Becky
an `sc` AI

Geoffrey D. Bennett

April 21, 2005

1 Introduction

This is a report on Becky, an `sc` AI written by Geoffrey D. Bennett which has been entered into the `linux.conf.au` 2005 HackFest competition in the "AI" category.

The AI category involves writing a computer program to automatically (that is, without human intervention) play a game of `sc`. `sc` is a clone of the original Spellcast game available at `http://www.eblong.com/zarf/spellcast.html`, and was reimplemented by Chris Yeoh with a client-server protocol suitable for interfacing AIs with (the original Spellcast game ran as a single process connecting to multiple X servers).

2 Spellcast Introduction

The Spellcast manual describes it as "a game concerning the imaginary conflict between two or more powerful wizards in a duel of sorcery. The opponents perform magical gestures with their hands to create their supernatural weapons — spells".

Spellcast is a turn-based game where each wizard chooses their gestures simultaneously. Once each wizard has chosen their gestures and made any further decisions necessary (for example, choosing the targets for any spells invoked), the gestures are revealed to each player and the effects of spells resolved simultaneously.

The object of the game is to kill every other wizard playing so that your wizard is the last one alive. Each wizard starts with 15 hit points, so once 15 points (or more) of damage have been inflicted, that wizard is dead.

Wizards can also create "monsters" which will inflict damage on the creature (any wizard or other monster) specified by the monster's owner.

2.1 Gestures

Five basic gestures — "digit" (also known as "point"), "fingers", "palm", "snap", "wave" — can be performed with either hand, plus "clap", and "knife" (also known as "stab"). A wizard can also do "nothing" with either or both hands, although this usually does not need to be considered.

In this document, the gestures are represented as follows:

Digit (Point)
Fingers
Palm
Snap
Wave
Knife
Clap

This results in a possible 35 sensible combinations of gesture-pairs that need to be considered in each turn:

							X
							X
			X				X
							X
							X
						X	X
	X	X	X	X	X	X	

The entries marked "X" above are because clapping requires both hands (a half-clap gesture can be made, but is useless), the wizard has only one knife, and is an immediate "surrender" resulting in the player losing that game.

In certain situations (such as the first move) we do not need to consider gesture-pairs which are mirror images of each other (such as vs.) — this reduces the possible combinations that need to be considered down to 20:

		X	X	X	X	X	X
			X	X	X	X	X
			X	X	X	X	X
					X	X	X
						X	X
						X	X
	X	X	X	X	X	X	

2.2 Spells

After completing a certain sequence of gestures, various spells may be cast. For example, the gestures or make up the "Cause Light wounds" spell which will inflict two points of damage, and the gestures or make up the "Protection From Evil" spell which will protect from monster attacks, "Missile" spells, and from stabs by wizards.

Most spells require a target to be specified, which may be one of yourself, another wizard, or a monster.

The surrender ( ) and knife (or) gesture combinations are described in the Spellcast manual as not being spells, but for the purpose of the AI implementation there is no reason to not consider them as spells.

All the gestures for one spell need to be made with the same hand (although some spells require both hands to be used), but otherwise, any spell can be made with either hand.

Two spells can be made at the same time provided that neither of them requires two hands, and that the surrender gesture (✋✋) does not get made. Also, if the gestures finishing one spell are the same as the gestures that start another spell, fewer turns are required in order to invoke the two spells than otherwise would be the case. For example, after invoking the "Missile" spell (✌☝) the "Amnesia" spell (☝✋✋) can be cast in only two further turns (✋✋).

In the Spellcast game, there are 43 different spells which can be cast, but `sc` does not have implementations for "Haste", "Time Stop", "Delayed Effect", or "Permanency", and so Becky does not have them implemented either.

3 Becky Implementation

To choose what move to make at each turn:

- given our current state (as supplied by the `sc` server) generate the list of possible moves that we can make
- given the other wizards' current states (again, as supplied by the `sc` server) generate the list of possible moves that the other wizards can make
- determine the new wizard states for each combination of moves
- calculate a score for each of these new states that is indicative of the relative strength of the wizards (more positive meaning that we are winning, more negative meaning that we are losing)
- determine which move of ours will maximise the minimum score resulting from all our opponent's possible responses (ie. the classic minimax algorithm)

3.1 Move Generation

Each move that gets played by a wizard during a turn is made up of not just the chosen gesture-pair, but also:

- whether any spells can be invoked, and if so what spells should be invoked and what creatures they should be cast at, and
- has the wizard been paralysed or is the wizard paralysing another wizard.

3.1.1 Gesture-Pairs

Four lists of possible gesture-pairs are pre-generated (see `gestures.h` and `gestures.c`):

1. `poss_gp[]` which contains the 35 standard gesture pairs,
2. `poss_gp_noswap[]` which contains the 20 gestures that should be considered at the start of the game or when mirrored gestures do not need to be considered, and

3. `poss_gp_para[]` which contains all the gestures from `poss_gp[]` as well as the less-useful gesture-pairs containing "nothing" and "half-clap" (which may be forced on the wizard even though they would not normally perform those gestures by choice).

4. `poss_gp_confused[]` which contains all the gestures from `poss_gp[]` except for those that contain ✋ or 🤚, as it is dangerous to play those gestures when confused.

3.1.2 Spells

A given sequence of gesture-pairs may invoke in a turn exactly zero, one, or two specific spells, or there may be a choice of what spells to invoke in some cases. For example, the gestures for "Cause Light Wounds" (🤜✊✋) end with the gesture for a "Shield" spell (✋), and only one of them may be invoked.

`spells.c:get_spellpair_list()` will return a list of possible spell-pairs (ie. the spells that the left and right hands can invoke) given a list of gesture-pairs.

When a choice of spell needs to be made for a given gesture-pair, each choice is treated as a separate move to consider even though the actual gesture-pairs are the same.

In some cases where there is a choice of spells, the choice can be predetermined. For example, the "Finger of Death" spell ends with 👉☝ which is the "Missile" spell, but there is no point considering to use the "Missile" spell in this case. In `spells.c` this is termed "trumping", where the "Finger of Death" spell trumps the "Missile" spell, and `get_spellpair_list()` will not return the "Missile" spell as one that can be invoked.

Most spells would only be invoked on either ourselves or on an opponent (`spells.c:spell_who[]` lists who is the usual target of each spell), but some spells can sensibly be invoked on either ourselves or an opponent. In this case, the choice of targets is treated as two separate moves.

Some spells can be cast on monsters as well as wizards; in these cases, if there is a monster in play, there is a further choice of whether to cast the spell on a wizard or a monster which again is treated as two separate moves.

3.1.3 Paralysis

If one wizard has invoked the paralysis spell on another wizard, the casting wizard has a choice of which hand to paralyse in the following turn. This choice is independent of what gestures we make, so it doubles the effective number of our possible moves, but greatly reduces the possible moves by the opponent.

3.2 Determining the New States

Given:

- the current state (number of hit points, previously-played gesture-pairs) of each wizard,
- each move that we can play, and
- each move that our opponent can play,

we need to determine new resultant states for each wizard. This is implemented in the `moves.c` functions:

- `apply_spells()` — which determines in what order the spells should be applied and calls `apply_spell_to_wizard()` or `apply_spell_to_monster()` as appropriate. The Spellcast documentation says that all spells are resolved "simultaneously", but their implementation of "simultaneously" is indistinguishable from applying the spells in the order defined in `spells.h` `spell_order[]`.
- `apply_no_spells()` — handles the timing out of spells once they have expired.
- `apply_spell_to_wizard()` — a large switch statement which makes the appropriate state changes to wizards.
- `apply_spell_to_monster()` — a smaller switch statement which makes the appropriate state changes to monsters.

3.3 Score Calculation

For each of the new wizard states, a score needs to be calculated to determine if this is a move that we should be making (as this is a zero-sum game, a single value suffices for the score). As Becky does a tree search, there are dynamic and static score calculations.

A static score calculation is performed when at a leaf of the tree (either because we're not searching any further down in the game tree, or because we have won or lost). This score is calculated by summing a score that is generated for each wizard which takes into account:

- if the wizard is us (no change) or an opponent (multiply the wizard's score by -1),
- the damage that has been inflicted on that wizard,
- the number of hit points of damage the monsters owned by that wizard can cause per turn, and
- a bonus or penalty for various wizard states (presence of fear, disease, invisibility, etc.).

A dynamic score calculation is performed at non-leaf nodes of the tree, and is calculated by doing a minimax search which results in us attempting to maximise our score regardless of what move they play. It is assumed that our opponent wizards will always play their best move (ie. attempt to minimise our score regardless of what move we play) and that they use the same algorithm as we do to determine what a good move is. Further details on this are in a following section.

3.4 Possible Moves Representation

Unlike games such as chess where players alternate moves, spellcast wizards choose their moves simultaneously. Becky treats the possible moves from each turn as a 2D array with the different rows representing our different moves, and the different columns representing our opponent's different moves.

For an example of this, look at the below table. Along the top are their possible moves (0–19), and along the left side are our possible moves (0–19) with the resultant score after playing those two moves shown in the body of the table. On the right hand side, the "min" column is the minimum score in that row (ie. the "worst" that our opponent can give us).

Our best move will therefore be the one that maximises the minimum score, which in this case is any of moves 3–19.

Them	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	min
Us 0	0	0	-3	-2	0	2	-1	0	0	2	-1	0	0	-1	-1	-1	0	0	0	2	-2
1	-3	-2	-3	-1	-1	-1	-1	-2	0	0	-2	-2	0	0	0	0	0	0	2	2	-3
2	0	0	-1	1	-1	1	2	2	1	2	3	3	1	3	1	1	3	3	5	5	-1
3	0	0	1	1	0	2	1	2	2	2	3	2	3	2	2	4	4	2	4	4	0
4	0	0	0	0	0	0	0	0	0	0	0	0	0	2	0	0	0	0	2	2	0
5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
6	0	0	3	0	0	0	3	0	0	0	0	3	3	0	0	3	3	3	3	3	0
7	0	0	1	1	0	0	0	0	0	1	1	2	2	2	2	2	2	2	2	2	0
8	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
9	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
10	0	0	0	0	0	0	0	0	0	0	0	1	1	3	3	3	3	1	3	3	0
11	0	0	0	0	0	0	0	1	0	1	1	0	0	0	2	2	2	2	2	2	0
12	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
13	0	0	0	0	0	0	0	0	1	1	1	0	0	0	3	3	3	3	3	3	0
14	0	0	0	0	0	0	0	0	1	1	1	0	0	0	2	2	2	2	2	2	0
15	0	0	0	0	0	0	0	0	1	1	1	0	0	0	3	3	3	3	3	3	0
16	0	0	0	0	0	0	0	0	1	1	1	0	0	0	2	2	2	2	2	2	0
17	0	0	0	0	0	0	0	0	0	0	0	1	1	1	2	2	2	2	2	2	0
18	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
19	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

3.4.1 Cut-off Optimisation

If the scores are calculated in the above table left-to-right followed by top-to-bottom, then after one row of scores has been calculated, a lower bound on the score has been established (in the above case, this is -2). Therefore when the next move gets calculated (score -3), it is apparent that no further moves in that row need to be examined. Note that before the first row of scores has been calculated, the lower bound is considered to be $-\infty$.

After calculating the scores for the third row, the lower bound will be reestablished at -1, and then in the fourth row, the lower bound will be 0.

In the fifth row after calculating the first column score, we can stop examining further moves in this row depending on whether we want to find all possible best moves or whether just one of the best moves is required. Currently only one of the best moves is determined.

In either case, because the tree levels below the first are only used to establish a score for that branch, only one of the best moves need to be determined.

3.4.2 Cut-off for Lower Depths

If not at the leaves of the tree, then once the first row of scores has been calculated, the lower bound can be passed to the lower depths of the tree so that the initial lower bound can be higher than $-\infty$.

3.4.3 Move Ordering

If we search their moves in order from best-for-them to best-for-us, and search our moves in order from best-for-us to best-for-them, we'll be able to determine the best move (in this particular case) in only $20 + 19 = 39$ calculations rather than $20 \times 20 = 400$.

Of course, if we already know what the best move is, we wouldn't need to be doing any of this, but some guesses can be made as to what moves should be tried first. For example, a move that completes a spell is more-likely than other moves to be found to be a best move.

Also, a move that has been found to be good for our opponent given one of our moves is likely to also be good for them against other moves of ours. Therefore when each score is calculated, if it is apparent that this one of the opponent's move should have been checked earlier, it is moved up the list for next time.

3.4.4 Tree Search

Becky implements a depth-first search of the game tree with iterative deepening (DFID). That is, it first does a search of the game tree considering only the current move (depth 0). It then repeats the depth-first search considering the current move and one move ahead (depth 1), and this repeats until the allocated time for the move selection has been reached.

As the tree expands exponentially with a high branching factor, the time taken for all of the depth $0 \ldots n-1$ searches is negligible when compared to the time taken for the last depth n search, so the "wasted" searches have a negligible negative effect, but there are two strong advantages to using DFID in this case:

- regardless of the allocated time, provided that a depth-zero search has been done (which takes nearly no time at all), a "best move as far as we know so far" is always available so the search can be stopped at any time, and
- it is assumed that the best move at depth n is often going to be the best move at depth $n+1$, so that move orderings from the depth n search are cached and reused to speed up the depth $n+1$ search.

4 Handling Multiple Wizards

So far, this document has only discussed play with two wizards. To incorporate play with more than two wizards, the 2D array of our possible moves vs. our (one) opponent's possible moves is extended to consider our possible moves vs. each of our opponent's possible moves. Doing this increases the number of move combinations that need to be checked from n^2 to an^2 where a is the number of opponents and n is the number of possible moves for each opponent. Opponent interactions are not considered as that would instead result in n^{a+1} move combinations to check.

The minimax algorithm is used as before, except that the minimum is taken over all possible (separately considered) opponent moves. This seems to give a good balance of time usage vs. intelligence.

5 Implementation Optimisations

5.1 Gesture Representation

As there are less than 16 gestures, they are internally represented in 4-bit nybbles. A gesture-pair can therefore be represented in 1 byte.

Since the maximum length of a spell sequence is 8 consecutive guestures, each wizard's gesture history (called a gesture-list) will fit within 8 bytes (a long-long integer).

In order to efficiently determine what spells can be performed with a given set of gestures, two things are done:

- Each spell is stored as a long-long value and mask pair for quick determination (one logical "and", and one compare) of whether any gesture matches a particular spell.
- An array of all possible last-two gesture values (65536) is pre-generated, with each array element containing a list of the spells that could end in those two gesture values. The most number of spells that can match is 5 (Cure Light Wounds plus Summon Goblin/Ogre/Troll/Giant which all end in ✋☇), so with one table lookup and up to 5 "and"/"compares", a list of all possible matching spells can be found for any gesture-list.

5.2 Monsters

Rather than considering monsters individually, Becky aggregates the remaining hit points and total maximum hit points for each monster per wizard owner. This means that the the number of moves combinations that need to be considered stays constant regardless of how many monsters a wizard owns.

5.3 Score

Part of the static scoring algorithm includes a factor based on various states that are tracked for each wizard. As there are only 15 states that are used, a 2^{15} entry lookup table is pre-generated. As the static score needs to be calculated at practically every leaf of the game tree, it is important that this be fast.

5.4 Hash Table

In order to retain the move list ordering between different iterations of the tree search, a hash table based on the wizard state is used. Entries in the hash table are locked as the tree is searched so that entries can be sorted in-place within the hash table without needing to care about a collision replacing a shallower entry with a deeper entry.

6 SMP Support

SMP support has been partially implemented (and may not work); in order to enable it, you can uncomment the `#define SMP` near the top of `moves.c`. When `init_moves()` gets called, it will fork `NUM_WORKERS` children which will be used to calculate the scores for levels one and below of the game tree. This theoretically would result in nearly a linear improvement in speed, but the hash table is not shared between the children, and the move reordering has not been implemented, so is likely to run slower.

7 Suggestions for Improvement

7.1 Mixing up our Moves

Rather than choosing the first move with the best score, if more CPU time is available, the cut-off algorithm could be changed (just for the top level of the tree) so that all equally-best moves are found, and then one is randomly chosen.

7.2 Predicting Opponent Moves

If there are opponents that are deterministic, tracking the WizardState vs. the played move would give a significant advantage. By retaining state between games, it could easily be deduced whether or not particular opponents are deterministic.

7.3 Handling Smarter Opponents

If an opponent wins against Becky (and, again, is determined to be a deterministic AI), it is possible that playing the starting moves that won for the other opponent may help narrow the margin between Becky and the other AI.

8 Acknowledgements

The gesture images included in this document are taken from Spellcast, which contains this copyright notice:

> The original paper-and-pencil version of this game was created by Richard Bartle (`76703.3042@compuserve.com`). This implementation is by Andrew Plotkin (`ap1i+@andrew.cmu.edu`). It is copyright 1993 by Andrew Plotkin. The source code may be freely copied, distributed, and modified, as long as this copyright notice is retained. The source code and any derivative works may not be sold for profit without the permission of Andrew Plotkin and Richard Bartle.