COMPUTER GENERATION OF CHINESE COMMENTARY ON OTHELLO GAMES*

Jen-Wen Liao and Jyun-Sheng Chang+

Department of Computer Science
National Tsing Hua University
Hsinchu, Taiwan 30043

Abstract

This paper describes the design and implementation of a text generation system which produces a mutil-sentence commentary in Chinese on a kind of board game call Othello. The system is built under a general framework of text generation. Discourse-related linguistic knowledge is encoded as rules and a production system is used to manipulate these rules to generate cohesive text. A previously constructed sentence generator based on systemic grammar is adopted to generate the final surface text.

1. Introduction

Natural language processing (*NLP*) is a main branch of *artificial intelligence*. There are many applications of NLP such as: *Machine Translation*, *Information Retrieval*, *Human-machine Interaction*, *Text Generation*, etc.

We choose a kind of board game called *Othello* as the application domain of text generation. We have implemented a text generation system creating commentary on Othello in **Chinese**. Despite the apparent simplicity of the task, the possibilities of producing text are rich and diverse. The commentary is intended to convey both the moves of the game and the significance of each move by showing errors and missed opportunities.

* This research is partially supported by National Science Council, grant No. NCS79-0408-E007-05 and NSC80-0408-E007-13.
+ To whom corresoondances should be addressed.

In next section, we present the description of Othello. In section 3, we introduce the general concepts of text generation. An overview of our system is also presented. In section 4, we introduce the first module of the system. In section 5, we give a detailed description about the second module of the system. In the last section, we show some examples generated by our system and summarize the paper.

2 Description of the game - Othello

Othello is a derivative of the *Go* family of board games; emphasizing capture of territory through the process of surrounding the opponent's pieces. It is played on an 8 x 8 board, with a set of dual-colored *discs* by two persons. Each disc is **Black** on one side and **White** on the other. Each color stands for one of the two players. The initial board configuration is shown in Figure 1.

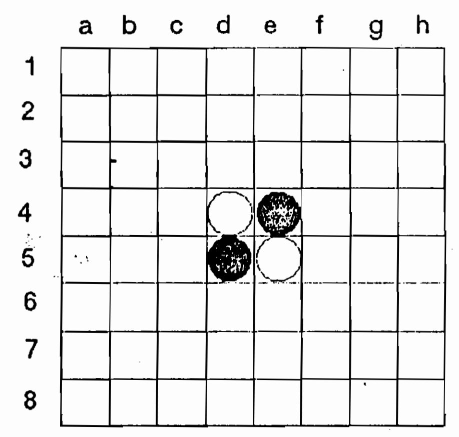

Figure 1. the initial Othello board configuration;

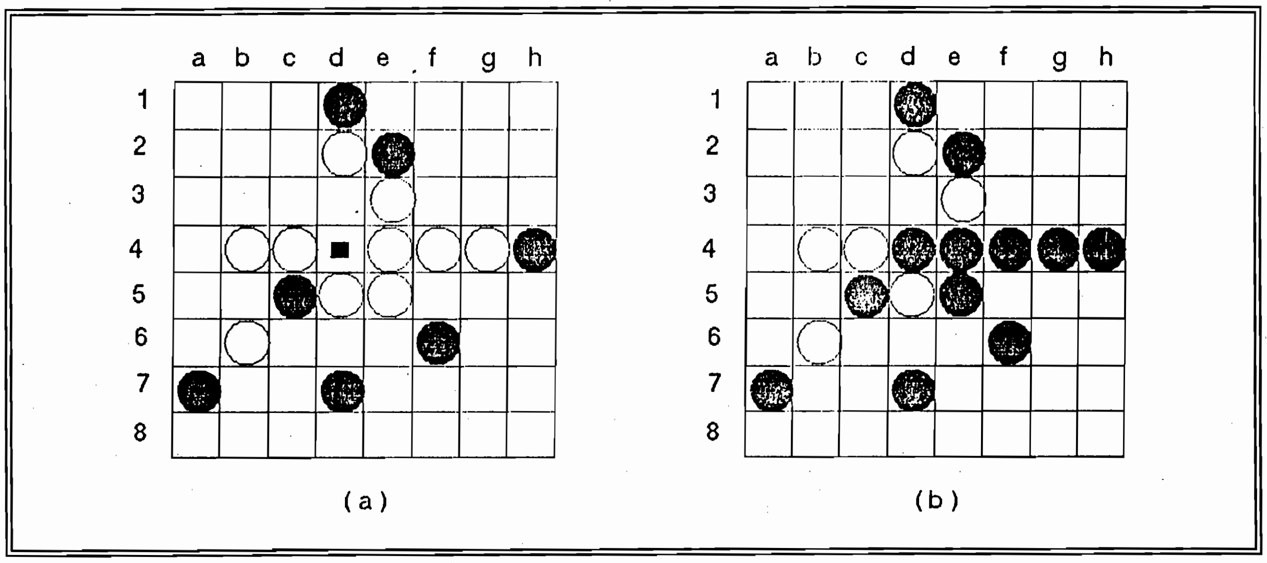

Figure 2. (a) Result of a legal play before BLACK's move at d4;

(b) after BLACK's play at d4.

Initially, WHITE owns the two central squares on the *major diagonal* (d4 and e5), and BLACK owns the two central squares on the *minor diagonal* (e4 and d5). BLACK plays first, and then the players take turns to play a move until neither side has a *legal* move. The player who has more disks at this point wins the game.

A move is made by placing a disc on the board, with the player's color facing up. In order for a move to be *legal*, the square must be empty prior to the move and it must result in *capturing* of the opponent's discs. The opponent's discs are captured by *bracketing* them between the disc being played and an existing disc belonging to the player. Bracketing can occur in a straight line in any of the eight directions(two vertical, two horizontal, and two in each diagonal direction). The captured disc (*discovered* discs) are flipped to the other color and become the opponent's. Bracketing and discovered discs does not further cause more discs to be flipped even if they result in a new bracket. Figure 2(a) and Figure 2(b) show an example of a legal move by BLACK on square d4, and the resulting change in disc ownership.

A number of simple strategies, such as maximizing the disc differential, have been suggested for Othello. For more details, see [Rosenbloom 1982] and [Liu, Huarng and Hsu 1987].

3 Text Generation

3.1 Introduction

Text generation is the reverse of natural language understanding. A generation system accepts a representation of linguistic information and goals and produces a sequence of sentences that realize the goals and convey the given information. The problem of text generation can be divided into two main areas which are not wholly independent: *planning the content of what to say* and *deciding how to say it*.

3.2 An Overview of Our System

Our system generates Chinese commentary on *Othello* games. After the user or the computer makes a move on the board, the system automatically produces a paragraph of text, analyzing the tactical pattern of the move, estimating the goodness of the move, comparing the move and the best move that the system knows of and showing the current board situation. The information of each step is stored in a list called *game history list* for later use. When the game is completed, this system displays a commentary on the whole game, including the mistakes the user or the computer has made, the information of the ownership of discs, and the winner. Our system consists of two principle modules (see Figure 3):

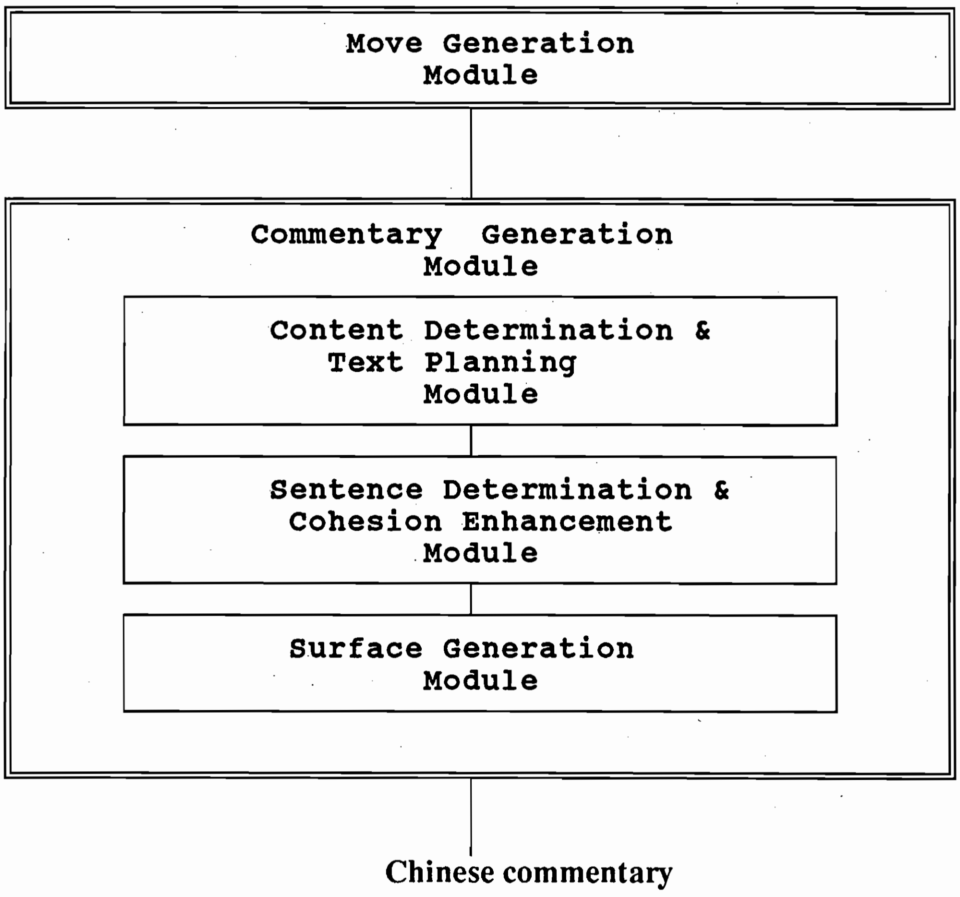
Move Generation Module
Commentary Generation Module
Content Determination & Text Planning Module
Sentence Determination & Cohesion Enhancement Module
Surface Generation Module
Chinese commentary

Figure 3. The text generation system

1. *Move Generation Module* which employs a minmax procedure and an alpha-beta cutoff procedure to select a move for the computer side or takes user's input as the move for the human side at each step. This module returns the *game history list* as output for the purpose of generating commentary on the whole game.
2. *Commentary Generation Module* which uses the methods described in subsequent sections to construct the commentary for one step or for the entire game. It can be further divided into three smaller modules:

   <A> *Content Determination & Text Planning Module* which determines what kind of information should be included in the output text.

   <B> *Sentence Determination & Cohesion Enhancement Module* which organizes the unordered, simple commentary into ordered, complex commentary.

<C> *Surface Generation Module* which transforms these commentary into Chinese sentences.

4. Move Generation Module

The purpose of *Move Generation Module* is to produce a legal move for the player (the computer side or the human side) and other relevant information such as game trees, the value of current evaluation function, etc. The move for the human side is taken from the input typed by the user and the move for the computer side is calculated by searching the game tree using a *minmax* procedure and an *alpha-beta cutoff* procedure. After each move of the game is generated, it is recorded in a so-called *game history list* for generating commentary on the entire game.

Usually the move returned by this module is not the best one since the depth of tree searching is limited. In order to generate versatile and meaningful commentary, the commentary generation module search at least one more level down the game tree to collect more knowledgeable information.

5. Commentary Generation Module

5.1 Content Determination and Text Planning Module

The first phase of text generation is to choose the information that is relevant and appropriate for inclusion in the output text. The decisions must be based on (1) what key objects involved that the system thinks is necessary to identify and (2) what aspects of situations are important and must be described. Also generation goals should be taken into account.

Text planning is the second phase in text generation. This phase accepts an unordered set of propositions generated by the content determination phase as input and organizes them into a series of ordered messages. The resulting messages must be appropriate (1) for the purpose of sentence generation and (2) to the intended audience. A knowledge base is also used to guide this process.

One of the problems that will occur if the content determination phase and text planning phase are totally separated is that some verbose or redundant message might be generated after the first phase such that the subsequent phase must waste time to remove them. This problem can be efficiently solved by integrating these two phases into one module as in our implementation. Besides, our system has a separate discourse module with explicit discourse structure which guide the content determination phase to generate propositions in a top-down, goal-directed fashion.

In our system, the input to this module includes the game tree, current board configuration, etc. We adopt the ATN (*Augmented Transition Networks*) framework to analyze these data and extract relevant information. The information generated from this phase is represented in frames like internal representation.

Transition Network formalism is commonly employed in NLP. A transition network consists of a network name, a set of nodes (states) and arcs. We express the transition network in a LISP-like *list* notation. The detailed description of representing ATN in list notation can be found in Charniak (1983). The registers in our notation is defined by beginning their names with character **?**. To define a network, we define a LISP function **def-net**:

(**def-net** *net-name* (*registers*) *commands*)

See Figure 4 for the commands for writing transition networks and Figure 5 for an example of network Generate-Partial-Comment in list notation.

We represent the internal structure of a sentence using a modified *slot-and-filler* representation, which reflects more the functional role of phrases in a sentence. This kind of internal representations is called *propositions*. In this notation, we assign a special semantic role (such as **Agent**, **Theme**, or **Focus**) to each major grouping of words. It identifies the properties of each semantic role as a set of *values.* A *slot name* distinguishes each value and indicates the role that value plays in the structure. Each slot statement is made up of the following components:

Commands	*Explanations*
(**traverse** (*network-name registers*))	Traverse the new network *network-name* with the values of *registers* are passed to the *network-name*.
(**seq** *actions*)	Perform the *actions* sequentially.
(**if** *test thenpart* **else** *elsepart*)	Check if *test* comes out true. If so, execute the commands of *thenpart*. Otherwise execute commands of *elsepart.*
(**either** *test1 action1 test2 action*2..)	If *test1* is true, execute commands of *action1*. If the test fails, try *test2* , *test3,* ... until one of them succeeds.
(**:=** *register value*)	Set the value of *register* to the value returned by the evaluation of *value.*
(**:=** (*registers*) *value-list*)	Set the value of *registers* according to the pattern returned by the evaluation of *value-list*.
(**jump** *tag*)	Jump to a *tag* of current network.
(**tag** *tag-name*)	Define a tag *tag-name*.
(**gen-comment** *comment*)	Generate *comment.*
=, <, >, and, or, dolist, first ...	some Lisp built-in functions

Figure 4.

--- a *slot name* (*operator*) indicating the type of the structure.

--- the *type* of the object.

--- the *modifiers* of the object, which may be a list of semantic role structures.

```
(def-net Generate-Partial-Comment (?Agent ?Opponent ?Move ?BestMove ?LastMove
                                    ?DiskFlipped ?GameTree ?Ev)
 (seq
  (traverse (Decide-Choices ?Agent ?GameTree ?Number-Of-Choices))
  (if    (= ?Number-Of-Choices 0)
         (jump NEXT)
   else (traverse (Show-Move ?Agent ?Opponent ?Move ?LastMove ?DiskFlipped)))
  (if    (= ?Number-Of-Choices 1)
         (jump NEXT)
   else (traverse (Measure-Goodness-Of-This-Move ?Agent ?Move ?BestMove
                              ?GameTree ?Other-Choices-Are-Equal ?MoveIsBest) ) )
  (if    (or ?Other-Choices-Are-Equal ?MoveIsBest)
         (jump NEXT)
   else (seq (traverse (Show-Best-Move ?BestMove))
              (traverse (Compare-BestMove-ThisMove ?Agent ?Move ?BestMove)))
 (tag NEXT)
  (traverse (Consequence-Of-This-Move ?Agent ?Opponent ?Move))
  (traverse (Summary ?Agent ?Opponent ?Ev))
 )
)
```

Figure 5.

1. The operators for simple NP will be used to indicate the determiner information. The possible combination for the operator of simple NPs are: DEF (for definite reference), INDEF (for indefinite reference), PRO (identifies an NP consisting of a pronoun), NAME (identifies an NP consisting of a proper name) and CONJ(represents two or more NPs connected by conjunction).

   The type for simple NPs will be as expected, and the modifiers will consists of adjectives, relative clauses, number information and so on.

2. The operators for clause structures include: PRES (for simple present tense), PAST (for simple past tense) and INF (for infinitive clause).

   The modifiers of a clause consist of: AGENT (the object that caused the event to happen), THEME (the thing that was acted upon), AFF-OBJ (the object that was affected by the event) and FOCUS (the focus of attention of the event).

3. The operators for simple sentences are simply ASSERT (for declarative sentence). As for compound sentences, the operators are the semantic relations between the clauses such as ALTHOUGH, CONJUNCTION, RESULT, etc.

   For example, the simple sentence "I have 3 choices" might be represented by the structure:

```
(ASSERT (PRES HAVE    (AGENT (PRO I))
                      (THEME (INDEF CHOICE (NUMBER 3)))
                      (AFF-OBJ nil)
                      (FOCUS (PRO I))))
```

Using the technique of transition networks, we can generate the internal representation for a versatile and comprehensive game commentary. During a network traversal, the command **gen-comment** creates a proposition and add it to a output list to be processed by the next module. If a register appears in the pattern of proposition, we must replace the register by its value at that position. To generate commentary for each step, we traverse the network *Generate-Partial-Comment*. When we traverse the network *Generate-Partial-Comment*, we also will traverse the subnetwork *Summary* (see Figure 6). Note that the symbol **$** means to evaluate the operation immediately following it and use the return value to fill the slot occupied by **$**.

```
(def-net Summary (?Agent ?Opponent ?Ev)
 (seq
    (:= ?AN (Count-disc-number '?Agent))
    (:= ?ON (Count-disc-number '?Opponent))
    (Gen-Comment '(assert (pres REMAIN (agent (pro ?Agent))
                                 (theme (indef DISK (number ?AN))) (aff-obj nil)
                                 (focus (pro ?Agent)))))
    (Gen-Comment '(assert (pres REMAIN (agent (pro ?Opponent))
                                 (theme (indef DISK (number ?ON)))) (aff-obj nil)
                                 (focus(pro ?Opponent)))
    (either (= ?AN ?ON)
            (Gen-Comment '(assert (pres TIE (agent (pro WE))
                                 (theme nil) (aff-obj nil) (focus (pro WE))))
       otherwise
            (Gen-Comment '(assert (pres $(if (> ?AN ?ON) 'LEAD 'LAG)
                                 (agent (pro ?Agent))
                                 (theme (indef DISK (number $(- ?AN ?ON))))
                                 (aff-obj (pro ?Opponent))
                                 (focus (pro ?Agent)))))
    (either (= ?Ev 0)
            (Gen-Comment '(assert (pres HAVE (agent (pro NOBODY))
                                 (theme (indef ADVANTAGE)) (aff-obj nil)
                                 (focus (pro NOBODY))))
       otherwise
            (Gen-Comment '(assert (pres HAVE
                                 (agent (pro $(if (> ?Ev 0) 'I 'You) ))
                                 (theme (indef ADVANTAGE) (aff-obj nil)
                                 (focus (pro $(if (> ?Ev 0) 'I 'You)))))
 )
)
```

Figure 6.

Suppose that the *?Agent* (the player who takes turn now) is 'You' (stands for the human side) and the *?Opponent* is 'I' (stands for the computer side). The current value of evaluation function (*?Ev*) is +5. After calculating the disc numbers on the board, we found that the human side owns 20 discs and the computer side owns 16 discs. The register *?AN* is set to 20 and the register *?ON* to 16. Therefore we will obtain the following propositions:

```
(assert (pres REMAIN    (agent (pro You))      (theme (indef DISK (number 20)))
                        (aff-obj nil)          (focus (pro You))))
(assert (pres REMAIN    (agent (pro I))        (theme (indef DISK (number 16)))
                        (aff-obj nil)          (focus (pro You))))
(assert (pres LEAD      (agent (pro You))      (theme (indef DISK (number 4)))
                        (aff-obj (pro I))      (focus (pro You))))
(assert (pres HAVE      (agent (pro I))        (theme (indef ADVANTAGE))
                        (aff-obj nil)          (focus (pro I))))
```

The corresponding commentary in Chinese is:

我 有 三 種 選 擇
你 剩 下 2 0 顆 棋 子
我 剩 下 1 6 顆 棋 子
你 領 先 我 4 顆 棋 子
我 佔 有 優 勢

5.2 Sentence Determination and Cohesion Enhancement Module

5.2.1 Introduction

After the content and overall ordered of the message in the text have been decided, there are still two problems that need to be resolved:

1. The first problem is deciding *how much information to put in a sentence*. For a set of propositions, it can be realized as a complex sentence or a few simple sentences. The factors influencing this decision include focus of attention [Derr and McKeown 1984], semantic and rhetorical relations between propositions [Davey 1979].
2. The second problem has to do with *cohesion in the text generated.* It is necessary in this phase to find and mark the cohesion links among propositions. These markings are subsequently used in surface generation for such activities as pronominal, demonstrative, verbal and clausal substitutions, ellipsis, selection of conjunction and lexical choice.

Some pragmatic knowledge is used to solve these two problems above. Under the consideration of easy understanding and modification, we have selected *rules* to represent the knowledge required for this module. Rule-based knowledge representation centers on the use of **IF** *condition* **THEN** *action* statements. Our rules are expressed with arrows (--->) to indicate the **IF** and **THEN** portions. For example,

the PH of the spill is less than 6 (**IF** part)

--->

the spill material is an acid. (**THEN** part)

5.2.2 Pattern Matching

To support rule-based approach described in the previous section, we need the technique of *pattern matching*. To enhance the capability of pattern matching scheme, we allow patterns to contain *pattern matching variables* beginning their names with the character ?. We also allow the user to attach an arbitrary *predicate* as a property of a pattern matching variable. Then when the matcher tries to match a variable against an item, it allows the match only if the item satisfies the attached predicate, if one exists. The pattern matching variable with a predicate is expressed as a list, in which the first element is the variable name and the second element is an arbitrary LISP predicate. For example, the expression

```
(age-of ?person ?(age (and (numberp ?age) (< ?age 20))))
```

would match

```
(age-of John 17)
```

with the result that ?person is bound to John and ?age is bound to 17.

It might be necessary to create a new pattern after we apply our pattern matcher to several patterns and get the variable bindings of a successful match. The pattern matching variables of the new pattern are required to be substituted with their corresponding value of bindings. In addition to replace variables, we might need to execute some other actions during the substitution to extend the new pattern such as:

!: Triggering another pattern matching process. The pattern following "!" is matched against a rule named by the first argument of the pattern. If it succeeds, the new pattern generated by that rule is used to fill the slot where "!" occupies. For example, consider the following rule Decide-Opening,

```
( (3 3) (3 4) ) ---> Parallel
```

the pattern before applying the rule: (The Opening is !(Decide-Opening (3 3) (3 4))) and the pattern after applying the rule: (The Opening is Parallel)

$: Evaluating the subsequent function and filling the slot with the return value. If a pattern matching variable is contained in the function, replace it with its binding value first and then evaluate that function. For example, if the bindings obtained after matching X, the **IF** part of a rule is ((**?A 3**) (**?B 5**)),

X ---> (The two variables are $(if (= ?A ?B) 'equal 'unequal))

the output pattern generated by the rule will be (The two variables are unequal)

@: Splitting the resultant pattern following it. It decomposes a pattern into a series of items. For example,

(a @(b c d) e) becomes (a b c d e)

5.2.3 Examples

The knowledge base in this module consists of two kinds of rules: one is called the *discourse-rule* and the other the *pattern-rule*. The former describes functional, semantic or rhetorical relation among propositions and the latter expresses facts in the knowledge base. The **IF** part of these rules is a list of one or more patterns. The main difference between them is that the **THEN** part of the discourse-rule has several patterns while the pattern-rule has only one. The pattern-rules can be applied in discourse-rules to check if a certain situation has happened, but the discourse-rules cannot be used in pattern-rules.

In this module, we encode *discourse-rules* to combine propositions. A discourse-rule consists of three components: *rule-name* , *rule-body* and *execution-priority*. The rule-name defines the unique name of a rule. The rule-body has an **IF** part and a **THEN** part. If a portion of the propositions matches the **IF** part of a rule-body, then it produces a new output proposition with every variables in the proposition replaced with its binding value. The execution-priority of rules are used to resolve the conflict where the **IF** parts of more than one rules matches a portion of propositions. When it happens, the rule with highest execution-priority fires. The output proposition of a rule can be tested for further combination with other propositions. We repeat this process until there is no possibility of combining propositions. The function **def-discourse-rule** is used to declare a discourse-rule.

A pattern-rule has two parts: *rule-name* and *rule-body*. The **IF** part of the rule-body consists of several patterns enclosed in a list and the **THEN** part specifies the new pattern to be generated if **IF** part of the rule matches the input patterns. The function **def-pattern-rule** is used to declare a pattern-rule.

Consider a discourse-rule <Although-but>. Suppose that a speaker makes an utterance that generates some expectation. But the expectation is contrary to the next utterance. Under such situation, combining these two utterances into a single complex sentence using conjunctions is better than producing two separate sentences since the former reflects the semantic relation while the later do not. To check if there is a certain relation among two sentences, the rule <Although-but> (See Figure 6) applies a pattern-rule Not-Expect:

There are other aspects of cohesion affecting the quality of output text. For instance, see input propositions below. Realizing these two propositions into two independent sentences (sequence 1) obviously is not a good idea since there are many redundant information. Instead we should join the two propositions by deleting the **PREDICATE, THEME** and **AFF-OBJ** of the proposition 2 and using conjunction to combine the two **AGENTs** (sequence 2). A discourse-rule named identity-delete-1 can be applied in such a condition by matching the values of **PREDICATE, THEME, AFF-OBJ** and **FOCUS** of one proposition with the corresponding arguments in the second proposition.

```
1. You lead me by 4 disks.
   I have advantage.

2. Although you lead me by 4 disks, I have the advantage.

(def-discourse-rule <Although-but>     7
 ( (assert (?T1 ?P1 ?A1 ?T1 ?O1 ?F1))
   (assert (?T2 ?P2 ?A2 ?T2 ?(O2
  !(Not-Expect (?P1 ?A1 ?T1 ?O1) (?P2 ?A2 ?T2 ?O2))) ?F2))
--->
 ((Although (assert (?T1 ?P1 ?A1 ?T1 ?O1 ?F1))(assert (?T2 ?P2 ?A2 ?T2 ?O2 ?F2))))
)

(def-pattern-rule Not-Expect
 ((LEAD ?P1 ?- ?P2) (HAVE ?P2 ?(Theme (= (Root ?Theme) 'ADVANTAGE)) ?- ))
---> True
)

input propositions:
    (assert (pres LEAD    (agent (pro You))      (theme (indef DISK (number 4)))
                          (aff-obj (pro I))      (focus (pro You))))
    (assert (pres HAVE    (agent (pro I))        (theme (indef ADVANTAGE))
                          (aff-obj nil)          (focus (pro I))))

output proposition:
    (Although  (assert (pres LEAD (agent (pro You))  (theme (indef DISK (number 4)))
                                  (aff-obj (pro I))  (focus (pro You))))
               (assert (pres HAVE (agent (pro I))    (theme (indef ADVANTAGE))
                                  (aff-obj nil)      (focus (pro I)))) )
```

5.2.4 Discussion

Our discourse rules are inspired by the research of Derr and McKeown (1984) who concentrated on using focus to generate complex and simple sentences. They encode tests on functional information within *Definite Clause Grammar (DCG)* formalism to determine when to use complex sentences. These tests look like our discourse rules except that they are written in Prolog and the pattern matching mechanism is implicitly embedded in the grammar. Our discourse rules have an extra component '*execution-priority*' to resolve conflicts while the tests have not. Checking semantic relation between propositions to generate complex sentences are also not included in their paper.

```
1. You have 5 remaining disks.
   I have 5 remaining disks.

2. You and I both have 5 reamining disks.

(def-discourse-rule identity-delete-1 1
  ( (assert (?T ?P (agent ?A1) ?T ?O ?F1))
    (assert (?T ?P (agent ?A2) ?T ?O ?F2)) )
--->
  ( (assert (?T ?P (agent (conj ?A1 ?A2)) ?T ?O ?F1)) )
)

input propositions:
      (assert (pres REMAIN(agent (pro You))        (theme (indef DISK (number 5)))
                          (aff-obj nil)            (focus (pro You)))
      (assert (pres REMAIN(agent (pro I))          (theme (indef DISK (number 5)))
                          (aff-obj nil)            (focus (pro I)))

output proposition:
      (assert (pres REMAIN       (agent (conj (pro You) (pro I)))
                                 (theme (indef DISK (number 5)))
                                 (aff-obj nil)
                                 (focus (pro You)))
```

The process of selecting and combining propositions in this module is similar to the *hill climbing* algorithm in Mann and Moore (1981). They designed so-called *aggregation rules* to combine sentences and then evaluate the resultant combinations. The best one is selected and the process is repeated. The difference between our method and theirs is that we always choose the best combination without generating many other alternatives. Another disadvantage of the aggregation rules is that they are language-dependent. Our discourse-rules are less dependent on language than theirs.

5.3 Surface Generation Module

The last module in the system is surface generation which takes the segmented and enhanced messages produced by last module as input and generates natural language sentences as output. It has to make such decisions as (1) what words and phrases to use in describing or referring to entities and their relationship, and (2) what syntactic structure to use in presenting these pieces of information.

Several grammatical formalisms have been used for surface generation: *Systemic grammar* [Halliday 1976], *Transformational grammar* [Chomsky 1965], *Augmented Transition Network (ATN) grammar* [Woods 1970], *Functional grammar.* Our surface generator use systemic grammar to produce Chinese sentences.

Our system employs the sentence generator implemented by Kuo (1989) as a part of the surface generation module for the production of versatile Chinese sentences. The form of input to the sentence generator looks like frames and has three parts:

1. *a frame name* --- specifies the constituent of a sentence that the systemic network is to generate.
2. *a list of features* --- provides the information about the functions that this constituent is intended to perform.
3. *an optional subframe list* --- gives the subconstituents that are to be handled by the lower level network. The subframes have exactly the same structure the we have described.

For example, to generate a sentence "<u>*You lead me by 4 disks*</u>" in Chinese, the input will be as follows:

```
(sentence (s-sentence)
    (clause (independent mood indicative transitivity transitive active double-obj)
            (agent (np head-noun pronoun (head-noun You)))
            (pred  (vp (verb lead)))
            (aff   (np head-noun pronoun (head-noun I)))
            (patient (np head-noun noun-mod class-phr (head-noun disk))
                        (classp (cp number (num 4) (class kir))))))
```

To prepare the input for the sentence generator, the propositions that we have produced need to be transformed into the form described in the last section. There are many decisions that we must make during this transformation such as the what features to include, how to fill the subframes, and which word to use. Instead of writing a subroutine to transform the propositions, we formalize the transformation by encoding these linguistic choices as pattern-rules. By matching the input propositions and the **IF** part of pattern-rules, the necessary form for the sentence generator can be obtained from the new pattern part of pattern-rules.

Consider the following pattern-rule for an example. The rule Transform-Clause transforms the clause element of a proposition into a subframe of the resultant input form according to the patterns specified in the **IF** part and the **THEN** part.

```
(def-pattern-rule Transform-Clause
  ( ?Type ?Pred (agent ?Agent) (theme ?Theme) (aff-obj ?Aff-Obj) (focus ?Focus))
 --->
  ((clause !(Determine-Clause-type (?Agent))
     mood !(Determine-Mood (?Agent))
     transitivity @!(Determine-Transitivity (?Agent ?Theme ?Aff-Obj ?Focus))
   )
     $(if '?Agent       '(subj @!(Transform-NP ?Agent)))
     $(if '?Pred        '(pred !(Transform-VP ?Pred ?Type ?Agent ?Theme)))
     $(if '?Theme       '(patient @!(Transform-NP ?Theme)))
     $(if '?Aff-Obj     '(aff @!(Transform-NP ?Aff-Obj))) )
)
```

A word in propositions generated by content determination and text planning module denotes only a concept and does not necessarily correspond to the actual word to appear in the resultant sentence. But the word system in the sentence generator only consists of locative particle subsystem. We can either extend the word system in the sentence generator, or handle the problem of lexical choice during the transformation process described in the last section. We choose to implemented the lexical choice mechanism in the transformation phase for the following reasons: First of all, these choices can be easily formulated and encoded as rules without making the grammar too complicated. Secondly, these rules could be gracefully incorporated with other natural language formalism such as semantic networks to enhance its capability.

For example, the concept "HAVE" might have the following alternatives.

HAVE CHANCE	corresponds to	GET CHANCE
HAVE ADVANTAGE	corresponds to	GAIN ADVANTAGE

Through the help of pattern-rules, we are able to implement some simple word choices. The choices that we have made primarily concentrate on verbs. The rule accepts a verb and a direct object of the verb as inputs and generates a new word which is then used to replace the original verb in the input for the sentence generator. The pattern-rule Generate-Word for verb "HAVE" is described below.

```
(def-pattern-rule Generate-Word
  ( HAVE CHANCE )
---> GET )
(def-pattern-rule Generate-Word
  ( HAVE ADVANTAGE )
---> GAIN)
```

5.4 Examples of Text Generated by the System

The following two paragraphs are the commentary produced by our system. The first paragraph is the comment on a certain step during the game and the second is the commentary on the entire game.

Commentary on a certain step during the game

你 擁 有 10 種 選 擇 .
你 下 "(5,8)" 使 得 2 顆 棋 子 被 你 吃 掉 了 .

最 好 的 一 步 棋 是 "(2,6)" .
"(5,8)" 是 危 險 的 位 置 但 "(2,6)" 不 是 .
雖 然 你 可 能 會 吃 掉 右 邊 的 1 顆 棋 子 但 是 我 可 以 阻 止
你 的 攻 勢 .
你 獲 得 佔 領 左 邊 的 機 會 .
你 剩 下 1 1 顆 棋 子 , 我 剩 1 2 顆 棋 子 .
雖 然 你 落 後 我 1 顆 棋 子 但 是 你 佔 有 優 勢 .

Commentary on the entire game

你 開 始 這 一 盤 遊 戲 並 且 下 了 "(3,5)".
我 緊 接 著 下 了 "(3,4)".
這 種 開 局 法 稱 為 平 行 式 開 局 法
在 第 14 手.
我 阻 止 了 你 的 攻 擊.
在 第 16 手.
我 錯 下 一 步 棋 使 得 你 佔 領 了 下 邊 的 有 利 位 置.
在 第 19 手.
你 採 取 了 錯 誤 的 策 略 使 得 我 吃 掉 了 右 邊 的 1 顆 棋 子.
在 第 22 手.
我 下 一 步 錯 誤 的 的 棋 使 得 你 吃 掉 了 右 邊 的 3 顆 棋 子.
在 第 28 手.
我 沒 有 採 取 正 確 的 策 略 使 得 你 佔 領 了 下 邊 的 有 利 位 置.
在 第 34 手.
我 採 取 了 錯 誤 的 策 略 使 得 你 吃 掉 了 上 邊 的 1 顆 棋 子.
在 第 40 手.
我 下 一 步 錯 誤 的 的 棋 使 得 你 吃 掉 了 左 邊 的 3 顆 棋 子.
在 第 43 手.
你 佔 領 了 右 上 角.
在 第 46 手.
我 佔 領 了 右 下 角.
在 第 50 手.
我 錯 下 一 步 棋 使 得 你 佔 領 了 左 邊 的 有 利 位 置.
在 第 52 手.
我 採 取 了 錯 誤 的 策 略 使 得 你 吃 掉 了 左 邊 的 1 顆 棋 子.
在 第 53 手.
你 佔 領 了 左 下 角.
在 第 55 手.
你 沒 有 採 取 正 確 的 策 略 使 得 我 佔 領 了 下 邊 的 有 利 位 置.

```
在 第 61 手.
你 佔 領 了 左 上 角.
最 後 的 結 果 是.
我 剩 下 23 顆 棋 子 ,  你 剩 下 41 顆 棋 子.
你 贏 得 這 一 盤 遊 戲.
本 程 式 謝 謝 你 的 使 用.
```

6 Conclusion

In this thesis, we have designed and implemented a text generation system for generating commentary on Othello games in Chinese. A framework for text generation is proposed and adopted in our system.

Our system is a complete text generation system that employes three independent phases to produce Chinese text. We hope that our work will give rise to other researches on the topic of text generation in Chinese.

References

[Charniak 1983]

E. Charniak, "A parser with something for everyone", in *Parsing Natural Language,* ed. M. King, Academic Press, London, 1983.

[Davey 1978]

A. Davey, *Discourse Production*,Edingburgh University Press,Edingburgh,1978.

[Derr and Mckeown 1984]

M.A. Derr and K.R. McKeown, "Using Focus to Generate Complex and Simple Sentences", *Proceedings of the 21st Annual Meeting of the ACL* (*COLING* 84), pp.319-326,1984

[Kuo 1989]

H.W. Kuo and J.S. Chang, "Systemic Generation of Chinese Sentences", *ROCLING II*, pp.187-212, 1989.

[Li and Thompson 1982]

C.N. Li and S.A. Thompson, *Mandarin Chinese - A Functional Reference Grammar*, University of California Press, California.

[Liu, Huarng and Hsu 1987]

P.F. Liu, D.H. Huarng and S.C. Hsu, "An Analysis and Implememtation of Othello", *Bulletin of The College of Engineering, National Taiwan University,* No.41, pp.17-26, 1987.

[Mann 1981]

W.C. Mann, "Two Discourse Generators", *Proceedings of the 19th Annual Meeting of the ACL*, pp.43-47, 1981.

[McKeown 1985]

K.R. McKeown, "Discourse Strategies for Generating Natural-Language Text", *Artificial Intelligence* 27, pp.1-41, 1985.

[Moore and Mann 1979]

J.A. Moore and W.C. Mann, "A Snapshot of KDS - A Knowledge Delivery System", *Proceedings of the 17th Annual Meeting of the ACL*, pp.51-52, 1979.

[Moore and Mann 1981]

J.A. Moore and W.C. Mann, "Computer Generation of Multiparagraph English Text", *American Journal of Computational Linguistics* 7:1, pp.17-29, 1981.

[Resenbloom 1982]

P.S. Rosenbloom, "A World-Championship-Level Othello Program", *Artificial Intelligence* 19, pp.279-320, 1982.

[Wilensky 1986]

R. Wilensky, *Common LISPCraft*, W.W.Norton & Company, 1986.