Blackjack++

making counterfactual regret minimization more accessible

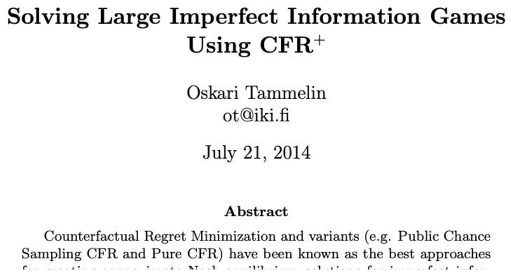
Mateo Maturana
Marcus Fong
Erin Liang

Intro

Motivation ++ Goals ++ Background Information

Motivation

- Algorithmic basis for the bots dominating annual poker competitions is **counterfactual regret minimization (CFR)** [1]
- Regret-based algorithms are nascent (~2014 for CFR+)
  - **Few materials available** to introduce students, researchers, practitioners
  - Best way to learn game strategy + CFR is to see how AI's optimal strategy iterates throughout a game



Solving Large Imperfect Information Games Using CFR$^+$

Oskari Tammelin
ot@iki.fi

July 21, 2014

Abstract

Counterfactual Regret Minimization and variants (e.g. Public Chance Sampling CFR and Pure CFR) have been known as the best approaches

Motivation

- Why C++?
  - CFR Games → large game trees
  - CFR requires a lot of iterations to converge
  - Object-Oriented → conceptually intuitive to understand
  - modern C++ opportunities: coroutines and modules
- Applying game theory class learnings! 🎉

Goals

- Design CFR decision tree class using the algorithm described in Tammelin’s “Solving Large Imperfect Information Games Using CFR+”
- Need an easy-to-understand game to apply CFR to → Blackjack
  - Blackjack is a game of imperfect knowledge, which makes it the perfect game to apply CFR
- We want modify an existing game to ensure that we don't repeat work that has already been done
- Design the game using modern C++ techniques

CFR Overview

- Recursive algorithm that attempts to minimize regret at each iteration vs our own current strategy
- Strategy at each iteration is computed using RegretMatching
    - Definition: Regret for Action A is defined as the amount we “regret” not doing this action. Mathematically, this is Utility[A] - Utility[current mixed strategy]
    - Strategies are then computed directly proportional to regret
    - Ex: The current utility of our strategy is 3. Utility of A is 7, Utility of B is 5, Utility of C is 1. Therefore the regret of A is 4, B is 2, and C is 0 (regret has a floor of 0). Thus, our strategy is the next iteration will be to do A with ⅔ frequency, B with ⅓ frequency, and C with 0 frequency.
- Final strategies are taken by averaging the strategy at each iteration
- Overtime, the average strategy will converge to nash equilibrium

CFR Overview: CFR+

- In order to reach faster convergence, we use CFR+ in our implementation.
- RegretMatching+
  - Instead of computing the regret at each individual iteration, we keep a sum of all regrets so far
  - Strategy is computed proportionally to the regret ***sums***
- Weighted Strategy Sums
  - Recall that in CFR, the final strategy is computed using the average strategy from each iteration
  - However, shouldn't strategies in later iterations be closer to equilibrium?
  - Weighted Strategy Sums weight later iterations more for the final strategy
  - strategySum[i] += curStrategy[i] * iterationNum
  - Sums are normalized after all iterations are complete
- CFR+ can help reach convergence orders of magnitude faster

Blackjack++ Overview and Rules

- All players and the dealer start with 1 card face up, 1 card face down.
- Players take turns choosing to either **hit/stand**. Once both players stand, their action is over
  - **Hit:** A random card is drawn and given to the player
  - **Stand:** The player indicates they no longer want any cards
- After player action is over, the dealer draws their cards according to an established protocol (hit if <17, stand if >= 17).
- The player/dealer with the highest sum is awarded the pot, Ace can be either 1 or 11.

Tutorial

How to Build and Run ++ “Hello World” ++ Common Errors

Usage: How to Build and Run

We included a Makefile to make building Blackjack++ easier:

```
$ make
c++  -I/usr/local/opt/ncurses/include  -c -o src/Cfr.o src/Cfr.cpp
[...]
```

To run a game of Blackjack++ to save under `sol` where all players start with a 10 with a maximum of 1 hit (players can draw up to 1 more card) and 10000 iterations of CFR:

```
$ ./a.out “10, 10, 10” 1 10000 sol
```

In the general case, we would run as follows:

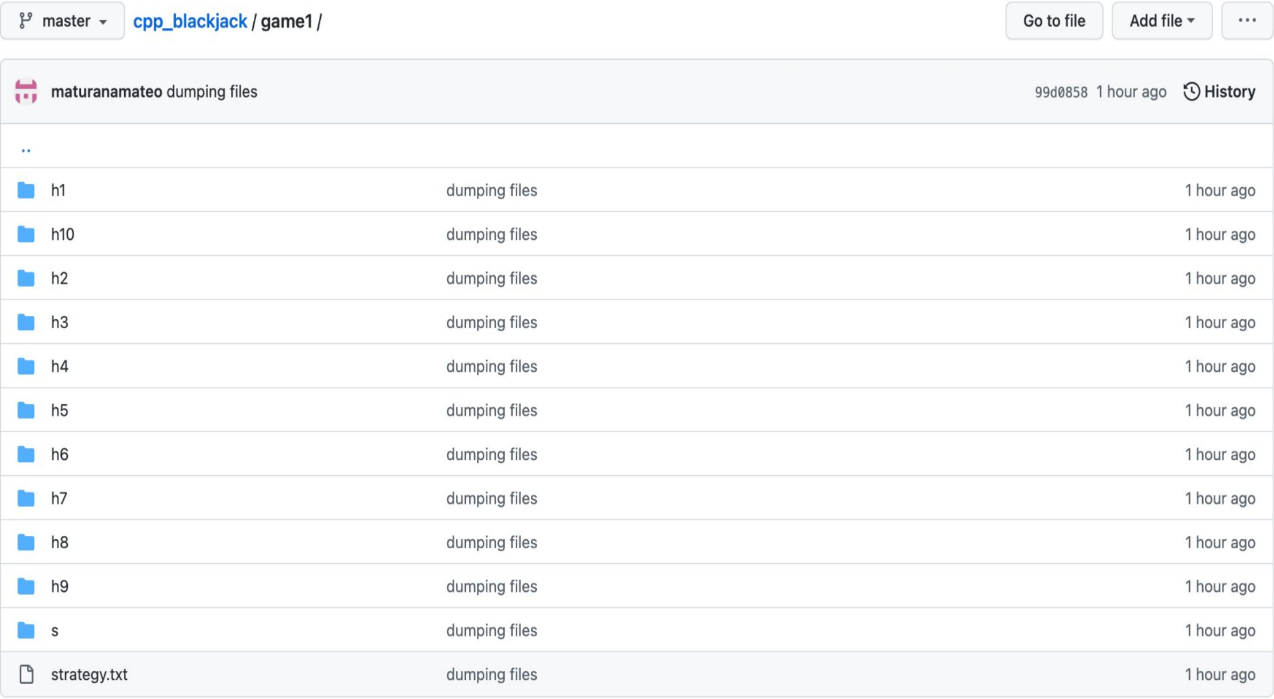
```
$ ./a.out “[player0 card], [player 1 card], [dealer card]” [maxHits]
[numIterations] [dump directory]
```

Blackjack++ Game Example

- How does our version of Blackjack work...
- Game start:
    - My deck: {5 (d), 3 (u)}
    - Other deck: {2 (d), 10 (u)]
        - I can only see the upcards of other players
    - I Hit and draw a 8: {5 (d), 3 (u), 8 (u)}
        - Because I saw that he has a 10 and I have an 8, I might as well hit because I never bust at this point level
    - P2 hits and draws a 8
        - Given that the other player hasn't busted, I know that the down card has to be a small number. I know that P2 has at least 2 more than me, so I might as well take the chance to win.
    - I Hit and draw a 5, putting me at 21 and a win

Blackjack++ Game Example Output (Directory Structure)

```
game1/  -
        | - strategy.txt
        |- s
            | - strategy.txt
            | - s
            ...
        |- h1
            | ...
        |- h2
            | ...
        | - h3
            | ...
        ...
```



Blackjack++ Game Example Output (Strategy File)

```
Current Acting: 1


Down Card: 1 || Stand: 0.000516931, Hit: 0.999483
Down Card: 2 || Stand: 0.000396455, Hit: 0.999604
Down Card: 3 || Stand: 5.89984e-05, Hit: 0.999941
Down Card: 4 || Stand: 1.70004e-05, Hit: 0.999983
Down Card: 5 || Stand: 4.04662e-05, Hit: 0.99996
Down Card: 6 || Stand: 0.0182729, Hit: 0.981727
Down Card: 7 || Stand: 0.055304, Hit: 0.944696
Down Card: 8 || Stand: 0.0703224, Hit: 0.929678
Down Card: 9 || Stand: 0.75498, Hit: 0.24502
Down Card: 10 || Stand: 0.990737, Hit: 0.00926331
```

Common Errors

Mostly errors with command-line arguments:

1. Setting `maxHit`s too high
   - maximum number of times that a player can request an extra card
   - would cause the game tree build time to be **very high** … foreshadowing for performance
   - Why? branching factor of ~100 for an increment of 1 in `maxHit`
2. Setting `cfrIterations` too high
   - number of iterations user wants CFR to take
   - due to CFR+, strategies converge fairly quickly

Design Manual

CFR ++ Game ++ Player ++ State Overview
(more details in the design doc)

`CFR` Class

- Runs all CFR relevant code
- `train()` - Runs all CFR iterations. For each iteration, generates a set of down cards
- `normalize()` - Normalizes strategy sums after all CFR iterations have finished
- `runCfr()` - Computes utilities and regrets
- `getStrategy()` - Uses RegretMatching+ to get the current strategy
- `getTerminalNodePayoffs()` - Generates payoff matrix for any terminal node
- All objects taken by reference. They are produced in tree creation, so want to avoid making unnecessary copies.

`Game` Class

- Defines the logic for the Blackjack++ game
- Starting from `main()`...
  - Game object created using command line args: `startingCards`, `maxHits`, `cfrIterations`
  - `constructTree()` — Builds game tree
    - Starts off larger Blackjack++ game logic
    - Keeps track of time and amount of nodes it takes to build the game tree for performance
- Other methods for debugging + pedagogical purposes
  - `printRandomPath()`
  - `dumpToFiles()`
- Alternative approaches?

`Player` Class

- Two Players, one dealer.
- What do we need to keep track of about the Player during Blackjack++?
    - Which player is this?
        - `id`
    - Is the player allowed to take any more actions?
        - `doneAction`
        - When is a player done?
            - cards sum > 20 (no feasible way they can take more cards because their minimum sum is 21)
- Constructed in Game class in `Game::constructTree()`

```
auto Player0 = Player(0, false);
auto Player1 = Player(1, false);
```

`State` Class

- What is a State? What does it represent?
- Where is our class used with respect to CFR algorithm?
- `State::State()`
- `State::populateChildren()`
- `State::createStandState()`
- `State::createHitState()`

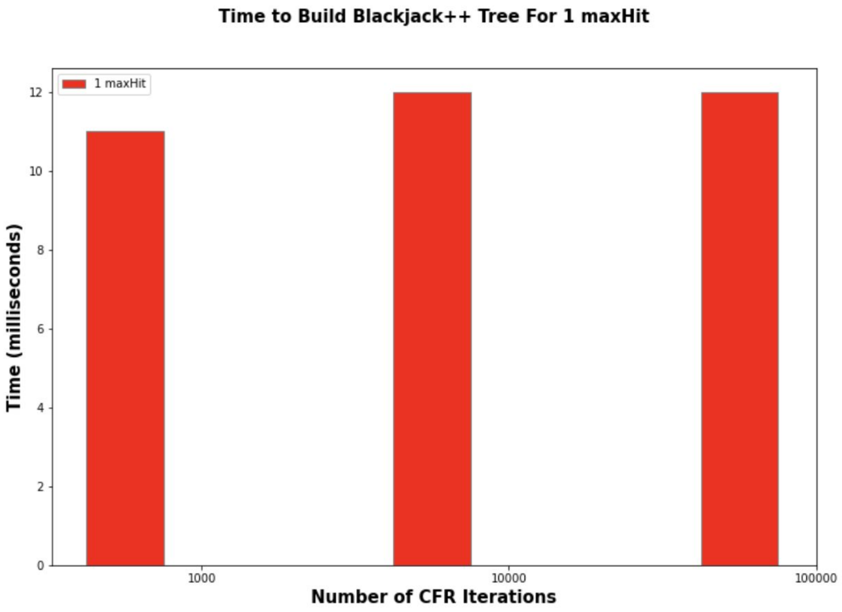
```cpp
std::vector<std::future<State>> futures;
futures.reserve(11);

futures[0] = std::async(std::launch::async, [this](){
    return State::createStandState();
});
for (int i = 1; i <= 10; i++) {
    futures[i] = std::async(std::launch::async, [this, i]() {
        return State::createHitState(i);
    });
}


for (int i = 0; i <= 10; i++) {
    if (i == 0) {
        const auto& state = futures[0].get();
        //state.printDetails();
        nextStates.emplace("s", state);
    } else {
        std::string formatted = "h" + std::to_string(i);
        const auto& state = futures[i].get();
        //state.printDetails();
        nextStates.emplace(formatted, state);
    }
}
```

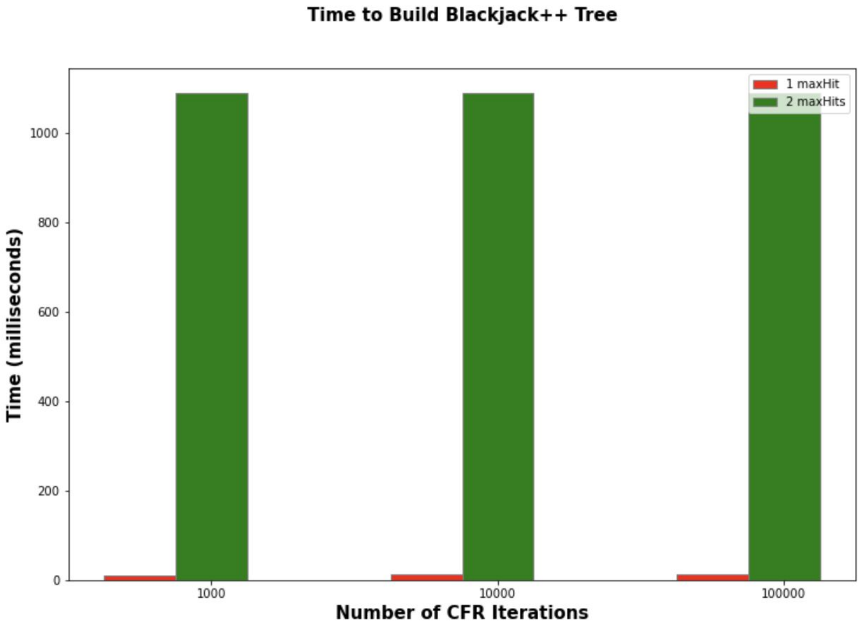
Performance

some pretty graphs!

Performance

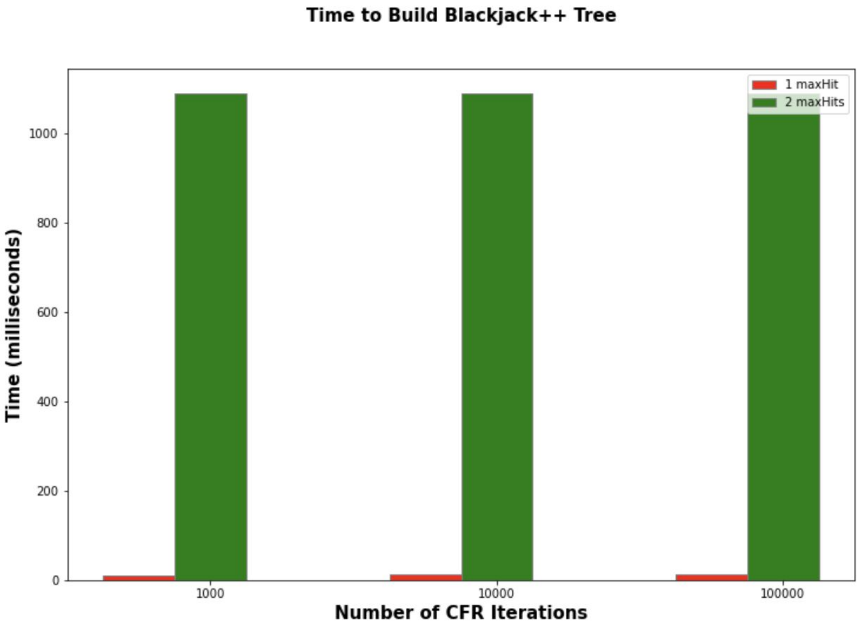
Impact of `maxHits` and `cfrIterations` on Game Tree Build Time

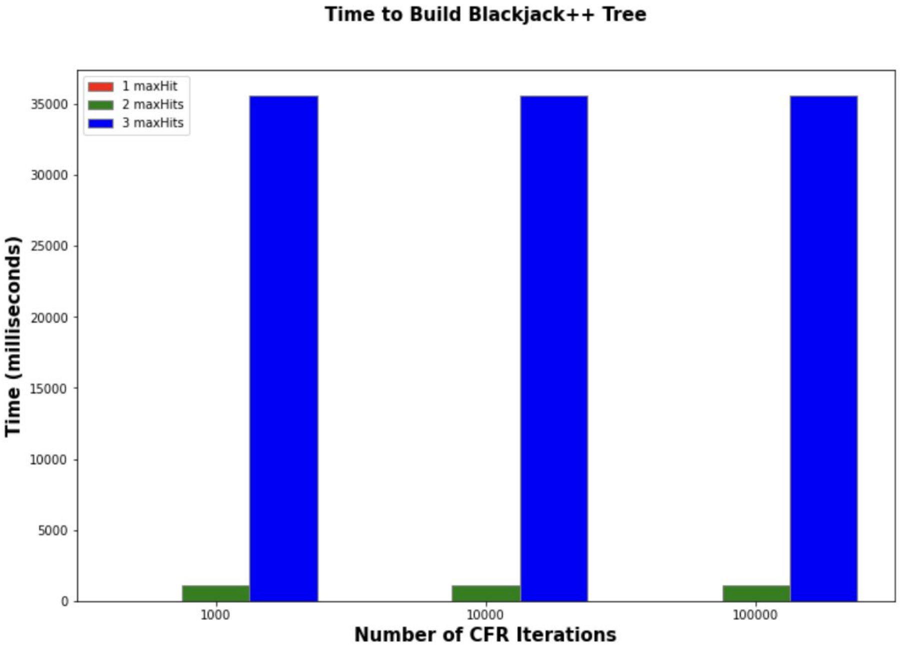




Performance

Impact of `maxHits` and `cfrIterations` on Game Tree Build Time

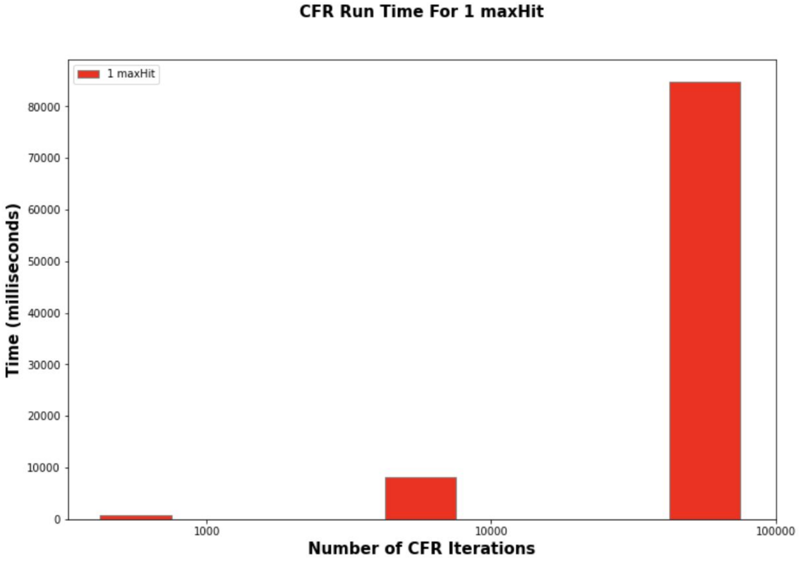


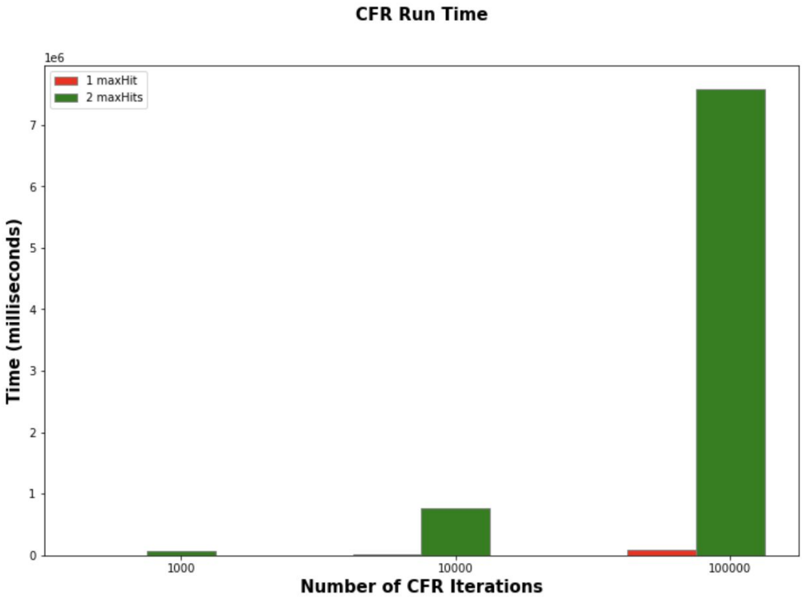


- factor of ~100 increase in build time when `maxHits` incremented —> branching factor
- no impact on build time when `cfrIterations` increased by factor of 10

Performance

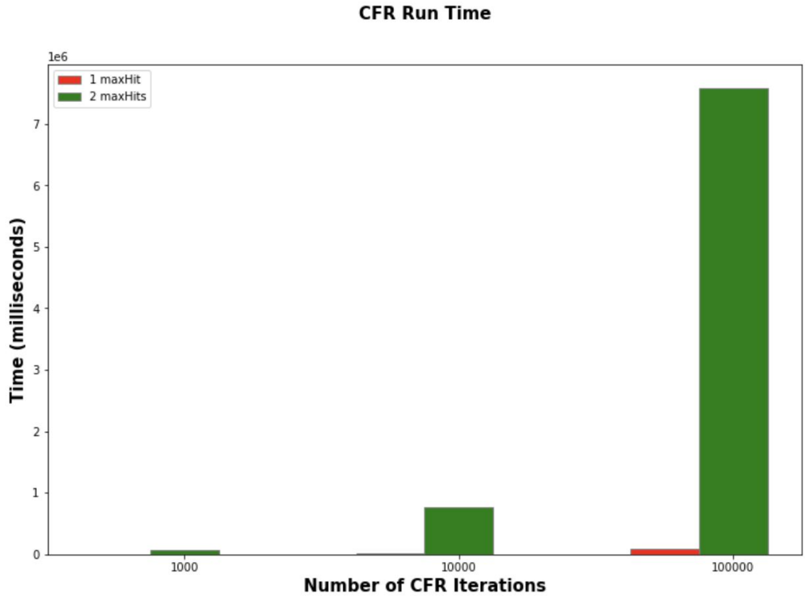
Impact of `maxHits` and `cfrIterations` on CFR Training Time

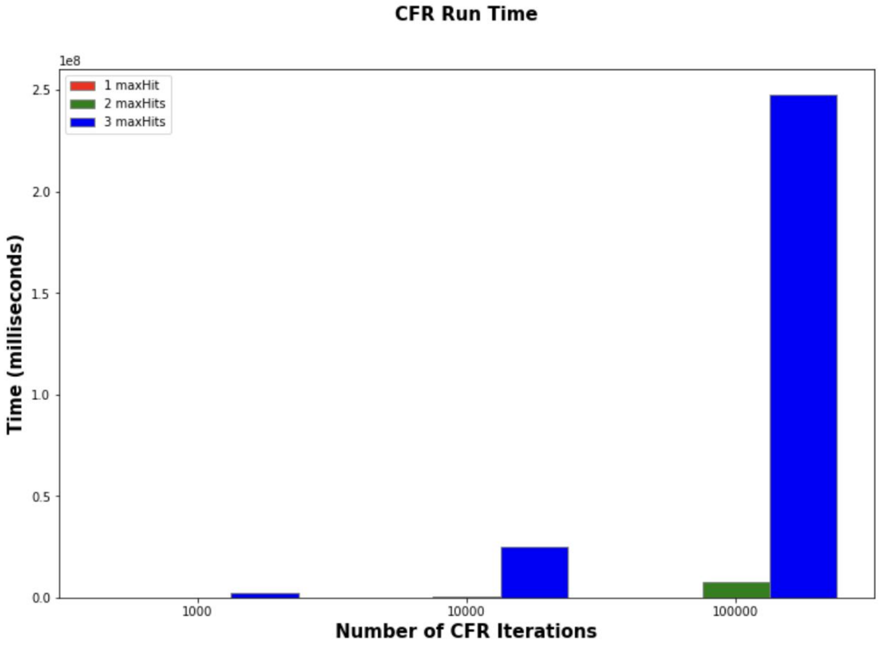




Performance

Impact of `maxHits` and `cfrIterations` on CFR Training Time





- factor of ~100 increase in CFR training time when `maxHits` incremented → branching factor
- factor of ~10 increase in CFR training time when `cfrIterations` increased by factor of 10

Post-Mortem

Reflections ++ Roadblocks ++ Future Work

Reflections + Roadblocks

- Current reflections
  - Positives
  - Negatives
- Integrating with modern C++
  - Coroutines
  - Modules

Future Work

- Interactivity
  - Being able to play vs the computed strategies
  - In theory, over the long run the computed strategies should not lose
- Generalizing to other Games
  - Building a CFR library that can build game trees for other types of games easily
- UI
  - Getting the current strategies requires moving through a lot of directories
  - In an ideal world, we would want players to easily find strategies for decision points
  - For example, a lot of online poker solvers have really good UIs (see next slide)

GTO Wizard
STUDY
PRACTICE
ANALYZE
UPGRADE
UPLOAD
CASH
6max
NL50
General
100 bb
UTG 100
FOLD
RAISE 2
ALLIN 100
HJ 100
FOLD
RAISE 2
ALLIN 100
CO 100
FOLD
RAISE 2.3
ALLIN 100
BTN 100
FOLD
RAISE 2.5
ALLIN 100
SB 99.5
FOLD
CALL
RAISE 12
ALLIN 100
BB 99
FOLD
CALL
RAISE 13
ALLIN 100
FLOP 5.5
J T 7
BB 97.5
CHECK
BET 1.8 (33%)
BTN 97.5
CHECK
BET 1.8 (33%)
BET 2.75 (50%)
BET 4.1 (75%)
BET 6.9 (125%)
STRATEGY
RANGES
BREAKDOWN
REPORTS
AA AKs AQs AJs ATs A9s A8s A7s A6s A5s A4s A3s A2s
AKo KK KQs KJs KTs K9s K8s K7s K6s K5s K4s K3s K2s
AQo KQo QQ QJs QTs Q9s Q8s Q7s Q6s Q5s Q4s Q3s Q2s
AJo KJo QJo JJ JTs J9s J8s J7s J6s J5s J4s J3s J2s
ATo KTo QTo JTo TT T9s T8s T7s T6s T5s T4s T3s T2s
A9o K9o Q9o J9o T9o 99 98s 97s 96s 95s 94s 93s 92s
A8o K8o Q8o J8o T8o 98o 88 87s 86s 85s 84s 83s 82s
A7o K7o Q7o J7o T7o 97o 87o 77 76s 75s 74s 73s 72s
A6o K6o Q6o J6o T6o 96o 86o 76o 66 65s 64s 63s 62s
A5o K5o Q5o J5o T5o 95o 85o 75o 65o 55 54s 53s 52s
A4o K4o Q4o J4o T4o 94o 84o 74o 64o 54o 44 43s 42s
A3o K3o Q3o J3o T3o 93o 83o 73o 63o 53o 43o 33 32s
A2o K2o Q2o J2o T2o 92o 82o 72o 62o 52o 42o 32o 22
UTG 100
HJ 100
CO 100
BTN 97.5
SB 99.5
BB 97.5
5.5 BB
5.5 BB
J T 7
Bet 6.9
125%
30.24 combos
6.2%
Bet 4.1
75%
76.79 combos
15.8%
Bet 2.75
50%
52.64 combos
10.8%
Bet 1.8
33%
47.23 combos
9.7%
Check
278.82 combos
57.4%
HANDS
SUMMARY
FILTERS
BLOCKERS
HANDS
Straight 1%
Set 1.9%
Two pair 2.7%
Overpair 3.7%
Top pair 12.5%
Second pair 12.4%
Third pair 9%
Low pair 5.8%
Ace high 26.6%
King high 13.3%
No made hand 11.3%
EQUITY BUCKETS

References

1. Solving Large Imperfect Information Games Using CFR+ Paper

Thanks for a great semester!

Any questions?