# AI Programming (IT-3105) Large Project Option # 2:
Texas Hold'em Player

**Due date:** Thursday, May 2, 2024 (final demo day)

**Purpose:**

- Implement a system for managing games of Texas Hold'em.
- Implement AI poker players that employ two different decision-making processes: pure rollouts and re-solving.
- Learn basic principles of implementing imperfect-information games.
- Gain experience in combining Monte Carlo tree search and neural network approximators.

## 1 Assignment Overview

In this project, you will implement a basic Hold'em poker simulator along with the *brains* of AI poker-playing agents. The simulator will incorporate all of the standard aspects of poker play: dealing cards, requesting player actions, managing the pot of poker chips, determining winners and losers, distributing winnings, etc.. The AI agents will employ strategies based on various forms of rollout simulations consisting of a) hypothetical deals and re-deals of cards, and/or b) hypothetical betting actions by itself and its opponent. This latter approach constitutes *re-solving*, a technique used by many popular AI poker systems and perfected by The University of Alberta's **DeepStack** in 2017.

The core of this project is a micro version of DeepStack that realizes many of it's fundamental components, but which you will, in all likelihood, not have the computational resources to train to anywhere near DeepStack's level of competence. Still, you should finish the project with a very thorough understanding of advanced AI systems for playing imperfect-information games. This knowledge should transfer readily to the automation of other situations, such as auctions, marketing and even international diplomacy, since many real-world scenarios manifest imperfect-information *games*.

## 2 Introduction

Within the past 25 years, AI game-playing systems have racked up a string of impressive achievements, often beating top-notch professional players, in games such as chess, backgammon, checkers, Jeopardy! and even go, which was (until 2016) deemed nearly impossible for AI to conquer with today's hardware and algorithms. However, all of these games are those of *perfect information*: Each player knows the complete state of the game at all times. Their only uncertainty involves the strategy of their opponent.

So despite the complexity of inferring another's *game plan*, all of these games present less of a challenge than Texas Hold'em and similar poker variants, all of which involve *imperfect information*: Each player is uncertain about various aspects of the game situation, such as the opponent's hidden (a.k.a. *hole*) card(s), while also having the standard ignorance of opponent' strategies.

For nearly two decades prior to 2017, the best AI poker players used abstraction techniques to reason with various equivalence classes of card collections (a.k.a. *hands*) instead of the actual hands. Though making good intuitive sense, these approaches yielded heavy losses against skilled human opponents, often much heavier than if the agent had just folded at the beginning of every game. These methods often attempted to generate, for their abstract version of poker, a complete *strategy*: a mapping from any possible game situation to an action or a probability distribution over possible actions. Designing a complete strategy for the unabstracted game of Hold'em remains intractable, so these abstract approaches seemed like the only feasible alternatives.

DeepStack turned those beliefs into mythology by dynamically generating complete strategies for subsets of poker games, a.k.a. *subgames*, for example the portion of a game from the beginning of the flop stage to the beginning of the turn stage **when** the three flop cards are: K ♣, 3 ♠, J ♢. It generates very intelligent strategies for very specific scenarios using only modest computing resources (i.e., a laptop) at game time. However, this relatively fast reasoning is made possible by neural networks that a) reside in shallow leaf nodes of the search tree, and b) serve as local approximators of value, but c) require extensive offline training in large machine parks to attain DeepStack's level of expertise.

DeepStack essentially *solves* these subgames in the sense that it finds strategies that approximate the Nash Equilibrium (when given enough offline neural-net training and enough runtime rollouts). It is therefore the first system to *solve* poker in at least a mathematical sense, and it does, indeed, perform well. It soundly beat 10 of 11 professional players who completed the prerequisite 3000 games of HUNL [1] against it. The 11th player also lost, but not by a statistically significant margin.

Although we cannot hope to achieve the poker prowess of DeepStack with our limited experience and computer resources, the bulk of its fundamental algorithms are reproducible from descriptions in various journals and conferences. The most important publication on DeepStack, which includes substantial supplementary material useful in understanding the *guts* of the system, is:

*DeepStack: Expert-level Artificial Intelligence in Heads-Up, No-Limit Poker*, Moravcik et. al. (2017), **Science**, May 2017.

A copy is provided on the course webpage, as are those of several related articles. Together, they provide a reasonable summary of the main techniques, which are also explained in the lecture notes.

# 3 The Main Components

The work on this project divides fairly conveniently into the four major objects of Figure 1: the poker game manager, the poker state manager, the poker oracle, and the poker resolver. The resolver further divides into two primary tasks: a) building and navigating subgame search trees (a.k.a. subtrees), and b) composing, training and deploying neural networks.

The five main components to implement in this project are therefore:

[1] HUNL is Heads-Up, No-Limit poker, where heads-up means 2-player and no-limit entails no significant bounds on the amount that a player can bet at any time.

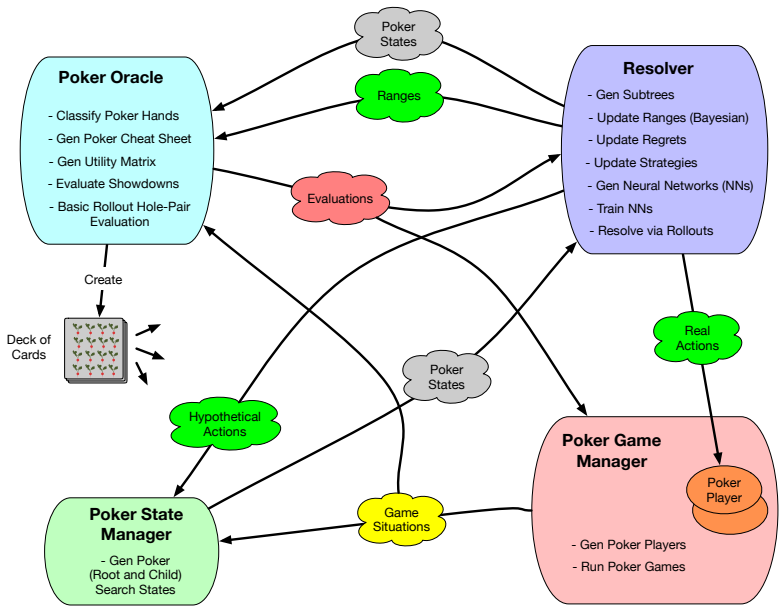


Figure 1: The four main poker objects and some of their primary methods. See lecture notes for more details.

1. A **poker game manager** that controls all basic operations of a Texas Hold'em poker game: rotating the role of dealer among the players, dealing public and private cards, invoking and monitoring agent play (i.e., betting actions), overseeing chips in the pot and in the agent's local piles, and determining wins and losses, as well as players who go *bust* (i.e., run out of chips). The manager should also create the poker agents at the beginning of a series of games.

2. A **poker state manager** for generating game-state objects, primarily for use by the resolver. This is a relatively simple object that can package up game situations from the game manager into state objects, which the resolver can then use as in the root of a subtree. When queried with a hypothetical game state from the resolver, it can generate some or all legal child states, thus aiding the resolver in building subtrees.

3. A **poker oracle** that handles many logical and probabilistic aspects of poker, particularly those involving card collections, ranges, value vectors and utility matrices.

4. A **poker subtree module** used by the resolver to both generate subgame search trees and control rollouts within them. Although subtrees may be generated in their entirety prior to all rollouts, they may gradually expand during rollouts, particularly near chance nodes, where computational resources may restrict complete generation of all chance-node children (e.g., all possible 3-card flop combinations) during initial subtree building but permit the gradual addition of a few new children each time search leads to the parent during a rollout.

5. A **neural network module** for configuration, training and deployment of the nets used in the resolver's subtrees. These networks reside in the leaf nodes of subtrees and allow the resolver to restrict subtrees to a manageable size, thus achieving a standard advantage of neural networks in search: providing evaluations for intermediate game states and thus obviating further tree expansion. See the lecture notes for more details on these networks.

The four main objects are summarized below, with some of the resolver's trickier details captured in pseudocode.

## 3.1 The Poker Game Manager

The poker game manager must be able to create two types of player objects: human and AI. The former is simply an interface to a real person who will play against the system, while the AI pokerbot is a hybrid that can either use pure deal-based rollouts or resolution to determine its moves. For example, it might use dealing rollouts in the pre-flop and flop stages but resolution in the turn and river. Or, it may stochastically choose rollout versus resolution based on an agent's parameter in [0,1]. Thus, the manager could setup games where one player uses resolution often, while the other uses it sparingly or not at all.

The manager must handle games with up to 6 players, but resolution should only be used when games consist of exactly two players (i.e., heads-up poker).

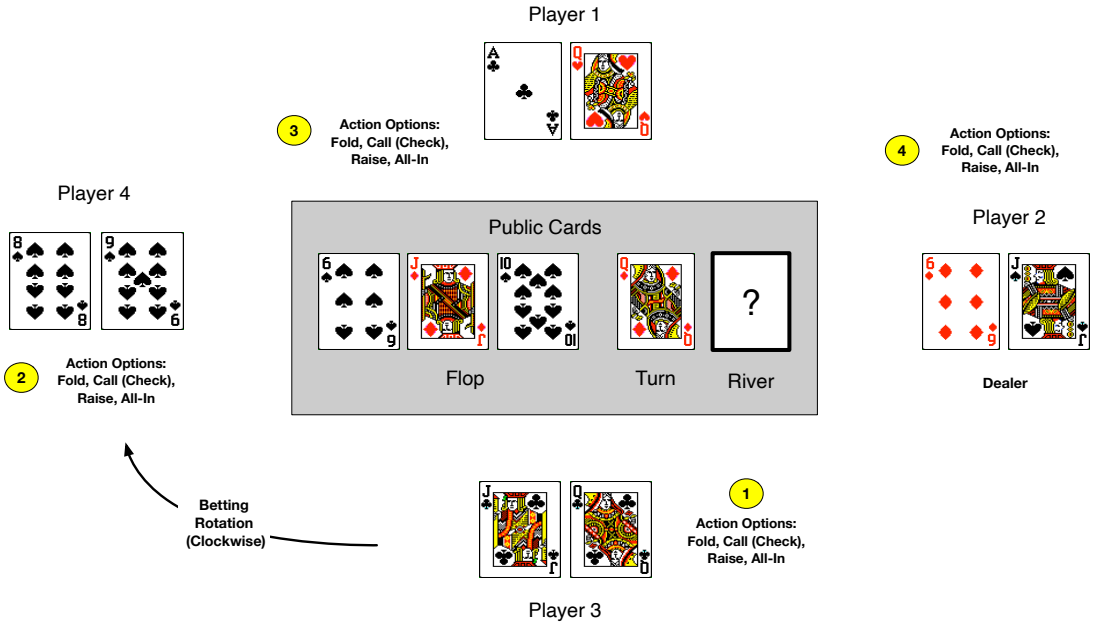


Figure 2: The turn stage of a Texas Hold'em game, from the perspective of an automated game manager (that can see the hole pairs of all players).

In coordination with the state manager, the game manager can enforce any number of restrictions on play. For example, it can limit the number of raises in a stage to a fixed number, or it can cap the size of the pot, or constrain raises to be of only one or a few amounts, e.g. the small or big blind.

The actual restrictions that you use should be clearly documented. Although severe constraints on betting are fine for this project, each game must consist of the standard 4 stages: pre-flop, flop, turn and river.

## 3.2 The Poker State Manager

Like the game manager, the state manager needs to know the specific rules of play so as to generate all and only the legal successor states to any given game state. It will provide this service of child-state production to the resolver. The game manager should only need the state manager to package up a current state of the real game for use by the resolver (as the root state of a subtree).

Practically speaking, the game manager and state manager should use the exact same set of game rules. If so, then one can query the other as to all legal moves in a given situation, thus avoiding redundant coding. For example, the state manager can just call the game manager for all legal moves when generating child states (for use by the resolver).

However, it is entirely possible that the two managers abide by different game rules, where the state manager would typically use a much more restricted version. For example, the game manager might allow bets of any (reasonable) size, but the resolver cannot generate subtrees with this unlimited number of actions. So the state manager would only supply the resolver with a small number of child states for any parent (e.g. based on fold, call, raise, or all-in). Then, if the resolver recommends a raise, the game manager could then permit a player to use some other logic to determine the amount of that raise. Your code should probably use the same rules for both managers, although you are free to explore a broader rule set for the game manager.

## 3.3 The Poker Oracle

This should handle many of the more complex calculations associated with poker, whether these involve collections of cards or probability distributions over hole cards.

It can classify poker hands into one of the 10 standard categories (straight, flush, full house, etc.) and further label them such that any two poker hands can be compared and ranked. It can also generate utility matrices for any collection of 3, 4 or 5 public cards.

As another key competence, it can use basic rollouts to evaluate hole pairs in combination with 0, 3, 4 or 5 public cards to yield a win probability for the pair. It may also generate card decks for internal use and for other modules, such as the game manager. In general, the poker oracle should restrict its computations to those involving cards, card collections, and the assessment of wins and win probabilities based on cards alone. It should not have to deal with betting actions nor pots nor player' chip piles.

## 3.4 The Poker Resolver

Although the game manager, state manager and poker oracle are complex in their own right, each is relatively straightforward to implement given basic knowledge of the rules of Texas Hold'em and many of the details covered in the lecture notes. However, the process of resolving (summarized in Figure 3) deserves an extra round of explanation, this time with very rough pseudocode that covers the resolver's main activities.

The pseudocode mirrors that of the supplementary material to the DeepStack article in *Science* (described earlier). However, whereas DeepStack employs a RangeGadget function to convert the evaluation vector of player 2 ($v_2$) into the range for player 2 ($r_2$), we skip this more advanced technique and treat both ranges ($r_1$ and $r_2$) similarly. Thus, our resolver finishes by returning both ranges, instead of $r_1$ and $v_2$.

Resolving begins with the production of a subtree, which might be produced in its entirety but may also be built incrementally, with expansions occurring during each traversal of the tree. The word *rollout* reflects both a) the possibility of incremental subtree expansion, and b) the possibility that the resolver may only explore some (not all) children of certain nodes. For example, if resolving crosses a stage border (and thus requires the dealing of more cards), it may be too computationally expensive to generate all possible cards (especially for the 3 flop cards). Hence, each visit to a chance node may invoke different events (i.e., different public cards) and thus different child nodes. Similarly, at a player node (i.e., one where a player must take an action), the algorithm may only consider one or a few (but not all) possible actions.

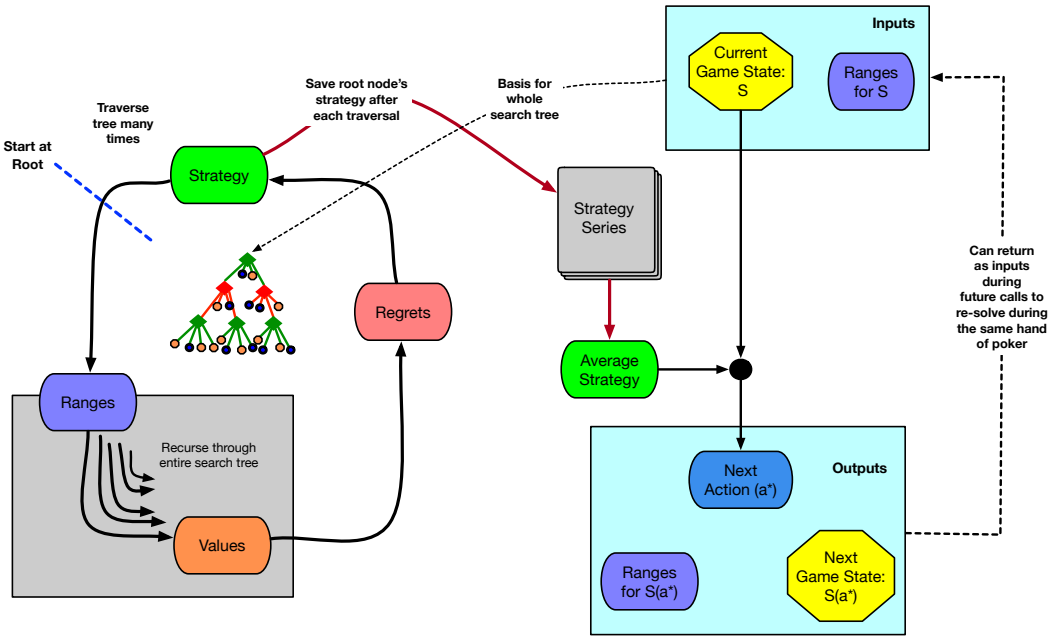


Figure 3: Overview of the Resolver's main activities.

The pseudocode mixes between notation involving nodes and states. In general, each subtree node houses a unique game state. Notation such as S(a) refers to the state attained by performing action a from state S, but it can also be interpreted as *the child node to S's node that corresponds to action a*. Since it is uncertain whether or not that child node actually exists when S(a) is computed (presumably by a poker-state manager), this is left as S(a) instead of Node(S(a)).

In addition to a state, each player node houses a strategy matrix, with one row for each possible hole pair and one column for each possible action. Range vectors ($r_i$) are sent down to each node, and evaluation vectors ($v_i$) are returned upward. The range of a player node's active player gets modified (via the call to BayesianRangeUpdate) as it passes down to each child, with m children entailing m different modifications. All of this is detailed in the lecture notes.

The procedure **Resolve** begins with a state in the current (real) game (i.e., the current game in the Poker Game Manager's series of games). It also receives an ending stage and depth (within that stage); these constrain the depth of the subtree. For example the current state (S) may belong to the flop stage, so a natural limit would be a depth of 1 within the turn phase. Thus, the subtree would grow to the chance state that separates the flop from the turn phase. That state resides at level 0 of the turn phase. **Resolve** would then generate one more level in the turn phase, by dealing out the turn card. At that bottom depth (level 1), all nodes should employ a neural network to produce value vectors, thus avoiding any deeper search.

**Resolve** performs T rollout traversals in the subtree, with each rollout a) passing ranges downward and values upward in the subtree, and b) modifying the regrets and strategies of each player node. These T different strategies at the root node are saved and then averaged to produce the final strategy, $\bar{\sigma_S}^T$, which is then sampled to produce **one** action, $a^*$, to be performed by the acting player of state S in the real game. This action also causes the range for the active player to be modified by the (now familiar) Bayesian update. The action ($a^*$), successor state ($S(a^*)$) updated range for the active player ($r_1(a^*)$) , and unaltered range for the non-acting player ($r_2$) are then returned. The ranges may be used in subsequent calls to **Resolve**.

**procedure** RESOLVE($S, r_1, r_2, EndStage, EndDepth, T$)
▷ S = current state, $r_1$ = Range of acting player, $r_2$ = Range of other player, T = number of rollouts
Root ← GenerateInitialSubtree($S, EndStage, EndDepth$)
**for** t = 1 to T **do** ▷ T = number of rollouts
$v_1^t, v_2^t$ ← SubtreeTraversalRollout($S, r_1, r_2, EndStage, EndDepth$) ▷ Returns evals for P1, P2 at root
$\sigma_S^t$ ← UpdateStrategy(Root) ▷ Returns current strategy matrix for the root
**end for**
$\bar{\sigma_S}^T \leftarrow \frac{1}{T}\sum_{t=1,T}\sigma_S^t$ ▷ Generate the Average Strategy over all rollouts
$a^* \sim \bar{\sigma_S}^T$ ▷ Sample an action based on the average strategy
$r_1(a^*) \leftarrow BayesianRangeUpdate(r_1, a^*, \bar{\sigma_S}^T)$ ▷ $r_1(a^*)$ is presumed normalized.
**return** $a^*, S(a^*), r_1(a^*), r_2$
**end procedure**

In procedure **SubtreeTraversalRollout**, there are four options:

1. The state represents a showdown situation, so the ranges are converted to value vectors by using the utility matrix corresponding to state S ($U_S$). The same utility matrix pertains to any state S and all of its descendants, S', but only if there are no chance states / nodes on the path from S to S'. At each chance state / node, one or more new public cards are dealt out, so the utility matrix must be recalculated. Note that the utility matrix entries are presumably from the perspective of player 1, so they are negated when calculating $v_2$.
2. The state is not a showdown but does reside at the lowest level (defined by a stage and depth within that stage) to which the resolver is expected to search. At this point, a neural network takes the ranges as inputs (along with the public cards, relative pot size, etc.) and produces $v_1$ and $v_2$. The pseudocode assumes that the state object contains its stage and depth.
3. The state is above the lowest level and requires a player to choose an action – from the set of all possible Actions(S). The resolver normally considers all such actions and thus recurses to each child state / node. In this third option, note the use of both $v_1, v_2$ and $v_P, v_O$. These are **not** four distinct vectors, only two. P will be 1 or 2, depending upon the active player in that state. So if P is 1, then $v_1$ and $v_P$ are the same vector, as are $v_2$ and $v_O$. Note that only the active player (P) has its range vector modified by the Bayesian updater prior to the recursive call. The returned value vectors are then scaled by the probability of choosing each action for each hole pair (as encoded in the strategy, $\sigma_S$).
4. The state represents the initiation of a chance event. In this case, no ranges are modified prior to the recursive calls to each child state, and all returned value vectors are scaled by the same factor: $\frac{1}{|Events(S)|}$.

**procedure** SUBTREETRAVERSALROLLOUT($S, r_1, r_2, EndStage, EndDepth$)
  **if** ShowDownState?(S) **then**
    $v_1 = U_S \cdot Transpose(r_2)$ ▷ $U_S$ = Utility matrix for state S
    $v_2 = -r_1 \cdot U_S$
  **else if** Stage(S) = EndStage and Depth(S) = EndDepth **then**
    $v_1, v_2 = RunNeuralNetwork(Stage(S), S, r_1, r_2)$
  **else if** PlayerState?(S) **then** ▷ A state in which a player needs to act
    P = Player(S), O = Opponent(S) ▷ Player(S) = player acting from state S. Opponent(S) = other player.
    $v_1, v_2 \leftarrow \mathbf{0}$ ▷ Both value vectors initialized to zero vectors
    **for** $a \in Actions(S)$ **do**
      $r_P(a) = BayesianRangeUpdate(r_P, a, \sigma_S)$
      $r_O(a) = r_O$ ▷ No change to range of non-acting player.
      $v_1(a), v_2(a) \leftarrow SubtreeTraversalRollout(S(a), r_1(a), r_2(a), EndStage, EndDepth)$
      **for** $h \in HolePairs$ **do**
        $v_P[h] \leftarrow v_P[h] + \sigma_S(h,a) v_P(a)[h]$
        $v_O[h] \leftarrow v_O[h] + \sigma_S(h,a) v_O(a)[h]$
      **end for**
    **end for**
  **else** ▷ Current state is start of a chance event
    $v_1, v_2 \leftarrow \mathbf{0}$
    **for** $e \in Events(S)$ **do**
      $r_1(e), r_2(e) \leftarrow r_1, r_2$
      $v_1(e), v_2(e) \leftarrow SubtreeTraversalRollout(S(e), r_1(e), r_2(e), EndStage, EndDepth)$
      **for** $h \in HolePairs$ **do**
        $v_1[h] \leftarrow v_1[h] + \frac{v_1(e)[h]}{|Events(S)|}$
        $v_2[h] \leftarrow v_2 + \frac{v_2(e)[h]}{|Events(S)|}$
      **end for**
    **end for**
  **end if**
  **return** $v_1, v_2$
**end procedure**

The procedure **UpdateStrategy** visits each node of the existing subtree, and if a node represents a player-action state, its strategy ($\sigma_S$) is updated – where $\sigma_S$ pertains only to the acting player. As described in the lecture notes, the current version of the strategy derives from the cumulative regrets ($R_S$) and positive regrets ($R_S^+$). Note that the strategy and both types of regret are represented by matrices with one row per hole pair (i.e., 1326 rows for a standard 52-card deck) and one column per possible action (e.g. fold,call,raise,all-in).

**procedure** UPDATESTRATEGY(Node)
  S = state(Node)
  **for** $C \in Successors(Node)$ **do**
    UpdateStrategy(C)
  **end for**
  **if** PlayerState?(S) **then**
    P = Player(S)
    **for** $h \in HolePairs$ **do** ▷ Each possible hole pair is an entry in the Information Set
      **for** $a \in Actions(S)$ **do**
        $R_S(h,a) \leftarrow R_S(h,a) + [v_P(S(a))[h] - v_P(S)[h]]$ ▷ Update cumulative regret for action a
        $R_S^+(h,a) = max(0, R_S(h,a))$ ▷ Compute positive regrets
      **end for**
    **end for**
    **for** $h \in HolePairs$ **do**
      **for** $a \in Actions(S)$ **do**
        $\sigma_S(h,a) \leftarrow \frac{R_S^+(h,a)}{\sum_{a' \in Actions(S)} R_S^+(h,a')}$
      **end for**
    **end for**
  **end if**
  **return** $\sigma_S$
**end procedure**

Finally, the procedure **GenerateInitialSubtree** will not be expressed as pseudocode but merely described. From the lectures, remember that none of the private information influences the structure of these subtrees: neither hole cards nor ranges over possible hole cards constrain the legal actions that a player can take, and these actions determine the child states of any parent state. Everything that constrains subtree growth is public: the number of raises so far in a stage, the maximum number of raises in a stage, the pot size, the chips of each player, etc. Hence, a complete subtree can be generated by simulating all legal actions in a game to a pre-defined depth (e.g., one step into the river stage).

This requires that the state manager keeps track of hypothetical chip counts for each player, raises per stage, etc. when computing the legal successors of any state S. So the resolver should be able to trust the state manager to generate all and only legal child states as it grows the subtree. And the state manager can do so without knowing the players' hole pairs or ranges.

Even if the subtree expands incrementally during a series of rollouts, the basis of expansion is always public information: a player's action or a new public card. Hence, the rules of poker enable a clean separation between the handling of public and private information such that the subtree's structure is fully determined by the former, while the primary computations of the resolver's rollouts involve the latter (i.e., the ranges, evaluations and strategies).

See the *simplications* section below for tips on reducing the size of the subtrees by restricting the legal actions in general and at various stages of the game.

# 4 Generality of Code

Ideally, your system should be general enough to handle different versions of poker and other games of imperfect information. However, Texas Hold'em has many special nuances (such as the 4 betting stages and different number of public cards dealt in each) that affect many aspects of this project, from the game logic to the number and configuration of the neural networks. Making each module completely generic would add a little too much extra work to this project.

However, your poker oracle should be able to classify any set of 5, 6 or 7 cards and thus be able to determine the winner hand(s) in any showdown across a wide variety of poker games, from Hold'em to simple 5-card draw. By insulating the oracle from betting behavior, it should immediately become reasonably useful across the poker spectrum.

In designing your game and state managers, try to make them as modular as possible such that a different version of each (for another poker variant) could conceivably be swapped in and would interface with the poker oracle and resolver.

When coding up the resolver, using terms such as *public information* and *private information* instead of *public cards* and *private cards* (or *hole cards*) will hint as to how the resolver could be extended beyond poker. Also, since an individual betting round has many similarities across most poker variants, the set of player actions need not be specialized for Hold'em; and even if they are, the resolver mainly just asks the state manager for all child states: it need not know the detailed semantics of the actions leading to those states in order to build its subtree, nor to do rollouts within it.

In general, the resolver can work with data structures for ranges, evaluations, regrets and strategies without knowing anything about their specifics. However, the actual sizes of these data structures will depend upon the precise game. For example, Hold'em involves a hole **pair**, and thus 1326 possibilities, while 5-card stud poker involves only a single hole card (and many publicly visible but non-shared cards, 4 per player) but 4 betting rounds, just like Hold'em. In 5-card stud, a player's actions should also affect the probability distribution over the opponents beliefs about their hole card (i.e., their range) just as in Hold'em.

To generate a utility matrix for 5-card stud, a poker oracle would only need to know each player's 4 visible cards, from which it could compute the winner for each pairing of single hole cards to fill a 52-by-52 matrix. The matrix operations for computing evaluations from ranges and a utility matrix would then be identical to those in Hold'em.

In short, a little thought in your design can yield relatively general code, but there is no need to go the extra mile to achieve full plug-and-play modularity and interchangeability to support multiple poker games or other imperfect-information tasks.

# 5 Simplifications

Due to the difficulty of making a resolver truly general, this project is hard to test on simpler imperfect-information games, such as the auctions covered in the lecture slides. However, there are several ways to reduce the search space for Texas Hold'em so that a) debugging is easier, and b) your Hold'em player may be able to show some intelligence despite our paucity of computational resources.

One helpful tactic is to restrict the number of raises to a small even number (e.g., 2 or 4) per stage (pre-flop, flop, turn and river). Removing the *all-in* option is another reasonable simplification. Restrictions can also be placed on the pot size such that once it reaches a maximum, both players can only call or fold.

It is also wise to have one basic *raise* action that always involves a fixed amount, e.g., the small or big blind.

Another reasonable modification involves the card deck. Instead of generating all 52 cards, use only 24: the 9 through ace of all 4 suits. This *short deck* still facilitates the use of all 10 hand classifications, although, by the end of the river phase, each player will have 7 cards to work with and thus be guaranteed at least a pair: the high-card category then becomes irrelevant.

You may also set a hard limit ($k_c$) on the number of random events that can be performed at a chance node during re-solving. Thus, each such node would only have $k_c$ children in the subtree, with each associated with a different public card (or cards for the flop stage).

Note that most of these simplications will not affect the poker oracle, since it should not have to consider player actions, only card combinations. Only when the size of the deck changes, does the oracle need to get involved.

**Any simplifications beyond those mentioned above should be discussed with and approved by the course instructor to insure that you can still get credit for the project.**

# 6 Deliverables

This project counts as 4 of the total of 6 points available in this class, and you need 5 of the 6 to get a passing mark for the course. All points are assessed independently in a simple binary fashion: you either get a 0 or 1 for each possible point (i.e., no partial credit).

1. The poker game manager and poker state manager need to properly handle game situations and game states, and the game manager needs to clearly and unambiguously display the current state of any game (either on the command line or using a fancier visualization).
2. The poker oracle must generate card decks and utility matrices, as well as performing diverse rollouts for evaluating hole pairs in the pre-flop, flop, turn and river stages of a Hold'em game. It must be able to use these rollouts to produce a poker cheat sheet.
3. The resolver must be capable of generating a subtree for any game state and then performing rollouts within it.
4. The resolver must be capable of configuring, training and using neural networks to restrict the size of subtrees. Trained networks for the flop, turn and river phases should be saved to file and then be available for resolving in the pre-flop, flop and turn phases, respectively. See the lecture notes for more complete details of the bootstrapped training process for these three networks.