Danmarks Tekniske Universitet

Making self-playable agents for the Hanabi card game using Dynamic Epistemic Logic, compact data representation and a rule-based strategy

Bachelor report

By: Christoffer Danborg Irvall (s174256)

Advisor: Nina Gierasimczuk

Contents

1 Introduction

Hanabi (meaning *fireworks* in Japanese) is a cooperative card-game designed by Antoine Bauza [2]. In the game each player has a set of cards in their hand and the players must cooperate in order to play the cards in a specific sequence to achieve the highest possible score of 25 points. The main obstacle for the players is that a player cannot see their own hand, but can see the other players' hands. Despite this limitation each player must play cards from their own hand and the only way in which they can know which card to play is based on specific hints and counting the cards. Since what a player can know for sure is quite restricted, each player, in practice, will have to guess the intention of the other player i.e. have a theory of mind. Since it is a game with imperfect knowledge, as well as good strategies have some theory-of-mind in place, it has sparked some notable interest in the AI research [1].

In this thesis I will focus on *self-play* i.e. only AI agents will have to cooperate in order to get as high a score as possible, as opposed to *cross-play* where some or most of the players are actually humans that try to cooperate with one or more agents.

Solutions to the problem includes hat-guessing strategies [3], Cox et al develop two strategies, one for recommending moves to other players, denoted recommendation strategy, and one for increasing other players' knowledge of what cards they currently hold, denoted information strategy. A hat-guessing strategy utilizes the fact that any legal move in the game can be interpreted as an encoding about some information or recommendation. This encoding can then be decoded by each individual, giving rise to some information or recommended action. Such strategies have proven effective for Hanabi, with [3] getting an average of 23.00 points for their recommendation strategy and 24.68 points for their information strategy. Looking into these strategies in detail, I see that these strategies are not only effective, but also efficient, because it is a very small amount of data each agent has to keep track of, and deducing the encoding and updating auxiliary data is quite trivial for a computer. Other solutions utilize Dynamic Epistemic Logic [4], which can very declaratively specify goals and try to find shortest paths in order to satisfy these goals. There are also machine-learning oriented solutions to this problem [8], that nowadays seem to be on-par with the rule-based ones achieving average score of 24.09.

In this thesis I will describe my solution based on principles of Dynamic Epistemic Logic, with emphasis on knowledge about ones hand, in order to make a group of agents work together and attempt to score as many points as possible in Hanabi.

The choice of implementation language is Zig [12], a new C-like language, which offers a lot of low-level control to the programmer, making it possible to create some highly-optimal data-structures as well as well as making some types of optimizations possible that would be hard to achieve with languages working on a higher level of abstraction.

1.1 Instructions on how to run the code

You will need the Zig compiler in order to run the code. Navigate to this page `https://ziglang.org/download/` and download a version 0.10.1, matching your computer's architecture.

Once you have the compiler, you can navigate to the folder `AgentsAndGameSimulator` and run the command: `$ zig build run-agents -Drelease-fast=true`

The `Drelease-fast=true` is optional, but makes sure it is compiled with optimizations.

1.2 Hanabi setup and rules

For completion I will give a short summary of the rules in Hanabi, which will also serve as terminology later on in the report.

Hanabi consists of a *Deck* of 50 cards, 4 *black tokens*, 8 *blue tokens*. Of the cards there are 5 *suits* and 5 *values*. The suits are red, green, blue, white, yellow. The values are 1 through

5. For each suit there is three cards of value 1, two cards of value 2, two cards of value 3, two cards of value 4 and one card of value 5. The number of cards in each player's hand depends on the number of players. If there are 3 players or less, they will have 5 cards each. Otherwise they will have 4 cards each.

The goal of the game is to play add as many cards as possible to the *colour piles*. There are 5 colour piles, one for each suit. You gain a point for each card in the colour piles, with a maximum of 25 points.

A player can spent her turn on one of the following actions

- Play a card
- Give a hint
- Discard a card

Playing When playing a card you attempt to add a card to one of the colour piles. You can add a card c to a pile of identical suit p if either,

- p is an empty colour pile and c is of value 1.
- p is non-empty and c's value is exactly one greater than the current card on top of the pile.

If a card does not add to a colour pile then it is put in the *discard pile* and then the game removes 1 black token. If there are no black tokens the game ends immediately.

Hinting If a player chooses to hint, he can hint any other player. Hints are restricted, so you can either hint about cards matching a suit or cards matching a value. Imagine a hand of following configuration: ((red,1),(blue,3),(red,2),(yellow,3)), each having respective indices 0 through 3. Then if you hint "cards of value 3", then you have to give the positions of both (blue,3) and (yellow,3) i.e. index 1 and 3. And if you give the hint "cards of red suit" then you have to give the position of both (red,1) and (red,2) i.e. index 0 and 2. You cannot give an "empty hint" like "there are no green cards".

When giving a hint it removes 1 blue token, if you cannot do that, you cannot give the hint.

Discarding A player can discard any card on their hand, and it will result in the card going in the discard pile and it adds 1 blue token (and can only be done if there are less than 8 blue tokens left).

End of the game The game ends immediately if there are no black tokens left. The game also ends if the last card from the deck has been drawn, in which case every player can play an additional round (including the player who drew the last card) and then the game ends.

After the game ends the score is decided by counting how many cards there is in the colour piles.

Definition 5.24 A model $\mathcal{M} = (W, (R_i)_{i \in \mathcal{A}}, L)$ of the multi-modal logic KT45^n with the set $\mathcal{A}$ of n agents is specified by three things:

1. a set W of possible worlds;
2. for each $i \in \mathcal{A}$, an equivalence relation R_i on W ($R_i \subseteq W \times W$), called the accessibility relations; and
3. a labelling function $L : W \to \mathcal{P}(\texttt{Atoms})$.

Figure 1: Definition of KT45^n (which is the same as S5^n) from [9]

2 Problem analysis

In this chapter I will go through the main difficulties in coming up with a good solution for Hanabi. Firstly, I will go through hardware constraints. Secondly, I will motivate my overall solution using Dynamic Epistemic Logic (DEL), which will be expanded upon in the Design section, and thirdly, I will analyse some lower bounds on time and space this solution would lead to.

2.1 Hardware and time constraints

In order to run the self-play of the agents I will have to pick a platform to work on. The setup I am going to run the self-play of the agents on utilizes an Intel® Core™ i7-10875H Processor, with 8 cores and 16 GB of ram. There is also 34GB of swap memory. So there is about 50 GB of memory as an constraint. In regards to time constraint I want to make have that each agent does not spent much more time than a real-time player. A couple of minutes at most - but this goal is less important, but mostly there to secure that it is actually testable. The chosen setup and time and memory constraints is mainly due to that is the computer I have, and I know I can develop quickly in iterations over a setup I have full control over.

2.2 Multi-modal logic of S5^n

In this problem I will have to use modal logic in order to solve the game of Hanabi. Modal logic is a field with a lot of depth and even notions like 'epistemic actions' can become part of the model[1]. I will chiefly focus on knowledge (what can be known) and what can be considered possible given the information an agent has. An approach to this is using S5^n [9], which seem flexible enough to adapt to the game. For completion, I put forward their definition of S5^n in Figure 1.

To give a correspondence to the Hanabi game, the world $w \in W$ will correspond to some configuration of hands for all the players/agents, and the deck, discard pile and colour piles. Some equivalence relation $R_i(w_x, w_y)$ means that agent i thinks that both w_x and w_y are possible based on the given information. And using the labelling function $L(w_k)$ could contain atoms like, "the first card in agent p's hand is possible to be played" for world w_k, so it can be used to answer queries.

Of course finding a good encoding for the worlds, such that it is easy to answer queries, as well as traversing the model in order to answer epistemic queries is no trivial task, and one might use a different encoding for a world than one strictly consisting of atoms (or booleans).

[1] like [4] logical system called Ostari

Number of players	2	3	4	5
Min	65574	47666	7406	6676
Max	93667	83848	16316	12962

Table 1: The number of distinct combinations for an individual player's hand as a function of the number of players in the game (initial round). On each column, I have data for 30 random games for k players, in which I dealt the cards to all the players except for 1 player p, and calculated how many distinct combinations p could get (see how data is generated by Appendix A).

2.3 Problems with possible worlds and modal logic

The problem with the approach of modal logic and possible worlds, is that we easily get combinatorial explosion. For some estimates on the number of possible hands for a single agent see Table 1.

To give some lower bound on memory usage for the case of 2 players, Alice and Bob, we can take the minimum on the table 65574 and assume that we spent 1 bit on each possible generated world. This means that we would at least have 65574 from the Alice's point-of-view, and 65574^2 for Bob (A set of worlds for each of Alice's worlds). 65574^2 bits alone is about 0.53 GB which is not a lot, but this is also a very theoretic lower bound. We do the same for calculating lower memory bound for 5 players, given that we generate a world for each fixed scenario, then we have at least $6676^2 \cdot 5$ worlds, which corresponds to about 0.028GB, which is an order of magnitude lower than the case with 2 players. So the scenarios with 4 and 5 players seem to be the least difficult to generate models for, or at least the models seem to be significantly smaller. To optimize this approach, each agent could store the given hints and shown cards, and at some proper time – when a significant amount of hints and cards have been revealed – make the model of possible worlds. A different approach is to work with a smaller version of Hanabi, which is what [4] does, which has the added benefit ones ability to test the code in a fast manner, rather than waiting for lots of possible worlds to be generated and processed.

The main thing the program will spent its time on will be generating the possible worlds, and iterating through all possible worlds in order to answer some query. Just take the above analysis and replace "assume that we spent 1 bit on each possible generated world" with "assume that we spent 1 microsecond on each possible generated world", so I will have to find an efficient solution to generating the hands, as well as efficient solution for iterating through it.

2.4 Summary

The main problems with trying to solve Hanabi is to both time and space constraints, where DEL, when applied naively, need to represent lot of possible worlds. I will have to use space-efficient data-structures in order to store the many possible worlds as well as fast algorithms for generating said possible worlds. Furthermore I will probably only be able to tackle the problem for 4 or 5 players, because otherwise there is not revealed enough information in order to substantially reduce the number of possible worlds.

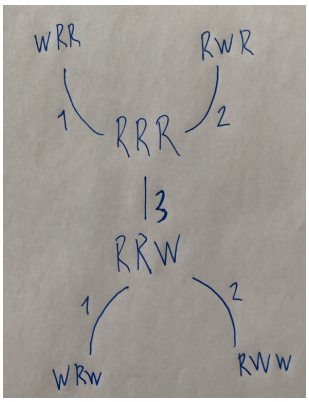

Figure 2: The initial model for the third wise man, with no other information than what he can see

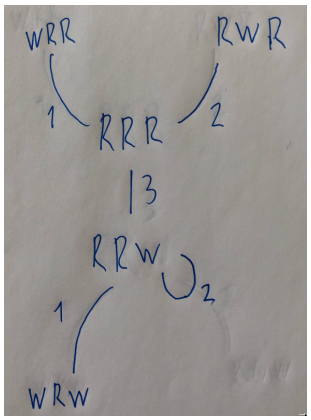

Figure 3: The model for the third wise man, after wise man 1 has said "I don't know"

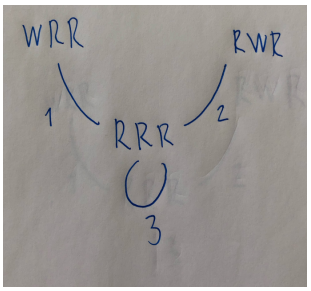

Figure 4: The model for the third wise man, after both wise man 1 and 2 have said "I don't know", in succession.

3 Design

3.1 Modifying S5^n in order to simplify the implementation

In order to simplify the implementation of S5^n, then instead of an agent p having the entire model $\mathcal{M}$ as described in section 2.2, it will have some subgraph of the model $\mathcal{M}$. This subgraph will be based on what is able to be seen by p. So given a model $\mathcal{M} = (W, (R_i)_{i \in \mathcal{A}}, L)$ from S5^n, we are only interested in the worlds $W_p \subseteq W$ which are non-contradictory with the current state of the game from p's point-of-view (shortened POV). For each $w_p \in W_p$, we also keep the worlds which has an equivalence relation to any w_p. For an agent p, that has such a model, we denote it $\mathcal{M}_p$ of sub-S5^n.

3.1.1 Example of this modification: Three wise men puzzle

In order to motivate this modification, I will take the example of the three wise men problem. There are three wise men, each standing in a row, enumerated 1 through 3. Then a king, who has 3 red hats and 2 white hats puts a hat on each of the three wise men in such a way that each person does not know the colour of the hat on their head. Then the king asks the first wise man whether he knows the colour of their own hat, where the answer could either be "I don't know" or "I know". He then asks the same of the second, then the third.

Let us say that the actual case is RRR i.e. wise man 1, 2 and 3 has a red hat on their head. In that case wise man 1 will answer: I don't know, the second wise man will answer: I don't know, but the third wise man will answer: I know. How so?

Figure 2 is the $\mathcal{M}_3$ model for the initial configuration. This means that wise man 3 does not know whether he has a red hat or a white hat. And for each of these scenarios he can imagine that 2 cannot know either, and neither can 1. I will refer to scenarios that are fixed from a particular agent's POV for *fixed scenarios*. So here the fixed scenarios are RRR and RRW with respect to wise man 3's model. All fixed scenarios implicitly has the equivalence relation R_3 in this case, because they are both possible given the information wise man 3 has.

When 1 states "he does not know" then that has to be equivalent to "not RWW is the case". This means that three can safely eliminate this state, as something 2 would imagine, given that this was a public announcement. This results in the updated model in Figure 3.

After 2 answers "I don't know", wise man 3 can deduce that the case was not RRW, since if it was the case, then 2 would have seen 1:R and 3:W and know that RRW was the only scenario in which this was possible, and hence would answer "I know". Which means that the final model that wise man three has is the one on Figure 4, for which the wise man 3 will know with certainty that the only scenario which is possible is RRR.

3.2 How the subgraph $\mathcal{M}_p$ is generated, given a game-state

Without worrying too much about the combinatorial explosion (more on that later), we can apply the principles described in chapter 3.1.1 to Hanabi.

So analogously instead of looking at a world as an assignment of hats, we look at a world as an assignment of hands and contents of the deck, discard pile and colour piles. The procedure of generating the model $\mathcal{M}_p$ for an agent p from a set of n agents $\mathcal{A}$ would go like this:

1. Generate all fixed scenarios based on the information accessible to p. Let's call this set of worlds for W_p.
2. And make elements in W_p have the relation R_p among each other.
3. For each $w_p \in W_p$ do
   (a) For each agent $a \in \mathcal{A}$ do
      i. Generate all worlds that a finds possible, given that w_p is the case
      ii. Make all elements generated this way have the relation R_a among each other.
      iii. Make all elements generated this way, w_a, have the relation $R_a(w_a, w_p)$

What each agent does not know, is namely their own hand and the contents of the deck. If the multiset of cards in the deck was known, then the multiset of cards on their hand would be known and vice versa. So we can restrict the problem of generating a world to either have to generate the possible decks, or the their own possible hand. I choose to generate their own hand, since it (for the most of the game) is much smaller. How such a possible hand for some agent p would be generated is to take the multisets of cards [2] : The initial full deck S_{deck}, the current discard pile $S_{discard}$, the current colour piles S_{color}, the other agents' hands S_{op}, and find what is left to take from. Then we can see that the set of possible hands p can generate must be taken from the multiset $S_p = S_{deck} \setminus S_{discard} \setminus S_{color} \setminus S_{op}$. The set of hands then is the distinct combinations of size k that we take from the multiset S_p. Here k is 4 or 5 depending on the size of the hand. This method does not take into account the hands which might conflict with a given set of hints. More on this in section 3.7.

Given the constraints on time and space, it would be nice to have a quite compact representation of each world, furthermore, given that query algorithms will go through most (if not all) of the possible worlds, iterating though the possible worlds should be fast. A fast way to iterate through elements is to make sure that each element is small as well as contiguous in order to make use of the CPUs caching system the most, i.e. small elements in a contiguous data structure which guarantees good spatial locality.

3.3 Formalizing accessing in a model and turning it into an array

In order to get good spatial locality I will explain how to take the model of sub-S5^n and turn it into a multi-dimensional array.

In the section 3.2 I went through the overall approach of generating the worlds from a specific agent's POV. In order to specify how such worlds might be accessed, I extend the solution with

[2] because there might be duplicates of the same card

some notation. Let $\mathcal{M}_p : W \to P \to \mathcal{P}(W)$ be the model for the agent p's POV. Here the interpretation is that the model takes some fixed scenario w_p of type W, and some agent b of type P, then we have all the worlds that p cannot rule out that b finds possible, given that w_p is the case. The fixed scenarios are strictly from the POV of p, which means that all fixed scenarios are all the worlds that p finds possible from the information she has got. So all fixed scenarios has an equivalence relation R_p. The set given by $\mathcal{M}_p(w_p)(b)$ has the equivalence relation R_b among each other, as well as for any $w_b \in \mathcal{M}_p(w_p)(b)$ has the relation $R_b(w_b, w_p)$. An interesting case is $b = p$ then we have the unit set, which is simply $\mathcal{M}_p(w_p)(p) = \{w_p\}$, which makes sense in the interpretation "given that we fix the world to w_p, then agent p can only imagine that p's only possible world is w_p".

It is clear from the above mapping that this could be encoded using a 3-dimensional array. Where W (the first argument) could be interpreted as an enumeration of the fixed scenarios for p, and P is an enumeration of the agents, and $\mathcal{P}(W)$ is an array of the possible worlds.

3.4 Choosing a compact encoding for a world

A world is simply a possible state of the entire game, i.e. everything is specified, including the hand you can't see. A naive approach to representing a world is to take the deck, the discard pile, all of the hands etc. into account. But this approach is definitely more than necessary, since what is in the discard pile and in the colour pile, as well as most hands, are information easily accessed by looking at the current state of the game. Furthermore I choose not to take order into account in any of the hands or piles, because it would lead to too many combinations (a hand of 4 cards has 25 permutations, assuming that all are unique). What is most interesting to represent is simply then what is uncertain. So in this case it is a agent's own hand and the current contents of the deck. Since an agent does not know their own hand, then assuming a hand will give the contents of the deck unambiguously, and vice versa, so it is sufficient to just represent the unknown hand. So a query $\mathcal{M}_p(w)(b)$, will return the set of hands that p cannot rule out that b finds possible for himself, given that the hand w is the case for p.

There are various approaching to representing a hand, I will go through the ones I have devised.

1. Represent each card in a sorted order. Called sorted-order representation
2. Enumerate each possible hand. Called Enumeration representation
3. Contingency table inspired multiset. Called table representation.

In each representation I will go through the number of theoretical bits needed per world as well as what the actual number of bits are if we have to make it directly byte-addressable. The reason I write "theoretical" is due to that most CPU architectures (as well as how Zig compiles code [10]) are byte-addressable. This means that the smallest addressable space is 1 byte on most modern architectures, even if you want to represent something that uses fewer bits. The theoretical number of bits can be reached, but it will require some bit manipulation in order to achieve, and comes with the trade-off of more CPU cycles spent extracting the information from a compacted model, rather than accessing the value directly.

Represent each card in a sorted order So firstly we want to look at how small can we get away with representing a card. A card can be one of 25 different instances (5 suits times 5 values), so a minimal representation could enumerate each card and a world be represented by 5 bits (which can represent from 32 different values). Then if we have a hand of 4 cards, that means we spent $4\frac{\text{cards}}{\text{hand}} \cdot 5\frac{\text{bits}}{\text{cards}} = 20\frac{\text{bits}}{\text{hand}}$. And using the same reasoning we use 25 bits for 5 cards. Then we can predefine some sorted order of the 25 different instances, and just sort a hand in order to get its multiset representation.

If each card has to be byte-addressable then it would have to use 1 byte (8 bits), which would then mean that a hand of 4 cards is 24 bits, and a hand of 5 cards is 40 bits.

Pros: Very simple and compact representation. No obvious cons.

Enumerate each possible hand If you draw 4 cards blindly from an initial deck of 50 cards, there are 18480 distinct combinations (data from Appendix A). If it 5 cards drawn it is 99455. Depending on the situation, you could enumerate each possible hand. Then a world need only an integer size large enough to store 18480 or 99455 depending on the case. This is respectively 15 bits and 17 bits. So this representation is definitely compact. Some edge-cases exists, for instance in a 5-player game, when the last card has been drawn, then there can exist both 4 and 5 cards representations at the same time, but that would require bits enough to represent $18480 + 99455 = 117935 < 2^{17}$ so 17 bits should still be sufficient.

If the representation would have to be byte-addressable in an array, then it would have to spent a whole number of bytes, which is respectively 2 bytes (16 bits) or 3 bytes (24 bits).

This enumeration would only create a mapping from an integer to an actual representation, so we would still have to choose a representation that this integer maps to. Which can be either sorted-order or table representation

Pros: Very compact representation, the enumeration map should only take some constant amount of memory at all times. Even if we assume the worst case, which is using the table representation and enumerating

Cons: Non-trivial to implement. A lot of conversion between values i.e. its hard to work on a world directly without looking it up in a map, so when iterating through a list of possible worlds I will have to look up what that world actually is, this I imagine will substantially slow down the algorithm, although I have not tested it.

Contingency table inspired multiset I stumpled upon this representation when I was trying to figure out how to generate the possible hands in a fast manner (more on the generation in section 3.5). Given that instance of a card can at most occur 3 times in a hand (or deck for that matter). Then an array of 25 elements, each element being able to store 0 through 3, will be a sufficient representation. This means that each element in the array need only be 2 bits. So a theoretic use of space is $2\frac{\text{bits}}{\text{elements}} \cdot 25\text{elements} = 50$ bits.

If each element has to be byte-addressable then it will have to spent 1 byte per element, in which case $8\frac{\text{bits}}{\text{elements}} \cdot 25\text{elements} = 200$ bits. Which is a pretty big increase from the theoretical 50 bits.

Pros: Very direct multiset representation of the cards, can even be used for the piles and deck.

Cons: Spends a lot more bits than strictly necessary to represent a world.

3.4.1 Choosing a representation

All representations have a convincing set of of trade-offs. I decided to go with the table representation, since it is very generalizable for other aspects of the game (i.e. can be used to represent decks, piles etc.), as well as easy to work on directly. Furthermore it is easy generate, see section 3.5. The other representations should definitely be kept in mind if memory proves to be a problem in practice.

3.5 Generating hands in an efficient way

Using the methods described in `math.stackexchange` posts "Efficient algorithm to find all unique combinations of set given duplicates" [5] and [7], I was able to design a good generation method for the hands. The main idea is that we represent the hands we want to generate in a contingency table, where we have some pool to take from. So a case with hand size of 0 and

																											Total
card pool	3	2	2	2	1	3	2	2	2	1	3	2	2	2	1	3	2	2	2	1	3	2	2	2	1		50
hand	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0		0

Figure 5: Contingency table with hand size of 0

																											Total
card pool	2	2	2	2	1	2	2	2	2	0	3	2	2	2	1	3	2	2	2	1	3	2	2	2	0		46
hand	1	0	0	0	0	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1		4

Figure 6: Contingency table with hand size of 4

the pool being the initial deck will be Figure 5 (although, in a practical game, you would know something about the other hands and the discard pile etc. so the pool would be less than the initial deck). Then the problem is to generate all such contingency tables, such that the sum of the hand is k (usually 4 or 5), and the column sums stay the same, for example see Figure 6.

Both the card pool and the hand can be represented by using the table representation described in section 3.4.

The simplest way to describe how the hand is generated, is to view the hand table entries as digits in an integer (or more mathematically specific, a multi-radix integer), with the leftmost digit being the most significant, and then you create the biggest integer with cross-sum k, and subsequently construct the number just below that, still with constraint that the cross-sum must be k. So continuing the example with Figure 5, we can create the "biggest integer" in this way Figure 7. Then the integer just below that is Figure 8. And continuing with 1s until we get Figure 9. Then we decrement the most significant digit and get Figure 10.

A more specific algorithmic description is given in section 4.2

3.6 How should an agent play?

How an agent should play based on its knowledge in sub-S5^n model is heavily inspired by [3], specifically the information strategy. In it they distinguish between *public and private information*. Where public information is what everybody knows, and everybody knows that everybody knows that etc. And private information, is what an agent individually knows based on the other agents' hands.

This can be compared to DEL where public information is what the model guarantees is public knowledge, and private information is simply queries that holds true for all possible worlds from that agents POV.

In my solution I will only be querying on the private information, but public information is interesting and is commented on in the Conclusion section.

Given a hand, which is a multiset of known or unknown cards, how does an agent know what card to play based on the private information? Firstly I will have to decide *when* a card should be played, discarded etc. Here I use the action algorithm from [3], with minor modifications:

1. Play the playable card with lowest index.

2. If there are less than 5 cards in the discard pile, discard the dead card with lowest index.

																											Total
card pool	0	1	2	2	1	3	2	2	2	1	3	2	2	2	1	3	2	2	2	1	3	2	2	2	1		46
hand	3	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0		4

Figure 7: First generated hand

																										Total
card pool	0	2	1	2	1	3	2	2	2	1	3	2	2	2	1	3	2	2	2	1	3	2	2	2	1	46
hand	3	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	4

Figure 8: Second generated hand

																										Total
card pool	0	2	2	2	1	3	2	2	2	1	3	2	2	2	1	3	2	2	2	1	3	2	2	2	0	46
hand	3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	4

Figure 9: The smallest integer containing 3 in the first position

3. If there are hint tokens available, give a random hint to a random agent.
4. Discard the dead card with lowest index.
5. If a card in the agent's hand is the same as another card in any agent's hand, i.e.,it is a duplicate, discard that card.
6. Discard the dispensable card with lowest index.
7. Discard index 0.

Here implicitly "playable card", "dispensable card", "dead card", "it is a duplicate", are epistemic queries on the $\mathcal{M}_p$ for the current playing agent p's hand. I denote these types of states a card can be in as *actionable states* for that card.

I define the actionable states for a card as follows (also based on [3]):

- Playable: If it is able to be played in the current game state.
- Dead: If the card has already been successfully played and is not needed any more.
- Dispensable: If the card could be played at some point (now or in the future), but there is at least one more of the same card left.
- Duplicate: There exists a duplicate in another agent's hand.

An agents would eventually have to interpret their hand as a list of cards (where order matters), due to the fact that they have to play some index from their hand (which they might have full, partial, or no knowledge about). It is for this list of cards that the agent has to derive the actionable states for each card, in order to come up with a move. Although, this interpretation of a hand as a list of cards is in contrast to how a world is currently defined: A multiset of cards, where order does not matter. So we have to define some mapping between an multiset of cards (a world) and a list of cards (the cards on an agent's hand).

To specify how an agent will view it's own hand as a list, we let what is known about a card be denoted by the pair: its suit, and its value. Where it is valid that a suit or value can be "unknown".

As an example: If agent p has the hand ((red,1),(blue,3),(red,2),(yellow,3)), and the current state of the game p only has her red cards hinted, then the list hand o_{hints} would be

																										Total
card pool	1	0	2	2	1	3	2	2	2	1	3	2	2	2	1	3	2	2	2	1	3	2	2	2	1	46
hand	2	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	4

Figure 10: The biggest integer containing 2 in the first position

$$o_{hints} = ((red, unknown), (unknown, unknown), (red, unknown), (unknown, unknown))$$

In order to create the mapping from a multiset in a world to an list that matches the hand indices, we have to take the hints given to agent p into account.

So given an unknown hand for agent p specified with the list of hints o_{hints}, and the model $\mathcal{M}_p$ we have to decide what possible actionable states is in each card.

For each fixed scenario for p, there is a specific hand for p, denoted $h_{specific}$. Since each hand is stored as a multiset, we have to use the hints o_{hints} to figure out how the set should be mapped to a list. How this can be done is to make $h_{specific}$ into any list $o_{specific}$ and subsequently take all permutations of $o_{specific}$ and see which ones matches the h_{hints}. If it is possible to find any permutation that is non-contradictory with o_{hints} then we can decide the actionable states for each card given by $h_{specific}$.

Continuing the previous example. We might have the multiset

$$h_{specific} = \{(green, 1), (red, 5), (green, 2), (red, 1)\}$$

Turning this into a list we have

$$o_{specific} = ((green, 1), (red, 5), (green, 2), (red, 1))$$

we see that there are 4 permutation that matches o_{hints}:

$$o_1 = ((red, 5), (green, 1), (red, 1), (green, 2))$$
$$o_2 = ((red, 1), (green, 1), (red, 5), (green, 2))$$
$$o_3 = ((red, 5), (green, 2), (red, 1), (green, 1))$$
$$o_4 = ((red, 1), (green, 2), (red, 5), (green, 1))$$

Then for each permutation we can decide the actionable states of each card, based on the permutation and the current state of the game. For instance, let us assume that the red colour pile in the current state of the game has a 4 on top. This means that the card in position 0 can both be playable (if it is a (red,5)), but also be unplayable (i.e. if it is a (red,1)). Since it is not certain that the card 0 is playable, we cannot perform step 1 in the action algorithm.

3.7 Removing other agents' possible worlds based on their hints

In section 3.2 I described how one might generate the possible worlds based on the revealed cards. Let's say we have two agents p and b, and we generate the model $\mathcal{M}_p$. But there is something important to take into account: if this generation is only based on the cards that are revealed then there will be some possible worlds which conflict with the hints given to b, therefore we can also go through all possible worlds for b and remove any that are in conflict with their hints. This can be done in much the same way as described in section 3.6, where we use permutations of a possible world to see if any permutation matches the hints of b's hand.

4 Implementation

4.1 Going through all permutations

In sections 3.6 and 3.7, I described how an agent might make a move and how to remove worlds based on hints. A crucial aspect of both of these procedures is that we need a permutation generation method. Generating permutations can be done using Heap's algorithm [6], which seem to be fast and not need that much data in order to compute the permutations. If there are k objects that you want to find the permutations of, you only need to maintain some auxiliary data of size $O(k)$. My implementation is in "`src/multi_agent_solvers/PermutationIterator.zig`".

4.2 Generating distinct combinations of size k

Continuing the section 3.5, I wanted to generate all distinct combinations from a multiset of elements. The approach I found is best described with the pseudo-code in Code listing 1.

It takes an array `taken_into_account` in which initially all elements are 0, which, when filled with `cross_sum` number of elements will represent a hand. `distinct_pool` which represents the multiset from which the k distinct combinations will be generated from. `sum_array` is a suffix-sum array of the initial `distinct_pool`. `cross_sum` which is initially the same as k. `current_id` is which element is being considered to be added to the hand and finally `accumulator` stores all the distinct combinations.

As an example, if we wanted to generate all distinct combinations of $k = 4$ from the card-pool array in Figure 5. Then the `distinct_pool` array would be equivalent to the "card pool" row (except for the "Total" column). And the suffix-sum array `sum_array` would be

$$[50, 47, 45, 43, 41, 40, 37, 35, 33, 31, 30, 27, 25, 23, 21, 20, 17, 15, 13, 11, 10, 7, 5, 3, 1, 0]$$

The way the algorithm runs is that it takes as much as it can, naively from the left, from the `distinct_pool` until the cross sum of `taken_into_account` is equal to `cross_sum`. After it has done this it will continually lower the rightmost non-zero entry until it cannot do so anymore, after which it knows that it has to lower the second rightmost non-zero entry etc. using the callstack. This way we trivially know that we go through all numbers of cross-sum k in the multi-radix interpretation of the problem, from biggest to lowest, and hence we get the distinct combinations of size k.

Let us denote the number of distinct elements to be n (i.e. the length of `distinct_pool` and let the number of distinct combinations of size k be equal to N_k, then I will argue that the runtime complexity is $O(N_k \cdot n)$ to generate all distinct combinations. This is due to the fact that the worst case for a single distinct combination, is that we have a call stack of size n, for which we have to pop $n-1$, to decrement some largest number and then naively go through the array in order to fill `taken_into_account`. This means that we spent $O(n)$ steps to generate a single combination. We do this exactly $O(N_k)$ times, so the desired complexity has been shown ■

The benefits of this is that n in the case of Hanabi is quite small, with only 25 different types of cards.

```
function distinct_combinations(
 taken_into_account: integer array consisting of 25 elements,
 distinct_pool: integer array consisting of 25 elements,
 sum_array: integer array consisting of 25+1 elements,
 cross_sum: integer,
 current_id: index,
 accumulator: growable array) {
    if (cross_sum == 0) {
        acc.append(taken_into_account)
        taken_into_account[current_id] = 0;
        return;
    }
    take = math.min(distinct_pool[current_id], k);
    for(int i = 0; i < take + 1; i = i + 1){
            taken_into_account[current_id] = take - i;
            if (sum_array[current_id + 1] < k - (take - i)) {
                taken_into_account[current_id] = 0;
                return;
            }
            distinct_combinations(taken_into_account,
     distinct_pool,
     sum_array,
     cross_sum - (take - i),
     current_id + 1, acc);
     }
     taken_into_account[current_id] = 0;
}
```

Code 1: Pseudo code for the hand-generation method. It is assumed that arrays are taken by reference and integer/indices are called via copy

4.3 Representing the world

I mentioned that I picked the table representation from the different choices I had (see section 3.4), but I was not aware, at the time, of how byte-addressability would affect the actual space, so I was surprised that the space was filled up so quickly. There are ways to mitigate these problems of space, and that is to make a packed array that uses bit manipulation in order to achieve this [3]. Due to time left to finish up this product I chose to have only 1 agent with 1 model at a time, so that the agents in total do not use too much memory.

4.4 Implementing the board game

I also implemented the board game, simply referred to as the Game struct (in file "`src/hanabi_board_game.zig`) in order for the agents to interact with the game. The board game keeps track on who is the current agent that must make a move, and has some simple methods that the agents can use when they want to make a move:

- `play(index)`
- `discard(index)`
- `hint_color(color, index_of_player)`

[3] Zig has this in its library called packed int array [11]

- `hint_value(value, index_of_player)`

In order to simplify the implementation of the board game, I simply let it crash if a agent does something that is illegal. (Like giving an index that is out of bounds, or hinting a colour not on the hinted agent's hand).

4.5 Agents

The logical agents in the game are implemented as an Agent struct. See a class diagram on Figure 11. An Agent has a hand that is represented with `CardWithStates`, here the states refer to each card's actionable states (related to section 3.6), but also its hints. It does not see the actual cards. It has a `KripkeStructure` which is sub-S5^n model. And lastly it has a `CurrentPlayerView` which is the current game that the agent is able to see, i.e. it cannot see its own hand, or the contents of the deck. An `Agent` has a method called `init(...)`, which, based on the view and which player index it is, is able to generate the entire model, as well as decide what states the `CardWithStates` should be in. After an Agent has been `init`ed it can make a move with the method `make_move(game)`, which modifies the game based on its `hand`, as well as its view.

4.6 Game simulation runner

The glue-code between the `Agent`s and the `Game` classes, is a class called `SimulationRunner` (defined in "`src/ai_simulation_runner.zig`"), which takes a fully initialized Game and on-demand generates the agents and their epistemic models, and simulates a course of a game. See Figure 12 for the components making up the game simulator. The Game class has an element of randomness (in order to simulate random draws), so a seed can be provided when initializing the game in order to facilitate reproducibility.

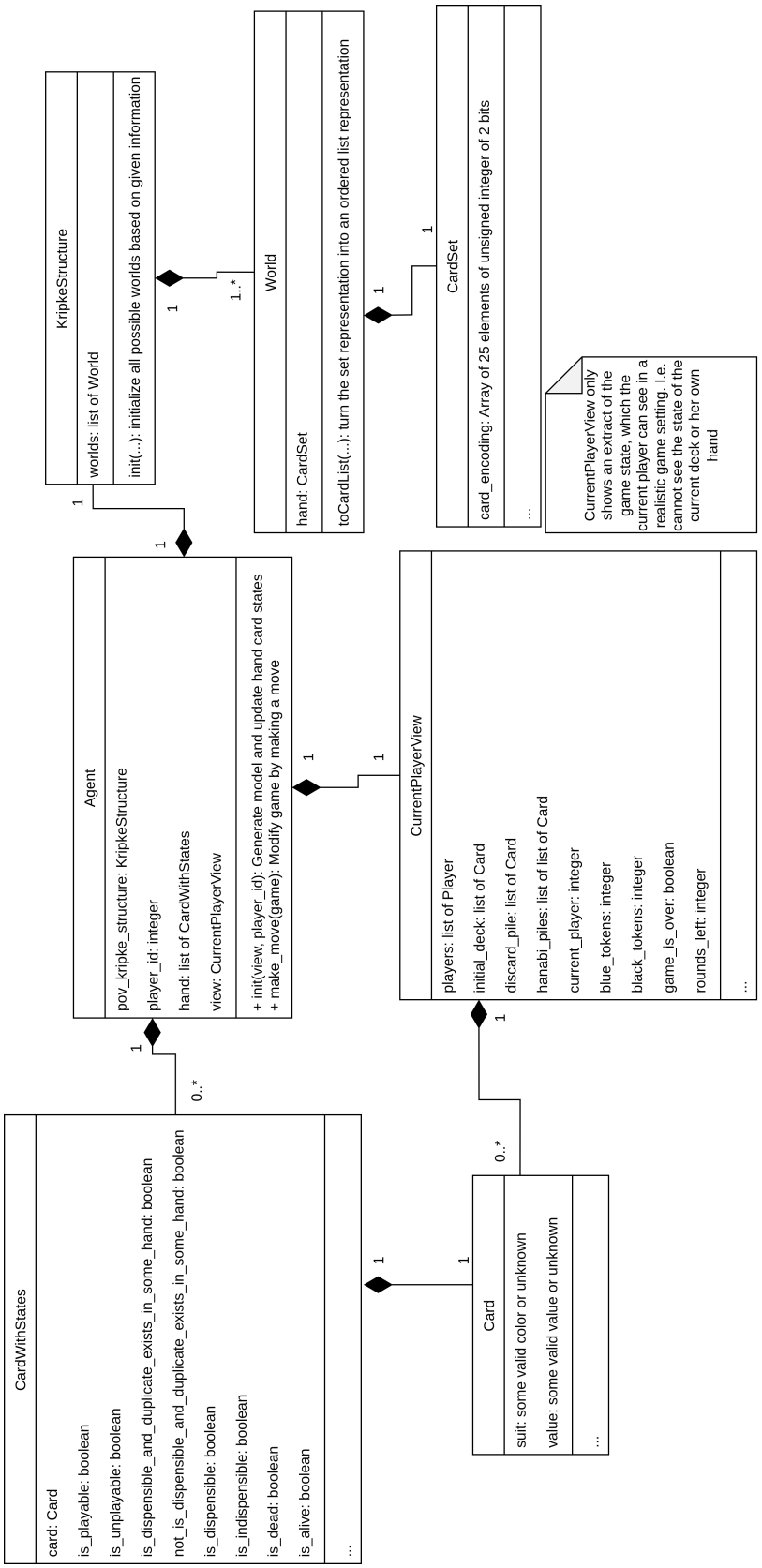

Figure 11: Low-fidelity class diagram for the things making up the Agent class

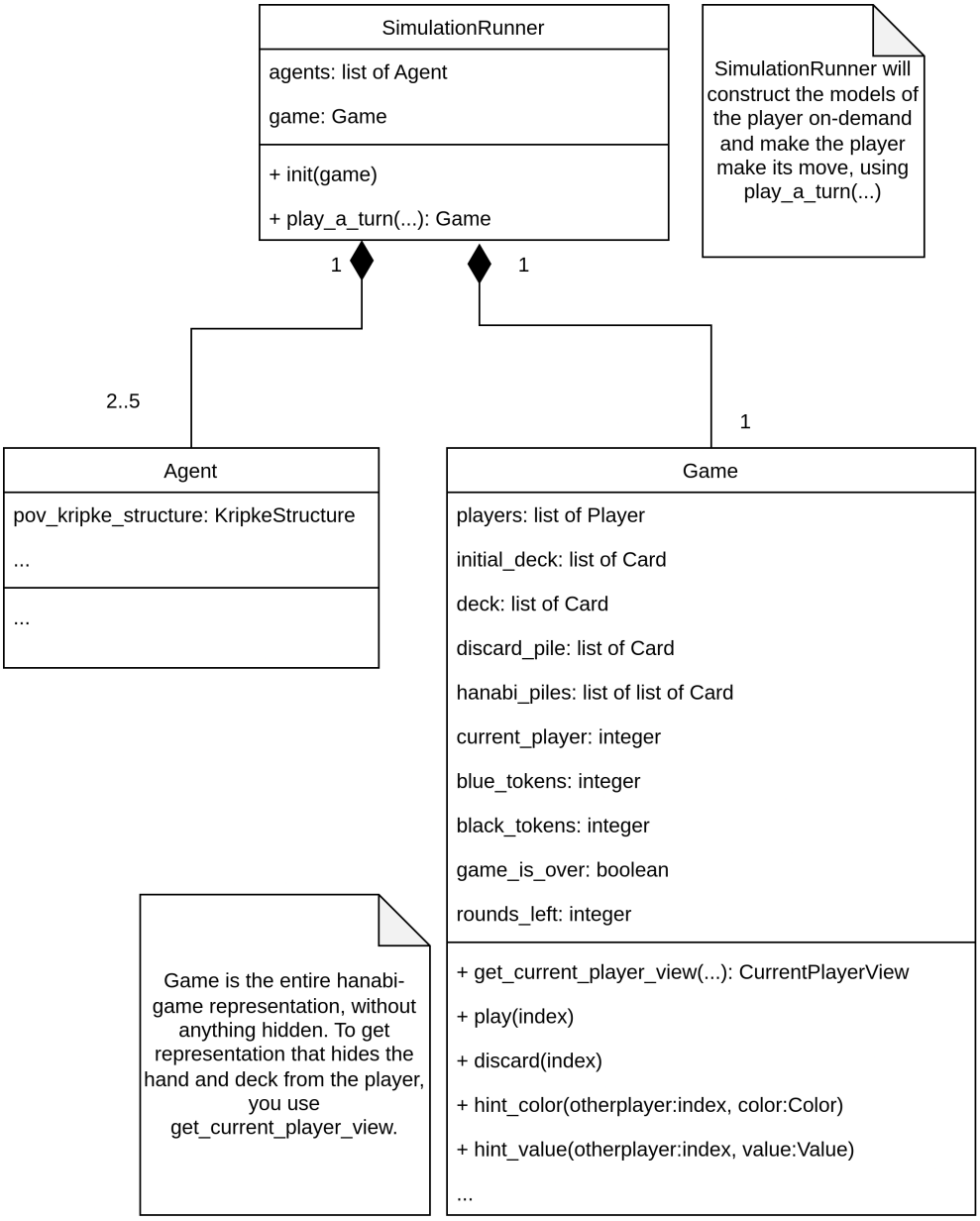

Figure 12: Low-fidelity class diagram for the things making up the SimulationRunner class

5 Testing

5.1 Unit testing and integration testing

I did some unit-testing in the project, which is directly integrated with the Zig language, in the form that you simply write the unit tests in the source files themselves, under a "test" environment. Things with a few dependencies are easy to verify and unit-tests give confidence to that these basic methods/classes can be relied on. These simple components are the `CardSet` class, the permutation and distinct combination methods, or printing methods.

Unit-testing becomes becomes infeasible when you have to combine and inspect more advanced properties. For instance, when testing whether the generated worlds are correct, I would have to compare it to something much simpler. In this case I tried to adapt my methods in order to make it match the three wise men problem, and then I would visually inspect it myself to see whether it was correct, this is done in the file `src/multi_agent_solvers/agent.zig` in the test called "Three wise men simulation". So simple tests like that functioned as an integration test.

Unit-testing is pretty crucial when working with manual memory management languages like Zig, since the risk of orphaned memory is much more severe, and Zig facilitates prevention of memory leaks by detecting this in its unit tests.

5.2 Early feasibility test for the world generation method

After implementing the table encoding for a world, I decided to make a feasibility test, whether it was realistic generating the possible worlds for Hanabi. The test is in `src/bigtest1.zig`. I generate 20 random initial rounds to see how much space and time each one uses. The most amount of space used by a model for a round was 11.27 GB, and the least space was 3.82 GB. The most amount of time spent on generating a model was 14.3 seconds, and the least time was 4.5 seconds. These numbers are well within the constraints given by Problem Analysis chapter and therefore I decided that it would be feasible to continue with the current solution. In this test I did not take into account how much time might actually be spent on querying and removing worlds, for more details on that see section 5.6.

5.3 Testing my game implementation by making an interactive interface

I wanted to test the `Game` class, to see if anything unexpected happened if some specific sequence of action occurs. In order to do this, instead of manually making some unit tests, I implemented a TUI interface of the game and tried various things in order to "break" the `Game` implementation. The TUI interface is in `src/cli_simulation_runner.zig`.

5.4 Ad hoc white-box testing system

Some things I wanted to continually inspect throughout the project, and some of these things were related to the internal behaviour of the methods. This is a type of white-box testing, where I inspect the internal structure of the methods. In order to do this, without sacrificing performance from the methods themselves, I utilized the fact that Zig can optimize code-blocks away, that it knows at compile-time will not be executed.

This is done using a file that contain some global-accessible flags: `src/multi_agent_solvers/globals_test.zig`.

So what I mean by "it optimizes code-blocks away" I will illustrate with the pseudo-code at 2. So we have some complicated function which does a lot of work on the `kripkestructure`, and I want to iterate through the elements, to verify that it does it correctly. The iteration

happens in the if-statement block. This block will be removed at compile-time if it is known that `globals.check_kripkestructure=false`, in this way I can enable it when I have a need for it.

```
function some_complicated_function(game, kripkestructure):
    // modifies kripkestructure in some way
    if(globals.check_kripkestructure){
        for each element in kripkestructure{
            print(element)
        }
    }
    // function continues to do other work
}
```

Code 2: An example of how globals are used for testing

5.5 Average score achieved by my product

The average score achieved through simulating 20 random games is 14.2, with lowest score of 12, and highest score of 17, which means that the program is far from being as good as many existing solutions.

5.6 Testing the time spent by `KripkeStructure` class

The bulk of the work done by the program is `KripkeStruture.init(...)` and `Agent.updateCardStates(...)`. I will here mainly focus on the `Kripkestructure` generation, and also test the time of `KripkeStructure.deinit()`, because I think that there might be a lot of time spent deallocating the structure as well.

The `KripkeStructure.init(...)` works in three steps

1. Generate all possible worlds based on the visible cards (hints are not used)
2. Eliminate all possible worlds in which they are in contradiction with the hints.

The `KripkeStructure.deinit(...)` method simply iterates through the structure and calls deinit on all substructures.

The reason that I examined these things was to 1) judge the efficiency of the distinct combination generator, 2) see if there was something to be gained from using a different type of allocator, because I know that I allocate and deallocate a lot of elements, 3) and if the method of removing worlds is actually efficient.

In 5 random games, it took 57.3 minutes to play all the games, for which it took 56.9 minutes for all the `KripkeStructure.init(...)`, `KripkeStructure.deinit()` and `KripkeStructure.remove_worlds_based_on_hints(...)` calls. Of the 56.9 minutes, each call took the following percentage of time: 24.5% of the time is spent on generating the distinct combinations, 1.2% of the time is spent on deallocating space and 74.2% is spent on removing worlds based on hints.

I think the removal of worlds takes so much time, is that I have not optimized for identical permutations. So no matter how many hints there are on a hand, it will generate all $k!$ permutations.

6 Project management

My main heuristic going into this project was to look for the minimal viable product, that made use of the theory of modal logic, and which was actually able to play Hanabi. So I tried to avoid premature optimizations in most places, except for parts I knew beforehand would be critical (like generating lots of possible worlds, and a compact representation for the worlds).

I used git throughout the project as source-control, which also has the added benefit of keeping a log of the different commits, so that I could write notes there as I complete the product.

7 Conclusion

I have made a product based on a modified model of S5^n, denoted sub-S5^n, in which each agent can deduce a big set of possibilities for the other agents (and itself), and is able to look through each possible world to extract some property about each world. The average score is 14.2 which is pretty far from many existing solutions. The memory-layout of sub-S5^n in the case of Hanabi, is quite simple, given that it is implemented using a 3-dimensional array, and each world spends only 25 bytes of memory. This compactness and good spatial locality is great for query algorithms that look through each world sequentially. Further space was saved by making sure that each world stores a multiset of cards, instead of a list of cards, while still retaining the ability to answer queries about specific cards on a agent's hand.

7.1 Further work

Public knowledge is very useful for strategies like the information strategy in [3], so if I had more time I would look into how public knowledge among the agents could be deduced from the sub-S5^n model, and whether it was actually possible to do this. Furthermore, it would be interesting to see which strategies would work great in conjunction with the sub-S5^n model. For instance the two rule-based strategies described in [3], I would argue, work completely orthogonally to the sub-S5^n model solution, which means that both of these strategies could be combined with sub-S5^n model in order to make better decisions. In the case of the information strategy, the information gained by encoding the moves could further modify each agent's model and thereby gain more information by using such a strategy. Similarly, if the recommendation strategy says one of your cards is "playable" then you can eliminate all worlds for which that card is not playable.

Data representation and optimizing the Zig code would also be of priority if I had more time. An interesting option is to make a more in-depth comparison of the data-representations of the worlds described in section 3.4, and see how they would have affected performance in practice. Furthermore a lot of redundant permutations are generated whenever inspecting a possible world, which can be optimized away by only keep unique permutations.

Another interesting aspect of the Dynamic Epistemic Logic is to look into how a model could be viewed as a probabalistic model, rather than a certain knowledge model. For instance if a lot (but not all) of the possible worlds suggest that agent p's 0th card is able to be played, then it is *probably* able to be played.

7.2 What I have learned

In this project I learned a lot about modal logic, which strikes me as a good balance between using a powerful type of logic, as well as being pretty realistic to implement with reasonable performance, although, I have only scratched the surface of the subject, and there is a lot more to be explored in regards to it.

I also learned a lot about the programming language Zig, which has been a hobby-language of mine for quite a while, but I have not been able to test it on a project of this scale. The main difficulty of using Zig is the fact that you have to take responsibility for how memory is managed. This can easily lead to decision-fatigue, especially if you are not accustomed to this style of programming. The advantage of Zig is that the language gives you a lot of tools in order to optimize code and data structures to a degree which is hard to do in a lot of languages.

References

[1] Nolan Bard, Jakob N. Foerster, Sarath Chandar, Neil Burch, Marc Lanctot, H. Francis Song, Emilio Parisotto, Vincent Dumoulin, Subhodeep Moitra, Edward Hughes, Iain Dunning, Shibl Mourad, Hugo Larochelle, Marc G. Bellemare, and Michael Bowling. The hanabi challenge: A new frontier for AI research. *CoRR*, abs/1902.00506, 2019.

[2] Board Game Geek. Hanabi. Accessed 25. April 2023 `https://boardgamegeek.com/boardgame/98778/hanabi`.

[3] Christopher Cox, Jessica De Silva, Philip Deorsey, Franklin H. J. Kenter, Troy Retter, and Josh Tobin. How to make the perfect fireworks display: Two strategies for hanabi. *Mathematics Magazine*, 88(5):323–336, 2015.

[4] Markus Eger and Chris Martens. Practical specification of belief manipulation in games. In Brian Magerko and Jonathan P. Rowe, editors, *Proceedings of the Thirteenth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE-17), October 5-9, 2017, Snowbird, Little Cottonwood Canyon, Utah, USA*, pages 30–36. AAAI Press, 2017.

[5] hardmath (https://math.stackexchange.com/users/3111/hardmath). Efficient algorithm to find all unique combinations of set given duplicates. Mathematics Stack Exchange. URL:https://math.stackexchange.com/q/1902945 (version: 2016-08-25).

[6] Heap's algorithm. Heap's algorithm. Accessed 28. April 2023 `https://en.wikipedia.org/wiki/Heap%27s_algorithm`.

[7] G Cab (https://math.stackexchange.com/users/317234/g cab). Efficient algorithm to find all unique combinations of set given duplicates. Mathematics Stack Exchange. URL:https://math.stackexchange.com/q/1903340 (version: 2017-04-13).

[8] Hengyuan Hu, Adam Lerer, Alex Peysakhovich, and Jakob Foerster. "other-play" for zero-shot coordination, 2021.

[9] Michael Huth and Mark Ryan. The modal logic KT45^n. In *Logic in Computer Science: Modelling and Reasoning about Systems*, pages 335–337, The Edinburgh Building, Cambridge CB2 8RU, UK, 2004. Cambridge University Press.

[10] Zig Software Foundation. packed struct. Accessed 28. April 2023 `https://ziglang.org/documentation/0.10.1/#packed-struct`.

[11] Zig Software Foundation. std/packed_int_array. Accessed 29. April 2023 `https://github.com/ziglang/zig/blob/master/lib/std/packed_int_array.zig`.

[12] Zig Software Foundation. Zig. Accessed 25. April 2023 `https://ziglang.org/`.

A Generating the distinct combinations of size k data from 30 random games and calculating mean, min and max

```
python version:
Python 3.11.2
packages version:
more-itertools==9.1.0
numpy==1.24.3
```

```
import more_itertools
import random
import numpy
random.seed(0)

def main():

    # Blind draw 4 cards
    print("Blind draw 4 cards has:",
        getNumberOfCombinationsGivenHandOfSize(generateDeck(),4),
                "unique combinations")

    # Blind draw 5 cards
    print("Blind draw 5 cards has:",
        getNumberOfCombinationsGivenHandOfSize(generateDeck(),5),
                "unique combinations")

    # Random draw
    n = 30 #random rounds simulated
    # Data 2D array depending on how many players
    data = [[],[],[],[]]
    for _ in range(n):
        for player_count in range(2,6):
            number_of_cards = 5
            if(player_count >= 4):
                number_of_cards = 4
            deck = generateDeck()
            for _ in range((player_count-1)*number_of_cards):
                removeRandom(deck,random)
            data[player_count-2].append(
                 getNumberOfCombinationsGivenHandOfSize(deck,number_of_cards))

    p = 2
    for playerData in data:
        print("===number of players:",p)
        p+=1
        print("median:",numpy.median(playerData))
        print("mean:",numpy.mean(playerData))
        print("std:",numpy.std(playerData))
        print("min:",min(playerData))
        print("max:",max(playerData))
```

```
def generateDeck():
    arr = [1,1,1,2,2,3,3,4,4,5]
    superarr = []
    for i in range(5):
        for elem in arr:
                superarr.append(elem+10*i)
    return superarr

def removeRandom(deck, random):
    toRemove = random.randint(0,len(deck)-1)
    deck.pop(toRemove)

def getNumberOfCombinationsGivenHandOfSize(deck,size):
    return len(list(more_itertools.distinct_combinations(deck,size)))
main()
```