AI Notebook

Derin Gezgin
COM316: Artificial Intelligence
Connecticut College

Fall 2024

Contents

I. Simple Search

In this chapter, I will discuss two readings:
Blind Methods [1]
Search Strategies [2]

Blind Methods explores the concept of Breadth-First Search (BFS) and Depth-First Search (DFS). Imagine a path-finding problem where you must find your way to the goal from a starting point. You have multiple path options with different lengths in your search process. If you will use this path only once, you can choose whatever path you want and hope for the best. However, if you are going to do this trip multiple times, you might want to find the shortest and the least costly path available. To achieve this, we can create a *search tree*, which is a semantic tree that can visualize the node paths and branches. If a location in our tree has b number of path options, this is defined as a **branching factor** of b. On the other hand, the times we went to a deeper level of paths in the tree is defined as the **depth** (d) of the tree. The total number of paths can be defined as b^d.

In our path-finding problem, if you commit to a road option -ignoring the alternatives- this can be defined as the *depth-first search*. When you get stuck at a dead-end, and you are not in the goal, you can start returning until you find an unexplored path. On the other hand, you can use a breadth-first approach, where you explore your options by increasing the depth level. You do not skip to the next depth level without checking all the current paths. The depth-first search can be done using a **stack** (first in, last out queue), while the breadth-first search can be done using a **queue** (first in, first out queue).

Choosing the right search type depends on the decision tree we have in our problem. The breadth-first search is a good idea when we have a long (maybe infinite) path length. On the other hand, the depth-first search would be helpful in a case in which we are confident that all the paths can lead up to the goal. In this case, we can directly explore a path rather than waste time exploring depth levels. As an alternative, we can use a *non-deterministic search* to balance the depth and breadth of the search process by expanding the nodes randomly so as not to get stuck in long branches or wide path options.

Uniformed search strategies discover depth-first and breadth-first search strategies. It gives a bit more insight into these search strategies by proposing some alternatives.

It first explores the *breadth-first search*. Breadth-first search can be implemented using a first-in-first-out queue. Using this type of queue ensures that the nodes are explored in order of exploration rather than prioritizing the nodes that have just been explored. Breadth-first search can find the optimal path if all the paths have the exact cost. Although it seems to work well, we must know its time and memory requirements. For a tree that generated b nodes at the first level, there will be a total of b^2 nodes at the second level of our tree, which will increase exponentially. In the worst case -for a tree with a depth of d- our time complexity will be $O(b^{d+1})$, which is also the same as the space complexity as we have to store every node we explored.

Uniform-Cost Search would effectively find the most optimal path, even when all path costs

are not equal. In this case, the uniform-cost approach always expands the lowest path cost node rather than expanding the least deep node in our search tree. This approach will perform the same as the breadth-first search if all step costs are equal. The uniform search approach will fail if it encounters a node that has a cost of 0, which will make it stuck in an infinite loop. We can prune our approach against this problem by expanding the nodes greater than or equal to some constant value. As this approach always expands to the lowest cost path, it is guaranteed to find the optimal solution. The time complexity of the uniform-cost search can be shown as $O(b^{1+\lfloor C^*/\epsilon \rfloor})$ where C^* is the cost of the optimal solution, and ϵ is the minimum cost of each action.

Depth-first search is an approach that always expands the deepest node using a stack. It is much more resource-efficient and only needs to store the current path. When a path is proven useless, it can be removed from the memory. For a certain space of a branching factor d and a maximum depth of m depth-first search requires a storage of $b.m + 1$. A variation of depth-first search is backtracking search, which uses even less memory by generating only one successor, significantly reducing resource usage. While depth-first search uses $O(bm)$ memory space, backtracking search uses $O(m)$ memory.

One of the drawbacks of depth-first search is that it can get stuck in an infinitely long path and lose time exploring a useless branch. *Depth-Limited search* - similar to the non-deterministic search to prevent this getting stuck problem from happening while not exploring all the nodes at the depth level. Depth-first search has a pre-determined depth limit l, which creates a cut-off point for the tree and prevents unlimited exploration of one path. The time complexity of depth-first search is $O(b^l)$ while its space complexity is $O(bl)$. Depth-first search is a special case of depth-limited search where $l = \infty$. There are some methods of choosing the l value that will ensure a perfect balance between exploration depth and breadth.

I think that -as a start of our whole AI journey- these search strategies (and their variations) are very interesting and simple. Their simplicity and how they can find the best path in small grids are interesting, but there are also obvious limitations or caveats in complicated grid environments. I think that variations of Breadth-First and Depth-First search are very handy. Non-deterministic or depth-limited searches are smart alternatives for BFS and DFS, which I find very useful in large-scale search problems. One of the caveats of this search type is it is mainly in an equally weighted grid. This problem is solved using the uniform-cost search -up to a certain point- but there is no way this can be considered a smart option. In general, I think this simple approach to path-finding is a good start for developing path-finding algorithms. However, they are too simple and resource-intensive and do not have a real-world application these days. The preliminary reason for this is that currently, our search strategy does not act smart and uses a fixed approach regardless of the search space or our problem. There is room for improvement as we can make our robot prioritize certain locations depending on the heuristics of the surrounding points. At the same time, this approach is not smart enough to adapt to a changing search space, as the whole space should be inputted into the program before calculating the path. If there is any change in the space, our robot has to make all the calculations from the beginning.

II. Heuristic Search

In this chapter, I will discuss;
A Formal Basis for the Heuristic Determination of Minimum Cost Paths [3]

This paper introduces the infamous A^* algorithm widely used to solve path-finding problems. The path-finding problems generally had two approaches. The *mathematical approaches* brute-forces a solution to find a path rather than being concerned with finding a computationally efficient solution to the problem. On the other hand, the *heuristic approach* uses knowledge about the problem, which will increase the computational efficiency of the problem compared to the mathematical approach, which does not care about it at all. The paper merges these two approaches to create a balance between them.

According to the paper, an algorithm is defined as *admissible* if guaranteed to find an optimal path for any graph from the start to the goal. A useful search algorithm should have an evaluation function to determine how useful and important a node is for expanding. The algorithm proposed in the paper aims to prioritize the node with the smallest evaluation function value. The evaluation function $f(n)$ can be defined as $f(n) = g(n) + h(n)$ where $g(n)$ is the most optimal path to the current node, and $h(n)$ is the cost from the current node to the goal. This approach is admissible but not optimal as it expands fewer nodes than the other solutions. It is important to note that the choice of the heuristic function (h) is crucial in the problem-solving process. The selection of the heuristic function can determine a solution's admissibility and computational requirement.

I think this paper being published in 1968 and A^* still being used is very interesting and impressive, especially considering the computational limitations at that time. As I mentioned in my thoughts section in the previous chapter, it is very intuitive to feel the need for a heuristic in our search problem, as both depth-first and breadth-first search approaches (and their variations) neglect a heuristic and have pre-determined search methodologies. At the same time, this paper explores the importance of determining a heuristic and how it can lead to several outcomes by changing the balance between admissibility and computational complexity. In our grid-search problem, this is a significant breakthrough as we were stuck with BFS and DFS, which were very pre-programmed algorithms that lacked smart decision-making. Finally, I can say that this approach can be easily improved (and has already improved since 1968) by adding a dynamic heuristic, improving the heuristic function, etc. It was the start of many other similar path-finding approaches like D^*. As A^* can work directly with unequally weighted graphs, it can be used in many different places like video games, robotics, path-finding in Google Maps, etc. It still has some limitations, like it has to use a significant amount of memory -despite being less than BFS/DFS-or it relies on an admissible heuristic function during the search process, and there is not a direct way to come up with a good heuristics function. Regardless of these problems, A^* is still widely used, considering its simplicity and generality.

III. Real-Time Search

In this chapter, I will discuss;
Genetic Algorithms for the Development of Real-Time Multi-Heuristic Search Strategies [4]

This paper investigates the evolution of a *heuristic function* that can be used in real-time heuristic searches. The paper mainly works on a 64x64 search space consisting of free or obstacle nodes placed 1.0 units away from each other. A real-time agent that will lose energy during each step performs the search.

The paper presents two types of heuristics: *stable* and *unstable*. Stable heuristics have values that will not change regardless of wherever the *real-time* agent is in the grid. Distance from the goal (the Euclidean distance from the goal), distance from the start (the actual path from the start), congestion (obstacle density in the area), and momentum (preference to go forward rather than a zigzag pattern) are the stable heuristic parameters. On the other hand, unstable heuristics have values that can change as the location of the real-time agent changes. Distance from current (the distance from the current node to the frontier node) and move away factor (preferring to stay in the same path as the current node) are the unstable heuristic functions. These heuristics are represented in a 32-bit integer, divided into four-bit alleles representing each heuristic.

The genetic algorithm aims to find the best combination of the bias factors for our heuristics, resulting in an optimum search strategy. The genetic algorithm is called for every five training cycles in the training session to update the population based on its fitness value. The genetic algorithm uses default operations like selection, allele crossover, bit crossover, and mutation. All these operations are done with a fixed random probability, which ensures a fair distribution. This algorithm is tested on different test benches, ensuring it does not adapt to a single terrain type. The paper concludes that the genetic algorithm solution was not as good as the persistent search despite being able to adapt to different terrain types, but it still showed significant improvement and could finish the task successfully.

I think genetic algorithms are a very interesting concept (which I will explore more in the relevant chapter) and have huge potential. As it replicates the real-life evolution steps, it can produce emergent solutions that can be useful in complicated problems. The adaptation skill of the genetic algorithms is interesting as it can adapt to different situations, such as terrain types. Following this paper, many studies on genetic algorithms and evolution strategies have proven that they can be helpful and a strong alternative to the currently used methods. On the other hand, real-time search is a significant breakthrough compared to previous ones, as we can perform our search process in the changing grid environments. For example, in real-world path-finding approaches, our search space rarely remains unchanged, which is a problem that is not addressed either in simple search or the heuristic search. In this case, we can continue our search problem, and our robot will be able to react to the changing search space. I personally liked this paper a lot, and I hope that I can meet with the primary author one day!

IV. Game Playing

In this chapter, I will discuss;
A Chess Playing Program for the IBM 7090 Computer [5]

This paper presents the development of a chess-playing program for the IBM 7090 computer. At the time when this paper was written -1962- chess was an already explored area, but there were no successful attempts for a significantly good chess program. The initial research started by writing a 3-move-checkmate program, which was completed in 1960. The research continued with the aim of a general chess-playing program. The proposed solution was a game-tree approach, which involved expanding the tree and backtracking it to the root with an evaluation function. This process was called **mini-max**, which chooses the best move with the opposition of a player by assuming it makes the best move possible. At the same time, it was possible to evaluate the game state by win/loss/draw, and a further re-evaluation method was proposed in the paper. In this method, some mid-game evaluation -considering the locations of the chess pieces and how much control a player has on the board- is also considered. The evaluation can be optimized by adjusting its weights, but the solution did not involve any learning due to the computing limitations.

While a best-move generation system was not possible, a new system called **REPLY** was proposed. This system scanned the legal moves table and evaluated it using material balance, center control, etc. The first test program took from 5 to 20 minutes per move. While it did not make any illegal moves, it explored many unnecessary branches, which led to high computational and time complexity. A new evaluation method called **alpha-beta** -which will be later called alpha-beta pruning- was proposed to solve this problem. This method ensured that there was no unnecessary exploration in the board state. The final program involved sub-programs responsible for managing the general program, tree management, tree expansion, and move evaluation.

The final results for the chess-playing bot were convincing. The program was comparable to an amateur chess player with around 100 games experience. The moves the program chose were not very significantly good or bad, with occasional blunders.

In my opinion, introducing mini-max (including its alpha-beta pruning variation) was a significant breakthrough in game-playing research. As far as I know- Deep Blue, a chess-playing program that mainly uses Minimax with Alpha-Beta pruning, defeated Garry Kasparov in a real chess game. Following this, game-playing research expanded to different tree search algorithms like Monte-Carlo Tree Search and games like GO, Mahjong, Shogi, etc. I also think that despite being the paper that introduces this algorithm, this paper does a fairly good job of explaining the limitations of this approach. For example, mini-max has a significant size and time complexity requirement, and modifications like alpha-beta pruning might not be helpful. At the same time, it is difficult to implement mini-max in games like chess as the branching factor is huge, and the process can get computationally intensive. Considering the computational limitations, it is interesting how this was thought in 1968. I feel like they were aware that there was a lot of room for improvement at that time, but what they had as computational power was insufficient for them to try those new ideas.

V. Propositional Logic

In this chapter, I will discuss;
Propositional Logic [2]

Following the path-finding systems, there was a need to formalize complex situations for computers to understand. **Propositional logic** was the first attempt of the new era of rule-based systems. The syntax of the *propositional logic* is fairly simple. There are symbols for prepositions P, Q, etc., logical connectors $\wedge$, $\vee$, $\iff$, $\implies$, $\neg$, etc. and parentheses () for structures. A logical sentence can be formed using parentheses, propositions, and constants. A sentence can also be formed by combining multiple sentences. A propositional symbol can mean whatever we want to assign as a value. $\vee$ can be considered true when either or both propositions are true. On the other hand, $\wedge$ is true when both propositions are true. It is important to note that propositional logic does not check any causal relationship between the propositions. For example, the sentence *Sky is blue implies roses are red* does not show a causal relationship between the variables. A truth table can be used to visualize the validity and test a statement's outcome.

The pre-determined nature of the logical connectors also allows us to develop a list of inference rules. For example, if $\alpha \implies \beta$ and α is true, we can say that β is also true. Or, if the statement $\alpha_1 \wedge \alpha_2 \wedge \alpha_3 \wedge ... \wedge \alpha_n$ is true, we can say that each element of this statement is true, considering $\wedge$ is true only all the propositions are true. Lastly, if α_i is true, the statement $\alpha_1 \wedge \alpha_2 \wedge \alpha_3 \wedge ... \wedge \alpha_n$ would also be true, considering $\vee$ needs only one proposition to be true.

Constructing a truth table is a complete solution as it can visualize all the possibilities in our statement and achieve a solution. However, its time complexity is also exponential, which is far from efficient. There are some attempts like *horn sentences* to solve this problem, but they were not successful in fairly complex problems as they are very complicated even to phrase in a propositional logic setting.

I think that propositional logic is an interesting concept as this is the first time so far that we are able to symbolize human-like sentences for computers rather than simple search problems we have been working on for a while. At the same time, this approach broadened our work area as we are no longer limited to the grid search. While it was a breakthrough, there was too much room for improvement and limitations of propositional logic, which was a topic of interest in the upcoming years. I will also discuss these approaches in the following chapters. One of the key things missing from propositional logic was the ability to create your logical prepositions and rules. This limited us to the default logical operators and restrained us from further developing our methodology. At the same time -as the paper also mentions- adding more complex structures to our problem can spike the complexity of our program as we have to create larger and more complex truth tables.

VI. First-Order Logic

In this chapter, I will discuss;
Rules and Rule Chaining [1]

This reading proposes a simple **rule-based system**, which I mentioned in the previous chapter. Rule-based systems contain *if* and *then* patterns, symbolizing conditions and actions that would be executed or new assertions if the conditional part is fired. To check whether the rules are true, rule-based systems have a *working memory* that stores assertions. If the then part of a rule-based system shows an action, this makes this system a **reaction system**. **Forward chaining** is the rule-evaluation process that is used in rule-based systems. This will start from our current working memory and expand it by firing the if conditions and adding new assertions from the *then* parts of the fired rules. Although rule-based systems mostly run using forward chaining, it is also possible to do **backward chaining** in a rule-based system. In this case, we would start with a hypothesis and work backward through the rules to satisfy this hypothesis.

While forward-chaining and backward-chaining are possible for the same problem, some cases should be considered while deciding which one to use. If a typical set of facts can lead to many conclusions, it is better to do backward chaining for simplicity and a straightforward approach. Otherwise, this might lead to multiple conclusions, which can be confusing. However, if a hypothesis can lead to many unsatisfied rules, forward-chaining can be preferable for our implementation.

A simple rule-based system can be used in different areas. One infamous example is the Zookeeper, which aims to identify an animal using the initial knowledge base. Similarly, a rule-based system can help physicians prescribe the correct drugs to patients.

I think that rule-based reaction systems help create simple decision structures as they involve straightforward decision structures. These structures present a significant breakthrough compared to propositional logic, as we can represent more complex situations easily without needing a truth table. Besides the two main examples in the reading -ZooKeeper and Mycin— we can solve problems presented as discrete rules with certain outputs such as customer service bots, simple NPC coding in Game-AI, etc. However, it cannot give non-discrete answers or perform self-learning. I found this interesting as this is the first time we could have general freedom in the context of our problem rather than a grid search or a simple true-false predicate logic. There is some room for improvement as we can try to add actions to our rule-based system so that it can modify the world model we have, or we can adapt a rule-based system to any larger inference system to perform fact-checking and validation of the results.

VII. Production Systems

In this chapter, I will discuss;
Rule-Based Reaction Systems [1]

As I briefly mentioned in the previous chapter, rule-based reaction systems have a specific type that involves actions called *reaction systems*. This chapter explores the reaction systems and their structure. While the reaction systems have the same if statement as the rule-based systems, the *then* part involves an action to be done. At the same time, the reaction system has an **add-delete syntax** that can add and remove assertions from our rule base. While this seems useful, it can create conflicts as one rule can remove an assertion, which can be crucial in firing another rule.

We should have some *conflict resolution* steps in this case. One of the most prominent options for conflict resolution is *rule ordering*. We can order the rules in a prioritized list and fire the rule with the highest priority to avoid conflicts. At the same time, we can apply *context limiting*, which can separate rules into groups to prevent contradicting rules from being active simultaneously. We can specify the ordering of firing of the rules to be sure that the add/delete structure of the rules does not conflict with each other. There are several more conflict resolution steps like *size ordering*, *recency ordering*, *data ordering*, etc.

I think that production systems are not much different from the first-order logic except the ability to have actions resulting from the if statements. I am not sure why this was even considered a breakthrough, but it is obvious that the *then* syntax changed to *add/delete*. However, production systems check for contradictions between the results of the if statements with conflict resolution, which the previous systems we studied did not attempt to solve. This is a nice feature as we do not have to worry about our program breaking itself. One of the main flaws of the production systems is they cannot self-decide the order of firing of actions and heavily rely on the conflict resolution steps that were implemented. At the same time, it heavily relies on pre-determined conflict resolution steps, which might be hard to use in large-scale structures. Production systems are a step closer to the systems that apply to the real world as they can perform actions and change our world model.

VIII. Planning

In this chapter, I will discuss;
STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving [6]

This chapter explores the **STRIPS** (STanford Research Institute Problem Solver) problem solver and explains the development & features of it. STRIPS is a problem solver that finds a series of actions to transform a given initial **world model** into a final one. It works similarly to a reaction system where a robot must perform actions to achieve the goal state. STRIPS consists of a set of *well-formed formulas (wffs)* that represent the positions and attributes of the attributes in the problem-solving space. WFFs can establish basic rules in our problem space, such as an object not being in two different places simultaneously, an object must exist at least in one place, etc. A STRIPS planner's available set of *operators* is named "**schemata**"s. Operators represent actions our agents can take, and they have specific built-in constants that are pre-determined during their usage. These constants include the conditions in which the operator can be used and the operator's effects. When we decide to use an operator, we must check whether any of the well-formed formulas in our space contradict our world model. If needed, we should backtrack multiple well-formed formulas and check all the prerequisites.

How we come up with a solution using a STRIPS planner is also crucial. While it is possible to do this by a brute-force approach where we try out all combinations and orderings of operators, it is not very feasible as we plan to use this in a large and complicated real-world setting, making this challenging considering computational requirements. Rather than this brute-force approach, we can adopt a strategy to check the differences between our current world and the final world state we want to achieve. After detecting the differences, we can start from the starting position and try to choose operators that can approach us to the final state. In this case, as long as we do not have a direct solution, we would have sub-goals to find how to satisfy using the operators we have. The selected operator, in this case, is called a **relevant operator**. We can repeat this process recursively until we achieve our desired world setting. This approach of setting sub-goals as long as we do not satisfy the final world state can also be represented by a *tree structure*. One of the main setbacks of the STRIPS planner is its reaction system nature. As some of the specific actions can have results that can affect some of the prerequisites of other actions, during our planning process, we should be mindful of the ordering of our actions. In most cases, using our tree representation, we can avoid this problem, but in some specific cases -especially when we have two options to choose from- we should check for contradicting situations. After the decision tree is structured, the operator path our agent will take is decided by the number of sub-goals that must be tackled and the complexity of these sub-solutions, other than the conflict resolution I mentioned.

One of the main challenges in STRIPS is determining the representation of the world model we are working on. While it is possible to set the well-formed formulas (wffs) in our world model during the problem-solving process, this is not feasible as we would have many copies of the unchanging wffs in our memory space. In fact, the only wffs that change throughout the problem-solving process are the wffs related to our robot and the objects it might manipulate.

As a solution to this, all the possible wffs are stored in a joint memory structure, and they are marked as visible to set the specific attributes of our search space. In our initial setup of the world, all the wffs we would like to set are marked as visible, and they stay that way as long as they are not changed. To keep track of the wffs that are changed in our world model, we have DELETIONS and ADDITIONS lists. This way, we do not have to create a copy of our world model by copying each wff in the memory space. This can save a lot of memory and time in a large world model. Considering the time this paper was written, this change can be considered a breakthrough.

As STRIPS planner is designed to be a general-purpose problem solver, it can be used in various real-world tasks. For example, it can be used for a box-moving task from Location-A to Location-B, a task that aims to create a tower using boxes, or a complicated task that aims to solve a Rubik's cube. Although the STRIPS planner can work in a real-world environment with a pre-set of rules, there is much room for improvement. For example, the STRIPS planner can be improved by creating a more robust evaluation for our decision tree. At the same time, we can use blocks of operators as a whole rather than a single-operator approach. Lastly, our STRIPS planner does not involve any learning, which limits its ability to adapt to more complex situations, generate different operators, and generalize its abilities.

I think that the STRIPS planner is a significant breakthrough in rule-based AI systems as it introduced dynamic problem-solving approaches with useful methods to simplify the problem and achieve a solution. Adding well-formed formulas allows us to set limitations to our world model and let the problem-solving algorithm act freely in between these borders. At the same time, relevant operators allow our agent to divide the large problem into sub-problems, speeding up the solving process and making some difficult problems solvable. At the same time, the additions/deletions structure provides effective memory usage, allowing representation of a larger world structure. I think that one of the nice things about the STRIPS planner is that it is sometimes smarter than humans as it approaches a problem in small pieces and does not attempt to solve it as a whole, which is a thing we sometimes forget to do.

IX. Semantic Networks

In this chapter, I will discuss;
Semantic Networks [2]

With their introduction in 1909 by Charles Peirce, semantic networks brought another perspective into the rule-based logic systems. One of the main breakthroughs semantic networks presented was the ability to present complex situations in a simple syntax. It can represent individual objects, object categories, and relations among them by establishing connections among them. It also can establish default connections, which can be helpful as we do not have to assert every known fact about an object when creating it. For example, if we have a person called Jim, we would know that -as he is a human- he will have two legs, arms, eyes, etc. As seen in this example, semantic networks are really good in inheritance. At the same time, Jim can be a member of two categories: a human and a judoka. In this case, he can inherit the features of a human -which I mentioned- and also the features of a judoka, like owning a red(?) belt, a judogi, etc. This shows that semantic networks are also good in multiple inheritance. It is important to note that if there is a more specific connection in our network, the default -inherited- values will be overridden by our semantic network system.

Semantic networks also can establish inverse links between two connectors that imply each other. For example, Derin can be studentOf Prof. Parker, and Prof. Parker hasStudent Derin. In this case, when we are looking for who is studentOf Prof. Parker, we can also search for hasStudent connections. One of the obvious drawbacks of semantic networks is that they can only represent binary information in its default form, which prevents us from representing complex sentences. We can create event objects which can be used to represent these attributes with individual links. Although we can use this structure to represent real-world sentences, we still miss negation, disjunction, etc. As the situations we want to present get more complex, semantic networks can become inefficient as there are many connections. In this case, a technique called procedural attachment can be used, which allows the filtering of results for the query to save time and resources.

I think that semantic networks are interesting structures as they can represent real-world relationships between people, objects, etc. Their ability to establish default connections makes them exceptional, as before this structure, we had no option to represent real-world connections between facts that we have in our working memory. At the same time, having default values and attributes for facts resembles object-oriented programming, which can also be considered a breakthrough. One of the caveats of this structure is it might get overcomplicated and non-efficient in large-scale applications as we do not have a direct option to index or access facts, and we have to go through them to access a specific one. At the same time, it represents binary relationships, as we can or cannot have a relationship between two objects, ignoring the possibility of a probabilistic relationship. I find semantic networks impressive as it is the first time we can represent real-world connections between entities. I feel like if I happen to do a very smart robot, I will probably include a semantic network-like structure (or an improved version of it) to give my robot a better sense of the context of its surroundings.

X. Frames

In this chapter, I will discuss;
Frames and Inheritance [1]

This reading mainly focuses on **frames** and their structure, starting from a basic frame structure and how much it can help us represent complicated situations. In the semantic networks, we introduced the (multiple) inheritance and how it can represent connections between objects. Frames re-explore this concept and take it a step further.

In a semantic network, a frame can be defined as the structure that contains an object and the links that connect it to other objects. Links can also be defined as slots, and the objects the slots point to can be defined as slot values. A frame representing a specific type of member is called an **instance frame**. A frame representing a class/type is called a **class frame**. Frames have several special slots, like an *Is-a* slot, which ties instances to classes, or an *Ako* slot, which ties classes together. A significant difference between frames and semantic networks is that frames can have procedures that can create a setup of predetermined actions. For example, constructors will invoke a set of procedures to set up an instance of a class. A **constructor** will connect the class frames to the new instance using Is-a slots and establish the Ako connections if needed. The Ako connections will also inherit the frames from the superclass to ensure that our low-level object has all the necessary frames. On the other hand, a slot writer can set values in frames, while a slot reader can get the slot values –similar to the getter and setters in Java.

During the setup of a frame using a constructor, we can set specific slot values for the slots of our instance frame. At the same time, the slot values of our instance frame can be predetermined and central. This central setup allows us to make changes that will easily apply to the rest of the program, prevents us from making mistakes, and allows easy distribution of values. This procedure of placing default values into slots is called a **when-constructed procedure**. In simple cases where there is only one class without upper-level classes it inherits from, invoking a when-constructed procedure will not cause problems as there are no conflicts. However, in multiple inheritances, the same slots might be attempted to be set to a different value. In this case, we have to have a mechanism to determine which classes are superior. We can use the **class-precedence list** to give us the specificity of a class compared to other inherited classes of the object. Using the class-precedence list, we can select the more specific class to invoke the procedure to set the relevant field.

In the case that an instance is connected to more than one class, this causes branching for that instance. This is an unsolved issue as we do not know how to handle classes in the same hierarchies. There are several approaches for resolving this, such as an **exhaustive depth-first search** (prioritizing the left branch) or an **up-to-join search** (do an exhaustive depth-first search but ignore repeated classes until the last time). These approaches are all formed in the graph representation on the class inheritance system, which can also cause a problem as the locations of the nodes can change while showing the same **class-inheritance graph**. This can result in different priority orderings and outcomes in the inheritance selection.

I think frames are a big spike compared to semantic networks as they introduce slots and

procedures, adding dynamism to our knowledge-based program. We can have instance and class frames separately to represent specific and general information using frames. I think that they are specifically interesting as my robot -that I mentioned in the last chapter- can have general world information and also specific information about objects without storing copies of the general information also thanks to inheritance. I think that one of the problems with the frames is the class-precedence list and how it handles complex situations. In a very complex real-world problem, this precedence-list approach might not be enough. I think that it is impressive that we are getting closer to modern software engineering concepts like inheritance, object-oriented programming, multiple inheritance, etc.

XI. Thematic-Role Frames

In this chapter, I will discuss two readings;
Frames and Commonsense [1]
Conceptual Dependency and its Descendants [7]

In this chapter, we expand the frame structure we had been working on and explore how to expand frames into real-world action series with sparse representation. Doing a real action in the real world involves a series of events, agent, and many other semantic structures depending on the situation. Our frame can represent a single piece of action, but it cannot make deductions on a series of actions that our initial action might require or trigger.

In our general action representation, we have some required fields, which come from the semantic representation of the action. Our main structure, in this case, is a simple noun. Each noun in an English sentence has a thematic role, which describes how that action takes part in the action in that sentence. The person who performs the action is considered the agent, the helper or partner of the agent is considered the co-agent, the person who the action is done for is considered the beneficiary, the object that is being affected is the thematic object, the object that is used is the instrument, old/new surroundings are the location, etc. A thematic frame does not have to include all of these attributes. Thematic-role frames can represent the primary meaning of a sentence, however, it cannot represent or deduct the implied information in a sentence there can be more to explore and interpret in a sentence, which it cannot do.

While establishing a thematic role frame, there are some rules that one should follow. For example, prepositions come before a noun can limit which rule it can have: *"by will be"* before an agent, *"with will be"* before a coagent, *"to will be"* before destination, etc. At the same time, the semantic meaning of a noun can also limit its possible roles in the thematic-role frame. There can be issues with ambiguous verbs, which can be solved using particles- a small family of words. For example, the difference between pick-up and pick-out can be differentiated by particles up and out. On the other hand, an abject can be in the physical world or one's mental world, depending on its semantic meaning. Lastly, ownership of an object can also be interpreted from the other elements of the phrase. The exact meaning of a thematic object can also be determined through a grammatical analysis, branching out the possible meanings and evaluating them based on the other frame slots.

While this seems like a good solution to show complicated real-world relationships and understand semantics, the wide range of English verbs and general vocabulary prevents us from creating a generalizable program. In this part, the chapter introduces 14 primitive actions such as **expel**, **propel**, **ingest**, **hear**, etc. We can use action frames and put these primitive actions in a specific field to represent a general type of action more easily and universally. At the same time, we can have a state-change frame that can show us causal relationships. Moreover, we can have actions involving sub-actions to achieve the frame's main goal. Thematic-role frames are also relatively good in paraphrased sentences as they accept two sentences as paraphrases of each other if they expand into the same primitive action, which is a disputed assumption and does not have a certain answer.

The reading by Lytinen takes this concept further by introducing a set of default representation structures to our system and the conceptual dependency theory. The conceptual dependency aims to create a generalizable representation of our world model, which can be understood in all languages. This is achieved by introducing primitive actions which can be used with several attributes to represent sentence structures universally. Conceptual dependency aims to symbolize sentences by removing the effect of paraphrasing and also uncovering the implied statements in a sentence. This requires the representation to be general and canonical, as our system should be reliable in its inferences and predictions.

To satisfy the mentioned requirements, conceptual dependency had three main types of grammatical structures. A set of primitives represents the actions in our world space, a set of states represents the preconditions and results, and a set of possible conceptual relationships between primitives, states, and objects. These are the set of primitives in our conceptual dependency program:

- PTRANS: The transfer of location of an object
- ATRANS: The transfer of ownership, possession, or control of an object
- MTRANS: The transfer of mental information between agents
- MBUILD: The construction of a thought or new information by an agent
- ATTEND: The act of focusing the attention of a sense organ toward an object
- GRASP: The grasping of an object by an actor so that it may be manipulated
- PROPEL: The application of a physical force to an object
- MOVE: The movement of a bodypart of an agent by that agent
- INGEST: THe taking in of an object (food, air, water, etc.) by an animal
- EXPEL: The expulsion of an object by an animal
- SPEAK: The act of producing sound, including non-communicative sounds

Each primitive object has a set of associated slots, providing fields that provide important context in our conceptual-dependency program. For example, PTRANS has slots like ACTOR (person who did the action), OBJECT (the affected object), FROM (origin of the action), and TO (destination of the action). While filling up these slots using the implicit information in the text, a set of inference rules was written to directly scrape the information from the text. Depending on the representation requirements of our situation, we can also chain these primitive actions to symbolize complicated situations.

Conceptual dependency is an interesting concept that brings the representation format of a semantic network to a frame-like structure. The semantic network theory is important in learning to visualize knowledge to interact with our world model. At the same time, frames were a step further as we could represent real-world actions with attributes in a frame-like structure. This conceptual dependency theory brings the systematic approach we had in the semantic networks and frames together and creates a universal deduction and symbolization language that can be used inter-languages and also to understand implied information in texts during the translation between languages.

I think that the thematic-role frames are generally interesting as they introduce a completely different type of language representation compared to the structures we have had so far. I think that one of the most important contributions of the thematic role frames is that they introduce general language structures we can use to represent the everyday sentences we

have. It is so hard to say that these are 100% right representations as -with the limited amount of primitives- we cannot represent the whole meaning of the sentence. However, I think they can represent the sentence's general idea -at least in most cases. One of the main problems of the thematic role-frames is that they cannot figure out the implied meaning of a sentence. I think that this is still a problem these days, as our best language models still cannot understand the basic semantic phrases. One of the infamous one is "How many r's in strawberry."...

XII. Scripts

In this chapter, I will discuss;
Scripts, Plans, and Knowledge [8]

The paper by Schank and Abelson describes the program called SAM (Script Applier Mechanism), which uses scripts to understand and create plans for particular situations. A script is a structure that can represent a specific set of actions in a particular context using its slots and requirements. An example of a script can be a story that involves a series of actions in a particular order. A script can be really simple and straightforward, or it can also implemented inside of another one. A script can also have several sub-scenes, representing a specific set of actions to achieve the main goal of our script. As a fail-safe, an action in the script must be completed to proceed to the next action. In a case where it is impossible to complete an action, we can have conflict resolution steps to solve the issue. As scripts represent real-life situations that can be very changeable, there are pre-set crucial points in a script that should be met. For example, eating and paying are crucial in a restaurant script. The rest of the process can be handled as different variations of the restaurant script. During the execution of the script, there can be errors due to the real-world application of the script. A script can fail in 3 different ways. It can be interrupted by another script -distraction-; it can be prevented from action as usual -obstacle-; and it can complete the action in an unexpected way –error. For example, in the restaurant script, if the waitress does not check on us or the restaurant is closed, we will not be able to satisfy all the critical components of this script. Our script will try to check for these requirements, but we can end it early as a fail-safe.

SAM (Script Applier Mechanism) takes conceptual dependency structures (which we covered in the previous chapter) as input. It will identify the script (which was already built-in) that we should use and fill in the script with new input values. It can make casual relationships between the inputs and conceptual dependency structures to detect and complete the script structure. The initial goal of SAM was to process the input text and paraphrase it using the conceptual dependency structure it detected. This program can also be used to create a summary of a text.

Plans are an upper-level structure that can be crucial in our script structure. In real-world narratives, it is not very easy to make sense of a piece of text by understanding the context. Plans include a set of actions that should be done to accomplish a goal or a set of prerequisites that should exist for the given situation to be true. A **deltact** is a set of actions that lead to our goal in the script. In SAM, there are five deltacts that can be used:

- $\Delta AGENCY$: A change in obligation to do something for somebody
- $\Delta CONT$: A change in the control of an object
- $\Delta KNOW$: A change in what an actor knows
- $\Delta PROX$: A change in the proximity relations of objects and actors
- $\Delta SOCCONT$: A change in social control over a person or a situation

A plan includes **planboxes** that can tackle the sub-goals for that specific action. This is similar to the STRIPS planning as we tackle the sub-goals individually. Each planbox has a

list of primitive actions (ACTs) that will achieve a goal. Planbox checks these ACTs and their pre-conditions to ensure no conflict in the program. There are three types of preconditions in the planning. A controlled precondition can be fixed as we can do the negative of the same thing. A negative uncontrolled precondition cannot be fixed, and we should change our approach. A negative mediating precondition can be altered, but it requires plans to change.

SAM (Script Applier Mechanism) and PAM (Plan Applier Mechanism) can be used to better understand the semantic information we have so far. With the help of conceptual dependency, they can summarize large chunks of text by simply using the conceptual dependency syntax. At the same time, SAM and PAM can bring a new level of real-world understanding to computer programs. One of the key missing parts of conceptual dependency, SAM, PAM, or anything at that time, was that there was no implementation of forgetting, which also led to storing unnecessary information in our system. The paper concludes that we should come up with a way to store only important knowledge and forget the unimportant parts, which can be done with heuristics.

I think scripts cover an important point the frames miss, as they can represent sequences of actions and events in specific contexts. I find the modular and sequential representation of actions that a script can do very fascinating. I think they are a good step for an AI agent to think about the necessary steps a final action might require. Moreover, if we can come up with an agent that can reason about these sequences, it can be interesting, too. For example, our script has a basic knowledge of some action sequences for preparing food. What if I add a new goal, like making a salad? If we can come up with a system that can understand that making a salad is like making a sandwich where we have to mix the ingredients in a bowl rather than a piece of bread, this agent can be used in very different areas. The current version of scripts cannot do this as they heavily rely on a previously defined set of actions. I think that another major problem with scripts is memory management, which the paper also mentions, as the Scripts cannot forget information. I think that -despite this being the first paper to mention this- most of our rule-based systems cannot distinguish between important and non-important information, and this can result in significant memory usage.

XIII. Case-Based Reasoning

In this chapter, I will discuss;
Case-Based Reasoning: Foundational Issues, Methodological Variations, and System Approaches [9]

In our previous knowledge-based AI systems, we were limited to our knowledge base and whatever we had beforehand. **Case-based reasoning** introduces the concept of re-using past experiences as a way to solve problems that we tackled in the previous knowledge-based AI problems. In a case-based system, a **case** means a problem our system must tackle. Any previous case that can be used in our knowledge-based can be called **past case**, **previous case**, **stored case**, **retained case**, etc. Case-based reasoning's main strength is that it can make generalizations and have a sense of memory/learning throughout the problem-solving using the previous cases. It is an approach that solves new problems by finding similar previous problems and applying them to the new problem case. At the same time, as I mentioned before, case-based reasoning can learn incrementally and increase its knowledge base, which we did not have in our previous knowledge-based systems. Case-based reasoning shows human-like behavior as real humans will also learn from their previous experiences and not make the same mistake again on the second try –or the third time, depending on the human:))). There are some sub-categories of case-based reasoning systems.

Examplar-based reasoning aims to have a simple classification loop where the answer or the current case is the most similar past case. In this variation, modification of a solution and learning is not available. Similarly, **instance-based reasoning** aims to have a non-generalization approach that uses inductive machine-learning methods for classification. **Case-based reasoning**—as I mentioned above—can adapt its knowledge base to past examples and generalize its knowledge base. On the other hand, **analogy-based reasoning** aims to solve problems from a different domain compared to case-based reasoning, which aims to solve problems in the same domain. This approach modifies the current case to be similar to those in our knowledge base.

Case-based reasoning involves a cycle of actions throughout the problem-solving and adaptation process. It starts by finding similar previous cases to our new case -**RETRIEVE**-. Then, it uses the retrieved case to solve the current problem -**REUSE**-. Using this solution, our system gets feedback from our world model -**REVISE**— and saves the useful parts of this learned case -**RETAIN**-. This cycle allows our model to learn by seeing more examples without outside intervention. While case-based reasoning seems like a perfect system that is self-sustainable, it has some issues related to representation and the methods it uses.

Firstly, a case-based reasoning system must find an efficient way to recall previous cases from memory. While this is not an issue in a small knowledge-base, it is a significant issue in a large knowledge-base required for generalization. At the same time, a solved problem should be inputted into the memory in a certain way to be usable for future examples. A solution to this is the **dynamic memory model**, which aims to store the examples using a hierarchical structure. It stores the cases with similar properties in the same structure, making it easier to find a specific case by navigating through this tree-like representation. In this representation,

it is possible to store generalized and domain-specific knowledge. Another way to store the knowledge base is to have categories for different cases –**exemplars**. In this method, cases are connected to general categories with specific features. There are also pointers between cases (exemplars) that show their differences. Cases are again stored by similarity and also connected like a semantic network.

Retrieval, reuse, and revision methods also have some challenges. In general, retrieval of the cases for usage is done mechanically by checking the similarity between two cases. There are attempts to check for semantic similarities between cases. During the identification of similarities, if the program encounters an unknown feature, it can ask the user to provide further information for identification. While our program can pick several candidates for the best matching case, the candidates can be evaluated for a second time to select the best case possible. In the reuse part of the program, it is important to select the important features of a case to store in our memory. While it is possible to store the complete case for reuse, it is also possible to store the deal-breaker part of the new case for case selection. Lastly, the feedback part of the case-based reasoning system can also have some flaws. During the feedback process, it is possible that the evaluation process can take a long time, which can result in many un-evaluated cases in our system. Similarly, when we get feedback from our world model, our reasoning model has to repair the existing case depending on the feedback to ensure that the error will not occur again.

A case-based reasoning system can be integrated into the problem-solving process of a rule-based system or a larger system as a helper. It can used to speed up the learning process and make the program integrated and adaptive to different situations. It can also make reactive changes depending on the situational changes in our world model. This separation of rule-based systems is done very explicitly to control their distinction and interaction. These integrated systems can be used in several areas, such as airplane composite material development, valve selection for boat construction, or help-desk systems.

I think that one of the main strengths of a case-based reasoning system is it can learn from experience without any outside help. Since the beginning of our AI journey, this is the first time we have learned, and I think this can be considered a breakthrough. I think that the basic 4-step cycle (Retrieve, reuse, revise, retain) is very self-sufficient. In my opinion, one of the main problems with Case-based reasoning is that it can end up being very good in a certain area rather than generalizing its knowledge, as the system would rather stay in its comfort zone than try different experiences. At the same time, -as in our previous AI systems- the high memory usage is still a problem, and our system can use extensive amounts of memory in large world models. I also think that -similar to the scripts- we can implement a forgetting mechanism to our case-based reasoning system which can help with this memory issue.

XIV. Current-Best Learning

In this chapter, I will discuss;
Learning Structural Descriptions from Examples [10]

In this paper, we discover a new learning system that can learn to recognize 3-dimensional real-world concepts by using abstract descriptions of the world model. It works on a simple arch-constructing example and how this task can be taught to a computer. It goes through the importance of representation of examples, good training sequences, and near-miss examples –an example that is nearly the same as the expected example.

In this learning form, the machine learns by seeing a sequence of examples. It first starts with the correct example of what the end goal looks like. It then evaluates more examples by comparing them to its current concept of a correct example. From every example -depending on the example being a correct example or a near miss, etc.- it adjusts its concept of correct and incorrect based on the requirements. Similar to case-based reasoning, the machine uses previous examples to improve itself, and it is seen that the model performs increasingly well.

I think current-best learning is a very interesting concept, but I feel like it is an obvious downgrade compared to case-based reasoning. I cannot see why this can be preferred over case-based reasoning. We can easily use case-based reasoning for this learning procedure. Considering that this paper was written in 1970, compared to the case-based reasoning paper, which was published in 1994, this difference in features makes sense. I think one of the main limitations of this system is its need for near-miss examples. At the same time, this system can learn very specific tasks which lack significant generalization. I find the idea of this paper interesting but it is far from being usable, considering the alternatives like case-based reasoning.

XV. Version Space Learning

In this chapter, I will discuss;
Learning by Managing Multiple Models [1]

This chapter takes current-best learning further by introducing direct positive and negative examples rather than near misses. A version space structure is a specific knowledge system limited to our example space. During our training session, depending on the example type, we specialize or generalize our models. To manage this two-ended structure, there is a specialization and generalization tree, representing a generalization model (which matches everything) and a specialization model (which matches only one thing). Writer functions connect a node -knowledge bit- with a model, while the reader functions produce a model for the node. Version space learning aims to intersect the generalized and specialized models by avoiding negative examples, which is possible with enough examples. As we have positive examples, we can prune the generalization tree (to make it more specific and prevent a match), and as we have negative examples, we can prune the specialization tree (to make it more general to ensure a match). We aim to avoid negative examples while generalizing our model enough to satisfy all positive ones.

Version-space learning has a symmetric handling for positive and negative situations. If our example is positive, we will generalize all specific models to include the positive example while checking that our specific model is a specialization of a general model and not a generalization of another specific model. Following this, we would prune the general models that cannot exist with the current positive example. On the other hand, if our example is negative, we can specialize all general models to eliminate the negative example while ensuring that the general model is a generalization of a specific model and not a specialization of another general model. Similar to the previous case, we would prune all the models matching the negative example.

One of the advantages of version space learning is that it can learn possible negatives and positives during the training. This is an important feature, considering we do not have to finish our training session to have a working model; we can start using the model—with a margin of error—at any point in the training session. This can be very useful when we do not have complete information but must make a deduction.

I think that version space learning is a very good deduction system, and maybe the best one. Rather than focusing on performing actions to achieve a certain goal, it focuses on making a deduction about a certain situation. Its ability to narrow down possibilities and achieve generalization/specialization is interesting as I genuinely think that the the working methodology of this system can still be used these days to solve logical puzzles. I think that, one of the main caveats of this system is that it requires complete information which is not very convenient in real-world examples as we can have noisy data which can miss some examples. At the same time, in my opinion, this system can struggle with more continuous data as it has to come up with a way to evaluate all the cases in this continuous data. I can say that this is definitely in the top 3 of my favorite systems in this class.

XVI. Feed-Forward Neural Networks

In this chapter, I will discuss;
Distributed Neural Network: Dynamic Learning via Back-propagation with Hardware Neurons using Arduino Chips [11]

An Artificial Neural Network is a set of neurons with connections mostly implemented in virtual or single-chip real-world environments. There are many different approaches for different neuron implementations, but none have been close to this paper's goal. This paper aims to create an artificial neural network with a one-chip-per-neuron approach while considering cost restraints. Considering these requirements, the paper utilizes Arduino Pro Mini (APM) microchips to use as neurons and perform back-propagation.

APM chips are selected as they are widely available, small enough, and significantly cheap (around 10$). Each APM chip is a single neuron that can learn similarly to a biological neural network. As this paper is a proof-of-concept, one APM chip is used as the input layer, two as hidden layers, and one as the output layer. The network can learn AND, OR, XOR, and XNOR functions only using the hardware APMs.

The input is established using switches, representing 1/0, allowing real-time user input. Starting from the input neuron -until the output neuron-every neuron takes the weighted sum of all the neurons connected to it and passes it into an activation function to determine its output. When the final layer is reached, the network's output is compared to the expected output, and an error rate is calculated. Depending on the error rate, all the previous values are updated in the magnitude of the error rate and the learning rate. Communication between the chips is established using the I2C (Inter-Integrated Circuit) bus, which also allows for the Master-Slave connection. The two main commands that were used are read and write.

The weights and inputs are stored in different arrays for each neuron, and the learning rate was set to 0.35 for all neurons. During the execution of the program, the hidden neurons constantly request signals from the output APM Master. When it's triggered, it starts to receive the signals, and after receiving 4 bytes, which means that the output neuron sent an error gradient as feedback and the receiving process is complete, this error gradient can be used to update the weights. Similarly, the final output layer has two functions, which read the hidden layer signal and its output.

Initially, all the layers are set to have a threshold of -1 and random weight values between -1 and 1. The training starts with a signal from the output neuron and continues as a training loop. The study tested this structure using different error margins, and it was observed that it is possible to successfully learn the OR, AND, XOR, and XNOR functions. While XOR took the least time to learn in a 30% benchmark (43 seconds), AND took the most (around 2 minutes and 36 seconds). At the same time, it is observed that the network could learn other operations easily after learning a specific operation. The paper concludes by adding that there is much more to explore in this area as the network structure can be widened and more complicated functions can be studied.

I think that this is a very interesting paper, as we can see a real-world execution of neural network learning. Using the Arduino chips as a real perceptron is an interesting idea as we can see how the learning happens. I think that one of the significant issues with this paper is that the scalability of this approach is not really possible while keeping the restrictions of 1 chip - 1 neuron and low price requirements. At the same time, I think that the development process of this circuit would be hard as a wrong connection can lead to burning the boards (maybe I am not very good at robotics yet...). Another important point is that network learning is significantly slower than virtual networks, which can be problematic in a larger network structure. Regardless of these problems, I still think that it would be interesting to see this idea on a larger scale.

XVII. Perceptron Learning

In this chapter, I will discuss;
Parallel Distributed Processes [12]

It is common knowledge that humans are significantly better than computers. But why are they exactly like this? Despite not seeming like it, the human brain is a very complex structure that can perform and think simultaneously in many tasks, such as reaching and grasping. Although reaching and grasping seem like simple actions, they involve many actions and background processing simultaneously, which shows how fascinating our brain is. At the same time, our brain can complete incomplete information such as reading a covered label. In all these examples, the whole information processing system goes through many small processing elements called the *Parallel Distributed Processing*. The parallel distributed processing framework is based on the connected net of neurons in the human brain that transmit signals through our brain to activate certain parts of our brain. As I said above, our brain can do this transmission process in parallel for more than one part of the body which computers still cannot do.

Although computers cannot do the parallel aspect of the brain, imitating the neural network architecture of the brain has been a point of interest. A PDP model can represent neuron-like connections of the brain in real-world examples and situations. It is mainly used in the fact-representation space as the attributes under the same category are grouped, and specific instances with specific attributes are connected to these attributes with links (like the dendrite). Having the necessary connections will fire the attribute, leading to firing the instance itself.

I think that this was the least fun reading among all the readings (including the optional one). I find the explanation of the topic interesting as it first starts from the humans and then switches to the computers, but as the current type of neural network we use is slightly different than what we had in the paper, I was not able to relate to or follow what is being told in the paper. At the same time, I think that the paper was also significantly long -despite the content was technically a repetition of itself- and I was not able to focus on it a lot. As an introduction idea, I think that parallel distributed processing is not the best way to explain this brain-like relationship, but other than this, functionality-wise, I think that it is impressive how the cluster-like structure can also be used to represent a neural network-like structure.

XVIII. Genetic Algorithms

In this chapter, I will discuss two readings;
Using a Genetic Algorithm to Replicate Allopatric Speciation [13]
An Introduction to Genetic Algorithms [14]

The chapter from the Mithcell book introduces the general outline of evolutionary computation, its history, and its application.

The history of evolutionary computation goes back to the 1950s-1960s. In these years, evolutionary systems were discussed mainly for the optimization of engineering problems. Following this, this idea evolved into genetic algorithms that can be used in finite-state machines. In the 1960s, Holland invented genetic algorithms to study adaptation and its nature. Holland's GA consisted of chromosomes, the idea of a population, selection, crossover, etc. This whole process replicates the biological recombination process between genes.

As problems computer scientists study get more and more complex, we end up needing ways to achieve a solution. Biological evolution was an appealing way for us to achieve solutions. Before going deep into the computer science side of genetic algorithms, it is useful to know the biological side of this area.

All living organisms are made of *chromosomes*, which store important information about the organism. A chromosome stores DNA with many *genes* inside it. The complete collection of the chromosomes is called the *genome*, and *genotype* means the genetic content of an organism. During the reproduction of organisms, a new copy of the chromosomes should be created for the new offspring. In this case, two chromosomes *crossover* by exchanging parts. This also ensures genetic variation throughout the generations. During this crossover process, there is a very low odd of a change in the genes, which is named *mutation*.

In a genetic algorithm, a chromosome generally represents a possible solution for the problem. Genes can be defined as parts of this chromosome that represent individual parts of this solution, like the hyperparameters. Generally, the genetic representation is in binary format. Like its natural version, crossover happens by exchanging parts between the chromosomes. On the other hand, mutation happens by flipping one of the gene values.

After defining the technical terms, we have to figure out how we will evaluate the chromosomes (possible solutions) in our genetic algorithm. This is where the term of *fitness function* comes into play. A fitness function can be in any form, as its main goal is to provide feedback to our genetic algorithm about which solutions are strong and which are weak.

A genetic algorithm goes through a cyclic process until it satisfies the ending condition, which can change from problem to problem. It starts by generating a random population of candidate solutions. It calculates the fitness of each chromosome in the population. After determining the fitness values, a selection of individuals must be performed. At this point, there are many different selection approaches, but we will use the generic roulette wheel selection as the probability of selection is higher related to fitness. After the selection, - with a certain probability and random factor- crossover and mutation operators are called. After crossover and mutation, the two new offspring are added to the population. This cycle

continues until the end condition is achieved.

Genetic algorithms can be used in various areas, such as search methods that look for a path to the goal, search for stored data and general search for solutions in any area. It can be used to evolve agents to play games like Prisoner's Dilemma or evolving sorting networks.

The paper by Parker and Edwards aims to replicate allopatric speciation using genetic algorithms. Allopatric speciation is one of the two speciation types where new species (has to) evolve from common ancestral species due to geographical changes.

For the simulation, a 100 x 100 grid environment is set up, which is made of discrete blocks that can be empty, have a small/medium/large seed (food), or an agent. Agents can freely move around and interact with the environment. The biological inheritance between the agents is shown using the agent color. At the same time, agents also vary in size, which also determines their food preferences. Each agent consists of two chromosomes, which decide the agent's actions (move, eat, reproduce, etc.) and the mating preferences of the agent, like the size, color, etc.

For the GA training, a medium-sized population was set. Following this, a random population was established in the grid, and after initialization and deployment, small/large agents were dead. After the medium agent population settled down in the middle of the grid, a barrier was introduced, and half of the grid started to spawn large seeds while the other half spawned small seeds. This resulted in medium agents evolving into small and large agents, respectively. After 1000 generations, the wall in between was removed. Large and small seeds were spread around in the grid which led to competition between the species as there was a rivalry for the food resources. As an alternative test, large and small seeds were spread separately, and it was observed that both populations survived.

I think that genetic algorithms are a very interesting concept as we can replicate the evolutionary steps to solve problems at different difficulty levels. The concept of the fitness function is very general, which allows us to use genetic algorithms in many areas, like creating a game agent, making robots learn certain tasks, etc. While it is considered that genetic algorithms are not the *trend* anymore, I still think that they have a huge potential as they can speed up the learning process for a problem while also showing emergent behavior. I think that there is still room for improvement in the selection of a fitness function, chromosome representations, etc.

XIX. ChatGPT

In this chapter, I will discuss;
What Is ChatGPT Doing... and Why Does It Work? [15]

This paper introduces the background of how Chat-GPT works. It depicts Chat-GPT as a machine that picks the *next best word* to follow a sequence of words. This selection is based on a list of probabilities generated by training the language model -Chat-GPT- on a training set. One of the concepts the paper explores the most is tokenization and how different token lengths can differ in text generation. The length of the token is explained by the concept of n-gram models. One of the main problems in this selection is that if we always select the most probable word, we can end up with the same chunk of text all the time, and our program might end up looping through the same set of words. The concept of *temperature* helps our model to not select the *most likely* word all the time by introducing a grain of randomness into our model.

The training process of Chat-GPT is interesting as we have to create a model that can generate a *human-like* text. The paper returns to convolutional neural networks and image recognition –specifically digit recognition. The first digit recognition systems utilized convolutions and neural networks to perform the classification. In order to have a neural network that utilizes convolutions to classify digit images, we have to train this network. This is where learning from examples comes into play. Our network can learn important features by adjusting the weights in the network and creating a general classifier. While our network can be small for a digit classifier, in certain cases, we might need a larger network. We can determine the size of our network just by common sense, and there is no way to come up with the perfect size for our network.

In the case of Chat-GPT, it performs unsupervised learning to adjust its weights. As it is basically a *text completion* machine, it starts with complete chunks of text and masks the last parts to guess the rest of that text piece. The matching amount between the generated text and the masked-out section determined the error rate of the Chat-GPT used in training.

Following this, the paper explores the input space of the Chat-GPT as we have to come up with a meaningful way to input the words into our network. The way the paper comes up is by assigning a unique number to each common word in English and measuring the distance between the words to find the *nearness* of the words. While this is certainly a way to set up an input model, our network is still large –175 billion weights. While the image classification networks use the basic convolution layers, Chat-GPT uses a transformer structure that has the ability to pay more attention to certain features of the input sequences than others. This way, our network does not have to go through all the input to detect a relationship but can select the important features. Another important part of this network is that it can pay attention to the previous inputs rather than only being limited to the current input. This is important as the network can catch the long-term relationships in long texts.

The training of Chat-GPT is done using the data from the web, which consists of at least several billion human-written pages consisting of trillion words of text, 5 million digitized books (another 100 billion words), text derived from speech in videos, etc. After the initial

training, Chat-GPT is used to self-train, which allows the engineers to use the initial training sample as a loss function and develop it. We can see that, after this training, Chat-GPT was able to discover the regularities in the language and kind of derived the laws of the language.

To be honest, after reading this paper, which deeply explains how Chat-GPT works, I was less impressed by how good Chat-GPT is as I know how it was trained and works. While it is still impressive, I do not think that the popular belief of "Chat-GPT being an exceptional invention" is far from an exaggeration. I do not think it is very simple or a gimmick, but it is basically a huge network trained by a huge sample of words and a sentence completion tool. The new version of Chat-GPT -like o1- is more than a sentence compilation tool. While I understand how this "choosing next word" situation works, I would like to know how the image/video generation models work behind the scenes as they work from a prompt. I think that one of the main caveats of Chat-GPT is that it cannot fact-check what it generates. While the new versions of the Chat-GPT can do web searches to verify information, the original model described in this paper never checks how true its completion is. Also, at the end of the day, we are limited to the training sample the model was trained on, which means that after a certain point -despite the temperature variable- most of the generations will look similar, and this is how most of the (allegedly) AI detectors claim to work.

XX. Final Thoughts on AI

I think this class was a joyful ride through the world of Artificial Intelligence. One of the things I noticed significantly was the current best methods that we use (for example, in Chat-GPT) are technically old. Neural networks started to appear in the mid-20th century, backpropagation in the 1980s, and deep learning in the 2010s. This does not mean that they are not good enough, but I think that there is a complete breakthrough that is waiting for us, which we have seen many times throughout this class. I feel like -after looking at what was behind us for a semester- the question to ask now is, where are we going next?

Our current AI methods and the best models heavily rely on huge data consumption. This gives us really good results, but it is costly, not easily adaptable, and inefficient regarding resources. The next step might be using symbolic representations of the data to generalize it and create hybrid models that can adapt to new domains without needing a large dataset. Similarly, we always aim for the **bigger** models with more layers, width, etc. Not only does this approach require a lot of resources, but they can only master a single (or similar) domain. There is no way to have a general intelligence that can rely less on the data. At the same time, rather than focusing on *learning everything*, we can also focus on models that will only keep the important information and learn how to **forget**. I think that this can be a breakthrough as it will save a lot of memory space in the models. I think that, at this point, we are expecting our model to develop common sense.

I think that one of the upcoming challenges of AI would be its verifiability and the ethical side as its usage becomes more and more common. Right now, the output system of the AI programs is a complete black box, even for the people who created them. One of the first steps that can be implemented is for AI to show its train of thought, verify its conclusions, etc. At the same time, the ethical part of AI will be another problem as there are nearly no regulations on AI right now. A hybrid model that uses the knowledge-based systems covered in this class can help us significantly with this problem.

Another area in which AI will emerge -in my opinion- is the multi-model structure. We currently have uni-modal AI platforms specializing in text, image generation, etc. I think a general -brain-like- model that integrates different modalities into its reasoning can be a breakthrough. I think that this can follow the integration of AI into robots.

I think that in the long term, we will see more brain-like structures that replicate how our brains work. While we have a concept called *attention*, I do not think that we still cannot figure out what is happening in our attention system in our brain. At the same time, quantum computing can also result in a breakthrough if we can figure out its caveats.

Another noticeable thing is humanity's exponential development in technology. I think that in the last 10 years, a lot has happened that we could never imagine before. This makes me think, what will we do in 2035 if we go this speed? It is scary that if we ever achieve *artificial general intelligence*, we would technically have a robot that can learn, think, and develop projects by itself.

Lastly, I think a lot more is waiting to happen in AI. I hope to live long and see many advancements, and I am excited to live in this era and witness all these advancements.

References

[1] P. Winston, *Artificial Intelligence*. Addison-Wesley, 1992.

[2] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*. Prentice Hall, 2010.

[3] P. E. Hart, N. J. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," *IEEE Transactions on Systems Science and Cybernetics*, 1968.

[4] M.-T. Shing and G. Parker, "Genetic algorithms for the development of real-time multi-heuristic search strategies," in *Proceedings of the 5th International Conference on Genetic Algorithms*, 1993.

[5] A. Kotok, "A chess playing program for the ibm 7090 computer," 1962.

[6] R. E. Fikes and N. J. Nilsson, "Strips: A new approach to the application of theorem proving to problem solving," *Artificial Intelligence*, pp. 189–208, 1971.

[7] S. L. Lytinen, "Conceptual dependency and its descendants," *Computers & Mathematics with Applications*, 1992.

[8] R. C. Schank and R. P. Abelson, "Scripts, plans, and knowledge," in *Proceedings of the 4th International Joint Conference on Artificial Intelligence - Volume 1*, Morgan Kaufmann Publishers Inc., 1975.

[9] A. Aamodt and E. Plaza, "Case-based reasoning: Foundational issues, methodological variations, and system approaches," *AI Commun.*, pp. 39–59, 1994.

[10] P. H. Winston, "Learning structural descriptions from examples," Tech. Rep., 1970.

[11] G. Parker and M. Khan, "Distributed neural network: Dynamic learning via backpropagation with hardware neurons using arduino chips," 2016, pp. 206–212.

[12] D. E. Rumelhart, J. L. McClelland, and C. PDP Research Group, Eds., *Parallel distributed processing: explorations in the microstructure of cognition, vol. 1: foundations*. MIT Press, 1986.

[13] G. B. Parker and T. B. Edwards, "Using a genetic algorithm to replicate allopatric speciation," in *2019 IEEE Congress on Evolutionary Computation (CEC)*, 2019.

[14] M. Mitchell, *An Introduction to Genetic Algorithms*. MIT Press, 1998.

[15] S. Wolfram, *What Is ChatGPT Doing... and Why Does It Work?* Stephen Wolfram Writings, 2023.