Computer Go Program

Report for the implementation gogogo

Xiao Jia
5090379042
stfairy@gmail.com

Abstract

The general weakness of computer Go programs compared with computer chess programs has served to generate research into many new programming techniques. Considering the difficulty of computer Go, we devised a combination of three approaches: heuristics based on stone power, pattern recognition, and Monte Carlo. In this report, our approach is presented along with the system architecture, the implementation details, and the evaluation result. Preliminary experiment results show that our approach is effective with respect to the simplicity and generality of the implementation.

1. Introduction

Computer Go is the field of artificial intelligence (AI) dedicated to creating a computer program that plays Go, a traditional board game. Go has long been considered a difficult challenge in the field of AI and is considerably more difficult to solve than chess. The techniques which proved to be the most effective in computer chess have generally shown to be mediocre at Go.

Considering the difficulty of computer Go, we devised a combination of three approaches:

1. calculating heuristic values as priorities for each position
2. pattern recognition
3. Monte Carlo algorithm based on random playing

We implement these approaches under a multi-phase framework [1] which is described in Section 3.

Our team consists of two members:

Shijian Li (5090379043) is responsible for (i) `e3gtp`, which is a stable and reusable implementation of the GTP protocol in Java but not used in our project in consideration of simplicity and efficiency; (ii) the pattern filter (see Section 3.2), which is a filter for pattern recognition and improves the overall performance dramatically. He works on parameter tuning and testing as well.

Xiao Jia (5090379042) is responsible for (i) building the application and algorithm framework, which plays an important role in our implementation and eases the burden of trying out new ideas; (ii) setting up the skeleton of the phases and basic filters for expanding territories and capturing stones aggressively; (iii) implementing Monte Carlo algorithm based on random playing agents in order to choose a good position in case there is a tie on the scores of the heuristics which are defined by the filters.

1.1 Related Work

Müller *et al.*[4] claims that combinatorial game theory can be applied to computer Go, and develops a sum game model for heuristic Go programming and a program for perfect play in late stage endgames. We devise power-related approaches with respect to the influence functions introduced in [4]. We also adopt some of the ideas from the patterns discussed in [4].

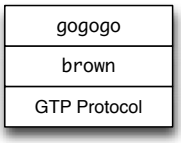
Kishimoto *et al.*[3] presents a practical solution to the GHI (Graph History Interaction) problem that combines and extends previous techniques. Making use of this algorithm will reduce the time cost for searching in Go program. However, considering the complexity of implementing this algorithm under our framework, it is not used in our implementation because the actual time cost in `gogogo` is reasonable and durable.

Gelly *et al.*[2] develops a Monte-Carlo Go program using UCT (Upper bound Confidence for Tree), and provides several patterns in the Go game. We adopt the idea of patterns and use some of the patterns in [2] as well as other patterns devised by ourselves. The use of patterns improves the performance dramatically.

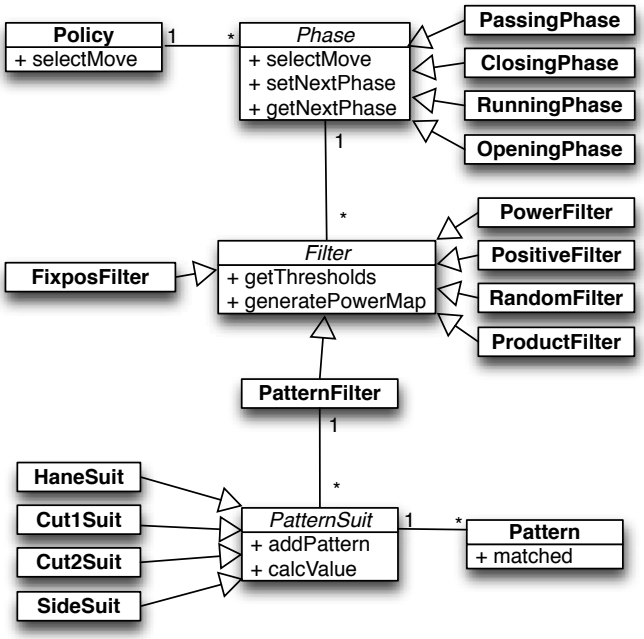
Cai *et al.*[1] surveys methods and some related works used in computer Go, and offers a basic overview for future study. They also present their attempts and simulation results in building a non-knowledge game engine, using a novel hybrid computation algorithm. We adopt the idea of phases and use it in our multi-phase framework.

2. System Architecture

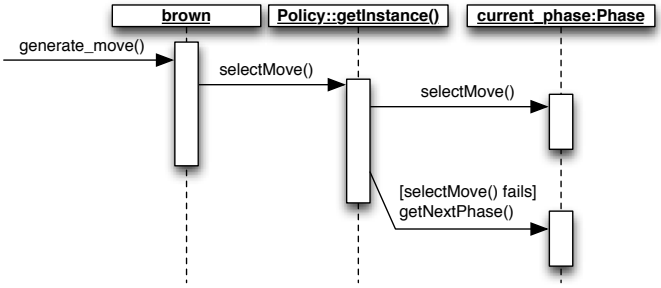
Figure 1 provides an overview of the `gogogo` system. In general, `gogogo` is built on top of `brown` for simplicity, as shown in Figure 1a so that the GTP protocol interface implementation can be reused. Figure 1b shows all the classes in `gogogo`. On generating a move, the function `generate_move` in `brown` will be invoked, which is modi-



(a) gogogo is built on top of brown



(b) Class diagram of gogogo



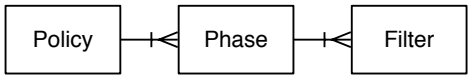
(c) Sequence diagram of move generation

Figure 1: System overview

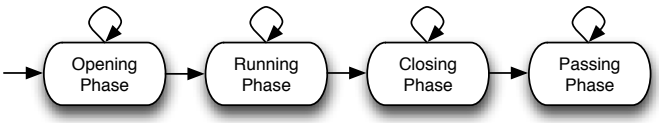
fied to invoke the `selectMove` method of the `Policy` singleton. Figure 1c shows the consequent method invocations for generating a move.

As described in Section 1, we make use of a multi-phase approach. Figure 2 depicts the relationships among the policy, phases and filters. The policy is a singleton and is the entrance of the whole algorithm. By design, a policy has one or more phases. In our implementation, there are 4 phases:

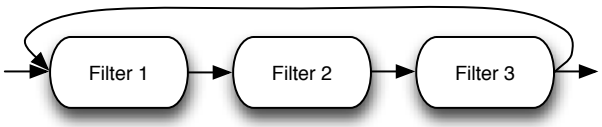
Opening phase is dedicated to occupying 9 special positions on the board.



(a) Entities relationship: A policy has one or more phases while a phase has one or more filters. In reality, the policy is a singleton in our implementation.



(b) Phases in a single policy: Arrows indicate possible transitions. The policy keeps track of the current phase, and falls through to the next phase only if none of the filters in the phase can generate a move.



(c) Filters in a single phase: Filters are matched one by one. The matching fails only if none of the filters matches. Once a filter matches and generates a move, the matching stops and the policy uses that move. In this case, the current phase is not changed.

Figure 2: Relationships among the policy, phases and filters

Running phase is the phase which tries to chase the enemy, making small eyes, and capturing stones.

Closing phase is responsible for the situation where the board is almost full. It will try to fill empty positions and to make small eyes.

Passing phase is a *sentinel* phase which always generates a pass move.

A phase consists of many filters. Each filter is responsible for matching a specific pattern or trying to occupy important positions according to power-based heuristic values. There are 6 kinds of filters:

`FixposFilter` is the filter which matches certain positions on the board (used for the opening phase).

`PowerFilter` is the filter which calculates *descending power values*. *Power* is a very important concept in our approach which is described in Section 3.1.

`PositiveFilter` works in the same way as `PowerFilter` except that it doesn't care enemy stones.

`RandomFilter` consists of two child filters. It chooses the first filter with probability p and the second filter with probability $1 - p$.

`ProductFilter` consists of two child filters and combines the effects of the two filters.

`PatternFilter` does pattern recognition and counts the number of patterns matched on each position as its power value. It is described in Section 3.2.

3. Algorithms and Implementation

Listing 1: `Policy::selectMove(moves, &pt)`

```
pair<int, int> pt;
while (!currentPhase->selectMove(moves, pt)) {
  currentPhase = currentPhase->getNextPhase();
}
return pt;
```

Listing 1 shows the skeleton of `selectMove` method of the policy, which tries to generate a move using the current phase, or falls through to the next phase if no move can be selected.

Listing 2: `Phase::selectMove(moves, &pt)`

```
for (int i = 0; i < filters.size(); i++) {
  Filter *f = filters[i];
  vector<pair<int, int>> fm = f->filter(moves);
  if (fm.empty()) continue;
  pt = randomChoose(bestMoves(fm));
  return true;
}
return false;
```

Listing 2 shows the process of `selectMove` method of a phase, which iterates over the filters one by one, trying to filter out a move. If the filter returns an empty list of moves, it goes to the next filter. Otherwise, it randomly chooses one of the best moves returned from the filter. The most important thing here is how to find the *best moves*. In section 3.3, we show how to evaluate moves according to the winning rate.

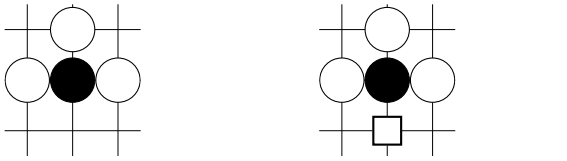
Listing 3: `Filter::filter(moves)`

```
vector<vector<double>> pm = generatePowerMap();
double mp = maxPower(pm, moves);
vector<pair<int, int>> filtered;
pair<double, double> t = getThresholds();
if (t.first <= mp && mp <= t.second)
  for (int k = 0; k < moves.size(); k++) {
    pair<int, int> p = moves[k];
    int i = p.first, j = p.second;
    if (pm[i][j] == mp) filtered.push_back(p);
  }
return filtered;
```

Listing 3 shows the general filtering process of a filter. First of all, the filter generates a power map which maps each position on the board to a real number, as the *power value*. Then the maximum power value of the moves is compared with the thresholds of the filter. If the maximum power value is in the range of the thresholds, the filter will return all the positions with the maximum power value. Otherwise, an empty list is returned.

3.1 Power Filter

`PowerFilter` is one of the most important filters in our system. It will spread out powers from one's own side stones and consider enemy stones which will weaken one's own side powers. The saying of `generatePowerMap` originates just from `PowerFilter`. Take a look at the following example in order to get a general idea of how `PowerFilter` works.



(a) Original board (b) Selected move (marked by square)

Figure 3: Partial boards for Example 1

Example 1. *Consider the board in Figure 3a. Suppose the* `PowerFilter` *is based on the following matrix*

$$\begin{bmatrix} 2 & 1 & 2 \\ 1 & 0 & 1 \\ 2 & 1 & 2 \end{bmatrix}$$

Then the power map for the white color is

$$\begin{bmatrix} -2 & -1 & -2 \\ -1 & 0 & -1 \\ -2 & -1 & -2 \end{bmatrix} + \begin{bmatrix} 1 & 0 & 1 \\ 2 & 1 & 2 \\ 0 & 0 & 0 \end{bmatrix} + \begin{bmatrix} 1 & 2 & 0 \\ 0 & 1 & 0 \\ 1 & 2 & 0 \end{bmatrix} + \begin{bmatrix} 0 & 2 & 1 \\ 0 & 1 & 0 \\ 0 & 2 & 1 \end{bmatrix}$$

which is eventually

$$\begin{bmatrix} 0 & 3 & 0 \\ 1 & 3 & 1 \\ -1 & 3 & -1 \end{bmatrix}$$

By choosing the maximum power, which is 3, the filter will generate a move at the position marked by the open square in Figure 3b.

3.1.1 Connecting Stones

This section contains the filters for connecting one's own side stones. Thresholds are omitted for brevity since they do not matter much in this case.

$$\begin{bmatrix} 1 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 1 \end{bmatrix} \tag{1}$$

$$\begin{bmatrix} 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 2 & 1 & 0 & 0 \\ 0 & 1 & 3 & 5 & 3 & 1 & 0 \\ 1 & 2 & 5 & 10 & 5 & 2 & 1 \\ 0 & 1 & 3 & 5 & 3 & 1 & 0 \\ 0 & 0 & 1 & 2 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \end{bmatrix} \tag{2}$$

3.1.2 Making Eyes

This section contains the filters for making eyes.

$$\begin{bmatrix} 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & -1 & 1 & 0 \\ 1 & -1 & 0 & -1 & 1 \\ 0 & 1 & -1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{bmatrix} \tag{3}$$

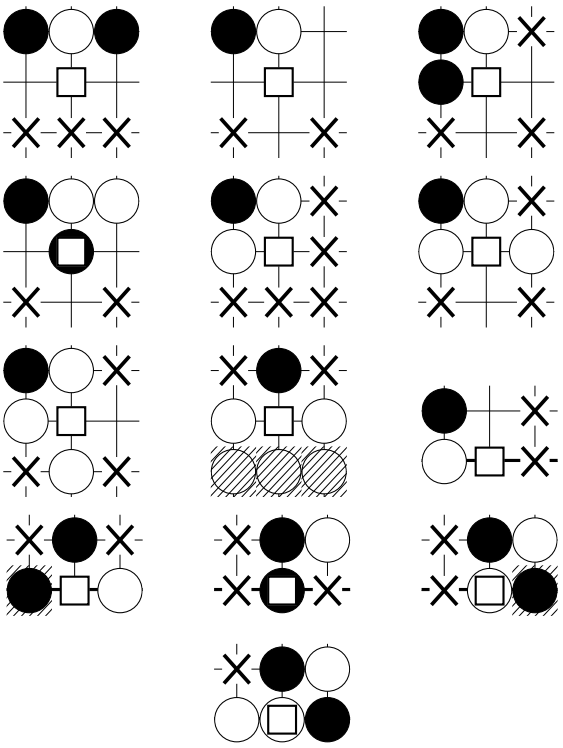


Figure 4: Patterns in gogogo

$$\begin{bmatrix} 0 & 1.5 & 1 & 1.5 & 0 \\ 1.5 & 1.8 & -1 & 1.8 & 1.5 \\ 1 & -1 & 0 & -1 & 1 \\ 1.5 & 1.8 & -1 & 1.8 & 1.5 \\ 0 & 1.5 & 1 & 1.5 & 0 \end{bmatrix} \tag{4}$$

3.2 Pattern Filter

Pattern recognition is used in pattern filters. We adopt the patterns introduced in [2] and consider each of them with 8 permutations. With local patterns, we can chase enemy, prevent the enemy from making eyes, compass enemy, capture enemy stones, etc. All the patterns are listed in Figure 4.

3.3 Monte Carlo

Monte Carlo is used when a filter returns multiple candidate moves. In that case, we use Monte Carlo to simulate 100 or more times for each move. Each simulation is just random playing until both sides generate pass moves. Finally, the candidate move with the largest winning rate is selected.

4. Evaluation

Three evaluation tests (see Table 1) are run among three Go programs: `gogogoX`, `gogogo` and `brown`. `gogogoX` is the version without pattern recognition, `gogogo` is the submitted version, and `brown` is the random playing version. Each test is run for 50 times, and statistics is given in Table 2.

Test	Black	White
1	`gogogoX`	`brown`
2	`gogogo`	`brown`
3	`gogogo`	`gogogoX`

Table 1: Evaluation tests

Test	Black wins	Black scores			
		avg	min	max	dev
1	94%	22.9(±2.9)	−23.5	82.5	20.5
2	100%	81.8(±4.2)	18.5	147.5	30.0
3	88%	17.8(±2.7)	−15.5	79.5	18.9

Table 2: Evaluation result

References

[1] X. Cai and D. Wunsch. Computer go: A grand challenge to ai. 2007.

[2] S. Gelly, Y. Wang, R. Munos, and O. Teytaud. Modification of uct with patterns in monte-carlo go. Technical report, 2006.

[3] A. Kishimoto and M. Müller. A general solution to the graph history interaction problem. AAAI, 2004.

[4] M. Müller. *Computer Go as a Sum of Local Games: An Application of Combinatorial Game Theory*. PhD thesis, ETH Zürich, 1995.