Incremental Updating of Objects in INDIGO

Bruno Bouzy
LAFORIA-IBP Paris 6 University, Tour 46 2ème étage
4, place Jussieu 75252 PARIS FRANCE
e-mail : bouzy@laforia.ibp.fr
Tél.: (+33) 44 27 70 10 Fax: (+33) 44 27 70 00

Abstract :

This paper shows the incremental updating that is used in the Go playing program Indigo. Due to the size of the board and time constraints, incremental mechanisms are relevant to update data. The evaluation of a position includes the construction of a taxonomy of objects which are linked by a lot dependencies. Therefore, classical incremental approaches use browsing of the taxonomy to udate objects. In Indigo, we use a different approach that use the spatial features of the game of Go. Each object is a "relying object" with a "location" and a "track". The later is the set of intersections on which the object depends. When a move is played somewhere on the board, browsing of dependencies is not necessary and "tracks" are used instead. The objects whose track meets the location of the move are deleted and the other ones are not. This mechanism symplifies the task of the programmer in pushing the incrementality problem into the specification of the track of each class of objects. This mechanism slightly reduces the response time of Indigo by a ratio of 50 on 19x19 boards without modifying the play of the program.

Keywords :

Incrementality, Object-oriented programming, Game of Go, dependency, "Track".

1. Introduction

The game of Go is a very complex two-person, complete information game. It is famous in China, Korea and Japan. Its complexity is high : there are between 0 and 361 possible moves on a position and game lasts about 300 moves. A position is hard to evaluate, and a good evaluation function needs a lot of knowledge. Despite of the effort in developing Go playing programs for thirty years, the level of the best programs still remains low on the human scale. The best Go programs are Goliath [Boon 1991] and Handtalk. Many faces of Go [Fotland 1992] is on the top of the Computer Go Ladder [Petersen 1994].

Indigo is the Go playing program [Bouzy 1995b] that we developed during our thesis [Bouzy 1995]. It is object-oriented (40 000 lines of C++) and it participates to the Computer Go Ladder [Pettersen 1994].

A very important guideline in developping a Go playing program is time constraints. This is crucial when the program plays on 19x19 boards. Most of the Go playing programs use some mechanisms to incrementally update their data. The aim of this paper is to show the *incremental updating of objects in Indigo*.

In Indigo, the evaluation of a board includes the construction of a hierarchy of objects with dependencies between them. The construction of the objects is quite complex. Part 2 shows the main idea - the *relying object* class definition - that supports it in order to make part 4 understandable.

An absolute updating may consume too much time on 19x19 boards. Then an incremental updating is relevant as it is shown in part 3.

Incremental updating can be a complex task for the programmer when he wants its incremental updating to be as accurate as the absolute updating is. Part 4 shows the mechanism of incremental updating. The main idea is to use the fact that the objects under construction are relying on a board and that their dependencies are mainly spatial. For each object, we define a special spatial set and we call it the *track*. The tracks allow to update objects without using the numerous dependancies between them. Therefore the task of the programmer becomes easier.

Before conclusion, we show what is necessary, what is gained and what is lost with such an approach and we compare it with [Klinger & Mechner 1996] which is the alone paper we found on this specific but crucial topic.

2. What to update ?

Object-oriented design and programming drove our work. INDIGO is object-oriented. This part shows the construction of objects that is done in the software. It does not handle the way it is updated after each move. It shows step by step the main concepts that are used in our design.

2.1. A relying object

A relying object is an object that is relying on the board. Each relying object has a relationship with the set of intersections it relies on. This set is called the *location* of the object.

A relying object is said to be simple if it is directly computed with the colors of the intersections of the board.

example of a simple relying object :
    a string of stones

2.2. The static state of an object

A relying object may have several states, each state may be dynamic or static.
A static state of a relying object is a state that can be deduced from the position without any lookahead.

example :
    'Won' if 3 liberties at least
    'Other' if 1 or 2 liberties

2.3. The dynamic state of an object

A dynamic state of a relying object is a state that is computed with lookahead on the static state of the object.

In Indigo, the values of a dynamic state is inspired of [Conway & al. 1982].

example :
    > The static state of the relying object is 'Won' whoever plays first.

2.4. A conceptual object

Abstraction is a standard technic in Computer Go [Ricaud 1995] [Klinger & Mechner 1996].

Conceptual objects in Indigo are built using other objects and with the states of these objects.

example :
    a group of strings

2.5. States of a conceptual object

As it is done about simple relying objects, one can define the static state of a conceptual relying object. In this case, the static state may depend on some dynamic states of other objects. That means that a static state of a conceptual object is static at the conceptual level of the object only. But, of course, a static state of a conceptual object may include the results of some lookaheads which are computed before on lower level objects.

2.6. Intrinsic or interactive state

For a static state, you have two kinds of state : *intrinsic* or *interactive*.

An intrinsic state is a state that depends on the object itself only. An instrinsic state does not depend on the neighbourhood of the object itself.

example of intrinsic state :

The lifebase of a group. This property reflects the ability of a group in having two eyes.

An interactive state is a state that depends on the states of the object itself but also on some neighbouring objects of the object itself.

example of interactive state :

The opponentship of a group. This property reflects the static ability of a group in winning a fight against an opponent of the group.

2.7. Global evaluation

The global evaluation of the board depends on the groups and the territories with their states. On can see the global evaluation as a state of the most simple relying object that is the board.

2.8. Dependancies

To sum up this part, figure 1 shows the global evaluation of a Go board in the object-oriented view. Relying objects and states are connected with a lot of dependancies. This figure is a simplification of what is actually built in Indigo.

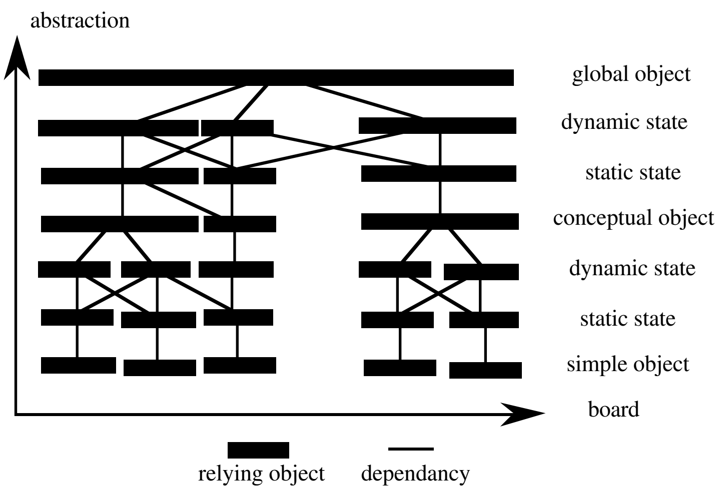

figure 1

3. Why incremental updating ?

3.1. Theoretic and combinatorial parenthesis

Papers have shown that most common board games Chess [Fraenkel & Lichtenstein 1981], Checkers [Robson 1982] and Go [Robson 1982b] generalized to NxN boards are theoretically of equal complexity. What has been shown is that, in each case, the problem of deciding whether one player can force a win from a given position is *exponential in time*.

[Allis 1994] defines A the "whole game tree complexity". Considering the average length of actual games L and average branching factor B, we get $A = B^L$. Literature on games and Human (H) versus Computer (C) results give the table 1 :

	Othello	Chess	Go
A	10^{58}	$35^{80} \approx 10^{67}$	$200^{300} \approx 10^{700}$
H v.C	H<C	H≈C	H>>C

Table 1 : Complexity and H v. C results

These games get increasing complexity and seem to be correlated with Human versus Computer results.

3.2. The practical aspect

In practice, when developing a Go playing program you don't care about theoretical complexity of the game of Go because, first you don't need an optimal move (Go playing programs are weak on the human scale), and secondly, the size of the board is fixed.

But you have to look carefully on the time in which your program play a move because, first, it is very boring to play against a weak opponent that is slow, second, time limits exist in official Computer Go competitions, third, the strongest Go programs are curiously the ones that play the fastest, fourth, to test the level of your program you cannot wait for years of computations.

On 19x19 boards, the first implementation of the model we presented in part 2, was using between 1 minute and 1 hour on a Sun sparcstation to evaluate a single global board. It was unbelievable to play against Indigo in such time conditions.

3.3. Cognitive parenthesis

Human players do not re-calculate each situations of the board after each move. In fact, they calculate only situations that are affected by the last move.

3.4. The necessity of an incremental construction

When you have a closer look, it would be required that an incremental updating only updates objects that have to be. On the example of figure 1, figure 2 shows which objects are to be deleted in case of a given move X played somewhere on the board.

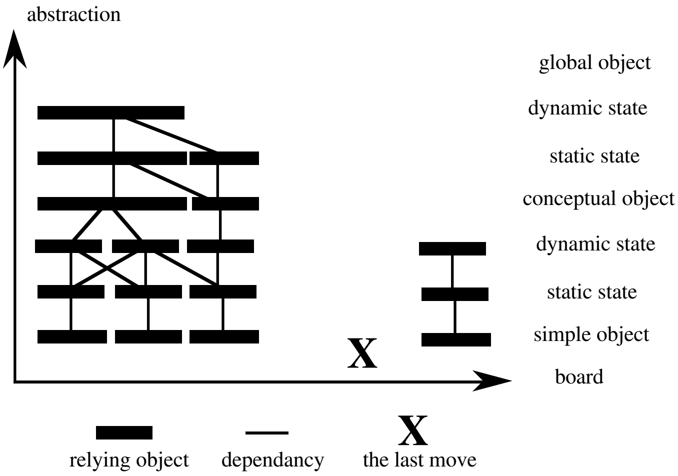

figure 2

4. How to update incrementally ?

4.1. Related works

[Klinger & Mechner 1996] describes some very crucial points of an architecture for Computer Go. Between other things they describe a mechanism of *reversion* and an incremental mechanism using *wards* and *dependencies*.

The authors describe a C++ class of "revertable" object. When a move is played or backtracked, this class allows to push an object into a list instead of destroying it, or to pull it instead of creating it again. This mechanism is very useful because it is quicker to pull an object off a list than to create it completely. But this mechanism does not say *which* objects to destroy, which objects to create, when a move is played somewhere on the board.

The authors describe also two C++ classes that allows to know which objects are affected by a move : the class "ward" and the class "dependency". The objects are explicitly linked. An object that depends upon another one inherits from the class "ward". Each "ward" class defines a function that handles each different alerts. An object that is depended-upon another one inherits from the class "dependency". Each "dependency" class defines a function that registers wards and sends alerts. This mechanism allows to destroy only objects that depend on another object when the later is modified.

But first, this approach does not say how to handle lookahead at different conceptual levels and second, the program must follows all the dependancies to destroy the objects : that can be slow or error-prone if objects are not deleted is the right order.

4.2. The "track" of a relying object

In Indigo, first we have lookahead at different levels and second we use an approach that shift the problem of following the dependencies in the taxonomy of objects.

The idea was to use the fact that the game of Go *is played on a board* and therefore that every object is created directly or indirectly with some intersections of the board. Each object that is created with a position is a "relying object". It has a slot that is the set of intersections it relies on. The idea was to add another slot : the *set of intersections it depends on*.

We called it the *track* of the object.

The track of an object has the following properties :

- if a move is played in the track of an object, then the object must be deleted;
- if a move is played outside the track of an object, then the object remains unmodified.

Of course, the track of an object is bigger than the set of intersections it relies on.

Figure 3 shows a vertical view of the board with one relying object. There are three areas : the "location" of the object, the "track" of the object and the outside of the object. If one move is played on the location, like move Z is, then the object is deleted. But even if one move is played on the track of the object, like move Y is, then the object is also deleted. If the move is played ouside of the object, like move X is, then the object remains unchanged.

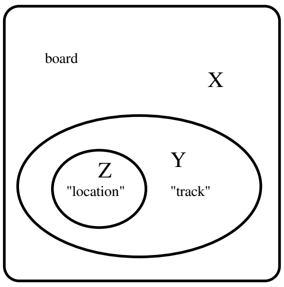

figure 3

4.3. The "track" of a static state

Sometimes, you have to make a distinction between the set of intersections the objet depends on and the set of intersections the *state of* an object depends on. In such a case, the programmer translates the state of the object into an object that is a relying object and therefore has a track. Another way to speak of this distinction is to write that the track of the state of the object is not equal to the track of the object. In these cases, we represented these static states with objects.

4.4. The "track" of a dynamic state

When you want your program to lookahead on the static state of an object and save the result in the dynamic state of the object, you may also define the track of a dynamic state.

For example, when a ladder takes place about a string and goes through the board along a diagonal, the static state of the string has a track that is located on the string and its very near surroundings, but the dynamic state of the string is the diagonal on which the ladder takes place.

The track of a dynamic state is always bigger than the track of a static state.

Without the mechanism of the track, things would be complicated because the dynamic state would depends on all static states that would be generated during lookahead. With a simple mechanism of dependencies, if the programmer wants to keep explicitly all the dependencies between the dynamic state and the static states, then you should save all the objects with their static states that lookahead have generated. In the case of the shicho, this would be absurd : it would mean that the result of the shicho depends on the fictive strings that are generated by the lookahead.

With the mechanism of the track, you simply put each track of each static state that the lookahead encounters into the track of the dynamic state. This way, you can delete the objects that are created during the lookahead without boring with a lot of dependencies.

4.5. Results

Let us define r as the ratio between the average time per move in the incremental mode and the average time per move in the absolute mode.

On 9x9, the results are a bit disappointing, the incremental mode playing program has about the same speed than the absolute mode playing program; $r \approx 1$.

On 13x13, $r \approx 6$.

On 19x19, $r \approx 50$. With the incremental mode the program plays much quicker than before on 19x19 boards : between 1" and 1' instead of between 1' and 1 hour.

On specific positions where some complicated sub-positions occur, the gain can be very big: when the last move has been played on empty areas far enough from these complicated sub-positions, the incremental mode playing program answers instantly when the absolute mode playing program computes again the evaluation of all the complicated sub-positions...

The drawback is that the incremental mode playing program plays as slowly as the absolute mode playing program when exactly one complicated sub-position occurs on the position and when the last move is played in the track of this complicated sub-position. That is the case of most of 9x9 positions.

4.6. Discussion

The main result of incremental updating is the one we expected for : response time is slightly reduced. But there are other side effects as well.

Then, the problem is to *correctly specify* the track of an object or static state. In the case of strings this is an easy doing. In the case of the static state of a string in a ladder, you have to think a little bit to specify the track. Moreover, in the case of more complex objects like "groups" (that are themselves difficult to define) specifying the track becomes harder. The programmer is forced to give up an optimal specification of the tracks and to approximate them.

If the programmer specifies a track that is bigger than the optimal one, then the program does not forget destroying objects and the behaviour of the program is still correct (i.e. equal to the behaviour of the absolute mode playing program) but it is slower than the optimal incremental mode playing program. If you specify a track that is the whole board for all objects then the program makes absolute updating.

If the programmer practically specifies a track that is smaller than the optimal one, then the program forgets to destroy some objects and the program quickly has an erratic behaviour and bugs appear.

So the problem is to correctly specify the track of objects and static states *as small as possible*. Track of a dynamic state is always the logical or of the tracks of the static states that are encountered during lookahead.

A new kind of bug appears with this technic : the *incrementality bug*. Assume your program plays a game in the incremental mode and that an erratic behaviour appears on a given position. When the programmer analyzes the game it makes the program running on the given position with the absolute updating. If the erratic behaviour also exist, it is a simple bug. If not, then this is symptomatic of the presence of an incrementality bug : the behaviour of the program is different in incremental mode than in absolute mode.

For an incrementality bug, it is hard to find which track is the source of the bug. Sometimes the smallness of the track makes the program forgetting to delete an object at move x, but the behaviour of the program becomes erratic only after move $y > x$. When you analyzes a game of your program, you know y but it is very difficult to guess x and trap it. You have no other solution than the try-error method to find x and reduce the incrementality bug to a classical bug.

5. Conclusion

In this paper, we have shown the incremental updating of objects in the Go playing program Indigo.

We have shown the concepts of our design that are dealing with incrementality :

- the *relying object* class,
- two important slots :
  - the *location* which is the set of intersections on which the object is relying,
  - the *track* which is the set of intersections on which the object depends.

We have shown how to incrementally update relying objects, static states and dynamic states in using the property of a track that is :

- if a move is played in the track of an object, then the object must be deleted;
- if a move is played outside the track of an object, then the object remains unmodified.

We followed such an approach because it simply uses the spatial features of the game of Go. It avoids to use the huge number of dependencies between objects when the program must decide which objects to delete.

We pointed that the incrementality mechanism in Indigo decreases the average response time by a ratio of about 50 on 19x19 boards.

We discussed the fact that the incrementality problem is now pushed in the *specification of the track* of objects and that the programming task is nicely simplified.

6. References

[Allis 1994] - Allis L. V. - Searching for Solutions in Games and Artificial Intelligence - PhThesis - Vrije Universitat Amsterdam - Maastricht

[Boon 1991] - Boon M. - Overzicht van de ontwikkeling van een Go spelend programma - Afstudeer scriptie informatica onder begeleiding van prof. Bergstra J.

[Bouzy 1995] - B. Bouzy - Modélisation du joueur de Go - Thèse de doctorat d'informatique de l'université Paris 6 - Janvier 1995.

[Bouzy 1995b] - Bouzy B. - The INDIGO program - Proceedings of the 2nd Game Programming Workshop in Japan - pp.191-200.- Kanagawa 1995.

[Conway & al. 1982] Conway J., Berlekamp E.R., Guy R., *Winnings Ways*, Academic Press, 1982.

[Fotland 1992] Fotland D., Many Faces of Go, documentation and playing algorithm, 1992.

[Fraenkel & Lichtenstein 1981] Fraenkel A.S., Lichtenstein S., Computing a perfect strategy for n by n Chess requires time exponential, *Journal of Combinatorial Theory*, Serie A, Vol. 31, N°2, September 1981, pp. 199-214.

[Klinger & Mechner 1996] - Klinger T. Mechner D.A. - An architecture for Computer Go

[Pettersen 1994] - Petersen E. - The Computer Go Ladder - http://cgl.ucsf.edu/go/ladder.html

[Ricaud 1995] - P. Ricaud - Une approche pragmatique de l'abstraction : application à la stratégie de début de partie au jeu de Go - Thèse de doctorat d'informatique de l'université Paris 6 - Décembre 1995.

[Robson 1982] Robson J.M., N by N Checkers is exptime complete, TR-CS-82-12, Australian National Unviversity Department of Computer Science, 1982.

[Robson 1982b] Robson J.M., The Complexity of Go, TR-CS-82-14, Australian National Unviversity Department of Computer Science, 1982.