Winter in Data Science(WiDS)

AI For Logical Thinking

Abhinav V
24B1007

Mentor: Abhilasha Sharma Suman &
Harshitha Inampudi

January 30, 2026

Contents

1 Conjunctive Normal Form (CNF)

Conjunctive Normal Form (CNF) is a standard way of representing logical formulas in propositional and first-order logic. A formula is said to be in CNF if it is expressed as a conjunction of clauses, where each clause is a disjunction of literals. A literal is either an atomic proposition or its negation.

Formally, a CNF formula has the form:

$$(C_1) \land (C_2) \land \cdots \land (C_n)$$

where each clause C_i is of the form:

$$(l_1 \lor l_2 \lor \cdots \lor l_k)$$

CNF is widely used because many automated reasoning algorithms, such as resolution and SAT solvers, require input formulas to be in this form. Here $l_1, l_2, ...l_k$ are literals which takes value true or false.

1.1 Examples of CNF

The following formula is in CNF:

$$(A \lor \neg B) \land (C \lor D) \land (\neg E)$$

In contrast, the formula:

$$A \to (B \lor C)$$

is not in CNF and must be converted before applying CNF-based algorithm.

1.2 CNF Conversion

It is the process of conversion of a formula that is not in CNF to cnf without changing the meaning or the satisfiability of the formula. To convert a formula to cnf follow the following steps:

1.3 Steps in CNF Conversion

1. **Eliminate implications and equivalences:** Replace implications and biconditionals using logical equivalences.
$$A \to B \equiv \neg A \lor B$$
2. **Move negations inward:** Apply De Morgan's laws and eliminate double negations so that negations apply only to atomic formulas.
$$\neg(A \land B) \equiv (\neg A \lor \neg B)$$
3. **Standardize variables apart (for FOL):** Rename variables so that each quantifier uses a unique variable name.
4. **Skolemization (for FOL):** Eliminate existential quantifiers by introducing Skolem constants or functions.
5. **Drop universal quantifiers:** After Skolemization, all remaining variables are universally quantified and the quantifiers can be omitted.
6. **Distribute disjunction over conjunction:** Use distributive laws to obtain a conjunction of disjunctions.
$$A \lor (B \land C) \equiv (A \lor B) \land (A \lor C)$$

Here steps 3,4,5 are optional there are done to get the formula in skolemized normal form.

1.4 Example of CNF Conversion

Consider the formula:

$$A \to (B \land C)$$

First, eliminate the implication:

$$\neg A \lor (B \land C)$$

Next, distribute disjunction over conjunction:

$$(\neg A \lor B) \land (\neg A \lor C)$$

The resulting formula is in CNF.

2 DPLL Algorithm

The Davis–Putnam–Logemann–Loveland (DPLL) algorithm is a backtracking-based search algorithm used to determine the satisfiability of propositional logic formulas in Conjunctive Normal Form (CNF). It forms the basis of many modern SAT solvers and improves upon basic resolution methods by systematically exploring assignments to variables.

The DPLL algorithm attempts to find a truth assignment that satisfies all clauses in a CNF formula. It recursively assigns truth values to variables, simplifies the formula based on these assignments, and backtracks when a contradiction is encountered. If all clauses are satisfied, the formula is satisfiable; if all possible assignments lead to contradictions, the formula is unsatisfiable. We take a decision on a variable then iteratively assign variables that are in unit clause to satisfy that clause, In this process if we get a contradiction we make a learning clause by backtracking else we conclude sat with its assignment as the decision we have taken.

2.1 Unit Propagation

Unit propagation is a key optimization in the DPLL algorithm. A unit clause is a clause containing only one literal. In order for the formula to remain satisfiable, that literal must be assigned a value that satisfies the clause.

For example, given the clause:

$$(A)$$

the variable A must be assigned true. This assignment is propagated throughout the formula, simplifying or eliminating clauses that contain A or $\neg A$.

2.2 Pure Literal Elimination

A pure literal is a variable that appears only in one polarity (either always positive or always negative) throughout the formula. Such a literal can be assigned a value that satisfies all clauses in which it appears without affecting the satisfiability of the remaining formula.

For example, if a variable B appears only as B and never as $\neg B$, it can be safely assigned true and removed from the formula.

2.3 Backtracking Search

After applying unit propagation and pure literal elimination, the algorithm selects an unassigned variable and assigns it a truth value. The resulting formula is then recursively evaluated. If a contradiction occurs, the algorithm backtracks and tries the opposite truth value.

This systematic exploration of the search space ensures that all possible assignments are considered if necessary. If we encounter a conflict clause we backtrack find learning clause by disjunction of negation of decision we made. This learning clause is then added to set clauses and we continue to find sat for new set of clauses.

2.4 Termination and Correctness

The DPLL algorithm terminates when either a satisfying assignment is found or all possible assignments have been exhausted. It is sound, meaning any satisfying assignment it finds is correct, and complete, meaning it will find a satisfying assignment if one exists.

Although DPLL is a decision procedure for propositional logic, its effectiveness is enhanced by heuristics and optimizations, making it practical for solving large satisfiability problems.

3 Understanding First-Order Logic

First-Order Logic (FOL) is a formal system used to represent and reason about facts, relationships, and rules in a precise mathematical manner. During the course of WiDS, we studied First-Order Logic as a foundational tool for modeling problems in artificial intelligence, algorithms, and formal reasoning systems.

3.1 Predicates

Predicates are logical statements that express properties of objects or relationships among multiple objects. A predicate becomes either true or false when specific values are assigned to its variables. In WiDS, predicates were used to model constraints, conditions, and properties in problem formulations.

For example, a predicate `IsConnected(x, y)` can be used to represent whether two nodes in a graph are connected. Similarly, `Visited(v)` can represent whether a vertex has been explored in a graph traversal algorithm.

Predicates allow abstract reasoning by separating the structure of a condition from the specific values involved. This abstraction was particularly useful when reasoning about algorithm correctness and constraints that must hold throughout execution.

3.2 Functions

Functions in First-Order Logic map one or more objects to a single object. Unlike predicates, functions do not evaluate to true or false; instead, they return a value. Functions are often used to model computations, transitions, or relationships that produce a result.

For instance, a function `Parent(x)` may return the parent of a node in a tree, while `NextState(s, a)` may represent the state reached after taking action `a` from state `s`. In algorithmic contexts, such functions help formally describe state transitions and recursive structures.

3.3 Quantifiers

Quantifiers specify the scope of variables and express whether a statement applies universally or existentially. The two primary quantifiers studied were the universal quantifier ($\forall$) and the existential quantifier ($\exists$).

The universal quantifier is used to state that a property holds for all elements in a domain. For example:

$$\forall v \ (Visited(v) \rightarrow Processed(v))$$

This expresses that every visited node must also be processed.

The existential quantifier expresses that there exists at least one element in the domain for which a statement holds. For example:

$$\exists p \ Path(p, s, t)$$

indicates that there exists a path from source s to target t.

Quantifiers were crucial in expressing algorithm guarantees, such as correctness conditions and termination properties, and in formally specifying problem constraints.

3.4 Combined Use in Problem Modeling

A key learning outcome was understanding how predicates, functions, and quantifiers work together to model real-world and algorithmic problems. Complex statements can be built by combining these components, enabling expressive and precise formulations.

For example, the statement:

$$\forall v \, \exists u \, (Edge(v, u) \land Visited(u))$$

can be used to reason about graph connectivity and exploration behavior in traversal algorithms.

4 Robinson's Resolution Algorithm

Robinson's Resolution Algorithm is a fundamental inference procedure used in First-Order Logic for automated theorem proving. It is based on the principle of refutation, where the goal is to show that a given set of logical statements is inconsistent. If a contradiction can be derived, the original statement is proven to be logically valid.

4.1 Resolution Principle

The resolution principle operates on clauses, which are disjunctions of literals. A literal is either an atomic formula or its negation. Resolution works by identifying complementary literals in two clauses and combining the remaining literals to produce a new clause, called the resolvent.

In propositional logic, the resolution rule can be written as:

$$(A \lor B), (\neg A \lor C) \Rightarrow (B \lor C)$$

In First-Order Logic, resolution is extended by allowing variables and applying unification before resolving clauses. It can be seen that the resolvent clause derived is always the resultant derivation of the clause that we used to get the resolvent. Hence the new resolvent doesn't change the satisfiability of the formula that we check.

4.2 Clause Normal Form

Before applying Robinson's Resolution Algorithm, all formulas must be converted into Clause Normal Form (CNF). This involves several transformation steps:

- Eliminating implications and equivalences
- Moving negations inward using De Morgan's laws
- Standardizing variables apart
- Skolemizing existential quantifiers
- Dropping universal quantifiers
- Converting the formula into a conjunction of disjunctions

Each disjunction in the resulting conjunction represents a clause. The resultant formula would be in skolemized normal form containing only universal quantifiers and constants.

4.3 Unification

Unification is a key component of resolution in First-Order Logic. It is the process of finding a substitution for variables that makes two literals syntactically identical.

For example, the literals:

$$P(x, a) \quad \text{and} \quad P(b, y)$$

can be unified using the substitution:

$$\{x \leftarrow b,\ y \leftarrow a\}$$

The most general unifier (MGU) is preferred, as it introduces the least amount of constraint while allowing resolution to proceed. There are some restriction while doing unification they are:

- A constant can't be renamed to a variable whereas its either way correct.
- A function can't be renamed to a variable but its either way good.

4.4 Resolution Algorithm

Robinson's Resolution Algorithm proceeds as follows:

1. Convert all formulas into Clause Normal Form.
2. Negate the statement to be proved and add it to the set of clauses.
3. Repeatedly select pairs of clauses containing complementary literals.
4. Apply unification to make the literals identical.
5. Resolve the clauses to produce a new clause.
6. Add the resolvent to the clause set.
7. Continue until either the empty clause is derived or no new clauses can be generated.

The derivation of the empty clause indicates a contradiction, proving the original statement.

4.5 Example

Consider the following set of clauses:

1. $P(x) \vee Q(x)$
2. $\neg P(a)$

Resolving clause (1) and clause (2) using the substitution $\{x \leftarrow a\}$ yields:

$$Q(a)$$

If Q(a) is true then the formula given is SAT else it is UNSAT. If the clause $\neg Q(a)$ is also present, resolving $Q(a)$ and $\neg Q(a)$ produces the empty clause, indicating inconsistency.

4.6 Properties

Robinson's Resolution Algorithm is sound, meaning any conclusion derived using resolution is logically valid. It is also refutation-complete for First-Order Logic, which means that if a set of clauses is unsatisfiable, resolution is guaranteed to derive a contradiction.

Due to its completeness and systematic nature, Robinson's Resolution Algorithm forms the theoretical foundation of many automated reasoning and logic programming systems.

5 Unification and Most General Unifier

Unification is a fundamental operation in First-Order Logic that determines whether two logical expressions can be made identical by substituting variables with appropriate terms. It plays a central role in automated reasoning systems, particularly in resolution-based theorem proving.

5.1 Most General Unifier (MGU)

A Most General Unifier (MGU) is a unifier that imposes the least possible constraints on the variables. Any other unifier can be obtained by further substituting variables in the MGU.

For example, consider the expressions:

$$P(x, y) \quad \text{and} \quad P(a, b)$$

One unifier is:

$$\{x \leftarrow a,\ y \leftarrow b\}$$

This substitution is also the MGU, as it makes the expressions identical without unnecessary constraints.

In contrast, for the expressions:

$$P(x, z) \quad \text{and} \quad P(a, y)$$

the MGU is:

$$\{x \leftarrow a,\ y \leftarrow z\}$$

whereas the substitution

$$\{x \leftarrow a,\ y \leftarrow z\ z \leftarrow a\}$$

is not a MGU as it can be obtained by applying substitution to our MGU. An MGU is a substitution where any other substitution can be derived from MGU by applying some genral renaming or substitution. This substitution ensures both arguments of the predicate remain equal.

5.2 Occurs Check

The occurs check is an important condition in unification that prevents a variable from being substituted with a term that contains the variable itself. This avoids infinite or circular substitutions.

For example, attempting to unify:

$$x \quad \text{and} \quad f(x)$$

fails due to the occurs check, since substituting x with $f(x)$ would lead to an infinite term.

5.3 Unification Algorithm

The unification process typically follows these steps:

1. If both expressions are identical, unification succeeds.
2. If one expression is a variable, substitute it with the other expression, provided the occurs check passes.
3. If both expressions are functions with the same name and arity, recursively unify their arguments.
4. If none of the above cases apply, unification fails.

This algorithm systematically computes the Most General Unifier when one exists.

5.4 Role of MGU in Logical Inference

The Most General Unifier ensures that resolution and other inference rules operate in the most general way possible. By using MGUs, inference systems avoid unnecessary specialization and maintain completeness.

Unification with MGU enables logical systems to reason symbolically rather than relying on explicit enumeration of all possible cases, making it essential for efficient automated deduction in First-Order Logic.

6 Undecidability of First-Order Logic

First-Order Logic is undecidable because there is no algorithm that can determine, for every possible formula, whether the formula is logically valid. This undecidability arises from the expressive power of FOL, which allows quantification over infinite domains and the representation of complex structures such as arithmetic. Through such representations, FOL can encode problems equivalent to the Halting Problem, which is known to be undecidable.

As a result, while First-Order Logic is sound and complete in the sense that all valid statements can be proven using inference rules, there is no general decision procedure that always terminates. Automated reasoning systems based on FOL may run indefinitely when attempting to prove certain statements, highlighting the fundamental trade-off between expressiveness and decidability in logical systems.

7 Given-Clause Algorithm

The Given-Clause Algorithm is a control strategy for applying resolution and related inference rules in automated theorem proving. Instead of resolving all clauses against each other indiscriminately, the algorithm carefully selects one clause at a time, called the *given clause*, and resolves it with a selected set of previously processed clauses. This strategy significantly reduces redundant inferences and improves efficiency.

7.1 Basic Idea

The algorithm maintains two disjoint sets of clauses:

- **Usable set**: clauses that have already been selected as given clauses
- **Set of support (SOS)** or **unprocessed set**: clauses that are candidates to be selected as the next given clause

At each step, one clause is chosen from the unprocessed set as the given clause. This clause is moved to the usable set, and all possible inferences between the given clause and clauses in the usable set are generated. Any new clauses produced are added to the unprocessed set.

7.2 Algorithm Outline

The Given-Clause Algorithm proceeds as follows:

1. Initialize the usable set as empty.
2. Place initial clauses into the unprocessed set.
3. Select a clause from the unprocessed set as the given clause.
4. Move the given clause to the usable set.

5. Generate all resolvents between the given clause and clauses in the usable set.
6. Add newly generated clauses to the unprocessed set.
7. Repeat until the empty clause is derived or no clauses remain.

The derivation of the empty clause indicates that the original set of clauses is unsatisfiable.

7.3 Otter-Style Provers

Otter-style theorem provers use the Given-Clause Algorithm as their central inference control mechanism. In these systems, clause selection is guided by heuristics such as clause weight, age, or complexity. This allows the prover to prioritize simpler or more promising clauses, improving performance without sacrificing completeness.

Otter also incorporates simplification techniques such as subsumption, demodulation, and tautology deletion to eliminate redundant clauses during the search process.

7.4 Example

Consider the following set of clauses:

$$\begin{aligned} &1.\ P(x) \lor Q(x) \\ &2.\ \neg P(a) \\ &3.\ \neg Q(a) \end{aligned}$$

Initially, all clauses are placed in the unprocessed set. Suppose clause (1) is selected as the given clause and moved to the usable set. Resolving clause (1) with clause (2) using the substitution $\{x \leftarrow a\}$ yields:

$$Q(a)$$

The new clause $Q(a)$ is added to the unprocessed set. If $Q(a)$ is later selected as the given clause and resolved with clause (3), the empty clause is derived:

$$\square$$

indicating a contradiction.

7.5 Set of Support Strategy

A common refinement of the Given-Clause Algorithm is the set of support strategy. In this approach, the initial clauses are divided into axioms and a set of support, which typically contains the negation of the theorem to be proved. Resolution is restricted so that at least one parent clause comes from the set of support. This restriction preserves completeness while significantly reducing the search space.

7.6 Properties

The Given-Clause Algorithm is sound and refutation-complete when combined with resolution and unification. Its strength lies not in new inference rules but in its disciplined control of inference application. By resolving one clause at a time against an accumulated usable set, the algorithm provides a practical and efficient framework for automated deduction in first-order logic.

8 Set of Support (SOS) Strategy

The Set of Support (SOS) strategy is a resolution control strategy designed to make theorem proving more efficient by focusing inference on clauses that are directly relevant to the goal. It is based on the observation that contradictions typically arise from the interaction between the negation of the desired conclusion and the given premises, rather than from the premises alone.

8.1 Motivation

Resolution-based theorem proving suffers from a combinatorial explosion in the number of derived clauses. Blindly resolving all pairs of clauses quickly becomes infeasible, even for relatively small problems. The SOS strategy addresses this issue by restricting resolution steps so that at least one parent clause is connected to the negation of the theorem being proved.

Since the original premises are usually assumed to be consistent, resolving only among them cannot lead to a contradiction. Therefore, focusing inference on clauses derived from the negated goal significantly reduces the search space while still preserving the ability to find a refutation.

8.2 Definition of Set of Support

Given a set of clauses divided into:

- **Premise clauses**: clauses representing known facts or axioms
- **Denial clauses**: clauses obtained from negating the theorem to be proved

The set of support initially consists of the denial clauses. During resolution, any clause derived using at least one parent from the set of support is also added to the set of support. Resolution is restricted so that at least one parent clause in every inference step must belong to the set of support.

8.3 SOS Resolution Rule

Under the SOS strategy, a resolution step is permitted only if:

$$\text{at least one parent clause is in the set of support}$$

Clauses derived solely from premise clauses are disallowed. This ensures that all derived clauses remain connected to the negation of the goal.

8.4 Example

Consider the following clauses:

$$1.\ P(A)$$
$$2.\ \neg P(x) \lor Q(x)$$
$$3.\ \neg Q(A)$$

Clauses (1) and (2) are premises, while clause (3) is a denial clause and forms the initial set of support.

Resolving clause (3) with clause (2) yields:

$$\neg P(A)$$

which is added to the set of support. Resolving this clause with clause (1) produces the empty clause:

$$\square$$

establishing a contradiction. Resolution between clauses (1) and (2) alone would not be sufficient to derive the contradiction under the SOS restriction.

8.5 Completeness of SOS Strategy

The set of support strategy is refutation-complete provided that the set of premises outside the SOS is satisfiable. This condition is typically met in theorem-proving tasks, where premises represent consistent background knowledge and denial clauses represent the negation of the desired conclusion.

Intuitively, any refutation must involve at least one clause derived from the denial of the theorem. Therefore, restricting resolution to steps involving the set of support does not eliminate any essential refutations.

9 Subsumption and Redundancy Control in Theorem Proving

Redundancy control is essential in resolution-based theorem proving to prevent the uncontrolled growth of the clause set. Many generated clauses do not contribute meaningfully to a refutation and can be safely removed without affecting completeness. Subsumption is one of the most important techniques for identifying and eliminating such redundant clauses.

9.1 Subsumption

A clause C_1 is said to subsume another clause C_2 if there exists a substitution θ such that:

$$C_1\theta \subseteq C_2$$

That is, every literal in C_1, after applying the substitution, appears in C_2. Intuitively, a subsuming clause is more general and imposes stronger constraints than the subsumed clause.

9.1.1 Examples of Subsumption

- $P(A)$ subsumes $P(A) \vee Q(B)$
- $P(x)$ subsumes $P(A)$
- $P(x) \vee Q(x)$ subsumes $P(A) \vee Q(A) \vee R(B)$

In each case, the subsumed clause does not provide any additional information beyond what is already expressed by the subsuming clause.

9.2 Redundancy Elimination

Redundant clauses are those whose presence does not increase the deductive power of the clause set. Eliminating such clauses reduces the size of the search space and improves efficiency.

Common sources of redundancy include:

- Clauses subsumed by other clauses
- Tautological clauses
- Duplicate clauses

Redundancy elimination is typically applied continuously during the proof search rather than only as a preprocessing step.

9.3 Subsumption Resolution

Subsumption resolution is a refinement of the resolution rule that avoids generating resolvents that are immediately subsumed by existing clauses. Before adding a newly derived resolvent to the clause set, the prover checks whether it is subsumed by an existing clause. If so, the resolvent is discarded.

Conversely, if a newly derived clause subsumes one or more existing clauses, those clauses can be removed. This bidirectional subsumption checking helps maintain a compact and informative clause database.

9.4 Clause Simplification Techniques

In addition to subsumption, several other clause simplification techniques are used to control redundancy:

9.4.1 Tautology Elimination

A clause is a tautology if it contains a literal and its negation. Such clauses are always true and cannot contribute to a refutation. For example:

$$P(x) \vee \neg P(x) \vee Q(y)$$

Tautological clauses can be safely removed at any stage of the proof.

9.4.2 Pure Literal Elimination

A literal is pure if its negation does not appear in any clause. Clauses containing pure literals can be removed, since the pure literal can always be assigned a truth value that satisfies those clauses without affecting the rest of the clause set.

9.4.3 Unit Clause Simplification

Unit clauses, which contain a single literal, can be used to simplify other clauses by resolving away complementary literals. This process reduces clause length and often leads more quickly toward the empty clause.

9.5 Importance of Redundancy Control

Effective redundancy control is crucial for practical theorem proving. While resolution is complete, unrestricted application leads to exponential growth in the number of clauses. Techniques such as subsumption, simplification, and controlled resolution preserve completeness while significantly improving performance.

Modern automated theorem provers rely heavily on these methods to manage large clause sets and make automated reasoning feasible in practice.

10 Search Algorithms

Search algorithms are used when we want to search for an target like a node from a set of search space like a graph. A search problem is defined by a set of states, an initial state, a set of actions (successor function), a transition model describing the result of actions, and a goal condition. A solution is a sequence of actions that transforms the initial state into a goal state.

Search problems are usually modeled as graphs or trees, where each node represents a state and each edge represents an action that moves between states.

10.1 Connection to Machine Learning and Heuristics

Search is a core idea in artificial intelligence and is closely linked to machine learning and optimization. Many learning problems can be thought of as search tasks, where the algorithm explores possible hypotheses, parameter values, or policies. Heuristics help by estimating how close a given state is to the goal, allowing the search to concentrate on the most promising areas. In modern AI systems, these heuristics may be carefully designed by humans, learned from data, or improved over time through experience, effectively connecting traditional search techniques with learning-based approaches.

10.2 Uninformed Search

Uninformed (or blind) search algorithms explore the search space using only the basic information given in the problem. They do not rely on heuristics or any domain-specific knowledge to guide them, which means they often take more time to find the exact target.

10.2.1 Breadth-First Search (BFS)

Breadth-First Search (BFS) explores nodes level by level, checking all states that are one step away before moving on to deeper levels. If a solution exists and the branching factor is finite, BFS is guaranteed to find it. It is also optimal when all actions have the same cost, because it always reaches the shallowest goal state first.

The main drawback of BFS is its high memory usage. Since it must store all the frontier nodes at each level, memory often becomes the limiting factor long before time does. Even so, BFS is very useful for finding the shortest path between two nodes in a graph, as it systematically explores the graph one level at a time.

10.2.2 Depth-First Search (DFS)

Depth-First Search (DFS) works by always expanding the deepest unvisited node first. Instead of exploring all possible options at once, it follows a single path as far as it can go before reaching a dead end. When that happens, the algorithm backtracks and tries a different path. In this sense, DFS follows a hit-and-trial approach: it keeps moving forward until no further progress is possible, then retraces its steps to explore alternative routes.

One of the main advantages of DFS is its low memory usage. Unlike Breadth-First Search, which must store all nodes at a given depth, DFS only needs to keep track of the current path and the unexplored sibling nodes, making it much more space-efficient.

However, DFS has important limitations. It does not guarantee an optimal solution, since the first solution it finds may not be the shortest or least costly. Additionally, DFS may fail to terminate if the search space contains infinite paths or cycles.

10.2.3 Uniform Cost Search (UCS)

Uniform Cost Search expands the node with the lowest cumulative path cost. It generalizes BFS to handle non-uniform action costs. UCS is complete and optimal as long as all step costs are strictly positive.

Unlike BFS, UCS explores paths in order of increasing cost rather than depth. This ensures that the first solution found is guaranteed to be the least-cost solution, but it may require exploring many low-cost paths before reaching the goal.

10.3 Informed Search

Informed search algorithms rely on heuristics to guess how far a given state is from the goal. By using this extra guidance, the search focuses on the most promising paths first, which often makes it much faster and more efficient.

10.3.1 Greedy Best-First Search

Greedy Best-First Search always expands the node that looks closest to the goal based on the heuristic. This often makes it run quickly, but the downside is that it doesn't guarantee finding a solution—or the best one—because it completely ignores the cost of the path taken so far and can be thrown off by a misleading heuristic.

10.3.2 A* Search

A* search combines the advantages of uniform-cost and greedy search using the evaluation function

$$f(n) = g(n) + h(n),$$

where $g(n)$ is the cost from the initial state to node n, and $h(n)$ is a heuristic estimate of the remaining cost to reach a goal.

If the heuristic is admissible (never overestimates the true cost), A* tree search is optimal. With consistent heuristics, A* graph search is also optimal. A* is a foundational algorithm that illustrates how heuristic guidance can preserve optimality while dramatically reducing the number of explored states. In finding the path that can be reached in shortest time via two nodes, we use heuristics like euclidean distance. This minimises the time that is actually required which is not in dijsktra.

References

1. Huth, M., & Ryan, M. (2004). *Logic in Computer Science: Modelling and Reasoning about Systems*. Cambridge University Press. Chapters 1–3.

2. Russell, S., & Norvig, P. (2021). *Artificial Intelligence: A Modern Approach* (4th ed.). Pearson. Chapters 7–9.

3. Newell, A., & Simon, H. A. (1956). *The Logic Theorist: A Machine That Proves Theorems*. IRE Transactions on Information Theory.

4. Wikipedia. (2024). *Logic Theorist*. Retrieved from https://en.wikipedia.org/wiki/Logic_Theorist

5. Otter Manual. (n.d.). *Otter 3.3 Reference Manual*. Retrieved from http://www.lcc.uma.es/~eva/asignaturas/lic/apuntes/otter3_manual.pdf

6. Schubert, L. (2023). *CSC 244/444 Lecture Notes: Resolution Strategies and Set of Support*. Retrieved from https://cs.rochester.edu/u/schubert/444/lecture-notes/lecture09.pdf

7. Sprinzl, C. (2021). *Unification in First-Order Logic*. Retrieved from https://www.cip.ifi.lmu.de/~sprinz/sprinz_2021_unification.pdf

8. IIT Bombay. (2025). *Lecture Slides: CS748 / CS747 Topics*. Retrieved from https://www.cse.iitb.ac.in/~shivaram/teaching/cs747-a2025/lectures/cs748-s2021-w02-l01.pdf

9. Lecture Notes (Google Drive). (n.d.). *Additional Lecture Material*. Retrieved from https://drive.google.com/file/d/1gu9JxmBPF1CwsJhLtiClgfwHBOZovHdf/view?usp=drivesdk

10. YouTube. (n.d.). *Lecture on Automated Reasoning / Logic*. Retrieved from https://youtu.be/WbzNRTTrX0g?si=T_eixvSYbRhAcqNZ