Data Structures for $2 \times n$ Go

RICHARD J. LORENTZ
lorentz@csun.edu

SIMON HA

Department of Computer Science
California State University
Northridge CA 91330–8281 USA

Abstract

We discuss the relative merits of various data structures and data representations used in programming go on a board with exactly two rows ($2 \times n$ go). We attempt to justify the choices we made and present some empirical evidence to show the worthiness of some of our decisions.

1. Introduction. We begin by answering the two obvious questions: (1) Why $2 \times n$ go? and (2) Why discuss something as simple as data structures?

We choose to study $2 \times n$ go for a number of reasons. We began looking at this game a few years ago with the thought that we could use a program to completely solve the game for small values of n and then quickly expand upon this program to write a very good, if not perfect, player of $2 \times n$ go [1]. It turns out both of these goals were unrealistic.

We still have not managed to solve the 2×9 case, and even some of our analysis for the 2×7 case has been called into question (though we continue to believe that our basic results are correct there). Also, writing a strong program for larger boards has proven irritatingly difficult. Despite the apparent simplicity of the game, many of the difficulties encountered in writing strong programs for the normal game of go appear here as well. We find ko, seki, capturing races (a situation that appears with great frequency here and so strength in capturing races is fundamental to writing a strong program), life and death, etc.

This, in fact, is the reason that encourages us to continue working on this problem. Though we still see all of the difficulties of the general game of go, they appear in a simplified setting. Some examples: there is really only one kind of ko in $2 \times n$ go; there is only one kind of false eye; there in no notion of edge since every stone is on the edge.

We feel that if we can write a strong program for this problem then we should be able to generalize much of it to the ordinary game of go. Working on this simpler problem should prepare us for the more difficult task ahead.

We have decided to emphasize data structures in this paper because they are fundamental to all go playing programs, but they do not seem to attract the attention they deserve. There is no question that there are other topics that are probably more important in the sense that a strong program will never be written until these areas are more fully developed. Certainly pattern matching is fundamental and a great deal of work is being done experimenting with the best ways to represent patterns, search patterns, attach meanings to patterns, maintain pattern data bases, etc. The same can be said about knowledge representation, life and death analysis, and many other similar topics.

However, the underlying data structures that represent the board, the stones, the strings, the groups, the eyes, the moyo, and whatever other structures are deemed primitive to the game must be chosen carefully so that the advantage of carrying this information around is not canceled by the burden of maintaining the information. This implies that it is not just the choice of the basic structures that is important, but their implementation as well.

In the rest of this paper we will discuss what properties of the go board we choose to represent in our 2 × *n* go playing program, the data structures we use to represent them, and some programming details.

2. Preliminaries. In writing a program to play 2 × *n* go there is a temptation to optimize the program for that game. Though we are, indeed, interested in writing a program that plays a strong game of 2 × *n* go, we are also interested in being able to apply our techniques to the general (19 × 19) game. Hence, most of the techniques we employ we use with an eye towards the general game.

For example, in 2 × *n* go one condition for a string to be alive is for it to have at least 10 liberties. Ignoring the fact that it is unusual for a string to obtain so many liberties in a 2 × *n* game, in the ordinary game of go there is no such condition possible. A dead string may have an arbitrary number of liberties. So, at least for now, we choose not to include such properties in our programming.

On the other hand, fundamental to both games are concepts such as strings (a collection of stones of the same color which are connected in terms of the graph imposed by the grid of the go board), groups (there is no firm definition of this concept – we will discuss this in more detail later), and eyes (again, we need to be careful of the definition) so these are the kinds of properties we implement in our game and we will be discussing.

Another important issue in writing any go program is deciding which structures should be "incremental" and which should be recalculated as needed. The distinction is best illustrated by an example. If a go program is maintaining a list of strings on the go board, after a move is made there are many changes that might occur to the strings. A new string might be created if the new stone was placed in isolation. A string might be enlarged by the addition of this one stone. Two strings might be merged by being connected by this stone. Some strings might actually be removed from the board if this stone captured some strings. To cope with these changes the programmer has two choices.

Since there will surely be some change(s) made to the strings after a stone is played we could choose to sweep over the entire board and *recalculate* all of the strings. This means that we throw away all the information we currently know about the strings and start from scratch to figure out what the string structure on the board is.

The other choice is to see which strings were affected by the placement of the new stone and *incrementally* change just those strings that are affected. This means going in and changing the data such that the appropriate strings are merged, removed, etc.

Wilcox argues strongly and eloquently against the use of most incremental data structures [4]: "Incremental modification is a tempting idea with any structure used to describe or represent an evolving situation over time. ... Incremental modification becomes a nightmare. Because I had incentive to save time and squander memory, incremental modification was used throughout the old program. ... I spent months debugging.... They were never fully debugged. ... The code grew complex. The number of special cases to consider when incrementally updating a structure is larger than if that structure were built from scratch. ... Tactics remained expensive. ... Thus I learned to hate incremental modification."

His point is well taken. Nevertheless, in order for a strong program to perform efficiently, a certain amount of incremental modification is necessary. It is, for example, unreasonable to expect to recalculate from scratch all the strings every time a move is made. Look–ahead would become unbearably slow. So we must discuss what will be done incrementally, how we will do it, and how we justify the programming overhead (and associated risks) involved in coding and debugging incremental structures.

3. The Board. In an effort to keep the representations simple in order to facilitate modifications and debugging we maintain our board as a simple two dimensional array of what we call "spots." Using a standard technique, our board actually has a number of extra rows and columns in each direction so that edge of board considerations can be dealt with using markers on the spots rather than a large number of conditional statements.

Having claimed that we are striving for simplicity, however, we point out that our board array and most of our data structures have pointers back and forth between each other so that we have ready access to needed data. This is so that in those areas where we do incremental updating we are able to move easily about the various components and do the updating quickly. After all, the only reason for doing incremental updating is to increase speed, so it is important that we try to do it as efficiently as possible.

Each spot in our board array contains the following information. There is a status item indicating whether the spot is occupied, vacant, or an artificial border point. If it is occupied it also indicates what color stone is there and if it is not it indicates if that spot is an illegal move because of ko.

There are a number of pointers to various structures from each spot. A spot may be a liberty of up to 3 strings (4 in standard go, of course) so there are pointers to each string that this spot happens to be a liberty of. If the spot is unoccupied and an internal point of an eye, there is a pointer to that eye. Also, for the sake of generality (but at the expensive of the usual notion of eye) a small eye may be internal to a larger eye of the opposite color and so if necessary we provide a pointer to that eye as well. We also have a notion of group and so a spot may point to the group it is part of. We say "may" because the group structures are not incremental. During look–ahead group information is not needed so often and the overhead in maintaining groups incrementally seems exorbitant, so we have decided to recalculate group structures as needed.

Finally, there are values associated with every spot that correspond to such things as influence values (there are actually eight of these corresponding to the various combinations of the color of the influence and the direction that influence may radiate), a guess as to how valuable a move made by each color at this square would be (in an effort to reduce the number of times a move's value needs to be recalculated), and whether this square is part of a "horse" or a "horse pair," a topic that is discussed in Section 6.

4. Strings. As mentioned earlier, strings are in some sense the most basic of the structures in a go game. Once formed a string can never be divided. It can only merge with another string (including strings of size one) or be completely captured. So it is not surprising that most of the data structures involve strings one way or another. And because of their importance, we choose to maintain them incrementally so we are not constantly recalculating them.

First of all we maintain global lists of all the black and white strings. This allows us to easily move around strings doing things such as looking for weak strings to capture or protect, finding "groups" of strings, identifying live strings, etc. Also, just as each spot on the go board points to the string it is a member of, each string contains a list of the spots that it comprises. This allows us to quickly move back from the string to the board when necessary, for example, to determine influence values in the vicinity of a string.

Each string also maintains a list of its liberties. Since strings are incremental, the liberties (and all other properties maintained by the string data structure) must, of course, also be maintained incrementally. So we must choose carefully what information we really want or need to maintain with a string since we do not want the overhead of maintaining too much information with each string to negate the advantages of doing incremental management. In this case, however, we believe that it is self evident that liberties must be maintained with strings since liberties are fundamental to so many aspects of strings including capturing (when the liberty count drops to zero), the health of the string (since more liberties often means a healthier string), tactical points (since liberties are often strong places to play when attacking a string), etc.

A string may also outline parts of eyes, or perhaps more precisely, every eye is defined by the strings that enclose it, so the string data structure maintains a list of all eyes that it is a part of. A long string can be part of many eyes so we place no limit on the number of eyes we associate with a string. We will define what we mean by an eye and how we represent it in the next section.

Finally, if a string is part of a horse (see Section 6) then there is a pointer to the data structure that represents horses as well as pointers to other strings that are part of the same horse.

We will not go into implementation details here, but related to the strings and some of our other data structures there is another data structure we need to mention that is a bit implementation dependent. This data structure has to do with deciding what to maintain so that moves can be "undone." As mentioned before, when making a move, strings may be created, merged, and even deleted. This, of course, not only has an effect on the strings involved but all other related structures such as liberty lists, horses, etc. So, after a move has been made and all these changes have been recorded what do we do when it is time to take a move back, remembering that taking moves back happens exactly as often as making moves when one is doing look–ahead? We don't want to lose all that we gained with the incremental data structure when it is time to take a move back.

The simplest solution to this problem seems to be to take snapshots of all the relevant data before a move is made. Relevant data includes all the strings that have been created, merged, or deleted, all the liberties of all the strings that are affected, the changes to horses, the ko status of points (which, of course, is at most one point), eyes that may have been created or destroyed, and some other basic things such as the move number. Notice this is not a snapshot all the data structures of the entire board. This would be an extreme way of dealing with the problem, where the other extreme would be the already mentioned method of recalculating everything after a move is taken back. Instead we have chosen a compromise method where most of the basic data is stored and so does not have to be recalculated, but this data has to be reinserted into the global structures to recreate the previous situation. This compromise method was originally chosen when memory (RAM and disk) was not quite so readily available as it is now, so it would be an interesting experiment to see if a method could be devised so that the a snapshot of all memory associated with the data structures could be easily captured, stored, and then restored as needed. A minor note related to this: we use the same snapshot technique for taking back moves during look–ahead and for taking back moves during play, through the menu system. If we were to try to take a snapshot of all of memory we would probably have to restrict that to the look–ahead portion only as even with current memory sizes, it would be unrealistic to expect to be able to store hundreds of total snapshots – and unnecessary as there no need to be able to retract moves at the user level with such blinding speed.

5. Eyes. Our notion of eye is imperfect at best, but the data structures involved are straightforward. Just as we maintain a global list of strings we also maintain a global list of eyes for the same sorts of reasons. Using the global list we can quickly move to critical areas of the board involving eyes. Also, the global list makes it easier to reconstruct the total data environment after a move has been taken back.

We view an eye as any region totally surrounded by strings of the same color and the edge of the board. This can be a bit problematic in general and even more so in the $2 \times n$ case. Consider Figure 1, on the top of the next page:

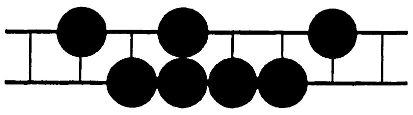
Figure 1

By our definition this group has two eyes but it is by no means alive yet. Hence, we must do more to determine whether a group is alive besides just counting the eyes. One way we do this is to recursively visit strings and eyes to be sure that every string associated with an eye touches two eyes and every eye is surrounded by strings that touch two eyes. Clearly the group in Figure 1 does not satisfy this condition since there are two strings (the single stones) that do not touch two eyes.

Unfortunately, this approach has proven to be too conservative in that very healthy groups do not satisfy this condition. So in an effort to improve the correctness of our evaluations we do some independent tests on an eye to see if it is healthy, and if so, we mark it as such. This, then, explains one component of our eye data structure, a flag that marks whether or not the eye is "real" or not. Perhaps an integer representing how close or far the eye is from being real would be better, but we find it difficult to do such an evaluation and equally difficult to then use such a value.

Another difficulty the "real" flag attempts to deal with has to do with situations similar to those in the following figure:

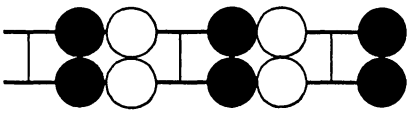
Figure 2

Are there two black eyes in Figure 2, just one white eye, a white eye inside a black eye, or what? Of course the answer depends on whose move it is and who ultimately captures whom. Suffice it to say that in 2 × n go capturing races play a critical role and so it is important to be able to recognize real eyes (as it is in normal go, of course) as so we have the "real" flag.

Continuing how we represent eyes, we keep a list of all the spots that are in the interior of the eye. This information is used for a number of things including determining how healthy the eye is (e.g., the more spots, the healthier), and seeing if it matches one of our eye patterns. To complement this second usage, eye data also includes information about the expanse of the eye in terms of the number of rows and columns it spans. The actual pattern recognition process is described in [2]. Also, there is a pointer to the actual pattern if there is a match, where information about the pattern itself is stored.

6. **Horses.** The last major incremental data structure that we maintain has to do with groups of strings. Groups are notoriously hard to define because they depend so much on context such as proximity to edges and corners, distances between components, friendly and enemy influences near connections, etc. But there are some situations where it is clear that two strings are part of the same group. Such strings connected in this way we call a *horse* from the Korean term for this situation.

In the case of 2 × n go, there are really only two ways strings can be connected to form a horse. They are shown in Figure 3 below.

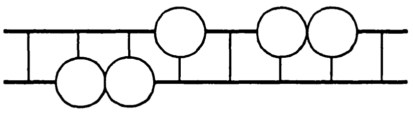
Figure 3

In this case all three groups are part of the same horse, where the middle string (the single stone) is connected to the string on the left in one way (diagonally), and connected to the string on the right in the other way (adjacently). Both of these connections have the property that the enemy, black in this case, must occupy two squares to separate the horse. For example, for black to cut the adjacent connection on the right he must occupy the point between the two strings and the point below. (Of course, he will actually need to occupy quite a few other points as well to give his cutting string sufficient strength to survive the incursion. A two stone string is unlikely to last very long after the cut.) If he only occupies one of these two points, white can occupy the other and connect. Another way to view this is that the strings are connected by miai. But this is the crucial point. If we can maintain these connections in our data structure, then as soon as black occupies one of the points, white immediately knows what point he must occupy to maintain the connection.

Because horses can remain connected if the program chooses to, we can then use horses as our basic unit of strategy, rather than strings. This, presumably, should have a positive effect on both the speed and the behavior of the program

To maintain horses incrementally is actually quite involved. We do most of it through the string data structure. Every string has a pointer to its horse (which will often be the string by itself) which maintains general information about the horse. Every string also has two pointers, one to the left and one to the right, in case it is part of a bigger horse and there are connecting strings to the left and/or the right. But these pointers don't point to other strings. Instead they point to another structure called a *pair* that represents the two points of the miai that are forming the connection between the two strings.

A pair, in turn, keeps the information about the status of the connection and what it is connecting. Clearly it will keep track of the two strings it is connecting by providing pointers to these two strings. It also has a pointer to its own horse for completeness. Finally, it maintains information about the two cutting points, such as which if any are occupied by the enemy. If somehow both points get occupied by the enemy then, of course, the horse gets split into two horses. Also, as we have done with all of our other data, since the essence of a pair is the two spots on the board, a board spot maintains a pointer back to the pair if it is part of one. This was mentioned in Section 3 where we discussed boards and spots.

However, there is a one technicality here that must be dealt with. It is possible for a spot to actually be part of two pairs (in a single horse) as seen in the following figure.

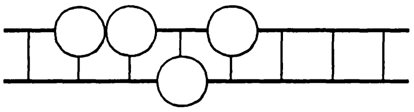

Figure 3

Here the spot above the single stone on the bottom row is actually part of two pairs associated with the two stones on the left and the bottom stone and associated with the bottom stone and the single stone on the right. So we deal with this case by allowing spots to point to two pairs, which is the most possible in 2 × *n* go.

Finally, the data structure for the horse includes information like a list of the strings comprising the horse (we don't provide a list of pairs since they can be quickly extracted from the strings list), the total liberties of the horse, how likely the horse is to live, etc. The idea is that the horse now becomes our basic unit of strategy (rather than the string) and so we access horses to see what is strong, what can be attacked, etc. Since horses are maintained incrementally because of their fundamental importance, we need to put horse information in our move snapshots so their structure can be quickly recovered when we retract a move.

7. Conclusions and further research. Despite the dire warnings of Wilcox, we have chosen to implement a great deal of our 2 × *n* go playing program using incremental data structures. We believe maintaining strings incrementally is absolutely necessary and have designed our program around this philosophy. As a result we have no empirical evidence to justify the claim that this is the best way to go since we have no experience with a program that recalculates strings. But we remain confident.

The situation with eyes is less clear. Our implementation of eyes is still not perfect, but we have chosen what seems to be a reasonable middle ground. We maintain the physical notion of eyes incrementally. By this we mean the aspect of eyes that has to do with an eye being surrounded by strings of the same color. However, the features of an eye (except those we extract through pattern matching) such as whether it is false or not, and if false, how likely it can become a real eye, these we calculate on an as needed basis, mainly during evaluation.

On the other hand, group information in the large sense, i.e., beyond the level of horses, is not done incrementally at all. This is mainly because our program is not sophisticated enough to use group information very often and because we are still struggling with the correct definition of group. Perhaps if we ever settle on a reasonable definition for group it will become a candidate for incremental implementation. And then we might find more uses for it, as we mention below concerning horses.

As for horses, we have versions of the program with and without incremental implementations. Empirical evidence indicates that the program runs about ten percent faster with the incremental horse. At first this sounds a bit discouraging. But there are other positive side effects of having incremental horses. Since they are calculated incrementally, this means that they are always available in the data structures. As a result we are encouraged to use this information more often as a result of its ready availability. The effect of this has been that not only is the program ten percent faster than it was, but it also plays better since we find we can now use horse information in many places such as during leaf evaluations, when pruning nodes, when ordering moves, etc. One is certainly less inclined to use information if it must be recalculated every time you want it.

As we said earlier, all of the implementations were done with the general game of go in mind. As a result we believe that much of what we have done can be quickly applied to a program that plays the general game.

However, we now find ourselves in the position where we want to know more about $2 \times n$ go. Can we write a very strong program? We are thinking of moving in the direction of programming $2 \times n$ go specifically to see how strong we can make the program. We are also wondering if the game isn't simple enough to evaluate analytically to see if we can find a perfect player. We have had no luck to date, but it is not at all clear that it can't be done. Also, Sanechika has suggested to us that we might consider different rules, e.g., forbidding passing. Clearly this will have an effect on small boards, but it is less clear what effect, if any, it would have on larger boards.

At the other end of the spectrum perhaps $2 \times n$ go is provably hard. Maybe it can be proven to be NP–hard (in which case it will probably be PSpace–hard, too, as seems to frequently happen with games, e.g., general go [3]). This seems less likely to be true though, because the restriction to two rows makes it difficult to reduce from the standard NP-Complete graph oriented problems which typically have a decidedly two dimensional feel to them and would seem to be the natural candidates to use in this situation.

8. References.

[1] Lorentz, R.J., $2 \times n$ Go, in *Proceedings of the Game Programming Workshop in Japan '97*, (October 1997), 65–74.

[2] Lorentz, R.J., Pattern Matching in a Go Playing Program, in *Proceedings of the Second Game Programming Workshop in Japan*, (September 1995), 167–174.

[3] Lichtenstein, D., and Sipser, M., Go is polynomial-space hard, *Journal of the Association of Computing Machinery*, 23 (1980), 393-401.

[4] Wilcox, Bruce, Reflections on Building Two Go Programs, *Special Interest Group on Artificial Intelligence*, (October 1985), 29–43.