Playing Clue®:

Presenting a Computer Player for the Classic Board Game

Arthur Riley Siebel ‘11
Advisor: Dr. Robert Schapire

Table of Contents

I. Introduction

Clue® is a murder/mystery themed board game originally published by Waddington's in Leeds, UK in 1949. The object of the game is for the players to discover the identity of three cards have been hidden away in the "case file". These cards represent the murderer, murder weapon, and crime scene of the murder of Mr. Boddy. The players must strategically move around the game board and reveal clues about the the hidden cards. The formulation of an optimal strategy for this game is a non-trivial task. In this paper, the development of an intelligent computer player for the game Clue® is presented.

While Clue® is a small problem, it may offer insight to more general principles. Clue® is one of a group of problems called "treasure hunts". A treasure hunt is a game where the players must traverse the environment and locate one or more "treasures." The players have "sensors" that they use to gather clues. The information the sensors reveal is dependent on the players' locations. Landmine detection is one treasure hunt with serious consequences. The robot must locate as many mines as possible, and is equipped with instruments that gather information to help it do so. In the best case, the robot would like to minimize then energy spent on locomotion and sensor activation as well as avoid getting blown up.

Solving a treasure hunt is a significant challenge, and involves overcoming several independent problems and integrating the solutions. The required functionality includes deciding where to move and which sensor measurement to take, maintaining a

belief state about the location of the treasure, and incorporating new information to updating the belief state.

Computer programs that play Clue® do exist. Cai and Ferrari, researchers at Duke University, published an approach based on Bayes nets in 2006.[1] Commercial AI's exist as well. One of the earlier computerized Clue® games incorporated a viciously difficult computer opponent.[2] These AIs represent important steps toward formulating a general strategy for the treasure hunt, but each has disadvantages. Exploring different approaches may offer alternate solutions and recast the problem in a new light.

This paper introduces a logical agent plays the board game Clue®. It incorporates decision-making algorithms to decide between the available actions. The algorithms choose between the choices by selecting the action that it expects to reveal the most information in the shortest amount of time. The information received is converted to statements of propositional logic and stored in a knowledge base (KB), which represents all the agent's knowledge. The agent's belief about the identity of the cards in the case file is represented by a sample of the possibilities and is kept up to date with new knowledge by a variant of particle filtering. In addition to the AI, a Clue® simulation was developed to allow the computer player to compete against human players.

The paper is organized in the following way. Section II provides a description of the game rules. Section III reviews the necessary background. Section IV provides a framework in which to organize the architecture of the overall AI. Section IV describes the logical agent, section V the belief agent, and section VI the action agent. Section VII

---

[1] Cai, Chenghui, and Silvia Ferrari, "On the Development of an Intelligent Computer Player for Clue®: a Case Study on Preposterior Decision Analysis." 2006.
[2] Interview with Chris Nash

describes the clue simulation that was developed and reports the results of informal testing. In Section VII similar work is presented and compared, including research by Cai and Ferrari.[3] Section IX presents opportunities for further development and research and section X concludes.

II. The Game of Clue®

The Clue® game board represents Mr. Boddy's mansion and contains of nine rooms: the dining room, library, billiard room, hall, kitchen, lounge, ballroom, study and conservatory, as well as hallways connecting the rooms. The game board is shown in figure 1. The players each assume the identity of one of the six suspects: Colonel Mustard, Miss Scarlet, Professor Plum, Mr. Green, Mrs. White, and Mrs. Peacock. The possible murder weapons are: the rope, lead pipe, candlestick, revolver, wrench and knife. The game also includes a deck of twenty-one cards, each representing a room, suspect, or weapon. At the beginning of the game, one card of each type is randomly selected and secretly placed in the case file; discovering the identity of these cards is the objective of the game. The rest of the cards are dealt, face down, to the players.

During game play, the players move pawns around the board by rolling a die; each player has a starting position. There are three ways for the players to enter one of the rooms: through a door, through a secret passage from another room, or by way of another player's "suggestion". Whenever a player enters a room they have the option of

---

[3] The format of this paper is heavily influenced by Cai and Ferrari's "On the Development of an Intelligent Computer Player for Clue®: a Case Study on Preposterior Decision Analysis." Both papers present AI's to play the game clue.

making a suggestion, a guess as to the contents of the case file. Whichever suspect the player suggests is brought into the room with the suggester.

Suggestions are the method by which the players reveal clues. When a suggestion is made, the other players must take turns trying to refute the suggestion: if a player has any of the three cards being suggested he/she must show the card (in secret) to the suggester. The information these refutations reveal is used to infer the contents of the case file, and making suggestions that elicit informative refutations has a lot to do with winning the game. Which suggestions can be made depends on the location of the suggester, since a suggestion must include the room from which the suggestion is made. After the suggestion is refuted (or not, if no player is able) play proceeds to the next player.

When a player feels confident enough, on any turn he/she may choose to make an "accusation". An accusation has the same format as a suggestion except that it can be made from any square on the board, and does not have to involve the room occupied by the accuser. After an accusation, the accuser checks the case file in secret. If the accusation matches the case file the player reveals the case file to the other players and wins the game. Otherwise, the player loses: they no longer get to move their pawn or make suggestions or accusations but they are required refute suggestions as before.

Fig. 1, The Clue® board

III. Development of a POMDP based AI

A. Review of Partially Observable Markov Decision Processes

Any treasure hunt, including Clue® can be modeled with a partially observable Markov decision process (POMDP). A Markov decision process (MDP) is a framework for describing a sequential decision-making process when the outcomes are partially random and partially under the control of the decision-making agent. In general, agents are able to determine an optimal strategy for most MDPs. POMDPs generalize MDPs to situations where the system dynamics are specified by an MDP, but the underlying state information is not necessarily observable by the agent.

Formally, a POMDP is a tuple (S, A, O, T, Ω, R) where:

- S is the set of all possible states,
- A is the set of all possible actions,
- O is the set of possible observations,
- T is a set of transition probabilities,

- Ω is a set of observation probabilities,
- $R : A,S \rightarrow \Re$, a reward function.

At each time step, with the environment is in some state $s \in S$, the agent takes an action $a \in A$ and receives a reward $r(a,s)$. At the next time step, the environment transitions to a new state *s'* according to the distribution $T(s' \mid s,a)$. In state s', the agent observes $o \in O$ according to the distribution $\Omega(o \mid s',a)$.

In an MDP, agents are generally capable of formulating an "optimal policy", a strategy that results in the highest expected total reward. In order to maximize its total reward, an agent only has to execute the action specified by the optimal policy for its current state. Because the agent in an MDP knows the current state at all times, it is able to use this policy to maximize its reward.

While agents in a POMDP can calculate an optimal mapping of states to actions, they cannot use this information directly to decide what to do. In a POMDP, the agent doesn't necessarily know what state the environment is in and might not know *T* or Ω either. Instead the agent maintains a "belief distribution", indicating how much the agent believes the world is in each possible state. The agent updates the belief distribution in the face of new evidence, and makes decisions by choosing the action with the highest expected reward given its current belief about the world.

B. Overview of AI Design

The approach presented in this paper uses the framework of a POMDP to structure the AI. A block diagram is provided in figure 2, the blocks are named

according to their functionality. The logical receives observations and translates them into clauses of propositional logic. It performs inference on these clauses using resolution. The particle filter maintains a sample of possible models to provide an estimate of the current state *s* as well as estimates for *T* and *Ω*. These samples are updated with the information maintained by the logical agent's model checking functionality. The decision maker uses the estimates provided by the particle filter to calculate the expected reward reward action. The decision maker chooses the action that maximizes the expected discounted value of information revealed by the action.

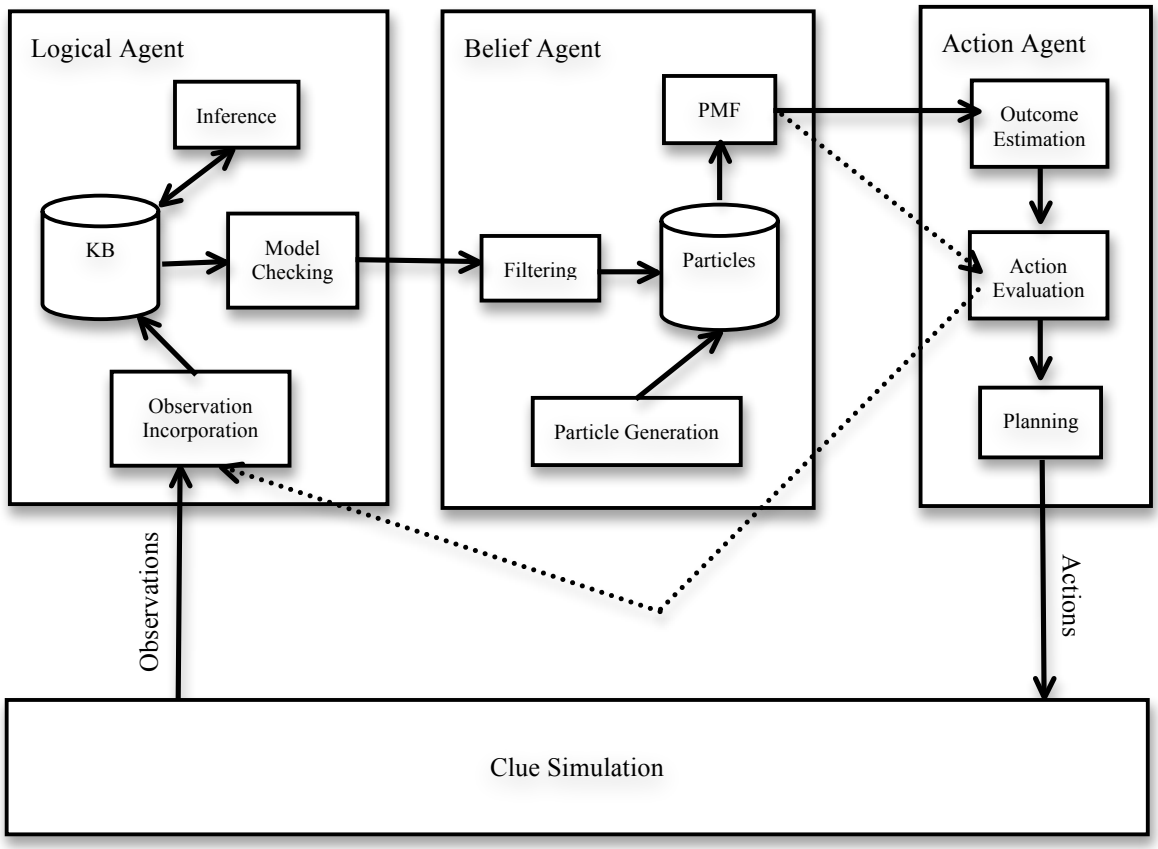

Fig. 2. Architecture of a Clue Agent

IV. Logical Agent

The logical agent is responsible for all the evidence gathering and determining whether a card can possibly be in a certain location given all the knowledge it has accumulated. Whenever a new observation is made, the agent incorporates the new clues into its knowledge. Beyond maintaining the information received, the agent requires the ability to synthesize new clues from what it already knows using logical reasoning techniques. Finally, the reasoner utilizes its sum total knowledge to judge which deals of the cards are possible at any point in the game. The framework for a propositional logic reasoner for the game of Clue® was borrowed from Clue® Deduction: an Introduction to Satisfiability Reasoning.[4]

A. Propositional Logic

The approach presented in this paper represents knowledge with clauses of propositional logic. Propositional logic is a framework for representing relationships between "statements". A statement is an assertion that might be true or false; for instance "the wrench card is in my hand," could be a statement, or symbol, in a propositional logic. The symbols used in the Clue® reasoner's logic represent assertions about the locations of the various cards.

Another approach would be to view the cards as random variables in a Bayes net. This solution views the observations as evidence, and updates a belief distribution

---

[4] Neller, Todd W., Zdravko Markov, and Ingrid Russell. *Clue: An Introduction to Satisfiabillity Reasoning*. 10 Aug. 2005.

describing the agent's belief about the setting of the random variables according to the relationships specified in the net. This approach was explored by Cai and Ferrari in 2006.[5] While this is a valid approach propositional logic offers an alternative framework for reasoning about Clue®,

Each symbol is an assertion that a particular card is in a particular location. The symbols are given unique identifiers specified by equation 1.

$$S = P \times num\_cards + C$$

To calculate the identifier of any assertion about the location of any card the identifier of the location (L) in the assertion is multiplied by the number of cards and added to the identifier of the card (C). The locations are numbered from 0-*num_players*, with the identifier *num_players* referring to the case file. The cards are numbered from 0-20. In the situation where all 6 players are playing there are a total of 147 symbols. By constructing clauses out of these symbols, the logical agent can represent its knowledge of the game.

Clauses of propositional logic combine symbols to form more complex and expressive logical ideas using logical connective. For example, a useful clause might state that the card representing the Library is in the hand of player 0 'or' player 1 'or' the case file. Since each of these statements is an assertion about the location of a card, we can calculate their identifiers according to equation 1. Assuming the library has identifier '2' and the case file has identifier 3, the example above would be represented as "2+23+65," where represents the logical 'or' operator. The agent represents its collection of clauses in a single "sentence" in conjunctive normal form (CNF). Each clause is a

[5] Cai and Ferrari, "On the development.."

disjunction of several symbols. Any symbol in a clause can be negated, but no clause may contain a symbol and its negation. Representing knowledge as a CNF sentence is useful because it allows relatively simple inference through implementation of the "resolution rule."

B. Prior Knowledge

Before play begins, there exists some general knowledge about the game that all the players possess. These facts represent knowledge of the rules of the game and the physical laws under which the game operates. Even before the first suggestion is made all the players know that a card can only be in one place at a time. The agent needs this sort of "intutitive" knowledge in order to perform inference on the clues it learns later on. There are four basic facts the logical reasoner needs to know; the algorithms to generate the CNF clauses that represent these facts and add them to the knowledge base is included below. (A location refers to a possible place a card might be, the set of locations includes the hands of all the players plus the case file)

- Each card is somewhere

```
for (Card c in deck)
   Clause cl = new Clause;
   for (Location l)
      cl.add(Symbol(c, l), true);
   KB.add(cl);
```

- If a card is somewhere, it isn't anywhere else

```
for (Card c in deck)
   for(pair of Locations [l1,l2])
      Clause cl = new Clause;
      cl.add(Symbol(c, l1), false);
      cl.add(Symbol(c, l2), false);
```

```
KB.add(cl);
```

- At least one card of each type is in the case file

```
for (CardType t in {Suspects, Weapons, Rooms})
   Clause cl = new Clause;
   for (Card c in t)
      cl.add(Symbol(c, casefile), true);
   KB.add(cl);
```

- If a card of one type is in the case file, no other card of that type is also in the case file.

```
for (CardType t in {Suspects, Weapons, Rooms})
   for(pair of Cards [c1,c2] in t)
      Clause cl = new Clause;
      cl.add(Symbol(c1, casefile), false);
      cl.add(Symbol(c2, casefile), false);
      KB.add(cl);
```

C. Learning from Observations

Throughout game play, the logical agent is kept updated with all the observations the player makes. Every time any player makes a suggestion the AI is exposed to potentially new information. Suggestions reveal different information in different circumstances: when the AI makes a suggestion, it gets to see which card the refuter reveals; when a different player makes a suggestion, the AI still learns the identity of the refuter, but not which card was revealed. The agent requires techniques to generate CNF representations of these observations in each circumstance and incorporate them into the knowledge base.

Suggestions can result in three kinds of observations. The first case is a suggestion with no refutation at all. This results when a player makes a suggestion and no other player was able to refute it because none of them have any of the cards being

suggested. The rules state that each player must attempt to refute every suggestion. If no player can make a refutation, then none of them have the cards.

The same reasoning holds when only some of the players fail to make a refutation. Every time a suggestion is made, the agent learns that each player that failed to make a refutation does not have any of the cards being suggested. The following algorithm incorporates the knowledge that comes from players failing to make a refutation in the knowledge base.

```
FR = set of Players that failed to refute;
for (Player p in FR)
   for(Card c in suggestion)
      Clause cl = new Clause;
      cl.add(Symbol(c, p), false);
      KB.add(cl);
```

The second kind of observation is made when the AI is not the suggester, but some player is able to refute the suggestion. In this case players who fail to make a refutation add to out knowledge as above, but we also gain information by learning the identity of the refuter. Since the refuter was able to refute the suggestion, it follows that he must hold at least one of the cards being suggested. This knowledge is incorporated in the knowledge base as follows.

```
Player p = refuter;
Clause cl = new Clause;
for (Card c in suggestion)
   cl.add(Symbol(c, p), true);
KB.add(cl);
```

The last case occurs when the AI *is* the suggester and a refutation is made. In addition to all the usual information about the players who fail to make a refutation, the player learns the identity of one card in the refuter's hand, namely one of the cards the player just suggested. The procedure for this case is very simple.

```
Player p = refuter;
Card c = card revealed;
Clause cl = new Clause;
cl.add(Symbol(c, p), true);
KB.add(cl);
```

D. Inference

Beyond translating observations into clauses and remembering them in a CNF sentence, the logical agent has the ability to infer new information from what it already knows. This process is based on repeated applications of the "resolution rule." The resolution rule provides a procedure that given two clauses in CNF containing complementary literals, produces one new clause synthesizing the information from both. Formally, given two clauses $l_1+...+l_k$ and $m_1+...+m_n$, where l_i and m_j are complementary literals, the resolution rule produces one clause $l_1+...+l_{i-1}+l_{i+1}+...+l_k+$ $m_1+...+m_{j-1}+m_{j+1}+...+m_n$. Resolution has been shown to be a complete inference rule, it finds every new clause that is implied by current knowledge.[6]

Each time information is added to the logical agent's knowledge base, it runs a resolution algorithm in a loop until all the new clauses has been found. On each iteration of the algorithm resolution is applied to each pair of clauses that contain complementary literals. The resulting clause is discarded as a tautology if it contains complementary literals. Otherwise, if the KB doesn't already contain the new clause, it is added to the knowledge base. When no new clauses can be added the resolution algorithm stops and the KB contains all the clauses that can be inferred at the current time.[7]

---

[6] Russell & Norvig, "Artifical Intelligence, a Modern Approach."
[7] Ibid. 255

```
for (pair of Clauses [C1,C2] in KB)
   resolvents = Resolve(C1,C2);
   for (Clause C in resolvents)
      if(!KB.contains(c))
         KB.add(c);
Set<Clauses> Resolve(Clause C1, Clause C2)
   resolvants = new Set<Clauses>;
   for (Symbol s in C1)
  if(C2.contains(!s))
         Clause c = (C1-s)+(C2-!s);
         if(c is not a tautology)
           resolvants.add(c);
   return resolvants;
```

E. Model Checking

The primary function of the logical agent is to determine whether a particular "model" is "satisfied" by the knowledge base. A model is a unique deal of the cards. The belief agent maintains a set of models that roughly constitute a representative sample of all the possible models given the knowledge base. To update its samples with new information the belief checks each model with the logical agent. If the logical agent identifies a model as not satisfying the KB, the belief agent replaces it with a new. This is the primary functionality of the logical agent and is the motivation for observation incorporation and inference.

To check if a model satisfies the knowledge base, the logical agent first represents the model as a particular setting for each symbol in the logic. Then for each clause in the KB, the agent ensures that at least one literal in the clause agrees with its setting in the model. Since each clause is a disjunction of literals, if any symbol is true the whole clause is true. Verifying each clause is necessary because each it is a conjunction of

clauses, all the clauses have to be true for the CNF to be true. The following algorithm verifies that a particular model satisfies the KB.

```
boolean Satisfies(Model model)
   boolean model_symbols[num_symbols];
   for (Card c in model)
      model_symbols[Symbol(c, c.location)] = true;

   for (Clause cl in  KB)
      boolean satisfied = false;
      for (Symbol s in cl)
         if(s.sign == model_symbols[s.id])
            satisfied = true;
            break;
      if(!Satisfied)
         return false;

   return true;
```

V. Belief Agent

The AI relies on a belief agent to maintain a belief distribution indicating with what probability the AI believes each card is in each location. Every time the AI makes an observation, the belief agent has to update the belief distribution to reflect this new information. In particular the belief agent must be able to estimate the probability that a particular card is in a particular location given all the knowledge available to the logical agent up to the current point in the game.

The simplest solution is to check each possible model. This approach requires storing every model that is allowed according to the current KB and throwing out models that are invalidated as new information comes in. The advantage of this approach is that the agent has on hand the exact probability of each card being in each place, with almost no calculation. The agent only needs to count the number of times a card appears in a

location in the sample set and divide by the number of samples to determine the actual, real probability that the card is in that location given the sum total of the observations it has made. Unfortunately the amount of space required to store all these models is prohibitively large. Assuming one byte to store the truth value of each symbol in the model, storing all 6.4×10^{17} possible deals would require 81 *exabytes* of memory! Clearly, the amount of memory required makes this approach intractable.

The approach presented in this paper is an adaptation of particle filtering. The belief agent stores a set of samples, or particles, each a guess as to the location of each card. These samples are filtered by incoming information: the agent removes the particles that are no longer possible given the new knowledge base. The sample set is repopulated using a partially-random seed, increasing radius, mutation algorithm whenever samples are removed. By counting the number of times a particular card-location pair comes up in the set of samples, the belief agent can approximate its true probability.

A. Initial Belief

At the beginning of the game, the belief agent has to create and store the set of particles that represents its initial wild guess. Each particle is a particular setting for each of the up to 147 symbols in the propositional logic, each is set to true or false. The belief agent uses rejection sampling to generate these particle. After generating a random true false value for each symbol the agent checks if the model satisfies the KB according to the logical agent. If it does the sample is added to the set of samples, otherwise a new

random sample is generated to ensure the belief distribution does not degrade in quality from losing samples.

```
for (int i = 0; i < num_particles; i++)
   Particle p = new Particle;
   while (!p.satisfies(KB))
      p = new Particle;
   particles.add(p);
```

While this approach works it leaves much to be desired in terms of speed. While there are approximately 10^{17} legal deals there are 10^{44} possible settings for the 147 symbols. If the agent has to generate samples as described above only 1 in every 10^{27} samples it generated would satisfy the knowledge base. Instead the agent utilizes the prior information it has (described in section IV. *B.*) to ensure that every particle it generates satisfies the knowledge base.

The information in the logical agent's prior knowledge describes the fact that each card is in exactly one place; this follows from the laws of physical reality. Similarly, there is exactly one card of each type in the case file, as specified in the game rules. The agent can ensure that the samples it generates always satisfy the prior knowledge by simulating an actual deal of the cards. Instead of randomly flipping the values of all 147 symbols, the agent simulates randomly selecting three cards to put in the case file, then deals the rest of the cards out to the players in the same way the dealer would have. In addition to satisfying the prior knowledge, this technique provides a more subtle benefit.

Say there are four players in a game of clue. After three cards are placed in the case file there are eighteen remaining to be dealt out. In this example two players would receive three cards and the other players would receive four. Also, since the room cards are dealt first, then the suspect cards and then the weapon cards, each player would get a unique number of each. Simulating the deal in the same way that the dealer would do

ensures that each player in each of our samples will have the correct number of cards of the particular types, in addition to ensuring each card is in exactly one place. After simulating the deal it is translated into a setting for each symbol. That logic is omitted below.

```
cf = casefile;
PN = player #N's hand;
cf.add(deck.getRoom());
cf.add(deck.getWeapon());
cf.add(deck.getSuspect());

n = 0;
while((Card c = deck.getRoom()) != null)
   Pn.add(c);
   n = (n + 1) % num_players;
while((Card c = deck.getSuspect()) != null)
   Pn.add(c);
   n = (n + 1) % num_players;
while((Card c = deck.getWeapon()) != null)
   Pn.add(c);
   N = (n + 1) % num_players
```

B. Incorporating New Information

The particle filtering algorithm above removes particles from the set when they are found to be invalid in the face of new evidence. In order to ensure that its probability estimates do not degrade over time as particles are rejected the belief agent generates new particles to fill the gaps. A simple solution would be to generate particles according to the procedure outlined above until a valid particle is stumbled upon.

While this strategy is appropriate before the game starts when the agent has not learned any specific information about the locations of the cards, during the course of the game the agent ought to be able to use the information it has obtained so far to focus its

search for a valid particle. The algorithm described below attempts to focus the search around particles that it knows to be valid, while still allowing for randomness to find new solutions. Every time the agent needs to replace a particle, it attempts to mutate other particles to find new solutions. Each mutation swaps the location of two cards in the particle.

The algorithm begins with a seed particle, and attempts to randomly mutate it until it finds a valid particle. After a few trials the algorithm picks a new seed and tries again, this time for twice as long. On each iteration the algorithm attempts more mutations before giving up and trying a new seed. Half of the time, this seed is a randomly generated new particle. The other half of the time, the seed is a valid particle from the current set.

By randomly choosing between the two different kinds of seeds, the algorithm strikes a balance between finding samples that are similar to known valid particles, which are easier to find, while still discovering enough diversity of particles to provide a representative sample. Every time the agent learns information, entire areas in the space of possible deals are eliminated. Starting with known valid seeds allows the agent to search for samples in an area it knows to contain at least one valid point. Using random seeds, on the other hand, gives the AI a chance to find unexplored areas of the space that might contain valid deals. Presumably, after one of these unexplored areas is found, more will be discovered in the same area when that sample is used as a seed in the future. Randomly seeding the mutation algorithm gives the belief agent the best chance to find a large number of diverse particles relatively quickly.

In addition to random seeding, the particle replacement algorithm utilizes an increasing "radius." On the first iteration, the algorithm has a radius of two meaning it attempts as many as two mutations before giving up and trying a new seed. On each subsequent iteration the radius is doubled; the algorithm will spend twice as long looking for a valid particle before giving up. The radius conceptually represents the maximum distance away from the seed the algorithm is willing to search in the state space starting with any particular seed. The increasing radius serves two functions. By starting the radius small, the program ensures that the agent tries several possibilities for easy solutions before trying hundreds of mutations on a seed that might not be anywhere near any valid solutions. Increasing the radius gives the agent a chance exhaustively search an area of the state space after it has failed to quickly find a new particle. The number of mutations is capped at 64 to ensure the algorithm doesn't wander too far looking for a solution.

```
for (Particle p in bad_particles)
   int range = 2;
   while(!p.satisfies(KB))
      r = {true:0.5, false:0.5};
      if(r == true && good_particles.size > 0)
         p = good.get(random);
      else
         p = new Sample;
      while(range > 0)
         if(p.satisfies())
           break;
         p.mutate();
         range--;
      if(range < 64)
```

The algorithm above describes how the belief agent replaces particles that get removed during filtering. The random mutation process strikes a good balance between

finding new samples quickly while still searching randomly in the state space. The belief agent uses the particles it maintains to estimate the PMF of location of the various cards.

VI. Action Agent

The action agent is responsible for achieving the AI's goal of making the winning accusation before the opposing players do. Every turn it decides where to move its pawn and potentially which suggestion to make according to the roll of the die and its current belief distribution. To have the best chance of winning the game the agent attempts to reveal as much information about the identities of the cards in the case file as quickly as possible. A suggestion might reveal different information depending on how it is refuted. The agent measures the expected reward of each refutation by computing how that refutation would change the entropy of the case file. The expected value of a suggestion is calculated by estimating the probability of each refutation given the suggestion. The agent plans a route to the room where it can make the best suggestion and takes into account the discounted value of future information using value iteration. An outline of the algorithm is provided.

```
MyTurn(int roll)
   double room_scores[];
   for(Room r)
      double best_score = 0;
      for(pair of Cards [s:Suspects,w:Weapons])
         score = SuggestionValue(r, s, w);
         if(score > best_score)
            best_score = score;
      room_scores[r] = best_score;
   ValueIterate(room_scores);
   MoveToBestSquare(roll);
```

A. Estimating reward

The action agent makes suggestions in order to elicit refutations from the other players. These refutations reveal information about the contents of the case file. Choosing good suggestions can maximize the chance of high value refutations, refutations that reveal more information about the case file. The value of a refutation is measured by estimating the reduction in the entropy of the case file that would result following the refutation. Entropy provides the measure of the uncertainty associated with a random variable. The case file is a random variable that can take on any suspect-weapon-room triplet as a value. A high entropy indicates that the AI is mostly unsure which cards are in the case file. When the entropy reaches 0, the agent knows for sure what the solution is. To win the game, the agent should choose to make actions that reduce the entropy of the case file to zero as quickly as possible.

Calculating the entropy of a random variable requires knowing its PMF. For a random variable X that takes on values $\{x_1,\ldots,x_n\}$ each with probability $p(x_i)$, the entropy is calculated according to the following formula: $H(X) = -\sum_{i=1}^{n} p(x_i)\log_b p(x_i)$[8], where b

[8] When $p(x_i) = 0$ the value of the summand is taken to be 0 as well, as is consistent with the limit $\lim_{p \to 0+} p\log p = 0$.

is the base of the logarithm. The units of entropy are called *bits* if $b = 2$, *nats* if $b = e$, and *digits* if $b = 10$.

To assign value to a refutation, the agent estimates the change in entropy that would result. The value of a refutation is equal to the original entropy minus the estimated new entropy, $V(r) = H(cf) - H(cf \mid r)$. The action agent estimates the new entropy by simulating the effect of the refutation on the logical agent and recording the belief distribution that results from filtering.

B. Evaluating Actions

To evaluate a suggestion, the action agent estimates the expected value of that suggestion over all the refutations it might provoke. When a player makes a suggestion, the opposing players each have a chance to refute the suggestion. The rules state that a player must refute the suggestion if they are able to do so, so if the suggestion is not exactly correct it will eventually be refuted and reveal some information. When an opponent has only one of the cards from the suggestion they are forced to reveal it. However, if the opponent has two or more of the cards being suggested choice of which to show is up to the refuter. The action agent assumes that in this situation the opponent chooses between its options randomly, but improvements to this scheme are suggested in section IX.

Assume player zero makes the suggestion $s = \{x,y,z\}$. Player zero is succeeded in turn order by player one, then player two, etc. Player one is given the first opportunity to make a refutation. Let $p(x,1)$ represent the probability that player one has card x. The

probability that player one will refute by showing x is calculated according to the following formula. Similar calculations can be made for y and z.

$$p(r_{x1} \mid s_{0xyz}) = p(x,1) + \tfrac{1}{2}\, p(x,1)\, p(y,1) + \tfrac{1}{2}\, p(x,1)\, p(z,1) + \tfrac{1}{3}\, p(x,1)\, p(y,1)\, p(z,1).$$

If player one does not have any of the cards then player two must attempt to make a refutation. The probability that player two will refute by showing *x* is equal to probability that player one did not have x or y or z times the probability that player two has card *x* plus one half the probability player two has both *x* and *y* etc. as outlined above. Because player one had the first opportunity to make a refutation, the number of possible worlds where player two can make a suggestion is limited to those worlds where player one failed to do so. The same relationship holds for all subsequent potential refuters. In general, the probability that the n^{th} player refutes a suggestion by showing card *x* is represented by the following formula

$$p(r_{xn} \mid s_{0xyz}) = \left(p(x,2) + \tfrac{1}{2}\, p(x,2)\, p(y,2) + \tfrac{1}{2}\, p(x,2)\, p(z,2) + \tfrac{1}{3}\, p(x,2)\, p(y,2)\, p(z,2)\right)\prod_{i=0}^{n-1}\left(p(\tilde{x},i)\, p(\tilde{y},i)\, p(\tilde{z},i)\right)$$

Equipped with the amount of information each refutation reveals and the probability that a suggestion will result in each refutation, the agent can calculate the expected information gain from making the suggestion. The expected value of a suggestion is calculated by summing the products of the value of each refutation and the probability that the suggestion will result in that refutation. Let E[s] be the expected value of suggestion *s* and let *R*(*s*) be the set of refutations that are possible in response to *s*, then $E[s] = \sum_{r \in R(s)} V(r)\, p(r \mid s)$. The action agent gives each suggestion a score equal to its expected value. A suggestion's score is a representation of the amount of information the

agent expects to reveal about the case file by making that suggestion. This calculation is outlined below.

C. Planning

Calculating the expected value of each suggestion allows the action agent to pick a suggestion to make. However, just as in any treasure hunt, each suggestion has to be made from a particular location on the board. In particular, a suggestion must be made from one of the nine rooms and the suggestion must include the room it is being made from in its guess. One solution is to simply plan a path to the room where the best suggestion can be made. The commercial video game *Clue: Murder at Boddy Mansion* implements this approach with A* search.[9] However, if getting to the best room requires travelling halfway across the board and a nearby room offers a suggestion that is almost as good, it makes sense for our agent to pick the closer room. The action agent needs to balance the quality of the information it gains with the amount of time it takes to do so.

If the score of the best suggestion in a particular room is *x,* that information is worth γx if it requires moving one space to get, $\gamma^2 x$ if it requires moving two spaces, etc. The γ term is called the "discount rate," and represents the fact that information is worth less in the future than it would be in the present. To plan a path that maximizes the

[9] Interview with Chris Nash.

expected present value of future information, "value iteration" is used to propogate the discounted scores across the board.[10]

Value iteration begins by assigning each room the score of its best suggestion, then iteratively updating each square's value to the value of its highest neighbor times the discount rate. As the algorithm progresses, the best paths emerge as trails of higher valued squares. The best path from any particular square travels to increasingly higher valued squares until it reaches the room that best balance the amount of information gained with the amount of spaces traversed to get there.

By simulating the results of a refutation, the action agent can predict how much information the refutation would reveal. This information is measured as the difference in the entropy of the case file. The belief agent's estimation of the PMF is utilized to predict the probability that a suggestion will result in a particular refutation and the expected amount of information gained by making the suggestion can be calculated. Value iteration is used to assign a score to each space on the board taking into account the discounted present value of future information. On its turn the action agent moves its pawn to the space with the highest score it can reach, when it reaches a room the action chooses the best suggestion that can be made from in that room according to the method described above.

VII. Game Simulation and Results

[10] Russell & Norvig.

In addition to a Clue playing agent, an interactive Clue game simulation was created to view the agent playing and to allow human players to compete against the AI. The game logic was written in Java and the GUI was written using Java Swing. The application begins by displaying a character choosing window, allowing the user to specify any number (3 – 6) of players, each of which can be a human or a computer player. At this point game-play begins with the deal occurring behind the scenes and the game board being revealed on screen.

Human players are controlled with the mouse; after being informed their random die roll, the reachable squares are highlighted and the player clicks on a square to move there. If the player enters a room, they have the opportunity to make a suggestion, choosing the suspect and weapon from drop down menus. The simulation then checks each player to see if they can make a refutation, and if they can it asks which card to reveal. If the refuter is a human there it is given a choice via radio menu, if the player is an AI a method is called which chooses which card to show.

Computer players interact with the simulation through a well defined API. A "move" a method is called to inform the AI of its roll, and requestiong the player to make a move. There are similar methods to determine if the player would like to make a suggestion, which suggestion the agent wants to make, etc.. In addition, each AI player is informed of all the suggestions and accusations made during the game.

The simulation successfully incorporates all aspects of the game's rules, including pulling players' pawns into a room after they are suggested as having committed the crime in that room. It also allows players to use the "secret passages" to travel from a room in one corner to the room in the opposite corner.

The AI players are notified of significant actions on the board (suggestions and accusations) through function calls. Each time an AI is notified, it adds the new information to its knowledge base, runs resolution on the new information, filters the particles in the belief state and refills the table of particles. The agent recalculates the best suggestion for each room and updates the scores of each board square to allow it to make a decision.

Unfortunately the performance of the AI has prevented any games from reaching completion. Even with a relatively low number of particles, the agent takes quite a while to go through the whole process of incorporating information, running inference, filtering the particles and generating new ones. This is an inherently complex procedure with many steps, but it shouldn't be taking as long as it is. In particular, generating new particles sometimes takes several minutes.

Much of this behavior is due to bad coding on my part. The overall architecture of the AI wasn't layed out until much of the code was written, so interoperability is an issue. For instance, the belief agent represents a particle by maintaining a mapping from players to cards, each mapping represents a player's hand. In contrast, when the logical agent checks models it expects to do so with an array representing the setting for each propositional logic symbol. Translating between the two representations is responsible for much of the slowdown, as checking the satisfiability of a model is a fundamental operation that happens a lot.

Besides the issue of speed the agent presented here is fairly successful. On each turn it correctly generates a score for each room according the belief distribution which depends on the information it has gathered so far. Whenever the agent is privy to an

observation it correctly incorporates that information into the knowledge base and can infer new clues using the resolution rule.

V. Similar Research

A. Bayes Net Approach

Silvia Ferrari and Chenghui Cai have written several research papers about the problem of developing an intelligent computer player for the game Clue. Their work uses Baysian network inference to maintain a belief distribution. The Bayes net they describe considers each 'slot' in a player's hand or the case file to be a random variable. Each time a slot has a card dealt into it influences which cards can fill the rest of the slots. However, performing inference on this Bayes net is an intractable problem, and a number of simplifying assumptions have to be made. In addition, their agent has no concept of balancing information with the time it takes to get the information, instead it simply enters the nearest room where it can make a reasonable suggestion.[11]

In another paper, Cai and Ferrari describe how this algorithm learns a Q function represented as a neural network. By improving the Q function this approach helps the

[11] Cai and Ferrari, "A Q-Learning Approach to Developing an Automated Neural Computer Player for the Board Game of Clue®," 2008.

Bayes net incorporate new evidence into its belief state more correctly. Their Clue® playing agent is based on strong theory, simplifying assumptions aside. However, their algorithm performs relatively poorly, winning 50% of the time against a 12 and a14 year old.[12]

B. Clue®: Murder at Boddy Mansion

A quick search on Google reveals the existence of another agent that was developed all the way back in 1998 by EAI.[13] This agent is famous for its difficulty, many players claimed that it cheated, but in fact it did not. Luckily I was able to interview Chris Nash, the lead programmer on the project, to find out how the AI worked on that project.

This AI was incredibly simple. Instead of implementing a formal logic, the team gave the AI a special notepad on which it could mark not only where cards were, but where they *weren't.* This has the same effect of doing formal logic, but without a lot of the general applicability to other treasure hunts. To decide which room to go to, the AI simply picked the closest room in which it could learn any information, using A* search to find the shortest path to that room, and choosing among the suggestions that might provide information more or less randomly.

This approach has almost no formalism associated with it. Indeed it clearly makes bad decisions sometimes, going to the closest room even if a further room has more opportunities for information gathering. Even then this algorithm was so good that

[12] Cai and Ferrari, "Q learning"
[13] Interview with Chris Nash.

the most accomplished players found it very difficult to beat, accusing the AI of "cheating." While there aren't any numbers to back this up, the testimonials seem to indicate that this AI was capable of winning much more often that 50% of the time.

VI. Further Work

While the logical agent described in this paper is capable of making use of most of the information revealed during a game of Clue, there is still room to improve its logic. In particular, when the agent believes an opponent has more than one card being suggested is models the opponent as choosing each between them randomly.. This approach ignores the complexities of attempting to hide information, assuming that the effect of revealed information on the outcome of the game is negligible compared to other factors. Most of the time the agent doesn't even have a choice, as the chances are that if it has one suggested card it probably doesn't have another. However, a smart opponent would remember having made a particular refutation and would attempt to reveal the same refutation instead of new ones to limit the amount of information it reveals. While simplifying out this factor is a reasonable approach, there are several steps that can be taken to improve performance on this margin.

The agent sometimes has the same choice of which card to choose . When refuting a suggestion, the goal of the agent should be the same as during every other action: maximize the amount of information the agent learns per turn, while minimizing

the same measure for the other players. When making a refutation, the agent is revealing information to all the players on the board. The player who actually sees the card gets the most information, while the other players also learn something. To minimize the information revealed, the agent needs to keep track not only of what it knows itself, but also what it suspects the other players know.

Every time another players' suggestion is refuted, the agent knows that the suggester just saw one of the three suggested cards. If the agent later learns that the refuter didn't have one of these cards, then the agent knows that the suggester must have seen one of two cards. As the agent learns more of where the cards are, it is capable of filling in more information about what cards the others have seen. After implementing this sort of logic, the agent should have at least an idea of what every player knows about each other, including what the other players know about it.

This information could be used in several ways: the most obvious being the ability to make a refutation that reveals the least amount of information. However this information could be used when deciding which suggestion to make as well. Currently, as described in Section (scoring rooms section), the agent assigns all refutations that an opposing player could make equal probability. But probabilities could be assigned to these refutations according to how much information the *opposing* player believes making that refutation would reveal. For instance, an opponent would definitely choose to refute a suggestion with the Library card if it knows the agent knows or suspects the opponent has that card, as that reveals almost no information. This would improve the suggestion evaluation process by assigning more realistic probabilities to the refutations.

VII. Conclusion

An intelligent agent was designed to play the game of Clue®. The agent's goal is to infer the identity of the cards in the case file in the smallest number of moves, in order to beat its adversaries. A propositional logic approach was developed to reason about the observations made during the game. This reasoning enabled a particle filter, which was used to maintain a belief distribution over the possible states of a POMDP. The entropy of the case file was used as the measure to be minimized by the agent. Each room is given a score based on the potential entropy reduction from making a suggestion in that room. Value iteration was utilized to propagate the discounted value of being in these rooms to the hallway spaces. During its turn the agent moves to the square with the highest value it is able to reach, and makes the suggestion with the highest score when it reaches a room.

While the agent demonstrated the feasibility of this approach to playing the game, more development is necessary to ensure the AI can play the game in real-time. This includes improvements to the particle filtering and resolution algorithms. In addition,

further work is proposed to improve the results of the AI. One such proposal is to introduce the ability to reason about the state of the *other* players' knowledge bases.

The framework presented in this paper can be generalized to many instances of the treasure hunt problem. However, this approach is particularly well suited to situations where the agent is searching a discrete, finite search space. Because the number of locations for the treasure (the identities of the cards in the case file) is finite and no card can be between two locations, this approach used propositional logic to represent knowledge about the treasure's location. To use a propositional logic framework to represent knowledge in a treasure hunt where the agent must search a continuous space for the treasure one approach would be to divide the search space into a collection of finite regions with a grid. Representing knowledge about an infinite space using propositional logic presents an even tougher challenge.

XI. Works Cited

Cai, Chenghui, and Silvia Ferrari. "A Q-Learning Approach to Developing an Automated Neural Computer Player for the Board Game of CLUE®." Proc. of 2008 International Joint Conference on Neural Networks, Hong Kong. 2347-353. Print.

Cai, Chenghui, and Silvia Ferrari. "On the Development of an Intelligent Computer Player for Clue®: a Case Study on Preposterior Decision Analysis." *Proceedings of the 2006 American Control Conference*. Minneapolis, Minnesota, USA. 4350-355. *Laboratory for Intelligent Systems and Controls*. Duke University. Web. 20 Apr. 2011.

Cover, T. M., and Joy A. Thomas. *Elements of Information Theory*. New York: Wiley, 1991. Print.

Nash, Chris. "Inverview with Chris Nash, Lead Programmer for Clue: Murder at Boddy Mansion." E-mail interview. 26 Mar. 2011.

Neller, Todd W., Zdravko Markov, and Ingrid Russell. *Clue: An Introduction to Satisfiabillity Reasoning*. 10 Aug. 2005. Lesson Plan.

Russell, Stuart J., and Peter Norvig. *Artificial Intelligence: a Modern Approach*. Upper Saddle River, NJ: Prentice Hall, 2010. Print.

XII. Appendix

A. Interview with Chris Nash

Thanks Chris,

I really appreciate you taking the time to email me.

First off, I wanted to know how the AI "scores" different states.
In my design I calculate the expected drop in the entropy of the case file (the solution) in the different rooms,
but I have seen another design that uses machine learning to achieve a similar effect.
Do you know how the Clue AI assigned scores?

The other major issue seems to be path finding, I'm thinking of doing value iteration with the values in the rooms, but there might a better way.
Did you find any interesting solutions to this problem?

Again, thank you so much for your time,

Riley Siebel

Hi Riley,

Like I said, I didn't write the AI portion for Clue, but the programmer who designed it didn't approach it the way you did at all. Let me tell you how it worked and perhaps that

will answer some of your questions.

The AI didn't cheat; that is, the program didn't give the AI any more information than it gave other players. What the AI did was to keep better track of who had which cards. It not only kept track of which players had which cards, it also kept track of which cards players *didn't* have. In this way, it was able to solve the game much faster than a regular player.

The best way to explain this might be to take a look at a standard Clue Detective Note Pad sheet. On it you'll see a list of Suspects, Weapons and Rooms. What you're supposed to do is keep track of what other players have. You keep track of this by making suggestions and by other players showing you their cards. If Player B has Colonel Mustard, for example, you know that Col Mustard isn't the murderer.

Suppose Player A makes the suggestion that Mustard did it in the Dining Room with the Candlestick. Player B cannot disprove it. Player C, can however, and shows Player A his card. For the sake of this example, let's say Player C has the Dining Room card. A standard player would mark that the Dining Room isn't part of the solution. A better player would mark that Player C has the Dining Room card. The Clue AI, however, would note that Player B doesn't have the Colonel Mustard, Dining Room or Candlestick cards. How does this help?

Let's say Player C next makes the suggestion of Mustard in the Kitchen with the Candlestick. Player A cannot disprove it. Play passes onto Player B and he shows Player C a card. Player A--the Clue AI--knows which card Player B is showing Player C. How? It knows from a previous turn that Player B *doesn't* have the Mustard or Candlestick cards. So the card B is showing C *must* be the Kitchen: it's the only card B could possibly show C.

The AI guys made up special note pads on which they could keep track of which cards players *didn't* have. We would play the board game, most of us playing the standard way, the AI guys using their special note pads. The AI guys *always* won. This is the technique the Clue AI uses. It works so good that many customers accused the Clue AI of cheating. It didn't cheat, it just played **better** than the standard player.

The Clue Windows game has different difficulty settings for the AI. To make the AI easier to beat, it just "remembers" the "doesn't have" cards for fewer turns. For the hardest setting, it remembers all the turns. For the medium setting, it remembered fewer turns (like 2 to 3). For the easiest setting, it just remembered the "doesn't have" for one turn, just like a standard human player might.

I hope that gives you a good idea of how it worked.

Our Clue AI didn't keep a "score" like a standard game; either you won or you lost. That's it. If you're talking about something else, please elaborate.

As for deciding which rooms to go to, the AI just decided which rooms it needed to go to for suggestion answers. For example, if it needed to know if the Conservatory is in the solution, it would head there. If it needed more than one room, it would head to the nearest one. But like I explained above, it could deduce who had which cards often without making any suggestions at all. We also used A* for pathfinding, but I don't think that's what you were asking.

I apologize if I didn't answer your questions. Please ask again with more details if you need more info.

Take care!

-Chris

Hey Chris,

Thanks for getting back to me.

My algorithm also doesn't cheat, and remembers everything just like yours (I haven't implemented difficulty settings, and remember all of history, so I guess mine is "Hard" all the time).
The question of "score" is related to the problem of deciding which room to go to, which is really the question of which suggestion to make.

My algorithm calculates the uncertainty of the solution, then picks a room to go to based on the suggestion that reduces this uncertainty (entropy) the most.
The reduction in entropy is the suggestion's score, and the score of a room is the score of the best suggestion that can be made in that room.
Is this how your AI determined the "best" question to ask and therefore the "best" room to go to?

The second issue is, once you decide on a room, how do you get there? My algorithm propogates the scores of the rooms out to the standard spaces using an algorithm called "value iteration" with a discount factor.
This results in every space on the board having a score, and the AI only needs to move to the square with the highest score it can reach.

I see at least one other way to plan the route, which involves running BFS or DFS or some search algorithm to find the path.
Do you know how your algorithm did path finding?

Thanks
Riley

As I said before, we used the A* (A star) search algorithm for pathfinding, the algorithm that is used in most modern games. There's a good article on it on Wikipedia.

Getting back to your first question, however, I don't think the AI was that complicated. It just figured out which rooms it didn't know about and just picked the closest one. The "closest" one is easily determined with the A* algorithm. I don't know if this sounds simplistic, but it was good enough to beat almost any human player every time.

Good luck!

-Chris