IOS-2-6

Suffix Tree Index Structure on Go Game Records

Shi-Jim Yen and Chen-Hsin Lee[1]
[1]Dept. of Computer Science and Information Engineering, National Dong Hwa University, Taiwan.

Jr-Chang Chen[2]
[2]Department of Computer and Communication Engineering, Ming Chuan University, Taipei, Taiwan.

Tai-Ning Yang[3]
[3]Department of Computer Science and Information Engineering, Chinese Culture University, Taiwan.

Shun-Chin Hsu[4]
[4]Department of Information Management, Chang Jung Christian University, Taiwan.

1. Introduction

In computer Go, game record database analysis is an important component. Many researchers try to apply matching learning technique based on game records. Researches regards game records as text strings by suitable encoding from each move to a character, and uses the approaches of natural language processes and statistics to acquire sequence patterns are proposed in Nakamura (1997, 1999, 2000). In Werfetal (2003, 2004), the authors proposed methods of learning to predict Life and Death and score final positions in the game of Go from game records by well designed data structure and good classifiers. Game records are used as the training data of neural networks (Konidaris and Nir Oren. 2002). A rule-based expert system could be build up by knowledge acquisition from game records (Kojima and Yoshikawa 1999). With regard to the programming techniques, a Go playing program may contain tactical look-ahead, pattern-matching, evaluation function and highly selective global search. This article focuses on pattern matching.

According to huge Go collection of records, the speedy and precise to judge next position during Go competition, and efficiency finding pattern in million databases is the big challenge of Computer Go programs. This is also a very important issue in Computer Go contest.

Go's search branching is wide and search depth is deeper. The standard board of Go is 19*19 which is much bigger than Chess (8*8) and Chinese Chess (9*10), furthermore, the moves of Go game is much more than other chess games. Therefore, Go game requires more search resource, such as operation speed and memory space, etc., For this problem, we provide a mechanism which apply the suffix tree algorithm to build an efficient data structure with indexing function for huge of Go records. This can speed up the search time for the pattern of Go, and show the position and occur times in Go game records, and furthermore provide the best next move for Go players.

2. Query Pattern and Suffix Tree Index Structure

Brute force algorithm is the simplest way to find the set of patterns matching on the board is to check for each board intersection and each pattern if it matches. However, it is not efficiently and it needs to take plenty of time to recognize pattern especially when Go game records are huge.

Obviously, Brute Force Algorithm is hopeless as search technique in Go; the search space is too large and the worst case complexity of this method is O(ds^2m) where s is the size of the board, d is the number of patterns and m the max number of elements by pattern. The details description can refer to (T.Urvoy, 2001). The Structure of Suffix Trees is robust and filters out impossible patterns which could instead of checking individually each pattern.

Contact: Shi-Jim Yen, Dept. of CSIE, National Dong Hwa Uni., Taiwan, Address: No. 1, Sec. 2, Da Hsueh Rd., Shoufeng, Hualien 97401, Taiwan. Tel: +886-3-8634031, FAX: +886-3-8634010, E-mail address: sjyen@mail.ndhu.edu.tw

The Go board has 3 states *Empty, Black, White*; in this paper, we use three number "*0*","*1*","*2*" to represent "*empty*", "*black*" and "*white*" on the board of Go. In a query pattern, use 3 bits to express 3 possible states of every point. Bit 1 means this point could be empty or not; bit 2 means this point could be black stone or not; bit 3 means this point could be white stone or not. (Yen, 1999). For example, a binary number "*101*" means this point could be "*White Stone*" or "*Empty* ", a number "111" express that point could be "*Empty*", "*Black Stone*" or "*White Stone*".

There are three possible query patterns in this article: 1. fixed size of patterns with fixed states in certain recognized area. 2. 5*5 Fixed size of patterns with unfixed states. 3. unfix size of patterns. The solutions are as in the follows.

2.1 5*5 fixed size of patterns with fixed states.

It is convenient to design the Go computer program by the fixed pattern $n*m$. For sure the exact patterns will be more precise and its contents will be more completely when n is bigger; however, the space of each pattern is much bigger as well, and knowledge database of all patterns will be extended reduplicated. In view of this, we take 5x5 as the base patterns size, because it contains the most of base patterns. For other bigger range patterns, it is allmost locate at border and the generality forms of pattern are rectangle (such as shape-size 5x10). And our technique could solve this problem as well. This idea is similar with Mark Boon's published paper (Mark Boon,1989),which mentioned 5x5 is large enough to cover more than 95% of all shapes.

Pre-process task:

Set up recognized area according to the position of last move on board for each Go record. Save each possible patterns $T_1 \dots T_{(\text{row}+5)*(\text{col}+5)}$ on board as an integer. (Hashing methods)

Searching Pattern Process:

Find the transform query pattern to an integer P.

Find the query pattern when $P = Tn$

2.2 5*5 fixed size of patterns with unfixed states

In other special sates of query patterns such as, some points must be our stone (usually means "Black Stone"), and some points must be opponent's stone (usually means "White stone") or some points must be "Empty"; sometimes, some points mustn't our stone, and some mustn't opponent 's stone or some must don't care point (it means it's ok for our stone or opponent ‘s stone) To begin with extracting Go records and reading row by row, then store into one-dimension array. We construct Suffix Tree base on Ukkonen’s algorithm.

Pre-process task:

1. Use 3 bits to representation those special sates and each board is transform to a string. we set up a recognize region 9*9 as a limit condition according to the position of last move. It can reduce search space, and speed up the search time. Sometimes, the recognize area will be extended depend on taken numbers of stones in rows and columns on board, when some stones were taken from board.
2. Build Suffix Tree for all Go records. We modify the suffix tree to fit the Go game program by using the first 5 character of each suffix string store in table, which define as “Super Nodes”, and store head position at index *i* in “Super nodes” as well. Then, use simple links to jump 19 points, which provide a mechanism for quickly traversing across sub-trees.
3. For Border-pattern, use an one-dimension array to denote the border. Through this mechanism, we could speedy recognize time for find the query patterns which locate at border.

Searching Pattern Process:

Find all possible matching patterns on the Suffix Tree and the Suffix table. Then use” Bitmapping” algorithm to verify pattern recognize correctness.

Step1: Read the matching pattern S_1 in super nodes.

Step2: Translate the query patterns P_i to all possible fixed size patterns.

Step3: Traverse the “super nodes” of the Suffix Tree and using the “bitmapping algorithm” to evaluate if match base on the query pattern P_i with allow state of each point on board.

Step4: Jump to next “supper nodes” by links.

Step5: Read all sub-strings to compare with query pattern P_i.

Step6: Use the “bitmapping algorithm” to evaluate if match or not.

Step7: Repeat Step4 to Step6 until all super nodes are searched.

Step8: Output results.

2.3 Unfixed size of patterns.

Search the query pattern on the Suffix tree row by row.

Pre-process task:

Build suffix trees for all Go records as in Case 2.

Searching Pattern Process:

Step1: Load query pattern *P*

Step2: Count total rows of *P* and store into variable ” *i*”.

Step3: Read query pattern row$_j$ (j++,$j \leq i$)

Step4: Traverse suffix Trees and find out all possible matching patterns with limit condition of allow states of patterns on board.

Step5: Store the position (*pos*) of first character for all possible match patterns on board into a table.

Step6: Let a formula to compute out a num for all possible matching patterns each row, and then determine if any number in each row is equal. (see table 3-13)

Formula: $num_i = pos$-19* k ($0 \leq k < i$)

Step8: Output Result.

3. Conclusion

According to huge Go collection of records, the speedy and precise to judge next position during Go competition, and efficiency finding pattern in million databases is the big challenge of Computer Go programs. For this problem, we provide a mechanism which apply the suffix tree algorithm to build an efficient data structure with indexing function for huge of Go records. Suffix trees are efficient access to all substrings of a Go string, and each of them can be constructed and represented in O(T) time and space, where T is the size of the board.

Moreover, this article address to three mechanisms that contain most situation of pattern matching on Go, such as varied pattern size, varied pattern states, query pattern on border, taken stones.

References

1. D.Gusfield. (1997) *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology.* Cambridge University Press.
2. George Konidaris Dylan Shell Nir Oren. (2002) ”Evolving Neural Networks for the Capture Game”, School of Computer Science Uni. of the Witw-atersrand, Johannesburg.
3. Giegerich, R., Kurtz, S. & Stoye. J. (2003) Efficient implementation of lazy suffix trees. *Software-Practice and Experience.* 33:1035-1049.
4. He, Y.J. (2005) An Efficient Multi-Feature Index Structure for Go Game Records. NDHU, Master Thesis.
5. Hsu, K. D. (2004) Pattern reorganization in Go game records. NDHU, Thesis for the degree of Master.
6. Kurtz S.(1999) Reducing the space requirement of suffix trees. *Software-Practice and Experience* 29 (13):1149-1171.
7. NAKAMURA, T. & KAJIYAMA,T. (1997) “Feature Extraction from Encoded Texts of Moves and Categorization of Game Records”, Department of AI, Kyushu Institute of Technology.
8. NAKAMURA,Teigo.(1999) “Acquisition of Move Sequence Patterns from Game Record Database Using n-gram Statistics”, Department of AI, Kyushu Institute of Technology.
9. T. Urvoy and gnugo team (2001) Pattern matching in go with DFA.
10. http://tanguy.urvoy.free.fr/Papers/dfabstract.pdf
11. Tata S., Richard, A., Jignesh, M. (2004) Practical Suffix Tree Construction. University of Michigan. *Proceeding’s of the 30th VLDB Conference*.
12. Takuya Kojima Atsushi Yoshikawa (1999) ”Knowledge Acquisition from Game Records”, NTT Communication Science Labs.
13. Teigo NAKAMURA, Takashi KAJIYAMA. (2000) ”Automatic Acquisition of Move Sequence Patterns from Encoded Strings of Go Moves”, Department of Artificial Intelligence, Kyushu Institute of Technology.
14. Weiner, P. (1973) Linear Pattern Matching Algorithms. *Proceedings of the 14th Annual Symposium on Switching and Automata Theory*.
15. Werf, E.C.D., Herik,,H.J.,& Uiterwijk , J.W.H.M. (2004): “Learning to Score Final Positions in the Game of Go”, Institute for Knowledge and Agent Technology, Department of Computer Science, Universiteit Maas-tricht.
16. Wikipedia. http://en.wikipedia.org/wiki/Suffix_tree
17. Yen, S. J. (1999): *Design and Implementation of a Computer GO Program JIMMY.* Ph.d. Thesis, National Taiwan University, Taiwan. (in Chinese)