peut choisir n'importe quel chemin dans le gadget B et garder la main pour le continuer avec la gadget (b) suivant. Il peut donc instancier n'importe quel valuation.

Q 45. Le correcteur à déjà largement dépassé le temps imparti.

Q 46. La question est intéressante. On en déduirait que *Go généralisé* est **NP-complet**.