# Using Go in teaching the theory of computation*

Graham Farr †
Faculty of Information Technology,
Monash University, Clayton, Victoria 3800,
Australia

29 July 2018

**Abstract**

With increased public interest in the ancient game of Go since 2016, it is an especially good time to use it in teaching. The game is an excellent source of exercises in the theory of computation. We give some exercises developed during our research on Go which were then used when teaching this subject at Monash University. These are based on One-Dimensional Go (1D-Go) which uses a path graph as its board. They are about determining whether or not a position is legal and counting the number of legal positions. Curriculum elements that may be illustrated and practised using 1D-Go include: regular expressions, linear recurrences, proof by induction, finite automata, regular grammars, context-free grammars and languages, pushdown automata, and Turing machines.

Keywords: theory of computation, Go, Wéiqí, Baduk, game, legal position.
MSC2010: 68Q01, 68R01, 68-01, 05C57, 05C15, 91A46, 97Q60.

## 1 Introduction

*Go* is one of the oldest board games still played, having originated in China over 2,000 years ago [2]. It is very widely played in East Asia, with over 40 million players [7], extensive media coverage and a dedicated 24-hour TV channel [1]. The game is known today as *Wéiqí* (围棋 (simplified), 圍棋 (traditional)) in China, *Go* or *Igo* (囲碁) in Japan, *Baduk* (바둑) in Korea, and *Go* in the West where it has long been popular among computer scientists and mathematicians.

Go is usually regarded as more complex than Chess. The number of legal positions is greater, the number of moves available at each turn is greater, and the best computer programs were well behind the best human players [13] until very recently.


†Email: Graham.Farr@monash.edu.

The profile of Go was increased significantly when Google DeepMind's AlphaGo program defeated first the European champion Fan Hui [15] in October 2015 (see also the *Nature* News article [6]), and followed this up with victory over one of the game's greatest players, Lee Sedol of South Korea [10], in March 2016. In May 2017 it defeated the then-top-ranked player in the world, Ke Jie of China [11]; the program was then retired. Fortunately, there are now open source programs based on similar techniques and with their own successes against top opponents. The first was Leela-Zero [12], since October 2017. Facebook's ELF OpenGo program was made available as open source in May 2018 [16].

Paradoxically, the end of human dominance in this game seems to have led to increased human interest in the game [14].

Another major development in the application of computers to studying Go, announced in January 2016, was the exact computation of the number of legal positions in the game by John Tromp [17, 18]; an example of coverage in the I.T. press is [9]. This is one area where Go has proved more mathematically tractable than Chess, for which the exact number of legal positions is still unknown (and practically unknowable).

With this rise in interest in Go as a topic for computer science research, it is an especially good time to engage students with the game. They will have a new problem domain to explore, as well as an excellent game to play with their peers. It also gives a valuable link to East Asian culture for students outside that region.

In ongoing research on counting legal positions in Go [3, 4, 5], I considered the characterisation and counting of legal positions on boards of various sizes and shapes. In some cases, languages of legal positions — when appropriately represented as sets of strings — give good examples of language classes studied in theoretical computer science. This led to the use of examples and problems on legal positions in Go when teaching the theory of computation.

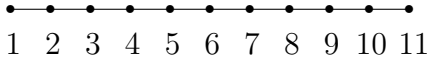
This article describes the use of such languages in a Theory of Computation subject (FIT2014 Theory of Computation) taken by second-year undergraduate students at Monash University. It is a core subject for the Bachelor of Computer Science and is taken as a Computer Science elective by some students in other degrees including the Bachelor of Science.

I am happy to make solutions to these exercises available to academic staff running Theory of Computation subjects, with some restrictions on further dissemination.

For information on how to play Go, see e.g., [8] or the many online resources. Knowledge of the game is not necessary for this article or the exercises we use.

## 2 One-dimensional Go

Go is played on a graph, usually a two-dimensional square lattice (grid) of $19 \times 19$ vertices. But we focus here on *one-dimensional Go (1D-Go)*, which uses a path graph with $n$ vertices and $n - 1$ edges, with vertices numbered from 1 to $n$, from left to right.




Positions in 1D-Go can be described by strings over a three-letter alphabet, and many basic Go concepts are much simpler in this context. We find that 1D-Go is a rich source of exercises

for many concepts in the elementary theory of computation, including regular expressions, finite automata, context-free languages, pumping lemmas, and Turing machines.

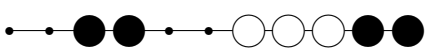
A *position* consists of a placement of black and white stones on some of the vertices of the graph. Each vertex may have a black stone, or a white stone (but not both), or be unoccupied (i.e., vacant). We may think of the vertices of the graph as each being coloured black, or coloured white, or left uncoloured. A position is *legal* if every vertex with a stone can be linked to an unoccupied vertex by a path consisting entirely of vertices with stones of that same colour (except for the unoccupied vertex at the end).

When the game is played, the players Black and White each take turns to place a single stone of their own colour on an unoccupied vertex, subject to certain constraints (or they may pass). A string[1] of identically-coloured stones is *captured* if it is surrounded by stones of the opposite colour; the captured stones are removed from the board. The aim, roughly, is to enclose more territory than your opponent. The detailed rules of playing and capturing do not concern us here; see, e.g., [8]. We focus only on *prima facie* legality of positions.

For example, the following position is legal, since each of its three "chains" of consecutive vertices of the same colour has an unoccupied vertex next to at least one of its ends.

But the following position is illegal, since it has a chain of white vertices with black vertices at each end. (The position has four chains altogether, and three are ok. But it only takes one without an unoccupied neighbour to make the position illegal.)

We say that a position on this path graph is *almost legal* if vertex $n$ has a stone (i.e., is occupied, or coloured) and its chain is not next to an unoccupied vertex, but every other chain is next to an unoccupied vertex. In other words, it is illegal, but the only chain making it illegal is the chain containing vertex $n$; all other chains are ok. The two positions given above are *not* almost legal: the first is legal (so it is not *almost* legal), while the second is illegal but the illegality is not due to the last vertex (which in this case is unoccupied). The following position is almost legal. All its chains are ok except the last one on the right.



# 3 Propositional logic

In the following exercises, propositional logic is used to describe position types, vertex colours, and how the colour of the last vertex affects the type of the position. This helps

[1] or connected subgraph, when the game is played on graphs more complex than a path

students clarify their understanding of legality and almost-legality of positions in 1D-Go, as well as giving practice in using logic.

Let $V_{B,n}$, $V_{W,n}$, $V_{U,n}$, $L_{B,n}$, $L_{W,n}$, $L_{U,n}$, $A_{B,n}$, $A_{W,n}$ be the following propositions about a position on the $n$-vertex path graph.

| | |
|---|---|
| $V_{B,n}$ | Vertex $n$ is Black. |
| $V_{W,n}$ | Vertex $n$ is White. |
| $V_{U,n}$ | Vertex $n$ is Unoccupied. |
| $L_{B,n}$ | The position is legal, and vertex $n$ is Black. |
| $L_{W,n}$ | The position is legal, and vertex $n$ is White. |
| $L_{U,n}$ | The position is legal, and vertex $n$ is Unoccupied. |
| $A_{B,n}$ | The position is almost legal, and vertex $n$ is Black. |
| $A_{W,n}$ | The position is almost legal, and vertex $n$ is White. |

**(3.1)** Use the propositions $L_{B,n}$, $L_{W,n}$, $L_{U,n}$ (together with appropriate connectives) to write a logical expression for the proposition that the position is legal.

Now consider how legality and almost-legality on the $n$-vertex path graph are affected by extending the path to vertex $n+1$.

**(3.2)** If $L_{B,n}$ is true, what possible states (Black/White/Unoccupied) can vertex $n+1$ be in, if we want the position to be legal on the $(n+1)$-vertex path as well?
Do the same for $L_{W,n}$ and $L_{U,n}$.

**(3.3)** If $A_{B,n}$ is true, what possible states can vertex $n+1$ be in, if we want the position to be legal on the $n+1$-vertex path?
Do the same for $A_{W,n}$.
Why is there no line for $A_{U,n}$ in the list of propositions above?

**(3.4)** Construct a logical expression for $L_{B,n+1}$ using some of the propositions $V_{\_,n+1}$, $L_{\_,n}$, $A_{\_,n}$ in the above table. (In other words, you can only use the $L$-propositions and $A$-propositions for the $n$-vertex path graph, and the $V$-propositions for vertex $n+1$.)
Do the same for $L_{W,n+1}$, $L_{U,n+1}$, $A_{B,n+1}$, $A_{W,n+1}$.

# 4 Linear recurrences

The insights gained using propositional logic allow us to write linear recurrences for numbers of legal and almost-legal positions, classified according to the colour of their last vertex. This helps develop students' skills in using iteration and recursion.

Define

$\ell_{B,n}$ := the number of **legal** Go positions on the $n$-vertex path graph, where vertex $n$ is **Black**.

$\ell_{W,n}$ := the number of **legal** Go positions on the $n$-vertex path graph, where vertex $n$ is **White**.

$\ell_{U,n}$ := the number of **legal** Go positions on the $n$-vertex path graph, where vertex $n$ is **Unoccupied**.

$a_{B,n}$ := the number of **almost legal** Go positions on the $n$-vertex path graph, where vertex $n$ is **Black**.

$a_{W,n}$ := the number of **almost legal** Go positions on the $n$-vertex path graph, where vertex $n$ is **White**.

**(4.1)** State the values of $\ell_{B,1}$, $\ell_{W,1}$, $\ell_{U,1}$, $a_{B,1}$, and $a_{W,1}$.

**(4.2)** Derive recursive expressions for $\ell_{B,n+1}$, $\ell_{W,n+1}$, $\ell_{U,n+1}$, $a_{B,n+1}$, and $a_{W,n+1}$, in terms of $\ell_{B,n}$, $\ell_{W,n}$, $\ell_{U,n}$, $a_{B,n}$, and $a_{W,n}$.

**(4.3)** How many of these quantities do you really need to keep, for each $n$, in order to work out the values for $n+1$? (Look for symmetry.)

**(4.4)** How do you work out the total number of legal positions on the $n$-vertex path graph, from $\ell_{B,n}$, $\ell_{W,n}$, $\ell_{U,n}$, $a_{B,n}$, and $a_{W,n}$?

# 5 Proof by induction

The recurrences of the previous section can be used to practise proof by induction. Students also learn about using linear recurrences to count things, or bound the number of them.

In §4, we saw that legal and almost-legal positions can be counted using a set of simultaneous linear recurrence relations.

We will now use these equations to prove upper bounds for the quantities $\ell_{B,n}$, $\ell_{U,n}$, $a_{B,n}$, and hence for the total number of positions. Note that all these quantities have the trivial upper bound $3^n$, since each vertex of the $n$-vertex path can be in one of three states (B/W/U), giving $3^n$ possible positions altogether, which includes all legal and illegal positions. We will do better than this.

**(5.1)** Prove by induction on $n$ that, for all $n \geq 1$, the quantities $\ell_{B,n}$, $\ell_{U,n}$, $a_{B,n}$ satisfy

$$\begin{aligned} \ell_{B,n} &\leq 1.8 \times 2.8^{n-1}, \\ \ell_{U,n} &\leq 1.8^2 \times 2.8^{n-1}, \\ a_{B,n} &\leq 2.8^{n-1}. \end{aligned}$$

The claim you are trying to prove here is the *single* claim that all three of these inequalities are true, simultaneously. You should *not* try to construct three separate proofs, one for each inequality (though it's fine for your proof to have different cases).

**(5.2)** (challenge) How much can you improve on these upper bounds? In particular, can you reduce the 2.8 to a smaller number? If so, what can you reduce it to?

This last question enables a link to be made to the theory of closed-form solutions to linear recurrence relations, which students may meet in their discrete mathematics courses. It may be worth mentioning to students the beautiful application of that theory to the Fibonacci sequence, showing that the ratio of consecutive terms is the golden ratio $(1+\sqrt{5})/2$, if they have not seen that before.

# 6 Regular expressions

After doing the logic exercises of §3, students should have a good grasp of legality and almost-legality. These 1D-Go concepts can then be used to practise writing regular expressions.

In this section and most later ones, we represent each 1D-Go position as a string over the alphabet {`b`,`w`,`u`}. The $i$-th character in the string represents the state of vertex $i$ in the $n$-vertex path graph, with `b`, `w`, `u` representing Black, White and Unoccupied, respectively. So the three positions given as examples in §2 are represented by the following strings in turn:

| | |
|---|---|
| `uubbubwwwuu` | This position is *legal*. |
| `uubbubwwwbu` | This position is *illegal*, and it is not almost legal. |
| `uubbuuwwwbb` | This position is *almost legal* (hence illegal). |

**(6.1)** Write a regular expression for the set of strings that represent legal positions.

**(6.2)** Write a regular expression for the set of strings that represent almost legal positions.

**(6.3)** Does there exist a regular expression for the set of strings that represent positions that are neither legal nor almost legal?

# 7 Finite Automata

Since the sets of legal and almost-legal positions are regular, they must each be recognisable by a Finite Automaton.

**(7.1)** Design a Finite Automaton that accepts precisely those strings that represent legal positions.

**(7.2)** How would you modify your FA so that it accepts precisely those strings that represent almost legal positions?
(No need to draw a new FA. Just describe clearly and precisely the change you need to make.)

**(7.3)** How would you modify your FA from (7.1) so that it accepts precisely those strings that represent positions that are neither legal nor almost legal?

(For (7.2) and (7.3), there is no need to draw a new FA. Just describe clearly and precisely the change you need to make.)

It is good for students to implement the FA they design. In our Theory of Computation subject, we have sometimes used Prolog for practical exercises. In the next exercise, we implement our 1D-Go FA in Prolog.

We can represent positions in 1D-Go using Prolog lists, with each member of the list being `b`,`w`,`u` according as the corresponding vertex is Black, White, or Unoccupied, respectively.

So the three positions given as examples in §2, and shown as strings in §6, are represented by the following lists in turn:

| | |
|---|---|
| `[u,u,b,b,u,b,w,w,w,u,u]` | This position is *legal*. |
| `[u,u,b,b,u,b,w,w,w,b,u]` | This position is *illegal*, and it is not almost legal. |
| `[u,u,b,b,u,u,w,w,w,b,b]` | This position is *almost legal* (hence illegal). |

**(7.4)** Write a Prolog program which implements a Finite Automaton for legal positions in 1D-Go. (We gave students a Prolog program for another, unrelated FA, as a template.)

For example, checking whether the above three strings are legal should give the following results in Prolog.

```
| ?- accept([u,u,b,b,u,b,w,w,w,u,u]).
true
| ?- accept([u,u,b,b,u,b,w,w,w,b,u]).
false
| ?- accept([u,u,b,b,u,u,w,w,w,b,b]).
false
| ?-
```

# 8 Lexical analysis

Lexical analysis is an important application of Finite Automata, and may be applied to 1D-Go.

Each position in 1D-Go may be regarded as a sequence of *tokens*, where each token is one of the following.

| Token | Interpretation |
|---|---|
| `free White chain` | White chain that has at least one unoccupied neighbour |
| `free Black chain` | Black chain that has at least one unoccupied neighbour |
| `captured White chain` | White chain with no unoccupied neighbour |
| `captured Black chain` | Black chain with no unoccupied neighbour |
| `gap` | sequence of Unoccupied vertices between chains |

**(8.1)** Convert your FA from (7.4) above to a *lexical analyser* for 1D-Go positions. This lexical analyser should detect each lexeme in the input, output it on a separate line, and

state on that line what the corresponding token is. It should give the answer `true` in all cases, so all states will be Final states. (In a more complete implementation, the lexical analyzer would use the presence of any captured chains to detect illegality, and would report at the end on whether the position is legal or not. This is not too difficult, but we did not require it in this exercise.)

For example, your program should enable the following interaction (where the positions used are the three we have been using as examples throughout).

```
| ?- accept([u,u,b,b,u,b,w,w,w,u,u]).
uu gap
bb free Black chain
u gap
b free Black chain
www free White chain
uu gap
true
| ?- accept([u,u,b,b,u,b,w,w,w,b,u]).
uu gap
bb free Black chain
u gap
b free Black chain
www captured White chain
b free Black chain
u gap
true
| ?- accept([u,u,b,b,u,u,w,w,w,b,b]).
uu gap
bb free Black chain
uu gap
www free White chain
bb captured Black chain
true
| ?-
```

# 9 Context-free languages

This section is about grammars for positions in 1D-Go.

Consider the following context-free grammar (CFG) for the language of legal positions in 1D-Go.

$$S \rightarrow P \quad (1)$$
$$S \rightarrow Q \quad (2)$$
$$P \rightarrow U \quad (3)$$

$$
\begin{aligned}
P &\rightarrow UB && (4)\\
P &\rightarrow UBS && (5)\\
P &\rightarrow UW && (6)\\
P &\rightarrow UWS && (7)\\
Q &\rightarrow BP && (8)\\
Q &\rightarrow WP && (9)\\
U &\rightarrow \texttt{u}U && (10)\\
U &\rightarrow \texttt{u} && (11)\\
B &\rightarrow \texttt{b}B && (12)\\
B &\rightarrow \texttt{b} && (13)\\
W &\rightarrow \texttt{w}W && (14)\\
W &\rightarrow \texttt{w} && (15)
\end{aligned}
$$

The non-terminal symbols have the following interpretations.

- Symbol $S$ is the Start symbol, as usual.
- Symbol $P$ is used to represent a position starting with `u`. The first chain of such a position will therefore be free (i.e., uncaptured), since it will have an unoccupied vertex on its left. That chain can then be followed by any legal position at all, hence the rules $P \rightarrow UBS$ and $P \rightarrow UWS$. (Here, the $B$ could stand for a completed Black chain, or it could be a prefix of a Black chain that is continued in the the string generated by the $S$ in $UBS$, and similarly for $W$.) Alternatively, that first chain could also be the last one, hence the rules $P \rightarrow UB$ and $P \rightarrow UW$.
- Symbol $Q$ represents a position that starts with a chain. For that chain to remain free, the rest of the position must be a legal position that starts with `u`, hence the rules $Q \rightarrow BP$ and $Q \rightarrow WP$.
- Symbol $U$ represents any sequence of Unoccupied vertices, $B$ represents any sequence of Black vertices, and $W$ represents any sequence of White vertices. (Note that the sequences represented by $B$ and $W$ need not be complete chains; they may be substrings of chains.)

**(9.1)** Give derivations of the strings (i) `buuw` and (ii) `buuwbuuw`.

**(9.2)** Prove, by induction on $n$, that for every $n \geq 1$, the string $(\texttt{buuw})^n$ has a derivation of length $7n$ using the above CFG.

Recall that $(\texttt{buuw})^n$ is $n$ repetitions of `buuw`. For example, $(\texttt{buuw})^2$ is just `buuwbuuw`.

In the next problem, you will design and implement (in Prolog) a pushdown automaton (PDA) for legal 1D-Go positions. We gave students Prolog code for a simpler PDA to get them started.

**(9.3)** Convert the above CFG for 1D-Go to a PDA that recognises the language of legal 1D-Go positions.

**(9.4)** Implement, in Prolog, your PDA from part (9.3) for the language of legal 1D-Go positions (with positions represented as lists, as in §7, (7.4)).

**(9.5)** The above grammar is not regular. Very briefly, why not?

**(9.6)** In fact, the language of legal 1D-Go positions does have a regular grammar. How do we know this? In answering this question, use known facts about regular languages, and what you already know about this particular language from our previous exercises.

**(9.7)** Find a regular grammar for the language of legal 1D-Go positions.

Legal positions in 1D-Go can be *scored* as follows.

A *gap* is a sequence of unoccupied vertices bounded on each side by a coloured stone or an end of the path graph. Note that a gap may have two, one, or no bounding stones, depending on where it appears in the position. The only way it can have no bounding stones is if it spans the entire path graph, so that every vertex in the path graph is unoccupied. A gap is *proper* if it does not span the entire path graph. If the gap has just one bounding stone, then it must sit at one end of the path graph.

A gap is *owned by Black* if it is proper and all of its bounding stones are Black. Similarly, a gap is *owned by White* if it is proper and all of its bounding stones are White. A gap is *neutral* if it is neither owned by Black nor owned by White. So, if a gap has a Black stone on one side and a White stone on the other, then it is neutral. The only other kind of neutral gap is when the entire path of $n$ vertices is unoccupied, so that the entire path is a neutral gap.

The *score* of a position is given by

$$\text{(number of vertices owned by Black)} - \text{(number of vertices owned by White)}$$

If the score is positive, then we say that *Black wins*; if the score is negative, then *White wins*; if it is zero, then the position is a *tie*.

For example, consider the 11-vertex positions below, represented as strings in our usual way. (The first of these has been used as an example several times previously.) Below each string, we indicate who owns each unoccupied vertex: `B` for Black, `W` for White, and `n` for neutral.

```
uubbubwwwuu   Three gaps. Score = 3 − 2 = 1. Black wins.
BB..B....WW

uubuwuuuuuw   Three gaps. Score = 2 − 5 = −3. White wins.
BB.n.WWWWW.
```

`uubuwuuwuub` Four gaps. Score = 2 − 2 = 0. Tie.
`BB.n.WW.nn.`

`uuuuuuuuuub` One gap. Score = 10 − 0 = 10. Black wins.
`BBBBBBBBBB.`

`uuuuuuuuuuu` One gap. Score = 0 − 0 = 0. Tie.
`nnnnnnnnnnn`

**(9.8)** Using the Pumping Lemma for regular languages, prove that the language of positions that Black wins is not regular.

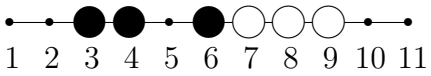
**(9.9)** (challenge) Prove or disprove: the language of positions that Black wins is context-free.

# 10 Turing machines

Having used 1D-Go for exercises on models of restricted types of computation — finite automata and pushdown automata — we now come to Turing machines, as general models of computation.

In 1D-Go, a chain is *vulnerable* if it has an unoccupied vertex at exactly one of its ends, with the other end being either the end of the path graph or adjacent to an opposite-colour stone.

For example, consider the following position.

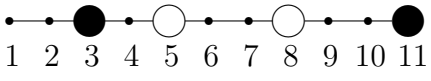



Here it is as a string.

`uubbubwwwuu`

In this position, the single-vertex black chain in the middle is vulnerable, as is the three-vertex white chain. But the two-vertex black chain on the left is not vulnerable, as it has two unoccupied neighbours.

In the following position, the only vulnerable chain is the single-vertex black chain at the right-hand end:




`uubuwuuwuub`

If a chain is vulnerable, then the opposite-colour player can *capture* the chain by placing a stone on the sole remaining unoccupied vertex next to the chain.

So, in the first position above, White could capture the single-stone Black chain by placing a White stone on vertex 5, while Black could capture the three-stone White chain by placing a Black stone on vertex 10. (In actual play, who gets to capture depends on whose turn it is. But in this exercise we do not worry about whose turn it is. We also ignore *Ko* rules for avoiding repetition in Go; since we are not actually playing the game here, these issues do not arise.) In the second example above, White could capture the single-vertex Black chain at the right end by placing a White stone on vertex 10.

When a capture is done, all the stones in the captured chain are removed, so those vertices become unoccupied. Here are the results of the three captures we have described.

| | |
|---|---|
| First position, White captures Black chain on vertex 6: | `uubbwuwwwuu` |
| First position, Black captures White chain on vertices 7–9: | `uubbubuuubu` |
| Second position, White captures Black chain on vertex 11: | `uubuwuuwuwu` |

**(10.1)** Build a Turing machine which takes, as input, a legal 1D-Go position (represented as a string over the alphabet {`b`,`w`,`u`} in the usual way) and carries out the capture of the leftmost vulnerable chain. (The first and third captures described above are of this type, but the second is not because the White chain is not the leftmost chain in the position.)

You do not need to check that the input position is legal; just assume that it is.

**(10.2)** Use your FA for testing legality (§7) to modify your TM from part (a) so that it *first* tests that the position is legal, and *then* captures the leftmost vulnerable chain. If the position is illegal, or legal but with no vulnerable chain, then the output position must be the same as the input position. If the position is illegal, then the TM must finish in the Reject state; if it is legal, it finishes in the Accept state.

**(10.3)** (challenge) Modify your TM from (b) so that it captures the *longest* vulnerable chain. If there are two or more vulnerable chains of equal-longest length, then it may capture any one of them (but only one of them).

All our exercises so far have been about 1D-Go. Standard Go is played on a 2D grid, so representing its positions as a one-dimensional input to an FA, PDA or Turing machine would sever some of the local connections and make computations complicated. But it is not hard to define two-dimensional Turing machines, and these may enable some of our exercises to be extended more naturally to standard two-dimensional Go. We have not tried this in our classes, but pose them here as challenges.

**(10.4)** (challenge) Write a 2D Turing machine to recognise whether a chain containing the initial location of the tape head is free or captured.

**(10.5)** (challenge) Write a 2D TM to recognise legal positions in standard $n \times n$ Go.

Our focus has been on the legality of 1D-Go positions, and using these in the Theory of Computation. But 1D-Go is also a game, and could serve as an instructive exercise in game-playing algorithms for students in artificial intelligence. I have not tried this, but offer

it as another challenge.

**(10.6)** (challenge)
(a) Write a 1D-Go playing algorithm. Implement your algorithm.
(b) Analyse the game, to try to determine which player wins with best play.

**Acknowledgements**

I thank the tutors in Theory of Computation at Monash University since 2015 for their work in helping students to do most of these exercises and learn from them, and the students for rising to that challenge. I am grateful to Harald Bögeholz, Michael Brand, Steven Kieffer, Ken Regan, Rebecca Stones and John Tromp for their comments.

# References

[1] BadukTV, http://www.tvbaduk.com/

[2] J. Fairbairn, Go in ancient China, London, 1995, http://www.pandanet.co.jp/English/essay/goancientchina.html

[3] G. E. Farr, The Go polynomials of a graph, *Theoret. Comput. Sci.* **306** (2003) 1–18.

[4] G. E. Farr, The probabilistic method meets Go, *J. Korean Math. Soc.* **54** (2017) 1121–1148.

[5] G. E. Farr and J. Schmidt, On the number of Go positions on lattice graphs, *Inform. Process. Lett.* **105** (2008) 124–130.

[6] E. Gibney, Google masters Go, *Nature* **529** (28 January 2016) 445–446, http://www.nature.com/news/google-ai-algorithm-masters-ancient-game-of-go-1.19234

[7] International Go Federation, *About the IGF*, 14 June 2010, http://intergofed.org/about-the-igf.html

[8] K. Iwamoto, *Go for Beginners*, Ishi Press, 1972, and Penguin Books, 1976.

[9] L. Johnson After 2,500 years, a Chinese gaming mystery is solved, *Motherboard*, 25 January 2016, https://motherboard.vice.com/en_us/article/vv7ejx/after-2500-years-a-chinese-gaming-mystery-is-solved

[10] C. Koch, How the computer beat the Go master, *Scientific American*, 19 March 2016, https://www.scientificamerican.com/article/how-the-computer-beat-the-go-master/

[11] C. Metz, AlphaGo's designers explore new AI after winning big in China, *Wired*, 27 May 2017, https://www.wired.com/2017/05/win-china-alphagos-designers-explore-new-ai/.

[12] G.-C. Pascutto, leela-zero, https://github.com/gcp/leela-zero, 10 May 2018.

[13] A. Rimmel, F. Teytaud, C.-S. Lee, S.-J. Yen, M.-H. Wang, and S.-R. Tsai, Current frontiers in computer Go, *IEEE Trans. Comput. Intell. AI Games* **2** (2010) 229–238.

[14] S. Shead, There is a worldwide shortage of the board game Go after Google DeepMind showcased it to 280 million people, 4 April 2016, https://www.businessinsider.com.au/go-board-game-in-demand-after-google-deepmind-put-it-in-the-spotlight-2016-4

[15] David Silver *et al.*, Mastering the game of Go with deep neural networks and tree search, *Nature* **529** (28 January 2016) 484–489.

[16] Y. Tian and L. Zitnick, Facebook open sources ELF OpenGo, https://research.fb.com/facebook-open-sources-elf-opengo/, 2 May 2018.

[17] J. Tromp, Number of legal Go positions, https://tromp.github.io/go/legal.html, Jan. 2016.

[18] J. Tromp, The number of legal Go positions, in: A. Plaat, W. Kosters and J. van den Herik (eds.), *Computers and Games: 9th International Conference, CG 2016 (Leiden, The Netherlands, 29 June – 1 July 2016)*, Lecture Notes in Computer Science 10068, Springer, 2016, pp. 183–190.

[19] J. Tromp and G. Farnebäck, Combinatorics of Go, in: H. J. van den Herik, P. Ciancarini, and H.H.L.M. Donkers (eds.), *Computers and Games: 5th International Conference, CG 2006* (Turin, 29–31 May 2006), Lecture Notes in Computer Science 4630, Springer, Berlin, 2007, pp. 84–99.