Synthèse :

Ma thèse s'intitule « Contributions sur la prise de décision séquentielle basée sur des simulations dans des environnements complexes de grande dimension ». Le cadre de la thèse s'articule autour du jeu, de la planification et des processus de décision markovien.

Un agent interagit avec son environnement en prenant successivement des décisions. L'agent part d'un état initial jusqu'à un état final dans lequel il ne peut plus prendre de décision. A chaque décision, l'agent reçoit une observation de l'état de l'environnement. A partir de cette observation et de ses connaissances, il prend une décision qui modifie l'état de l'environnement. L'agent reçoit en conséquence une récompense et une nouvelle observation. Le but est de maximiser la somme des récompenses obtenues lors d'une simulation qui part d'un état initial jusqu'à un état final. La politique de l'agent est la fonction qui, à partir de l'historique des observations, retourne une décision.

Nous travaillons dans un contexte où (i) le nombre d'états est immense, (ii) les récompenses apportent peu d'information, (iii) la probabilité d'atteindre rapidement un bon état final est faible et (iv) les connaissances a priori de l'environnement sont soit inexistantes soit difficilement exploitables.

Les 2 applications présentées dans cette thèse répondent à ces contraintes : le jeu de Go et le simulateur 3D du projet européen MASH (Massive Sets of Heuristics).

Afin de prendre une décision satisfaisante dans ce contexte, plusieurs solutions sont apportées :

1. simuler en utilisant le compromis exploration/exploitation (MCTS)
2. réduire la complexité du problème par des recherches locales (GoldenEye)
3. construire une politique qui s'auto-améliore (RBGP)
4. apprendre des connaissances a priori (CluVo+GMCTS)

L'algorithme Monte-Carlo Tree Search (MCTS) est un algorithme qui a révolutionné le jeu de Go. A partir d'un modèle de l'environnement, MCTS construit itérativement un arbre des possibles de façon asymétrique en faisant des simulations de Monte-Carlo et dont le point de départ est l'observation courante de l'agent. L'agent alterne entre l'exploration du modèle en prenant de nouvelles décisions et l'exploitation des décisions qui obtiennent statistiquement une bonne récompense cumulée. Nous discutons de 2 moyens pour améliorer MCTS : la parallélisation et l'ajout de connaissances a priori.

La parallélisation ne résout pas certaines faiblesses de MCTS ; notamment certains problèmes locaux restent des verrous. Nous proposons un algorithme (GoldenEye) qui se découpe en 2 parties : détection d'un problème local et ensuite sa résolution. L'algorithme de résolution réutilise des principes de MCTS et fait ses preuves sur une base classique de problèmes difficiles.

L'ajout de connaissances à la main étant laborieuse et ennuyeuse, nous proposons une méthode appelée Racing-based Genetic Programming (RBGP). Cette méthode permet d'ajouter automatiquement de la connaissance. Le point fort de cet algorithme est qu'il valide rigoureusement l'ajout d'une connaissance a priori et il peut être utilisé non pas pour optimiser un algorithme mais pour construire une politique.

Dans certaines applications telles que MASH, les simulations sont coûteuses en temps et il n'y a ni connaissance a priori ni modèle de l'environnement; l'algorithme Monte-Carlo Tree Search est donc inapplicable. Pour rendre MCTS applicable dans MASH, nous proposons une méthode

pour apprendre des connaissances a priori (CluVo). Nous utilisons ensuite ces connaissances pour améliorer la rapidité de l'apprentissage de l'agent et aussi pour construire un modèle. A partir de ce modèle, nous utilisons une version adaptée de Monte-Carlo Tree Search (GMCTS). Cette méthode résout de difficiles problématiques MASH et donne de bons résultats dans une application dont le but est d'améliorer un tirage de lettres.

L'algorithme Monte-Carlo Tree Search construit de façon asymétrique un arbre des futurs possibles. Elle se déroule en 3 grandes étapes : descente dans l'arbre, évaluation des feuilles et développement et mise à jour de l'arbre. L'évaluation se fait à partir de simulations de Monte-Carlo. Une formule (typiquement, une formule de bandit) est utilisée pour biaiser la construction de l'arbre.

Pour améliorer l'algorithme, nous pouvons paralléliser l'algorithme ou ajouter de la connaissance a priori. Nous allons voir que la parallélisation a ses limites et que l'ajout de connaissance a priori n'est pas si aisée.

Pour parler de parallélisation, nous présentons la variante appelée Slow Tree parallelization. Chaque machine construit son propre arbre. A chaque pas de temps *T0*, on fusionne les statistiques de l'arbre de toutes les machines pour les noeuds qui sont à une profondeur inférieure ou égale à *K* et qui ont au moins *N* simulations. Une variante est Slow root parallelization où seule la racine est considérée (*K=1*).

Nous avons implémenté et testé cet algorithme sur notre logiciel de Go, MoGo. En terme de speed-up, nous constatons qu'un plateau est atteint autour de 16 coeurs en 9x9 mais nous obtenons une amélioration régulière en 19x19. Nous avons également mesuré la scalabilité de MCTS dans le cadre des jeux. La scalabilité est ici définie comme la capacité à jouer mieux si on ajoute plus de puissances de calculs ou si on donne plus de temps de réflexion. Nous avons testé MoGo avec *2n* simulations contre MoGo avec *n* simulations. Nous constatons que la performance décroît avec *n*.

Une autre solution pour améliorer MCTS est d'ajouter de la connaissance a priori. Ce point n'est pas trivial notamment dans la partie simulation de Monte-Carlo. Sylvain Gelly a constaté que MoGo+GnuGo (= fort simulateur) était plus faible que MoGo combiné avec un faible simulateur.

Dans cette thèse, nous allons ajouter/construire de la connaissance a priori dans MCTS. Il est possible d'ajouter cette connaissance dans la partie simulation de Monte-Carlo mais ce n'est pas le propos de cette thèse. Les endroits où nous ajouté de la connaissance sont dans l'arbre, un cas à la fin de la simulation de Monte-Carlo et 2 cas où nous générons de la connaissance a priori non pas pour améliorer MCTS mais pour pouvoir l'utiliser : l'algorithme de détection de semeai dans GoldenEye et l'algorithme de clustering CluVo.

Le premier endroit pour ajouter de la connaissance a priori est dans l'arbre. Nous avons ajouté de façon statique de la connaissance a priori. La formule de bandit de MoGo est constituée de 3 termes. Le premier est l'estimée en ligne (le terme d'exploitation). Le second terme est les valeurs RAVE apprises à partir des simulations et le troisième terme est l'apprentissage hors ligne ainsi que l'exploration. Quand il y a peu ou pas de simulations, c'est le troisième terme qui est prépondérant. Quand il y a un nombre moyen de simulations, c'est le second terme qui est le plus important dans la formule. Enfin, après beaucoup de simulations, c'est le premier terme qui guide principalement la descente dans l'arbre.

Nous avons ajouté la connaissance a priori (expertises) à la fois dans le troisième terme et

dans le second terme. Dans le troisième terme, nous cumulons les expertises. Dans le second terme, grossièrement, la valeur cumulée des expertises initialise les valeurs RAVE (sorte de gains virtuels). Nous avons ajouté des connaissances humaines, typiquement des bonnes et mauvaises formes au jeu de Go. J'ai notamment réglé les coefficients associés aux expertises.

Les résultats sont plutôt bons. Bien qu'un motif apporte peu d'améliorations (< 1%), tous les motifs cumulés donnent un gain de 63.5% +/- 0.5 contre la baseline.

Malgré la parallélisation et l'ajout statique de connaissances, certains problèmes restent des verrous au jeu de Go. C'est le cas des semeais. Le semeai est une course de vitesse entre 2 groupes qui ne peuvent pas vivre sur place à moins de capturer le groupe adverse. Un semeai est généralement une situation locale.

Nous avons proposé comme solution de résoudre ces problèmes de façon dynamique en implémentant un solveur tactique GoldenEye. Le solveur doit être capable (i) de détecter ces situations et (ii) ensuite de les résoudre.

La première phase consiste à trouver ces situations locales sur le goban (plateau de Go). Nous nous sommes demandés si nous ne pouvions pas les trouver par motif. Les semeais peuvent prendre des formes nombreuses et variées. Nous avons opté pour une approche statistique en se disant qu'un semeai est un *ou exclusif* entre la vie de 2 groupes. Nous avons défini un terme *Ad_sim(s1,s2)* comme étant la fréquence d'une pierre *s1* en vie et une autre *s2* morte en un certains nombres *sim* de simulations. Nous avons défini ensuite *sem_ sim(sn,sb) = AD_sim(sn,sb) +AD_sim(sb,sn)*. Une pierre noire *sn* et une pierre blanche *sb* sont dites en semeai si le terme défini précédemment est supérieur ou égal à un paramètre ρ, valeur comprise entre 0 et 1. Les semeais sont ensuite construits par agrégation de pierres.

Nous avons mené une expérience suivant la valeur de ρ. Idéalement, ρ devrait valoir 1. Mais comme les données proviennent de simulations aléatoires, nous considèrons cette valeur trop stricte. Avec $\rho = 0.5$, des pierres qui n'ont rien à voir entre elles (d'un point de vue du jeu) sont regroupées dans un seul et même semeai. Avec $\rho = 0.75$, ces mêmes pierres sont cette fois-ci regroupées dans des semeais différents. Nous avons choisi $\rho = 0.83$ car les solutions étaient satisfaisantes.

Maintenant que GoldenEye est capable de détecter les semeais, nous entrons dans la deuxième phase de l'algorithme : la résolution. Comme un semeai est une situation locale, nous allons effectuer une recherche locale.

La recherche locale comprend le contrôle de l'expansion des coups et un module d'évaluation très efficace capable de voir très rapidement la capture d'un groupe. Les coups considérés sont à une distance de 2 du semeai. Le module évite de faire trop de simulations mais il coûte cher en temps.

L'algorithme de résolution est appelé MCTS* ; c'est une combinaison entre A* et MCTS. En effet, l'algorithme parcourt une partie de l'arbre pour choisir une feuille (A*) puis lance une simulation de Monte-Carlo à partir de cette feuille (MCTS) et ensuite l'arbre est updaté. Notez que toute la simulation est conservée dans l'arbre et l'algorithme prouve si le semeai peut être gagné ou pas. Un sérieux concurrent à l'algorithme est Proof Number Search (PNS). MCTS* n'utilise pas les informations de PNS tels que le nombre d'enfants qu'il reste à prouver.

Comme un joueur d'échec, GoldenEye essaie de gagner d'une certaine façon en choisissant un coup. Soit il y parvient, il cherche alors à réfuter cette ligne en considérant des lignes secondaires de l'adversaire, soit il n'y parvient pas, dans ce cas, il recherche une autre attaque.

Pour améliorer la rapidité de la résolution, plusieurs heuristiques ont été proposées, notamment l'*inhibition*. L'*inhibition* est un élagage temporaire. Un nœud est inhibé si ses *k* dernières simulations ont été perdues (*k = 10* dans GoldenEye). Un nœud est réactivé si tous ses nœuds frères sont inhibés ou résolus. Quand un nœud est réactivé, tous ses frères sont aussi réactivés. Le point important est quand un nœud est inhibé, ce nœud et le sous-arbre résultant ne sont plus visités. L'*inhibition* a 2 avantages : la partie de l'arbre à explorer pour choisir une feuille est considérablement réduite et nous évitons de dépenser du temps dans des variantes qui semblent récemment réfuter.

Nous avons testé notre algorithme sur une base classique de 10 semeais réputée difficile pour les programmes. Cette base a été créé en 2008 par Yoji Ojima, l'auteur du programme Zen, actuellement (en avril 2013) le meilleur programme au monde. Nous avons d'abord testé l'impact de l'heuristique *inhibition* sur la résolution complète de problèmes. Nous constatons que les semeais sont effectivement résolus plus rapidement. Nous avons aussi regardé son impact d'un point de vue temps d'exécution et mémoire sur un problème. Nous constatons qu'avec un même nombre de simulations, sans l'*inhibition*, le temps d'exécution explose après un certain nombre de simulations tandis qu'avec l'*inhitibion*, le temps d'exécution reste convenable. De même, nous constatons qu'il y a moins de mémoire utilisée avec l'*inhibition*.

Le fait qu'on gagne du temps est logique car par exemple, l'arbre à parcourir pour choisir une feuille est nettement plus petit. Par contre, je n'ai pas d'explication logique quant au gain de mémoire. Peut-être dans ce problème, l'*inhibition* a évité d'entrer dans des variantes compliquées et profondes. Je pense qu'il faudrait le tester sur d'autres problèmes pour confirmer ce résultat.

Nous avons ensuite comparé MoGo et GoldenEye sur les 10 semeais. En terme de simulations, GoldenEye est meilleur. Cependant, les simulations de GoldenEye sont beaucoup plus coûteuses que MoGo. En terme de temps, les résultats sont similaires.

Maintenant que nous avons un solveur tactique capable de résoudre des semeais, nous nous sommes demandés comment ajouter cette connaissance a priori dynamique dans MoGo. Nous avons proposé 2 solutions : par l 'expertise et par le conditionnement.

L'expertise introduit un biais dans le score. A travers le conditionnement, toutes les simulations qui ne sont pas consistantes avec le solveur tactique sont éliminées et sont rejouées. Nous avons testé sur 2 problèmes de semeai. Le premier est un cas dans lequel le premier joueur qui joue dans le semeai gagne. Le second est un cas où quelque soit le joueur qui commence, Noir gagne le semeai car il dispose de 2 coups d'avance. Nous constatons une amélioration quand le semeai devrait être jouer avec les variantes MoGo+expertise et MoGo+combinaison expertise et conditionnement. Cependant, nous n'observons aucune amélioration dans le second cas. Au contraire, nous avons peut-être même une détérioration des résultats. Avec peu de simulations, MoGo avec ou sans GoldenEye ne joue pas dans le semeai. Mais plus le nombre de simulations augmentent, plus MoGo y joue fréquemment.

Nous avons proposé une solution pour introduire de la connaissance statique ou dynamique à la main. Mais c'est ennuyeux, non trivial et biaisée par les idées humaines. Il existe des outils pour automatiser cet ajout : les algorithmes évolutionnaires.
Un algorithme évolutionnaire construit automatiquement un programme au travers d'essais (mutations) et d'erreurs (tests) dans le but de résoudre une tâche. Les 3 principaux problèmes rencontrés sont :

- le coût d'une évaluation d'une mutation
- la taille du grand ensemble possible de mutations
- la stochasticité de la fonction *fitness*.

Nous avons un programme *P*, un ensemble *S* de mutations possibles sur *P* et une évaluation de Monte-Carlo de *P+m* avec *m* appartenant à *S*. Soit nous obtenons une victoire contre *P* (*fitness = 1*), soit nous obtenons une défaite (*fitness = 0*). Nous souhaitons trouver une bonne mutation dans l'ensemble *S* et ne pas valider une mauvaise mutation.
Dans les algorithmes évolutionnaires, 2 principales questions sont en suspens :

1. la répartition de charges (sur quelle mutation va se porter nos efforts de calculs?)
2. la validation statistique.

Les approches bandits résolvent la première question. Cependant, l'approche dite de course résout les 2 questions à la fois.

Nous souhaitons automatiser la génération automatique de motifs aléatoires et automatiser la validation. La validation est une étape très difficile car le risque est cumulé avec le nombre d'essais. Avec un risque d'erreur classique à 5% par test, après 100 tests, nous avons une probabilité de 99.4% d'avoir accepté une mauvaise mutation : l'effet des tests multiples.

Des outils mathématiques existent pour valider une mutation : les bornes en particulier les bornes de Hoeffding et les bornes de Bernstein. Les bornes de Bernstein sont meilleures si la variance est petite. Or comme l'impact d'une mutation est faible, la variance est proche de ¼. Nous avons conservé les bornes de Hoeffding.

Les algorithmes stopping font tourner des tests jusqu'à ce que la mutation soit validée ou rejetée. Nous proposons une approche de course qui combine à la fois un algorithme de type bandit et un algorithme de type stopping. L'algorithme s'appelle Racing-Based Genetic Programming (RBGP).

Nous partons d'un ensemble *S* de mutations. Tant que l'ensemble *S* n'est pas vide, nous choisissons un individu *s* dans l'ensemble *S*. Soit *n* le nombre de simulations de la mutation *s*, nous effectuons *n* nouvelles simulations de la mutation *s*. Le nombre de simulations de la mutation *s* a donc doublé. Notez que nous n'effectuons pas un test après chaque simulation car le test a un coût. Ces *n* nouvelles simulations nous permettent d'affiner notre intervalle de confiance de la mutation *s*. Si la nouvelle borne inférieure de la mutation *s* est strictement supérieure à 50.1% alors une bonne mutation a été trouvée et nous sortons du programme. Veuillez prendre en considération qu'il existe peut-être d'autres bonnes mutations et qui sont éventuellement meilleures que la mutation *s* mais l'algorithme RBGP ne les cherchera pas. Si la nouvelle borne supérieure de la mutation *s* est strictement inférieure à 50.4%, alors nous éliminons la mutation *s*. La mutation *s* apporte peut-être quelque chose au programme *P* mais nous considérons que ce bonus n'est pas intéressant. En outre, elle ne serait pas validée dans un temps raisonnable.

Pour calculer les bornes, nous sommes confrontés au problème que nous ne connaissons pas à l'avance le nombre de tests que nous allons faire. Lorsqu'on effectue un test, quel risque « local » doit-on choisir pour que globalement le risque cumulé soit inférieur à un seuil choisit par l'utilisateur? Nous utilisons comme astuce les suites convergentes et en particulier, la somme des inverses au carré qui tend vers $\frac{\pi^2}{6}$. Sans entrer dans les détails, nous choisissons le risque local δ_i telle que $i = \frac{6 \times \delta}{\pi^2 \times i^2}$ avec δ le risque global et i le numéro du test. Cette astuce nous permet de prendre en compte les tests multiples.

Les propriétés de RBGP est qu'avec une probabilité *1-δ*, l'algorithme termine, il est efficace (s'il existe une mutation dont le score est supérieur ou égal à 50,4%, une mutation sera trouvée) et consistant (aucune mauvaise mutation n'est acceptée).
Nous avons testé notre algorithme sur 3 applications :

- le jeu de Go 9x9
- le jeu de Go 19x19
- NoGo en taille 7x7

NoGo est une variante du jeu de Go où il est interdit de passer et le premier des 2 joueurs qui capture une pierre a perdu. NoGo est un bon défi pour les développeurs de jeu selon le séminaire Birs sur les jeux. Un des aspects intéressants de NoGo est qu'il n'y a aucune connaissance humaine.
Nous avons appliqué notre algorithme RBGP dans un algorithme de type $1+\lambda$ *EA*. Tant que vrai, nous générons notre ensemble *S* de mutations aléatoires et nous appliquons RBGP sur notre ensemble *S*. Un pattern aléatoire est une mutation.

Nous obtenons de bons résultats dans le Go 9x9 : 53.5% contre la baseline et 4 mutations ont été trouvées. En 19x19, aucune bonne mutation n'a été trouvée. Mais il faut savoir que notre logiciel de Go, MoGo, est déjà équipée d'une grosse base de motifs. Comme nous avons toujours utilisé dans nos expériences un coefficient fixe de 0.75 associé à nos patterns, celui-ci était peut-être inadapté dans notre cas de figure. Néanmoins, nous avons adopté une autre approche pour que l'algorithme RBGP trouve de bonnes mutations : nous avons supprimé la grosse base de motifs de MoGo. Cette version est appelée MoGo light. Cette fois-ci, RBGP a trouvé 6 mutations et MoGo light + 6 mutations bat MoGo light 61% du temps. Cependant, MoGo light + 6 mutations reste bien moins fort que MoGo avec sa grosse base de motifs.

Par contre, les résultats sont excellents sur l'application NoGo. Nous obtenons quasiment 70% d'amélioration et 43 mutations ont été trouvées. Ce qui est vraiment très difficile pour les programmeurs. Chaque mutation apporte peu (environ 1%). Même une mauvaise mutation donne des résultats proches de 50%. Nous aurions certainement réussi à valider manuellement 43 mutations mais nous aurions certainement commis des erreurs et nous n'aurions certainement pas obtenu 70% de succès.

Nous avons proposé également une étude sur le choix de la taille λ de la population *S*. Nous avons montré théoriquement que λ devrait être égale à $\log(1-f)/\log(\delta)$ avec δ le risque global choisi par l'utilisateur et *f* la fréquence des bonnes mutations. L'inconvénient de cette formule est que nous ne connaissons pas la fréquence *f*. Néanmoins, expérimentalement, les résultats sont similaires. Nous avons considéré que $\lambda = 1$ est suffisant.
L'algorithme présenté optimise Monte-Carlo Tree Search mais il peut aussi être utilisé pour construire une politique. Dans ce cas, une mutation est simplement une sous-politique.

Nous avons proposé différentes méthodes pour améliorer Monte-Carlo Tree Search Néanmoins, MCTS nécessite un modèle du problème. Or l'application MASH que nous allons considérer est sans modèle. Nous allons nous interroger si nous ne pouvons pas rendre MCTS utilisable dans un cadre partiellement observable sans modèle ? Nous allons essayer de construire un modèle et d'appliquer MCTS sur ce modèle.

Dans l'application MASH, le solveur ne dispose pas de connaissance a priori et aucun modèle n'est fourni. Dans un premier temps, nous présentons l'application MASH et dans un second temps, l'algorithme de résolution CluVo+GMCTS.

L'environnement MASH est un simulateur 3D. Il est constitué d'une salle rectangulaire entourée par 4 murs dont les textures sont grises. Un avatar est dans cette salle ainsi que 2 drapeaux : un bleu et un rouge.

L'observation de l'avatar est constitué de 3 heuristiques :

- *bleu* qui renvoie un booléen indiquant si un pixel est de couleur bleu

- *rouge* qui renvoie un booléen indiquant si un pixel est de couleur rouge
- *identité* qui renvoie le niveau de gris d'un pixel.

Chaque heuristique donne environ 100.000 caractéristiques (une caractéristique par pixel). Comme une observation composée de $3 \times 100.000 = 300.000$ caractéristiques est trop grande, nous réduisons la taille de l'observation à 10.000 caractéristiques. Nous effectuons un échantillonnage aléatoire de 10.000 caractéristiques au début de l'expérience. Cela fait tout de même plus de $2^{10.000}$ observations possibles.

Ce qui complique la tâche du solveur est qu'il n'a aucune notion d'heuristique. Il reçoit juste une séquence de nombres.

L'avatar se déplace dans la salle rectangulaire. Il dispose de 4 actions :

- Avancer
- Reculer
- Tourner à gauche
- Tourner à droite

Néanmoins, le solveur est agnostique sur les actions. Pour lui, il a 4 actions 0, 1, 2 et 3; il en ignore la sémantique.

Dans la plateforme MASH, nous disposons de plusieurs applications. Nous nous sommes intéressés en particulier à l'application appelée « bleu puis rouge ». Pour accomplir la tâche, l'avatar doit d'abord toucher le drapeau bleu puis ensuite le drapeau rouge. C'est l'unique façon d'atteindre un état final.

Les récompenses associées sont :

- toucher un mur : -1
- toucher le drapeau bleu pour la première fois : +5
- toucher le drapeau une autre fois : 0
- toucher le drapeau rouge
  - après avoir touché le drapeau bleu : +10
  - sans avoir touché le drapeau bleu : -5
- par défaut : 0

La récompense cumulée optimale est donc +15 (+5 quand l'avatar touche le drapeau bleu puis +10 quand il touche ensuite le drapeau rouge).

La tâche est compliquée pour le solveur car la plupart du temps, il bouge sans toucher quoique ce soit et il reçoit donc très fréquemment 0 comme récompense. La récompense apporte peu d'information et il n'a aucune idée de la direction dans laquelle il doit aller pour atteindre son état but.

Résoudre la tâche « bleu puis rouge » est donc un gros challenge. Ce qui facilite la tâche de l'avatar est :

1. l'espace d'action est très petit
2. les mauvais états finaux sont inexistants.

Cependant, beaucoup d'éléments rendent la tâche très ardue :

1. l'immensité de l'espace d'états
2. l'absence totale de connaissance a priori
3. le peu d'information apportée par la récompense
4. la faible probabilité d'atteindre rapidement un bon état final surtout si on procède de façon aléatoire
   la longueur d'une séquence correcte d'actions peut-être supérieure à 100 (l'avatar doit par exemple effectuer 60 décisions pour faire un tour complet sur lui-même)
5. l'environnement partiellement observable
6. les simulations coûteuses en temps
7. l'absence de modèle.

Pour résoudre cette tâche, la philosophie n'est pas d'optimiser ou améliorer une politique. (i) La récompense apporte peu d'information, (ii) Il n'y a aucune connaissance a priori ; ce qui implique que la direction vers un bon état final est inconnue, (iii) la probabilité d'atteindre rapidement un bon état final est petite. Une recherche aléatoire n'est donc pas une option.
Le premier objectif avant tout est d'atteindre un bon état final. Nous allons procéder ainsi :

1. Acquérir des savoir-faire en utilisant des méthodes non supervisées. Etre capable d'accomplir des sous-tâches qui sont :
    1. presque impossible à réaliser en procédant aléatoirement
    2. peut-être inutiles pour effectuer la tâche réelle.
2. Combiner ces savoir-faire pour accomplir la tâche.

Nous proposons comme solution d'acquérir des savoir-faire :

1. construire des macro-actions
2. regrouper des caractéristiques.

Nous construisons un modèle à partir de ces savoir-faire et nous utilisons une variante adaptée de Monte-Carlo Tree Search sur ce modèle.

Pour construire des macro-actions, nous catégorisons d'abord les actions. Nous utilisons ensuite ces catégories pour définir des macro-actions ayant du sens.
Pour catégoriser une action, nous répétons un même scénario afin d'étudier l'impact de l'action sur l'observation. 3 catégories ont été identifiées :

- actions stationnaires (après avoir appliqué l'action plein de fois, l'état devient fixe)
- actions périodiques (après avoir appliqué l'action plein de fois, on revient à notre état initial)
- actions inverses ($action\,a + action\,b = action\,b + action\,a = aucune\,action$)

Une action stationnaire est par exemple avancer, une action périodique tourner à gauche et 2 actions inverses tourner à gauche et tourner à droite.

Comme les simulations sont très coûteuses en temps (l'avatar effectue seulement 15 actions par seconde), une recherche aléatoire est inefficace.

Nous avons besoin d'utiliser des macro-actions. Une macro-action est dans notre cadre une action répétée plusieurs fois. L'environnement est ainsi mieux exploré.
Pour rendre encore plus efficace les simulations aléatoires, nous utilisons les catégories d'action pour définir des macro-actions ayant du sens. Par exemple, si la dernière action a été « tourner à gauche », on évitera de tourner à droite.

Comme nous pouvons faire des simulations plus efficaces, nous cherchons à structurer les composantes de l'observation en les regroupant. L'algorithme proposé est un regroupement avec une approche agglomérative. A l'initialisation, chaque caractéristique constitue un groupe. Nous répétons le processus suivant. Nous générons une collection de listes de caractéristiques grâce à des simulations. Nous fusionnons 2 groupes qui ont des caractéristiques corrélées. Ce processus est répété jusqu'à ce que le nombre de groupes ne change plus. Le comportement attendu sur « bleu puis rouge » est de trouver 2 groupes : un groupe de caractéristiques issus de l'heuristique rouge et un groupe de caractéristiques issus de l'heuristique bleu.

Grâce à ces groupes, nous allons définir des sous-buts. Un sous-but sera un groupe de caractéristiques. Nous allons ensuite utiliser une politique avec mémoire. La mémoire sera un arbre de sous-buts. Nous commencerons à la racine. Dans un noeud, nous utiliserons le vote. Cela consistera à utiliser des actions qui tendent à activer des caractéristiques du noeud. L'arbre de sous-buts sera construit en utilisant Monte-Carlo Tree Search.

Parfois, il faut accomplir plusieurs sous-tâches (sous-buts). Dans notre cas, il faut d'abord toucher le drapeau bleu puis ensuite toucher le drapeau rouge. Le solveur doit être capable de changer d'objectif une fois le drapeau bleu touché. La solution est donc d'utiliser une mémoire représentée par un arbre de sous-but. Un sous-but est l'activation du plus grand nombre possible de caractéristiques d'un regroupement (par exemple, dans l'application MASH, toucher un drapeau). Pour ce faire, nous utilisons un mécanisme de vote. Ce mécanisme est appris de façon statistique grâce à des simulations. La mémoire est donc représentée par un arbre de sous-buts : un nœud étant un sous-but et une arête la décision d'accomplir un sous-but.

Pour construire cet arbre, nous utilisons une variante de Monte-Carlo Tree Search appelé Goal Monte-Carlo Tree Search (MCTS appliqué sur des sous-buts). Jusqu'à un certain nombre de simulations, on réinitialise l'application, puis on répète le processus suivant : on choisit le sous-but le plus prometteur (exploitation) ou bien on créé un nouveau sous-but (exploration), tant que le sous-but n'est pas atteint, on prend une décision suivant le mécanisme de vote et on conserve les récompenses obtenues jusqu'à ce qu'un état final est atteint et enfin on met à jour l'arbre avec les récompenses conservées comme dans MCTS classique. Une fois cette étape terminée, on retourne la séquence de sous-buts la plus simulée.

Nous donnons les résultats de la combinaison Macro-actions+CluVo+GMCTS sur l'application MASH. Le temps d'exécution de CluVo a été de 2h par « run » (incluant 10 minutes pour catégoriser les actions). 12 « restarts » ont été effectués ; 3 ont été des succès. GMCTS a ensuite été lancé sur les 3 « runs » réussis. A chaque fois, GMCTS a duré environ 8h par « run » et 100 simulations ont été faites. On constate que GMCTS donne une récompense cumulée autour de 11, ce qui est un très bon score (sachant que la récompense cumulée optimale est de 15 et que l'avatar n'est pas capable d'éviter les murs). En outre, dans le troisième « run », on a environ 80% de succès (c'est à dire 80 fois sur 100 l'avatar a touché le drapeau bleu puis le drapeau rouge dans un temps limité de 1000 décisions). D'un point de vue temps d'exécution, 48 heures ($12\times 2h+3\times 8h$) sont nécessaires sur une machine séquentielle tandis que 10 heures (2h de CluVo+8h de GMCTS sur 12 machines) suffisent sur des machines parallèles.

En résumé, nous avons rendu les simulations efficaces en utilisant des macro-actions pour explorer l'environnement et en catégorisant les actions pour éviter de faire des actions inutiles. La politique finale est à partir d'une séquence donnée de sous-buts (apprise par GMCTS), si le sous-but courant (selon un regroupement de caractéristiques appris par CluVo) est atteint, alors on prend le prochain sous-but. A partir de ce sous-but, on choisit une décision suivant le mécanisme de vote. L'algorithme est-il générique ? Même s'il est encore trop tôt pour l'affirmer, nous l'avons néanmoins appliqué avec succès sur 2 applications très différentes : le simulateur 3D de MASH et l'optimisation d'un tirage de lettres.

En somme, nous avons ajouté dans MCTS de la connaissance a priori de façon statique (motifs) mais également dynamique (solveur tactique). Nous avons tout d'abord ajouté cette connaissance à la main. Puis comme l'ajout n'est pas facile, ennuyeuse et biaisé par les idées humaines, nous avons automatisé le processus en validant rigoureusement l'ajout d'un motif.

Nous avons généré de la connaissance a priori afin de pouvoir utiliser Monte-Carlo Tree Search. Finalement, nous avons proposé une même approche pour résoudre les problèmes de semeai dans GoldenEye et toucher les 2 drapeaux dans le problème MASH. Tout d'abord, nous effectuons un apprentissage non supervisé. Dans les 2 cas, à partir de simulations, nous extrayons statistiquement de la connaissance a priori en effectuant des regroupements. Dans la phase de détection de GoldenEye, nous regroupons des pierres afin de localiser notre problème. Dans

l'algorithme CluVo, nous regroupons des caractéristiques afin de pouvoir définir des sous-buts. A partir de ces connaissances a priori, nous utilisons Monte-Carlo Tree Search (méthode supervisée). Dans GoldenEye, nous utilisons une combinaison de MCTS et A* (MCTS*) afin de trouver le bon coup pour gagner le semeai. Dans MASH, nous utilisons une variante de Monte-Carlo Tree Search adapté aux sous-buts (GMCTS) afin d'accomplir la tâche « drapeau bleu puis drapeau rouge ».

En perspective, dans GoldenEye, nous pourrons comparer MCTS* avec l'état de l'art (notamment avec PNS), trouver de nouvelles solutions pour ajouter de la connaissance a priori dynamique car les variantes proposées restent un peu décevantes mais sont encourageantes et trouver de nouvelles applications pour MCTS*.

En ce qui concerne GMCTS, nous pourrons essayer d'utiliser GMCTS avec une autre définition de sous-but et un autre mécanisme (au lieu d'un mécanisme de vote) pour accomplir une sous-tâche. Nous pourrons également vérifier la généricité de l'algorithme.

Pour RBGP, nous pourrons choisir aléatoirement ou adapter le coefficient des motifs aléatoires, ne pas choisir les motifs de façon aléatoire ou encore améliorer les formules de borne. Nous pourrons par exemple utiliser une meilleure paramétrisation des risques locaux δ_i. Nous voulons que la somme des δ_i converge vers le risque global δ pour prendre en compte l'effet des tests multiples. Nous avons utilisé la somme des inverses au carré qui converge vers $\frac{\pi^2}{6}$ mais il existe de nombreuses autres suites convergentes (par exemple, $\sum \frac{1}{n^{100}}$).

Notre problématique de départ était « comment résoudre des problèmes de décisions séquentielles sous les contraintes suivantes :

1. le nombre d'états est immense,
2. les récompenses apportent peu d'information,
3. la probabilité d'atteindre rapidement un bon état final est faible
4. les connaissances a priori de l'environnement sont soit inexistantes soit difficilement exploitables. »

Nous avons apporté 4 réponses :

1. simuler en utilisant le compromis exploration/exploitation (MCTS)
2. réduire la complexité du problème par des recherches locales (GoldenEye)
3. construire une politique qui s'auto-améliore (RBGP)
4. apprendre des connaissances a priori (CluVo+GMCTS).

Nous avons travaillé sur une application de Go où un modèle nous était fourni. Comme dans MASH, nous pourrions essayer de jouer au Go avec les mêmes contraintes que MASH : sans modèle et sans connaissance a priori. L'algorithme MCTS ne s'appliquerait donc pas.

Dans l'application MASH, nous avons construit un modèle mais pas suffisamment précis pour pouvoir utiliser MCTS tel que nous l'utilisons au jeu de Go. Ne serait-il pas possible d'établir un modèle construit par un algorithme générique suffisamment précis pour utiliser MCTS classique? La question est ouverte et de nombreuses portes seraient alors ouvertes.