Design Notes for the Sancho Game Player

Version 0.1

March 2nd 2014

Steve Draper

Contents

[Introduction 2](#_Toc381542495)

[High Level Overview 2](#_Toc381542496)

[Underlying State-machine 2](#_Toc381542497)

[Polymorphic Propnets 3](#_Toc381542498)

[Multi-Instancing 4](#_Toc381542499)

[TestForwardDeadReckonPropnetStateMachine 4](#_Toc381542500)

[Playout Implementation 5](#_Toc381542501)

[Network Optimization 5](#_Toc381542502)

[Network Splitting 5](#_Toc381542503)

[Goal Removal 6](#_Toc381542504)

[MCTS Implementation 6](#_Toc381542505)

[Node Storage 6](#_Toc381542506)

[Score Vectors 7](#_Toc381542507)

[Transitions 7](#_Toc381542508)

[Node Completeness 8](#_Toc381542509)

[Node Selection for Expansion 9](#_Toc381542510)

[Playout Result Update 9](#_Toc381542511)

[Heuristics 10](#_Toc381542512)

[Blending state-heuristic signals 10](#_Toc381542513)

[Blending move-heuristic signals 11](#_Toc381542514)

[Choosing the best move to play 12](#_Toc381542515)

[Threading Model 13](#_Toc381542516)

[Meta-Gaming 13](#_Toc381542517)

# Introduction

This document outlines the overall structure, and key algorithmic elements of the Sancho General Game Player. It is intended as a guide for developers to enable efficient collaboration on the codebase. The document is organized primarily into sections outlining the structure and design choices for each of several (somewhat) independent layers, within which sub-sections detail notable components and algorithms.

# High Level Overview

Sancho is built upon the Java GGP framework from Stanford, which provides:

* A low-level class library for interpretation of GDL
* An outline propositional network construction factory
* A framework for the implementation of GGP players, including all handling of necessary network communications

These have been modified in some cases, to enhance their functionality within the Sancho repository, but are largely used fairly directly. Sancho itself is built upon a sample player from the ggp.org codebase.

Sancho is essentially an MCTS player, utilizing a propositional net state-machine. Both the state-machine and MCTS implementations are however, significantly modified in a number of important regards, as will be detailed in later sections of this document.

# Underlying State-machine

The state machine implementation is based loosely upon the propositional net state machine code from ggpbase. However, it has been extensively modified in a number of important ways, as detailed below.

## Polymorphic Propnets

GGPBase uses a fundamental base class called ‘Component’ to abstract possible components in a propositional network (with subclasses for specific component types). In the Sancho codebase this is replaced by an interface ‘PolymorphicComponent’, to allow for multiple derived implementations, which can be switched between at runtime, and for multiple-inheritance (of interfaces). The idea here is that we can instantiate propnets with differing runtime characteristics (for instance an implementation that is very stripped-down and performance oriented, which we might use during actual gameplay, vs. one which is heavily instrumented and gathers statistical usage data, which we might use during meta-gaming analysis). Each implementation will provide a factory class which can manufacture PolymorphicComponent instances of its implementation type. The overall propnet factory (OptimizingPolymorphicPropnetFactory) then takes a component factory, provided to it at runtime, to instantiate the appropriate implementation, potentially duplicating an existing propnet with a different implementation type. PolymorphicComponent also supports a few capabilities that are used for higher-level semantic purposes:

* Components have a ‘signature’. This is a random number (from a 64-bit space to make duplication extremely unlikely) for an input proposition (or constant), and is calculated for other components by hashing together (in an order-independent way) the signatures of its inputs with a fixed signature for its component type. The result is a signature for the quantity it computes (in the context of a given network)
* Propositions support a get/settable opaque property, which may be used by client code for its own semantic purposes (use of this will be discussed in a later section)

The main component implementation used by the current version of Sancho is the ForwardDeadReckonPropnet. This provides a highly optimized implementation where (for example) collections of component inputs/outputs exist as arrays rather than collections, which have much better runtime performance characteristics. The key features of the ForwardDeadReckonPropnet classes are:

* When initially constructed they are mutable (component inputs can be added and removed, as can outputs), and use collections to enable this. This allows optimizing transformations to be made on them during an analysis phase.
* They support a ‘crystalize()’ method which removes the mutability, dynamically instantiating fixed size arrays for all collections, and maximizing runtime performance thereafter
* Propagation occurs by forward-propagation synchronously when (input) propositions are modified, working differentially relative to the existing state encoded on the network – that is to say that states are not propagated through the entire network (necessarily), but only as far as components whose outputs do not change as a result of the new input. Thus if (say) an AND component has 3 inputs, and 2 are false in the previous state, then the change of one input to true, will not influence the output state, and propagation (along that branch) will cease at that component.
* AND and OR components both work by maintaining a true-input count rather than actually enumerating and AND/ORing together their input values at each change of one input. This is far more efficient for large numbers of inputs
* Both active base propositions, and active legal move propositions, are maintained as an explicit list, efficiently and iteratively updated during state propagation. This allows client code to determine legal moves and active propositions without enumerating all such to find the active ones. These ‘lists’ are actually BitSets.
* The class library also introduces an internal representation of a machine state, which is simply a bitset of the active base propositions, with a master lookup table mapping from bit-index to proposition component. This is far more efficient to deal with at runtime than a GDL-based MachineState object. In particular it very efficiently supports bitwise comparisons and operations on states, which make calculation of things like state-distance metrics highly efficient. Similarly a ForwardDeadReckonLegalMoveSet does the analogous thing for legal move propositions.

## Multi-Instancing

The PolymorphicPropnet library also defines a second interface called MultiInstanceComponent. This abstracts the concept of ‘channels’, allowing components to have a vector of values rather than just a single value. Setting a proposition’s value then becomes setting it on a given channel id (index), and propagates it along that channel. If proposition nets are built with multi-instance components they are effectively multiple instances of the same network with independent state (but common topology). ForwardDeadReckonPropnet family components implement this interface, and it is used to provide a simple multi-threading capability (each thread owns one channel, and thus state is manipulated on any given channel by only a single thread, making it a lockless implementation)

## TestForwardDeadReckonPropnetStateMachine

This is the (not really very aptly named any more!) state machine implementation used by Sancho, on top of the ForwardDeadReckonPropnet propositional net implementation. It provides the abstract state machine interface for the current game to the higher layers. It has a number of interesting features as below.

### Playout Implementation

The state machine is capable of running playouts from a provided starting state, both with high efficiency, and added functionality beyond completely random playouts traditionally associated with MonteCarlo players. The key points are:

* Entire playouts remain within the efficient internal representation, so no transformation between state representations is required during a playout
* Playouts may optionally be ‘greedy’ or not. If greedy playouts are enabled then at each state reached all child states are calculated, and any wins that are at the discretion of the role whose choice it is are deterministically taken. Similarly any choices that immediately lead to a deterministic win choice by the next role are rejected. This effectively adds a one-level tree search to each playout step, and avoids tainting the playout statistics with move sequences that would always be (expected to be) avoided in actual play. Because this is quite expensive, its use is dynamically enabled and disabled depending on meta-game analysis (see later)

### Network Optimization

When building the propnet for a given set of GDL rules a number of optimization steps are run. These include:

* Removal of anonymous propositions
* Removal of unreachable bases and inputs
* Removal of redundant constants (there ARE only 2 binary constants!!) and of redundant logic (e.g. – AND inputs that are constant)
* Refactoring of large gates (large input count) and of large fanouts (large output count), using common factors of all or a large number of the inputs/output (e.g. – an AND with 10 inputs, all of which are ORs of a common value with independent other values, can refactor the common OR to the far side of the AND)
* Optimization of inversion using DeMorgan’s law transfromations (e.g. – an AND whose inputs are all inverted can be rewritten as a NOR of uninverted inputs, removing all the inverters on the input lines)
* Removal of duplicate logic. This utilizes the PolymorphicComponent’s ‘signature’ to identify components whose outputs are the same thing, allowing one copy to be removed, possibly removing large duplicate sub-networks)

### Network Splitting

It is frequently the case (that is to say, with many games) that a small set of base propositions change very frequently (canonically the ‘whose turn is it’ propositions), and that states ‘oscillate’ between sets that contain different configuration of these highly active propositions (black’s turn, white’s turn, black’s turn, …). This tends to lead to a lot of repeated-very-other-state-transition calculations in the state propagation, which is an overhead we can eliminate. This is done by splitting the state machine’s network into sub-nets (which overlap), wherein the highly active propositions are hard-wired to one particular configuration in each split (with the set of splits covering the possible configurations). Currently a single split, pivoting on a single proposition, into 2 sub-nets is supported, and automatically discovered during state machine initialization. For two player games this almost always maps to whose-turn-is-it (though it is calculated based on whichever base proposition changes most frequently during simulated games). Propnets for each subnet are initialized independently of one another (referred to as the X-net and the O-net), and used when performing operations on states that correspond to their hard-wired base prop value (e.g. – during X’s turn the X-net is used to calculate legal moves/nets state etc., during O’s turn the O-net is).

*Note* – a structure is held for each LOGICAL base proposition or legal move which references the actual proposition in both the X and O nets. A reference to this structure is stored as opaque data on the propositions to which it refers. This allows us to get from a proposition to its logical role in the network, whichever subnet it is from, and visa versa (in either subnet)

### Goal Removal

For quite a few games the size of the logic to calculate goals is very large (consider Reversi), and much of the effort of propagating proposition changes through the propnet goes to recalculating goals. However, goals are only defined in terminal states, and only queried in those states (typically). Consequently we do not care about the goal sub-net during playouts (until the terminal state is reached). To make this more efficient a separate goal-net is constructed (which is then trimmed down to ONLY those components needed to compute the goal values), and all goal logic is removed from the X-net and O-net. Requests for goal values are then computed using the goal-net, and other requests are not burdened by extra computation to propagate changes through the goal logic. This makes a request for a goal value a higher overhead than it would otherwise be (the goal net is typically not already in more-or-less the right state, so differential propagation will take longer), but reduces the overhead on state transition. Provided goal requests are rare compared to state transitions this is a (potentially major) win. This will be the case for non-greedy playouts (but not greedy ones to a significant extent, as they explore all nearby-terminals and need to know who they are wins for). Consequently goal-net separation is optional, and is enabled when greedy playouts are disabled and visa versa.

# MCTS Implementation

Sancho is evolved from an initially standard MCTS player, but has added a large number of improvements (?) and changes, which are discussed below

## Node Storage

We want to restrict ourselves to bounded memory, even for games whose state-space is extremely large (and regardless of how long we are given to think). Consequently we cannot just keep adding nodes to our MCTS structure without end. To handle this efficiently, Sancho uses a fixed array of nodes (the node pool) from which nodes are ‘allocated’ (become referenced from the MCTS structure). This removes any object allocation/initialization overhead (since we reuse a fixed set of object instances). During play, if the table passes a certain fullness threshold (basically allowing a margin that should suffice for any one node expansion, so is essentially the maximum branching factor we can cope with in the worst case), then (between expansion selection instances) a trimming function is run to remove nodes from the MCTS structure until the threshold is no longer breached. Freed nodes are returned to a free-node chain from which the allocator will preferentially take them.

Because this leaves us with a (slightly) sparse structure we need to then also answer two questions:

1. What do we do when we are selecting from a node which has trimmed children? In this case the statistics for the child are not available and so UCB cannot be applied. We instead choose randomly (i.e. – degrade from MCTS to simple MC at that node). Obviously this is undesirable if it happens frequently. When this occurs we also re-expand the partially trimmed node again.
2. Which nodes do we trim? There is some literature on this for regular MCTS trees, though applying it at low overhead (that is, computational overhead to the search for nodes to trim) proved problematic. The literature suggests trimming non-recently visited nodes from at least depth 3 of the tree. In Sancho we run an analogue of the MCTS selection procedure, from the root, selected the LOWEST scoring nodes, also dividing by the log of the number of descendants of each child. This favours nodes that are least likely to be selected for expansion with large numbers of descendants – the second part is important or we’ll soon end up trimming one subtree out completely and back it up to, or near, the root (at which point selection though it becomes very likely).

## Score Vectors

To properly handle multi-player (>2 roles) games you cannot just propagate a single score (indeed not even for 2-player if it’s not fixed sum). Instead we store a vector of scores (one for each role) at each node, and rollout results also provide such a vector, which is then propagated during score update. Each node has a ‘decidingRoleIndex’ member which is the index of the role that is represented by the edge leading TO that node. Note that role indexes are always normalized such that we are role index 0.

## Transitions

Many games exhibit multiple paths from one state to another (e.g. – re-order a given set of moves in Connect5). In such cases one path transitions into a state already encountered by another. It is desirable to have a single node in our MCTS structure (necessarily no longer simply a tree) that represents any given state for several reasons:

* It’s more memory efficient
* The score stats are more accurate when aggregated than they would be on separate nodes with fewer individual playouts though them

Consequently Sancho changes the MCTS tree structure to a more general graph structure. This complicates the upward propagation of playout results (you cannot propagate through all upward links because that can explode exponentially, making propagation of playout results computationally infeasible). After a significant amount of empirical experimentation, Sancho now records the (downward) selection path through the graph used to determine where to play out from (this is the TreePath class) and uses that to also control the upward propagation of results.

## Node Completeness

In an MCTS tree, a non-terminal node has a score that is a statistical measure of it actual value, based on propagated playout results. Terminal nodes however, have exact scores (that is information about their exact value is complete). We generalize this to allow any node whose score is perfectly known to be deemed ‘complete’. This can arise based on the completeness and scores of its children. Thus:

* In a non-simultaneous turn game only one role can make a decision at any given node. Hence if a child of that node is a complete node that is a win for the choosing role, then that node is also a complete win for that role
* If all children of a node (in a non simultaneous turn game) are complete then the choosing player will choose the best result, and the parent node can also be considered complete with that score
* In a simultaneous turn game if a move leads to a win for a role, and so does that same move in all cousin states, then it is a win for that player regardless of the move other roles make, and hence the parent is a win for that player

Using these rules (and variants thereon) we can often propagate completeness up the tree, and trim all nodes below the highest level complete node (all we need to know is the score at the point it becomes fully determined).

On completing a node, its score estimate loses all uncertainty, and we can further leverage this by observing that some of the uncertainty to the values of its parent(s) has also been removed. We thus recalculate the parent values from scratch as a weighted average of its children’s values (using a UCB-like formula accounting for uncertainty of each child). This formula is somewhat empirical and can probably be improved, but it seems to work reasonably well. This process also provides a level of correction for statistical uncertainty introduced by the fact that we have imperfect result propagation because of the graph structure (that is we only propagate along the path the was selected down)

## Node Selection for Expansion

This is based on the UCB rule for MCTS trees, but with a number of tweaks:

1. We use UCB-tuned (common in the literature)
2. The constant factor used to determine the ratio between the exploration term and the exploitation term varies in optimal value by game. We try to set this appropriately during meta-gaming (it can make a huge difference)
3. Some experimentation has been done on mixing other values into the UCT selection value to influence selection. This amounts to adding in heuristic signals to cause parts of the tree to be explored more than other parts, and is more fully discussed in the section on heuristics
4. Selecting through a complete node can never provide new information (but importantly it can increase its weight higher up the tree). This means running new playouts from a complete node is pointless (at best, and misleading at worst). Two approaches have been experimented with:
   1. Immediately propagating a pseudo-playout repeating the (already known, completely determined) value of the selected node
   2. Selecting the next best node instead

Empirically just doing the first of these behaves badly and leads to too much time spent increasing the strength of already known values without actually gaining any new information. Instead we only take this approach when all children are complete or we have found the node to be complete immediately upon expansion (i.e. – it’s terminal). In other cases we perform a hybrid action based on the second option, but with modified upward propagation rules (see later)

## Playout Result Update

Propagation of playout results is more or less as in a standard MCTS implementation, with a couple of small differences:

* Because the MCTS structure is actually a graph rather than a tree, when we update a node’s values in response to a playout we need to take account of the fact that the child we’re being back-propagated to from may have received playout results along other paths, not involving this parentage. Consider the extreme situation where the first hundred visits to a node occurred via one parent, and in those visits it gains a strong score. Then, due to the action of the exploration term of the UCT value), a different path is selected and we transition into the same child node. Again due to exploration, we select a descendant of that child for expansion that is not the highest scoring node and get a poor playout result. This is fine for the child itself, since the new poor result is averaged in with all the previous good ones (it’s just normal MCTS at the child’s level), but for the parent this is the first playout propagation we’ve seen from that child, and it gives a distorted view of the value of the state, not taking account of other paths to it that resulted in different results.  
  To address this we store a visited count on both nodes AND edges, and can thus determine how much of a child’s value was accrued along the path from a given parent. On update of the parent’s score value instead of just applying the playout result we apply an adjusted value which is a weighted average of the playout result (multiplied by the proportion of child playouts which were selected through this parent) and the child’s current score (the proportion due to other routes). This allows us to propagate information received by the child in other expansion paths, and acts as a statistical ‘fixup’ to the fact that we cannot propagate along all parent paths for expense reasons.
* As mentioned in a previous section, when we are selecting nodes for expansion and find the ‘correct’ choice is a complete node, we choose the next best choice. This means that when propagating up from the choice node in question, higher nodes should see the playout values that would have resulted from the correct choice (i.e. – the value of the complete child node that UCB told us to select). This is handled by recording such decision points in the TreePath as override values for the propagated score. The net result is that the actual chosen path results in a useful playout to improve our knowledge of the node in which we made the ‘incorrect’ choice, while providing the correct values upwards.

## Heuristics

For games where we can identify heuristics we can use them to help the MCTS structure converge more rapidly. We may have heuristic values for states, or for moves – just for historical reasons I focused more on heuristic values for states first, and did considerable experimentation on ways to blend heuristic signals in the MCTS process.

### Blending state-heuristic signals

Generally there are a couple of ways one might do this (which are somewhat explored in the literature):

1. Use the heuristic to modify the UCT score used to select which node path to follow in expansion.
2. Use the heuristic value of newly expanded nodes to provide one of more pseudo-playout values to ‘seed’ the new node.

Experimentally I found the first approach to be problematic because it causes a divergence of node score from node selection rate, which means the greatest certainty (of the score estimate) does not necessarily correlate with the highest scores. When selecting the move to actually play this can lead to pathological situations where you mistakenly play a move with an apparent high score but very low confidence, which actually turns out to be bad (and would have obviously been so with a little more exploration of that choice).

Although I have now somewhat addressed that issue, because it was unavoidable for move heuristics (see later), it lead to me adopting approach (2) preferentially for state heuristics.

Since any pseudo-playout results will propagate upward through all ancestors, if we simply apply the heuristic at every node we will tend to multiply-count it in a way that is proportional to the tree depth at that point (it will apply every time we expand a descendant that still has whatever condition the heuristic is detecting). This effectively makes it very noisy, so instead I apply a differential measure - for example, if we have a piece count heuristic it will apply a boost across a state transition that captures a piece, but we don’t want it to apply in all states that are a piece ahead if the entire tree has that condition (that just introduces noise). Ideally the best way to do this is probably to compare the heuristic value of a newly expanded child with that of its parent and use the difference, but having to continually recalculate parent values is expensive, so I actually use a differential relative to the current turn’s root (which can thus be calculated once).

Sancho performs state heuristic blending as follows:

* The heuristic function returns a heuristic score vector for the state (that is a value for each role, such that the values total 100)
* The vector is compared with the root’s vector, and if the difference is below a threshold no values are applied
* The vector is normalized against the root vector (since it’s based on heuristic comparison to the root state it represents a difference in value relative to the root’s value) effectively resulting in a vector which is the root’s score vector ‘adjusted’ by the heuristic vector – in particular this will have the property of totaling 100 across roles.
* The resulting vector is applied as a pseudo rollout result with a pseudo visit count of a heuristic weight for the heuristic in question
* Currently the only state heuristic used (and only in some games) is a piece count heuristic, when identified during meta-gaming (see later). The heuristic weight is dynamically set at the start of each turn according to the material difference then (for more even states the weight is higher because material advantage will be significant, but for uneven states it is set lower because either we’re well ahead, in which case focusing on position with unbiased MCTS is likely to be better, or we’re well behind, in which case we’d better find a non-material way out!)

### Blending move-heuristic signals

One might also have heuristics for moves (somewhat) independently of states. Because these pertain to (sets of) edges rather than individual nodes one cannot use the same technique as is used for state heuristics. The only mechanism I know of to cope with move-based heuristics like this, is to use them to adjust selection probability (i.e. – an extra term in the UCT value in effect, though possibly it should be thought of as a term in the exploration element of the value). This leads to the problem outlined earlier, that best score no longer correlates reliably with most-explored (least uncertainty). Since there doesn’t seem to be any way to side-step this I instead cope with it in the algorithm used to choose the actual move to be played (see later section). Assuming this is indeed coped with, then using the heuristic value to influence selection probability works to increase search density in the part of the state space preferred by the heuristic.

Currently this is experimentally (mixed results, really ongoing work to be done here) used to implement move action histories, whereby moves in a given node are given a weight based upon the rollout results that propagate to the node as a result of selecting through that move choice. During playout result propagation, move weight vectors are also propagated upward and accrued into the averages for the move weights at each node on the path (decaying by a fixed factor at each propagation step). During the expansion phase move weight vectors are propagated DOWN the selection path (and again decayed at each step), with each node’s own vector being averaged in at each step, and the result applied to provide a heuristic to modify move selection probabilities in the UCT calculation (treated a bit like the exploration term in that the weight given to the heuristic is reduced as the child gets more sampled and thus loses uncertainty). In effect this propagates around the tree the benefit of selecting a given move in ‘nearby’ states – that is it captures the notion that if a move is good in one situation it probably is in similar situations.

This mechanism certainly still needs work and appropriate tuning is (as yet) not obvious!

## Choosing the best move to play

When we need to actually respond to the selectMove request we need to decide which child of the root to actually play. Naively just choosing the one with the highest score is the best bet, but there are some issues with this and some alternatives to consider:

* Because of heuristic bias, the scores may be divergent from the selection rate, and it’s never a good idea to select a move with very low certainty
* If one or more children of the root are complete then we have an absolute result if we choose it – if one such is a win then clearly it should be chosen
* If we are in a situation where the upper bound on our score is less than 100 then the logic above about complete wins applies to complete nodes with the upper bound score
* Some consideration might be given to actually searching the MCTS structure (shallowly) to determine the move choice that has least bad outcomes a little deeper down
* If all nodes are losses which should we choose!?

To handle these issues, Sancho works as follows (roughly!):

1. It calculates a ‘selection value’ for each move which is the score, down-weighted by a measure of the uncertainty, and chooses the child with the highest selection value, not just the highest score
2. In the event of ties it prefers complete nodes to incomplete ones
3. It uses the highest and lowest observed playout scores as approximation of the achievable upper and lower bounds, and will not choose a complete node with score equal to the lower bound if there is a choice (this is partly a don’t throw your king away in escort latch breakthrough measure!)
4. If all choices are complete losses we choose the one with highest variance (if known) – this maximizes the opponent’s opportunity to make a mistake!
5. I did some experimentation with applying a shallow minimax to the MCTS structure, and this was effective in some games but unreliable (sometimes resulted in choosing based on nodes with too little certainty). This is currently disabled, but still present and may warrant further experimentation later.

# Threading Model

Sancho uses the following threads:

* A pool of playout threads, each of which owns one channel in the statemachine (i.e. –a thread safe instance). These threads drain a playout request queue and place their results in a playout response queue. By default the pool size is 4
* An MCTS structure control thread, encapsulated by the TreeSearcher class. This is responsible for building the graph. It selects nodes to be expanded, queuing them to the playout queue when first expanded, and also drains the playout response queue, implementing structure update in the processing of the responses so obtained. Once kicked off (at the end of meta-gaming) this thread basically free-runs, and is the only thread that updates the MCTS structures
* Player request response thread – called on receipt of a network request to play a move. Synchronizes with the MCTS structure thread to give it new base states, and to read results (i.e. – the move to play)

*Note* – for iterated games and for some puzzles the MCTS structure thread is never spawned and separate search routines (much simpler on the whole) are run directly from the player request response thread. I have not documented these alternate search routines, as they are very peripheral (will add sometime)

# Meta-Gaming

Sancho uses meta-gaming to simulate the game and extract characteristics from it to tune search parameters. The following are determined (or at least educated guesses are made!):

* Is the game a puzzle (single player)? If so is it one we didn’t actually see any useful results for in random simulation (i.e. – all we ever saw is a score of 0), and if so can we construct a target state that would be a solution? – if we can we do, and that set of conditions triggers the use of a separate puzzle search algorithm with a state distance heuristic relative to the discovered solution state (how I original solved 8-puzzle, but I intend to change this to A\* sometime).
* Is the game multi-player (>2 roles)? This triggers use of VERY crude player modeling whereby we ascribe a ‘rationality’ score to each player, and use that to modulate their exploration bias (that is a totally irrational player will explore the tree uniformly). This is just the beginnings of a mechanism that needs significantly more work, and it’s motivated by multiplayer games where some of the roles are the ‘Random’ player on Tiltyard. The idea is that by measuring the correlation between what a player actually does, and what we think they should do, their rationality can be updated (currently we don’t do this – we just assume a bit of irrationality for all other players in >2 role games). A highly irrational player will have randomly explored the tree, and thus feed into the MCTS graph a result that assumes that behavior. By way of example, consider a game of N-player free for all, where one or more opponents is random. If we can detect this as the game goes on we can start playing moves that rely on that player not making the best move (e.g. – not capturing us when we move next to it!) and extract a significant advantage
* Is the game an iterated game? If so a different search routine is used (this is very crude and I only did it because Sam suddenly deposited several iterated game-theoretic games such as iterated prisoner’s dilemma on Tiltyard a once, and everyone was doing really badly at them!)
* How fast can we simulate this game? (used to set the sample size for a rollout)
* What is its branching factor and its observed range of lengths? These are used to determine a sensible (hopefully – this is very empirical) exploration bias
* How effective do we think greedy rollouts are likely to be? Also how much will they cost? These measures are used to decide whether to enable or disable them
* Can we see any piece heuristics? – this works by analyzing the GDL for sets of propositions that look like they MIGHT represent pieces, and then gathering correlation stats for those heuristics with game results in random sampling. Those that do have a positive correlation (and one or two other characteristics) are selected, and if this is not the empty set, use of heuristics is turned on
* Is the game a simultaneous move game? If so then lots of special handling occurs during MCTS search because we have incomplete information in any node from which we’re making a move choice for any role but us (the resulting state is not just dependent on the role choice but also on the choice of all other roles). This basically means that the selection value for a move depends on its average across all cousins, and the rules for propagating completions upward are much more complex.
* Is this a special case of simultaneous game that is actually factorable into two non-simultaneous games (simultaneous connect4, Chinook, …)? We attempt to identify these (very heuristically) and treat them as NOT being simultaneous after all! (we don’t try to actually factor them yet)
* Once we’re done analyzing (we currently allow up to 10 seconds of random simulation to determine all the above stats, which is generally plenty), we kick the MCTS thread off (for non-puzzle/iterated games anyway) so it can be thinking about the initial state for the rest of the allotted meta-gaming time (this is a huge help in some games, notably Bidding-TicTacToe10, where MCTS takes a long time to converge and will play a move that is actually a loss with best play if not given time to settle on the initial move)