Recitation 10: Leiserchess Test Infrastructure

This recitation will help you familiar with the final project code base. You will extend the project test infrastructure so that you can make optimizations to the Leiserchess board representation and check that those optimizations preserve output of the search algorithm.

Note: You need not work with your project partners for this recitation, despite it being about the project code. If you don’t work with your partners, be careful about committing the changes that you make to the code to your group’s Git repo, as you might create conflicts with other group members (they are also doing this recitation after all). We suggest that you get together with your group afterwards to decide how to merge your changes.

1 Getting started

We recommend that you work on the course machines:

```
$ ssh <username>@cloud
NcsailNmitNedu
```

If you have not done so already, clone a copy of your project team’s Git repo:

```
$ git clone /afs/csail/proj/courses/6.172/student-repos/fa14/
> projects/project4/<team-name>.git project4
```

If you do not yet have a project 4 group, you should get a project 4 group. In the meantime, however, you can clone a copy of the project 4 code from GitHub:

```
$ git clone git@github.com:mitV1WR/projectTNgit projectT
```

2 Testing scout search

As discussed in lecture, scout search performs three main steps for each node\(^1\) in the search tree:

1. First, it tries to guess which of the legal moves from that node is in the principal variation. Scout search will make this move first and evaluate its score using normal alpha-beta search.

2. To verify whether the move from Step 1 was really in the principal variation, scout search then performs null/zero-window searches on the other children.

3. If one of these children results in a higher score than the principal variation move, the algorithm must have guessed the principal variation incorrectly and falls back to performing normal alpha-beta on that child. Otherwise, the algorithm propagates the score for the principal variation up the tree.

\(^1\)A node is a board position.
This algorithm works well in practice, if the game guesses the principal variation correctly most of the time, because null-window searches are (relatively) cheap.

To help you test your scout search algorithm, we have included a test harness in `search.c` and `search_mod.c` that (conceptually) performs the following check in `searchRoot()` and `searchPV()`, respectively:

```c
for each move m in list_of_moves:
    ...
    if (TEST) {
        game_state_reference = copy(game_state)
    }
    score = scout_search_mod(game_state, -alpha, ...)
    if (TEST) {
        score_reference = scout_search_ref(game_state_reference, -alpha, ...)
        assert (score_reference > alpha) == (score > alpha)
    }
    ...
```

See the section of the code labeled “Test framework assertions” in `search.c` for the implementation. This test checks whether your scout search code (`scout_search_hi` in `search_mod.c`) produces a different number of beta-cutoffs than a reference code (`scout_search_hi` in `search_ref.c`), and triggers an assertion in that case. This type of check is invaluable in discovering whether some optimization you make to your search routine is changing (breaking) functionality.

- **Exercise:** Enter the `player/` directory and compile the program with `make test`. To quickly check that the test harness is working, make some changes (it’s up to you) to `scout_search_hi` in `search_mod.c`. Run the program and type `go` to trigger scout search. If your change created a different number of beta-cutoffs, you will see a message like this:

```
Beta-cutoff MISMATCH! [expected = -517, actual = -521, alpha = -518, depth = 1, mv index = 2]
```

Keep making changes until you trigger the assertion.

In the pseudocode, `score > alpha` and `score_reference > alpha` should be interpreted as “did a beta-cutoff happen?” We compare against `alpha` because our scout search is coded like a Negamax algorithm: `alpha` to a parent node is `beta` to the child node.

- **Exercise:** Why does the assertion check `(score_reference > alpha) == (score > alpha)` as opposed to `score_reference == score`? Hint: look at how the parallelized code uses `do_abort()` from `abort.h`.
3 Extending the test harness

Although we have gotten you started with the test harness, it isn’t as useful as it could be. In particular, `scout_search()` in `search_ref.c` calls the same functions in `move_gen.h` as `scout_search()` in `search_mod.h`, meaning that both your code and the reference code must share board representations. Why is this a problem? If you optimize `move_gen.c` and introduce bugs along the way, those bugs will propagate to both reference and your code — and the beta-cutoff assertion likely won’t catch the problem.

- **Exercise:** What other functions are shared by the implementations of `scout_search()` in `search_ref.c` and `search_mod.c`?

For the rest of the recitation you will change the test harness so that `search_mod.c` and `search_ref.c` can use different board representations. To give you some idea of how you might do this, suppose that you want to preserve the reference implementations of `low_level_make_move()` and `make_move()` in `move_gen`. Here is a sketch of what you can do:

1. Copy the declarations of `low_level_make_move()` and `make_move()` from `move_gen.h` into a separate file, such as `move_gen_ref.h`, and copy their definitions from `move_gen.c` into a separate file, such as `move_gen_ref.c`.

   *Note:* In general, you should be very skeptical about what needs to be copied. For example, `move_gen.h` has constants that should be consistent across all reference/optimized codes.

2. Rename the functions within `move_gen_ref.c` and `move_gen_ref.h`, and rename the calls to those functions in `search_ref.c`. You can use preprocessor macros to vastly simplify this process. If you add these macros to the top of each file:

   ```c
   #define low_level_make_move low_level_make_move_ref
   #define make_move make_move_ref
   ```

   and these macros to the bottom of each file:

   ```c
   #undef low_level_make_move
   #undef make_move
   ```

   Then all instances of `low_level_make_move` and `make_move` within the file will be properly renamed.

3. Add `#include "move_gen_ref.h"` to the bottom of `move_gen.h`, and add `#include "move_gen_ref.c"` to the bottom of `move_gen.c`, if they are not already present.

- **Exercise:** Demonstrate to your TA that, by changing a function from `move_gen.h`, you can trigger a Beta-cutoff MISMATCH! assertion failure.

Once you actually changed the board representation from the provided implementation, you will need to change `pos_to_fen()` in `fen.c` to handle your optimized board representation as
input. As described in the “Test framework assertions” section of search.c, right before the call to scout_search_ref(), we translate your optimized board representation to FEN (which serves as an intermediate string representation) and then from FEN back to the reference board representation. This allows the reference code to only ever have to interface with the reference board representation.

Using the FEN translation will help you minimize the number of changes you have to make to support multiple move_gen implementations. For example, scout_search() also calls eval() (which takes a board position as input) from eval.h. Yet, eval() takes as input your optimized board representation, not the reference version. An elegant way to deal with eval() is to use the FEN conversion in the opposite direction: right before the eval() call, translate the reference board representation back to the optimized representation.

- **Exercise:** The fact that scout_search() calls eval() is problematic. One potential solution is to follow the process sketched above to maintain a “reference” version of the eval code. Why will this not work in the context of checking the beta-cutoff assertion?

If you finish early, feel free to start optimizing your move_gen implementation. Recall from the code walk that the board border (i.e., ARRAY_WIDTH in move_gen.h) is 16 on a side for two reasons. First, calculating adjacent rows can be done using cheap, shift operations. Second, we can use sentinel values to check whether pieces move off the board if there is at least a 1-row 1-column buffer between the end of the array and the end of the board. Because we are using an 8 × 8 board this semester, it would be preferable to get a 64-bit board representation. Use the test harness to debug your logic to handle board edge cases.

You can also start optimizing scout search itself. The sorting kind of feels like overkill, don’t you think?

---

2This tends to be true in practice for several reasons. First, as you write more versions of the move_gen code, you will find that there is a canonical style of program that will convert it to FEN (as opposed to converting directly between two representations). Second, the sheer number of translation functions becomes linear with the number of move_gen implementations. Suppose that you want to convert between N move_gen implementations. You need N² translation functions (in the worst case) if you don’t use FEN, but only 2N functions (N to get to FEN and N to get from FEN) if you do use FEN.