Othello AI Tournament!

CS2 Recitation Series
Friday, March 4, 2016

Administrative notes

- Upcoming two-week project
  - Two milestones (worth up to 20 pt each)
  - Project can be tackled in teams of 2
  - First introduction to Git source control
    - will go over briefly in this recitation
    - http://git-scm.com/book/en/Git-Basics
  - Easy 5-pt bonus if you start before the weekend!
- Let us know if you have questions!
  - Questions clear up ambiguity!
  - They also help us find errors and address concerns.
  - Office hours will be held as normal
  - Feel free to email cs2-tas@ugcs with questions

Source control - some motivation

You're working with a small team on a module for an operating systems project. You keep your code in a shared folder online.

A teammate tries to optimize a critical section of the module, replacing hundreds of lines of code with poorly documented hacks.

You start seeing terrifying bugs - with no easy way to debug - and no way to roll back!

Source control - some motivation

You're working with a small team on a project for a networking class. You pass files back and forth by email.

Two of your teammates, while working on different tasks, end up changing the same three files - without coordinating.

Reconciling these changes is a full-time job - and your next project milestone is tomorrow!

Enter source control

- Called by various other names (version control, revision control, ...)
- A source control system makes life easy!
  - manages changes made to files
  - keeps a complete history of all changes
    - can rollback to an earlier state in the event of a problematic change
  - solves the concurrent-editing problem
    - controls file checkin/checkout; merges changes
  - (optionally) can provide off-site backup
    - can restore entire source tree in the event of disaster

Source control - many solutions

- Some solutions are centralised
  - one central server, many clients
  - CVS, Subversion
- Other solutions are distributed
  - every client is a repository
  - often there is a 'master' repository with which changes are merged
  - Git, Mercurial, Bazaar, Fossil
- We concentrate on Git for this course
  - you've been using it to fetch assignments (and updates)
  - you'll use it to turn in your Othello Project

Source control - repository hosting

- Git can synchronize with remote repos
  - many hosting services (GitHub, Bitbucket, Gitorious, ...) let you store Git repos in the cloud
  - you can also sync with other computers you own
  - we use Bitbucket to host this course's code

Source control - getting started

- Create a new repository (`git init`), or clone an existing one (`git clone`)
  - In each case we create a local copy
- In our case, we're just cloning something from Github
  - more specifically, we're forking the CS2 Othello code (i.e. creating an independent copy)
  - (you will need your own Github account to do this - use your Caltech email while signing up)
  - and then cloning that

Source control - local workflow

- Make changes at will
  - track newly added files: `git add foo.cpp`
  - also use `git add` to include specific changed files for the next commit
  - if you change your mind about including a change: `git reset HEAD foo.cpp`
  - `git status` lists which files are (un)staged
  - `git diff` shows details of unstaged changes
    - `git diff --cached` - staged changes
    - `git diff HEAD` - all changes

Source control - local workflow

- Commit when you have enough changes.
  - commit all your staged changes: `git commit`
    - shortcut for lots of changes: `git commit -a` (automatically includes all changed files)
  - every commit needs a commit reason - make it short but descriptive
    - other people will read it to figure out what you changed and why
  - every commit receives a commit ID
    - tools can use ID to refer to specific changesets
  - commit often; commit code that works
    - small commits = easier to revert single changes
    - don't break the build = happier team members

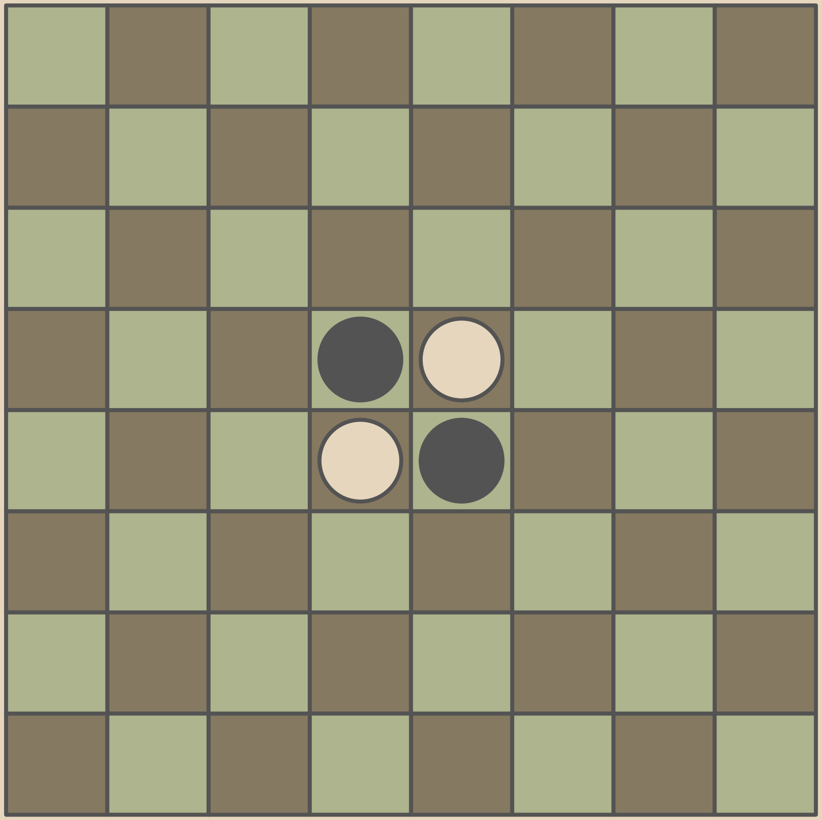
Source control - local workflow

- Pull any changes from the remote side.
  - `git pull` updates current branch
  - if someone has made changes in the meantime, Git will try to merge them automatically
  - if Git runs into conflicts, you'll be prompted to merge files manually
  - if changes were made, retest things to see if things still work!
- Push your changes to the remote side.
  - `git push` uploads to current remote branch

Source control - other features

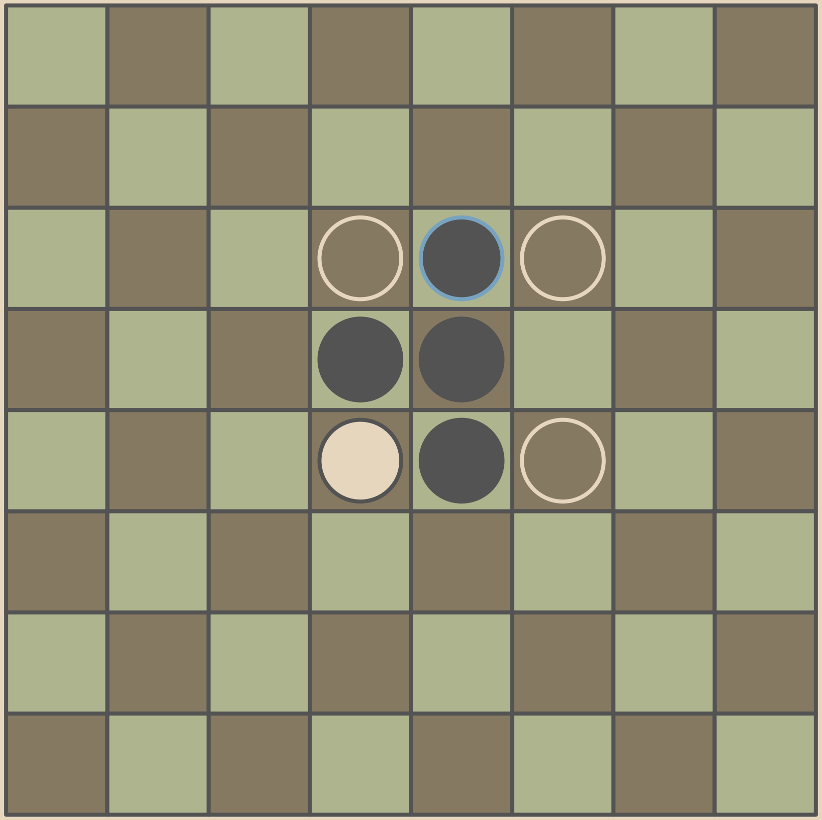
- Branching
  - pursue feature development independently of other features
- Change stashing
  - storing uncommitted changes for later use
- Time travel
  - inspect code as it existed in the past
  - revert to non-problematic changes if it becomes necessary

Othello: the game



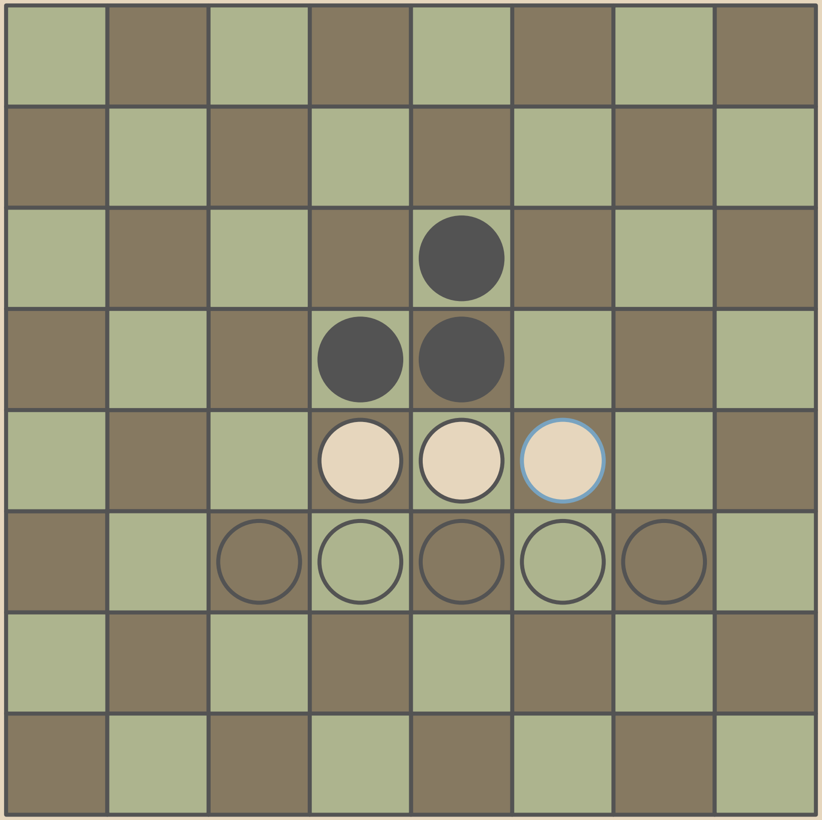
- Played on 8x8 board
- Starting position as shown

Othello: the game



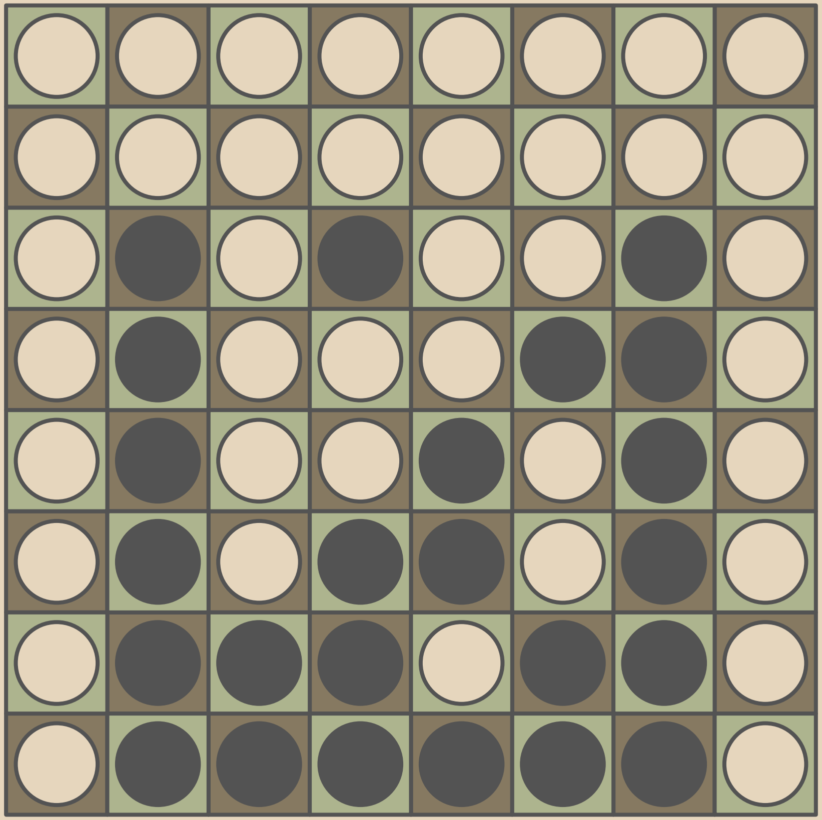
- Black makes the first move
- Moves must 'sandwich' opponents' pieces
- Pieces 'sandwiched' by a move are flipped to the player's color
- Iff a player has no legal moves, then that player passes

Othello: the game



- Black makes the first move
- Moves must 'sandwich' opponents' pieces
- Pieces 'sandwiched' by a move are flipped to the player's color
- Iff a player has no legal moves, then that player passes

Othello: the game



- The game ends when there are no more legal moves
- The player with the most disks of their color wins
- If both sides have an equal number of disks, the game is a draw

Your task

Write an AI that plays the game of Othello.

Hard task to do in a vacuum!

So we give you an Othello Project

- Project contains a tournament framework written in Java, and some (minimal) scaffolding so you can get started
- You will be writing a C++ program that plays Othello
- Let's take a tour of the Othello Project...

Files we provide

- To let you program in C++, we give you some wrapper code:
    - WrapperPlayer.java runs your C++ player and communicates using stdin/stdout
    - A basic C++ player consists of the following:
        - wrapper.cpp - handles communication
        - player.cpp - plays Othello
- You only need to modify player.cpp

Files we provide

- common.h - provides some data types
  - Move
    - an (x, y) coordinate pair
    - does NOT encode which player played
  - Side (enumeration)
    - can represent either BLACK or WHITE

Files we provide

- board.cpp - represents a game board
    - can be used to track the current board state
        - your AI will need to do this
    - can check move legality
    - can be updated with a given move
    - individual squares can be checked to see what's there
    - works, but not very efficient or featureful!
        - you can modify it as you please, or write your own replacement

Files we provide

- testgame.cpp - tests your player
    - invokes the tournament framework and displays a graphical Othello board
    - lets you test your AI against opponents
        - SimplePlayer, ConstantTimePlayer, BetterPlayer
        - **`./testgame player SimplePlayer`**
    - lets you play interactively against your AI
        - **`./testgame player Human`**
    - swap the order of the arguments to switch sides

What you'll be doing

- You'll be modifying the Player class
  - `Player(Side side)` - constructor
    - sets up the AI's copy of the board
    - does any precomputation / setup
    - limited to 30 sec. of runtime
  - `Move * doMove (Move * opponentsMove, int msLeft)`
    - takes the opponent's move, records locally
    - calculates a legal move, records locally
    - returns the move made
    - must take less than msLeft milliseconds!
      - Total time is 16 min. per side per game

What you'll be doing

- Makefile compiles everything to an executable `player`
  - change this to something containing your team name by changing PLAYERNAME in Makefile
- You may declare additional functions and classes as you need them

Note - language independence

- If you have a strong preference for a language that is not C++, you can use other languages for this project.
- May require more work on your part!
- See the assignment writeup for details on how this is done.
  - Note that using a higher-level language may place your player at a speed disadvantage...

Suggested workflow

- This is how we suggest you do things:
    - First, implement a class that returns some arbitrary legal move.
        - Passes if no legal moves.
        - Otherwise, picks one completely randomly.
    - Test it against yourself or against SimplePlayer to make sure there are no bugs.
    - Once you can get such a player working, then you can start to think about how to make it better
    - This is where thinking about strategies comes in

Some basic heuristics for Othello

- We would like to succinctly represent the "score" of a particular position relative to us
- More precisely, we're looking for some function that takes a board position and returns a number describing how "good" it is for us
- How can we accomplish this?

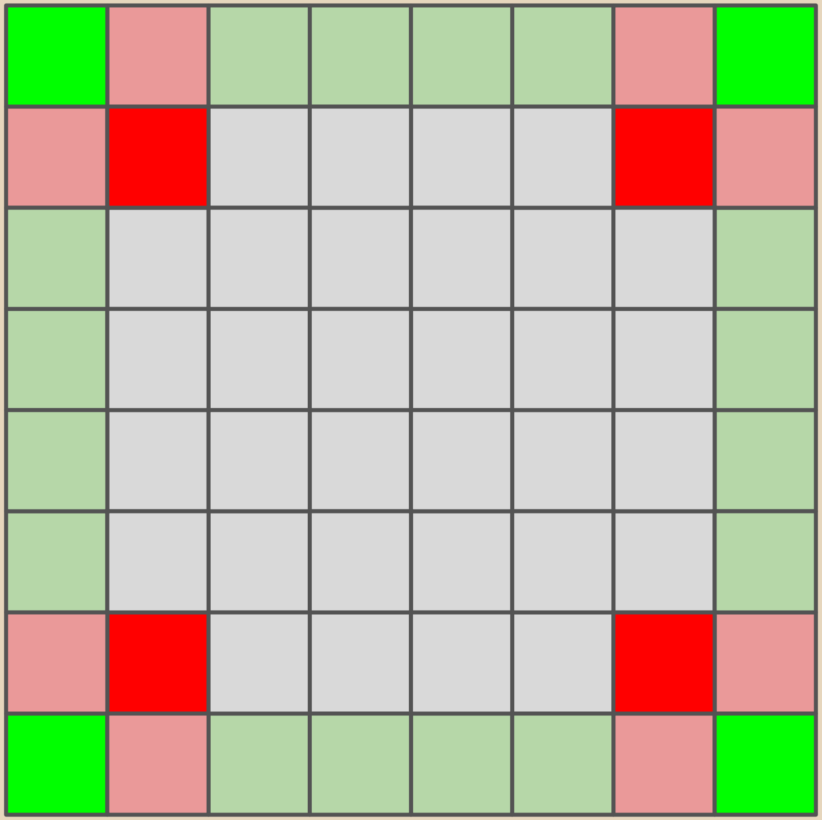
Some basic heuristics for Othello

- First idea: recall that the victory condition is to end up with the most pieces
- So we could set our "score" to be the number of pieces we have on the board!
- Simple and accurate, right?
  - All we need to do is to calculate the scores of all the possible end positions,
  - and then follow the line that lets us do best (minimax)...

Some basic heuristics for Othello

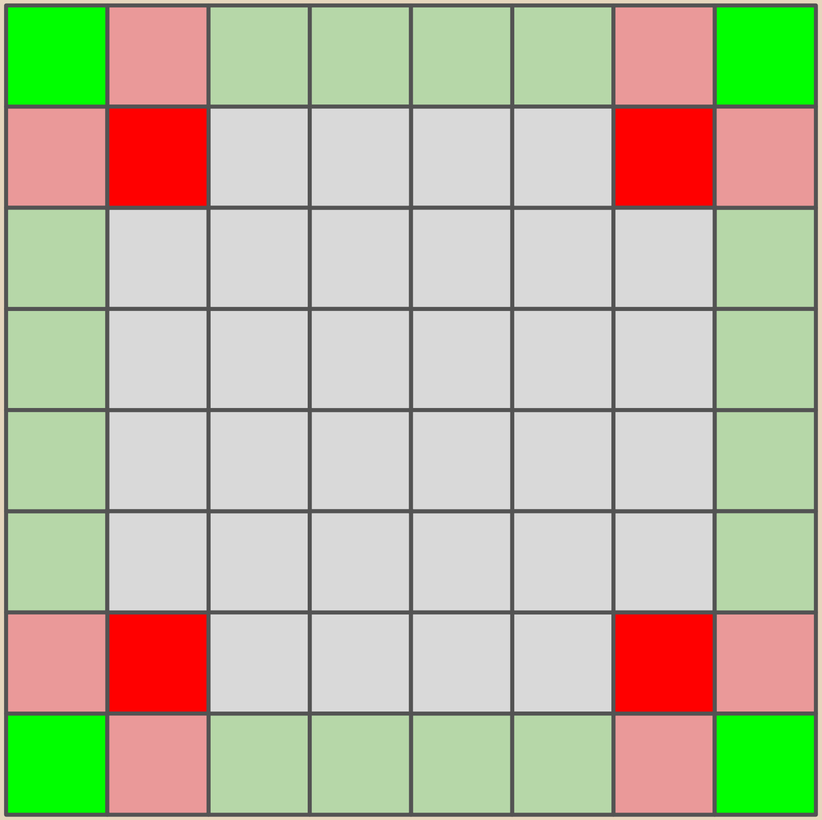
- Correct IF we live in a perfect world...
- but we don't!
  - 8x8 Othello game tree is HUGE (>> 10^50 possible games!)
  - We *cannot* hope to analyze them all in any short time.
  - Your programs, in particular, are limited to the memory of the cluster machines and the tournament time limit!
  - We want something that gives decent results WITHOUT checking the entire game tree.

Some basic heuristics for Othello



- Some squares are more or less 'valuable' or others
- Corners are excellent!
  - A piece in the corner can never be captured.
- Edges are pretty good!
  - A piece on an edge can only be captured by other pieces on the same edge.
- Squares next to corners are terrible!
  - Playing there often gives opponent a corner play.

Some basic heuristics for Othello



- Can take this into account when calculating our score
  - Pieces on special squares receive a multiplier in value
  - Pieces on edges are worth more than normal
  - Pieces in corners are worth a lot
  - Pieces in undesirable squares receive negative weights

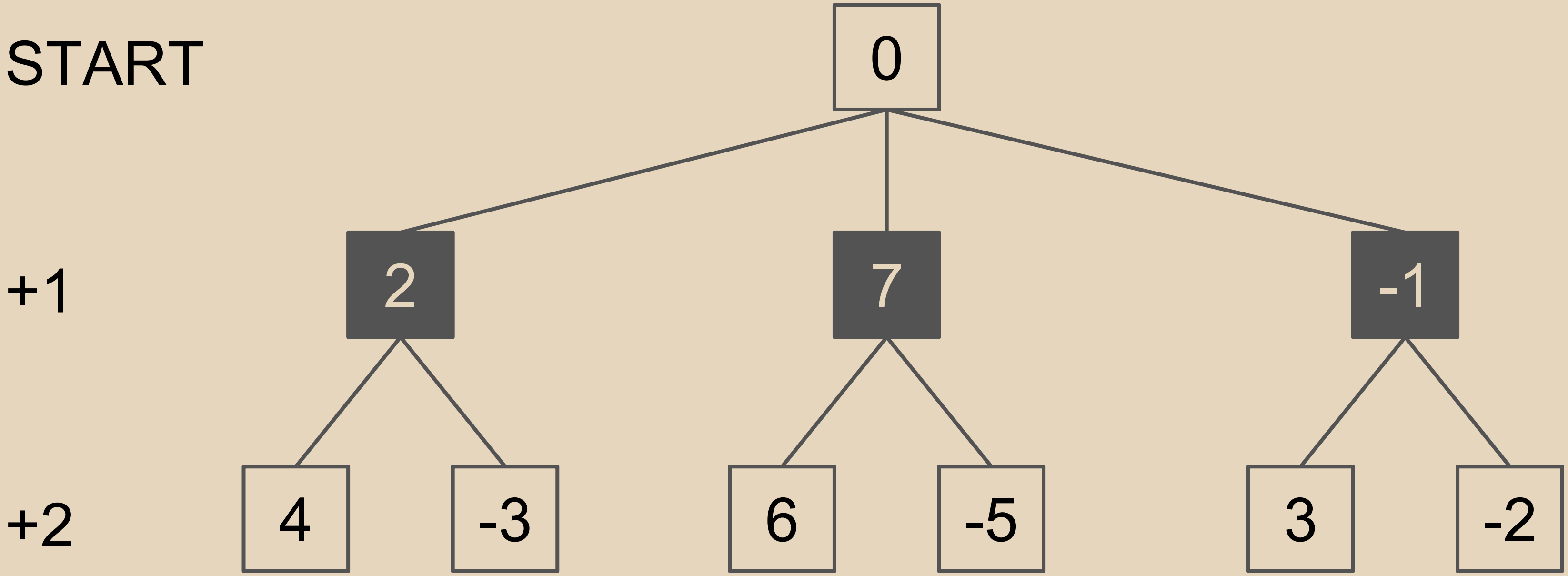
Suggested workflow - continued

- After your super-basic Othello player is working, you can improve it
  - Extend your Board class to 'score' a board position
  - Choose the move that yields the highest score
- There is no need (yet) to consider more subtle strategies
  - Concentrate on getting a decent working AI even if the heuristic is simple
- Once your AI performs well enough, we can think about improving it further...

Heuristics are not enough...

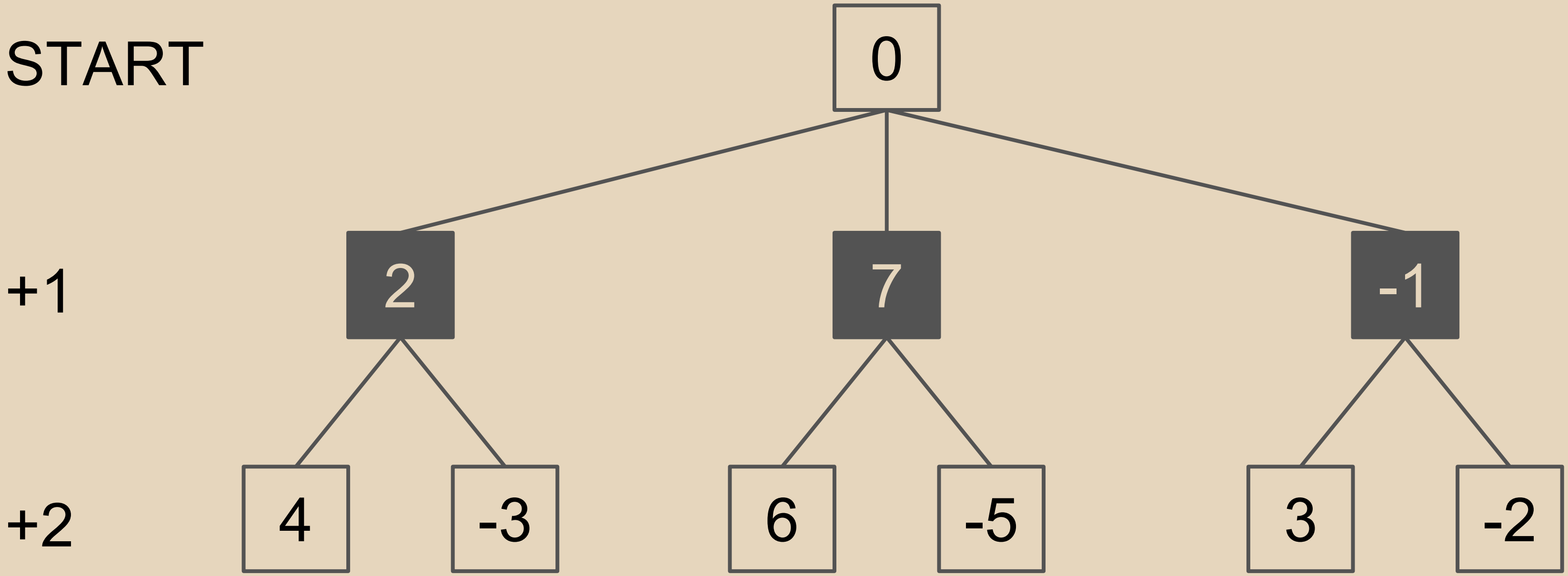
- ...not on their own.
  - A move that is good in the short term may open up a devastating counterattack!
  - We must protect against this by looking ahead and anticipating such problems.
  - Conversely, a winning line can look like you're at a disadvantage for most of the game.
- We can't look ahead forever, but we could look ahead to some reasonable depth.

Decision trees!



- Sample decision tree with arbitrary heuristic (scored from White's perspective)
- Start: White to move
- White can choose any of the three choices in +1
- Black can choose from connected choices in +2

Minimax Idea



- Idea: White wants to *maximize* its *minimum* gain.
- White is trying to maximize score…
- while Black is trying to minimize it.
- White is trying to achieve the best outcome under these circumstances.

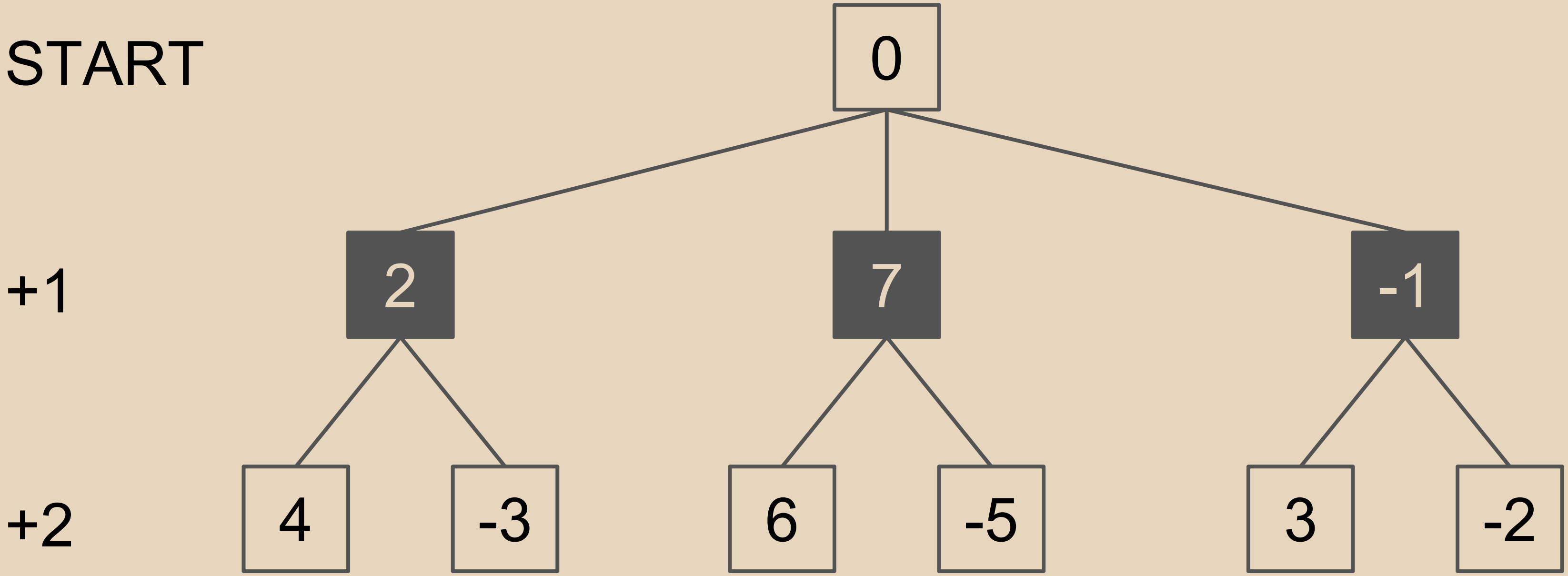
Minimax - Implementation Notes

- Every implementation will differ in details.
- We provide a testminimax.cpp to help you validate your minimax algorithm
  - invokes your Player with a special `testingMinimax` variable set to true
  - you will need to load up the test case into your Player's internal board state
- For the minimax test, you will need to override your heuristic
  - difference in number of pieces
  - call different board evaluation code based on the value of `testingMinimax`

Minimax - Implementation Notes

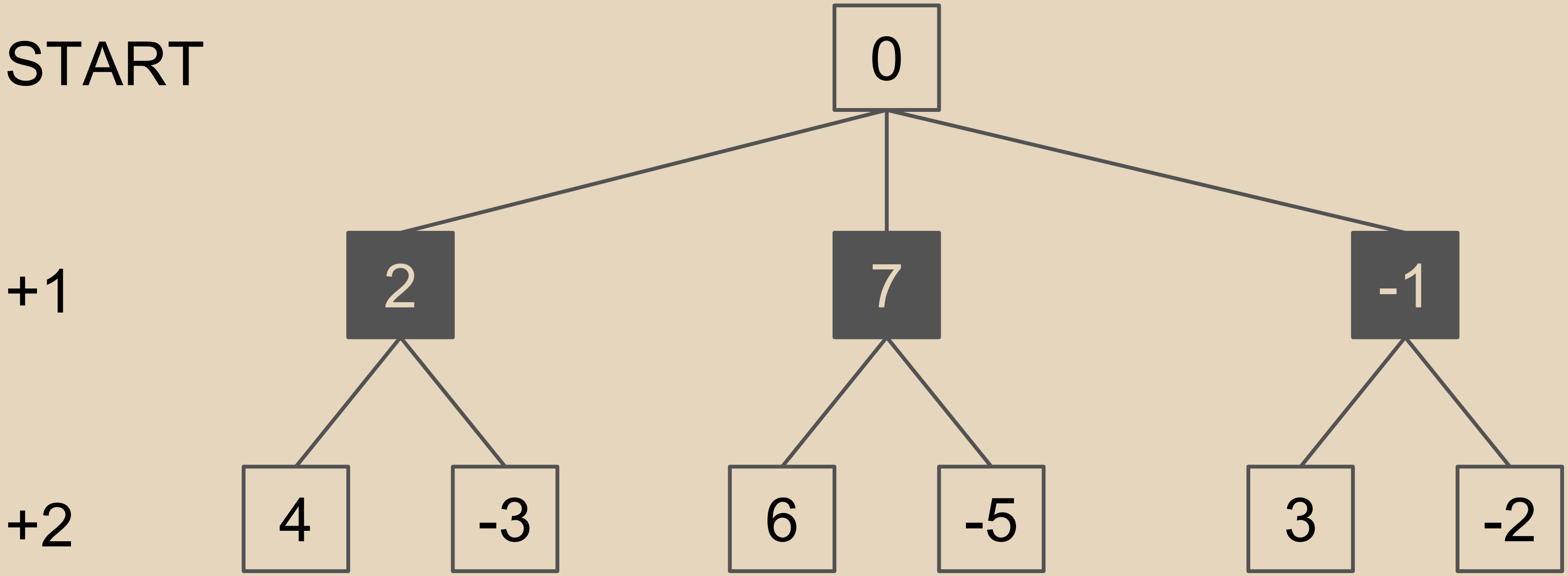
- Minimax search necessarily visits a large number of board states (several million even with not much depth)
- This can generate huge memory leakage
- Be careful with memory management!
  - keep code layout as simple as possible
  - keep data in local scope where possible
  - where data must transcend scope, keep track of what is responsible for its lifetime

Minimax Idea



- This is a fairly small tree.
- Could represent a fairly simple game.
- Othello trees are much bigger
  - average branching factor of ~7
  - maximum depth of 60

Minimax Idea



- For relatively shallow depths (2 to 4 ply), we can still generate the entire tree in time
- Deeper than that, we need to focus our efforts.
- Can we somehow identify branches that are not promising to search?

Next time...

- Alpha-beta pruning!
- More advanced ideas
  - Iterative deepening
  - Opening books
  - Transposition tables
  - ...?