
Symbolic AI vs. Large Language Models

A comparative study on game theory and optimization

Lorenzo Blanco, 715351,
1.blanco2@studenti.unipi.it

Luca Gallone, 715429,
1.gallone@studenti.unipi.it

With this project we aim to implement an intelligent agent designed to address two different types of problems: adversarial game strategy in Connect Four and combinatorial optimization in the Knapsack problem. First, we implement from scratch symbolic AI techniques, using different approaches for each problem. Finally, we compare their performance and reasoning capabilities against the latest Large Language Models.

1 Problems Description

1.1 Connect Four

Connect Four is a 2-player game played on a 6-row by 7-column grid, initially empty. On each turn, a player chooses a non-full column and drops a token, which occupies the lowest available cell in that column. The game ends when a player aligns four of their tokens consecutively, either horizontally, vertically or diagonally. If the grid fills up without any player forming a line of four, the game ends in a draw. The objective of each player is to maximize the chances of achieving a line of four while simultaneously preventing the opponent from doing so.

1.2 Knapsack problem

The knapsack problem is a classical combinatorial optimization problem in which, given a set of n items, each one characterized by a weight $w_i \in \mathbb{R}^+$ and a value $v_i \in \mathbb{R}^+$, and given a knapsack with a maximum load capacity $W \in \mathbb{R}^+$, the objective is to determine the subset of items to insert into the knapsack such that the total accumulated value is maximized, strictly respecting the imposed capacity constraint. Formally, the problem can be modelled as an integer linear programming problem:

$$\text{Maximize } Z = \sum_{i=1}^n v_i x_i \quad (1)$$

Subject to constraint:

$$\sum_{i=1}^n w_i x_i \leq W \quad (2)$$

Where x_i represents the binary decision variable: $x_i = 1$ if the i -th item is selected, 0 otherwise.

The 0/1 Knapsack Problem is classified as NP-Hard (Non-deterministic Polynomial-time Hard). This classification implies significant challenges from a computational perspective, especially as the instance size grows: the intrinsic difficulty of the problem lies in the size of the admissible solution space. Since each item has two possible states (taken or left), for n items there are 2^n possible combinations.

2 Experimental Setup

The source code of the following from-scratch implementations is available at:

<https://github.com/blanco003/AIF>

2.1 Connect Four

This work considers two complementary approaches to decision-making in adversarial games: Minimax, which relies on systematic search, and Monte Carlo Tree Search, which relies on stochastic simulations. We implemented the game logic ourselves, allowing us to fully control the internal representation of the board and adapt it precisely to the used algorithms.

2.1.1 Minimax

Connect Four is a deterministic, zero-sum and fully observable game. These properties allow the Minimax algorithm to explore the state space consistently, identifying optimal decisions. The possible sequences of alternating moves can be represented as a game tree, where each level corresponds to a player's turn and each branch represents a legal action.

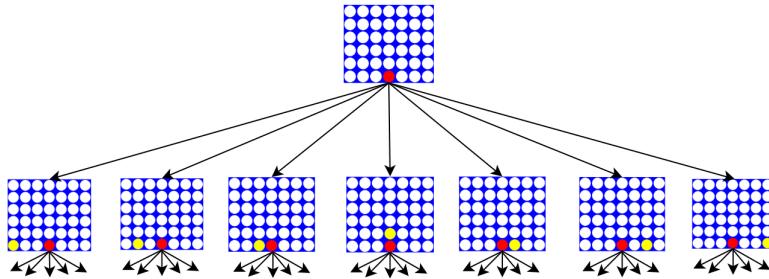


Figure 1: Connect Four Game Tree

Minimax determines the optimal move assuming that both players play optimally: *MAX* chooses actions that maximize backup utility while *MIN* chooses actions that minimize it, producing a contingent strategy that guarantees the best possible outcome for *MAX*; this is achieved by recursively evaluating terminal states, returning $+\infty$ for an AI win, $-\infty$ for an opponent win, or 0 for a full board (tie), and propagating these values along the game tree.

The Minimax algorithm performs a depth-first search of the game tree. Its time complexity is $O(b^m) = O(7^{42})$, where $b = 7$ is the branching factor, which is the number of legal moves at each turn, and $m = 42$ (6×7) is the maximum depth of the tree. The exponential time complexity makes Minimax impractical for games with a such large search space, as it is infeasible to explore all the possible ones.

$\alpha - \beta$ Pruning We can significantly reduce the exponential complexity of game tree search by computing the correct Minimax decision without examining every single state. This is achieved by pruning parts of the tree that cannot influence the final result.

During the search the algorithm keeps track of two additional parameters α (best value, which is the highest score, found so far along the path for the maximizing player) and β (best value, which is the lowest score, found so far along the path for the minimizing player), updating them and pruning the remaining branches of a node (terminating further recursive calls) as soon as it becomes clear that the current node cannot produce a better value than the existing bound: α for Max nodes and β for Min nodes.

With this technique the algorithm examines only $\approx O(7^{42/2})$ nodes instead of $O(7^{42})$ for standard Minimax. This means the effective branching factor becomes $\sqrt{7}$ rather than 7, resulting in a substantial reduction in computation.

Depth Limited Since a full game can last up to 42 plies, even if $\alpha - \beta$ pruning reduces the number of nodes visited, it cannot fully eliminate the exponential blow-up. For this reason, we introduced a depth limit d , stopping the search before reaching terminal states and relying on a heuristic evaluation instead. The higher d , the more future moves the algorithm considers. This generally improves decision quality, but it also increases the computational cost, as both the number of explored nodes and the required time grow exponentially. A comparison between the depth limited search with Minimax and Minimax with $\alpha - \beta$ pruning can be seen looking at the Table 5.

Heuristic Evaluation With depth-limited search, if the status of the game in which the search ends is not a terminal state, assigning a score of 0 is not very useful to discriminate between good and bad moves. To overcome this, we defined a heuristic function, which assigns a score to any given board state from the point of view of a specific player. The goal is to estimate how favorable the position is for that player, so the Minimax algorithm can make informed decisions. The score of the heuristic combines:

- **Positional Weighting:** Each column of the board is assigned to a weight that reflects its strategic value: [40, 70, 120, 200, 120, 70, 40]. The idea is that controlling the center increases opportunities to form 4 in a row in multiple directions. Each piece on the board contributes to the score according to its column. If the piece is of the current player the score is added, otherwise is subtracted.
- **Pattern-based evaluation:** The board is scanned for every possible set of 4 consecutive cells (windows) in horizontal, vertical, and diagonal directions. Each window is evaluated using the following rules:
 - **4 in a row** (base case):
 - * 4 token of the current player $\rightarrow +\infty$
 - * 4 token of the opponent player $\rightarrow -\infty$
 - **3 in a row with 1 empty cell** \rightarrow count the number of **open ends** (empty cells at the start and end of the window); from the current player point of view:
 - * 2 open ends \rightarrow double threate \rightarrow inevitable win $\rightarrow +\infty$
 - * 1 open end \rightarrow high chance to win $\rightarrow +900000$
 - * Same for the opponent player, with negative scores.
 - **2 in a row with 2 empty cell**: \rightarrow count the number of open ends; from the current player point of view:
 - * 2 open ends \rightarrow good chance to win $\rightarrow +50000$
 - * 1 open end \rightarrow check the **extendability** (how many additional empty cells are available in that direction) and assign a score proportional to it (5 $\rightarrow +40000$, 4 $\rightarrow +30000$, 3 $\rightarrow +20000$, 2 $\rightarrow +10000$, 0 $\rightarrow 0$)
 - * Same for the opponent player, with negative scores

The evaluation function assigns both positive and negative contributions to the board score: positive contributions reward moves that increase the player's chances of winning, while negative ones penalize moves that allow the opponent to create immediate threats. Without considering the opponent's threats, the heuristic could choose visually appealing moves that ignore imminent victories by the opponent. To prevent this, the heuristic is defined as the sum of the player advantage minus the opponent threats. The goal of the algorithm is to maximize this total score.

The heuristic evaluates the current board state as a whole, before Minimax propagates scores upward. The value that ultimately determines the chosen move is the score returned to the root node, after all possible counter moves have been considered.

2.1.2 Monte Carlo Search Tree

Monte Carlo Tree Search estimates the value of moves through simulations rather than exhaustively exploring the entire state space, which is too large to fully search in Connect Four. By balancing the exploration of rarely tried actions with the exploitation of moves that have performed well in previous rollouts, MCTS efficiently focuses the search on promising regions of the game tree.

We modelled the game as a tree of nodes, where each **node** represents a game state and stores key properties such as the move that led to it, the player who made that move, the number of visits, the

number of wins accumulated during simulations, a reference to its parent node, and a dictionary of child nodes corresponding to possible next moves. The algorithm keeps a search tree and grows it on each iteration of the following steps:

- **Selection:** starting from the root, the algorithm traverses the tree by choosing at each step the child node that maximize a selection policy, until a non fully expanded or terminal node is reached. As **selection policy**, we used the **UCT** (upper confidence bounds applied to trees), which ranks each possible move based on **UCB1** upper confidence bound formula:

$$UCB1 = \frac{\text{wins}}{\text{visits}} + C \times \sqrt{\frac{\log(\text{visits of parent node})}{\text{visits}}} \quad (3)$$

where C is a constant the balances exploitation and exploration, which we empirically fixed to $\sqrt{2}$.

- **Expansion:** adds new child nodes based on the available moves.
- **Rollout:** simulates a random game from the current state to the end and calculates the outcome, either a win/loss for the current player or a tie.
- **Back-propagation:** updates the statistics (wins, visits) along the path traversed.

These steps are repeated iteratively. We implemented two types of stopping criteria: a **time limit** (s), or the **number of rollout** to be executed (which is however less intuitive to fix a priori).

We also implemented a **best move selection** method, which chooses the action to take by selecting the child of the root game tree with the highest number of visits, and, in case of ties, the one with the best estimated win rate. This is related to the idea that a node with 65/100 wins is better than one with 2/3 wins, because the latter has a lot of uncertainty. Moreover, the **UCB1** formula ensures that the node with the most playouts is almost always the node with the highest win percentage, because the selection process favors win percentage more and more as the number of playouts goes up.

After each move in the actual game, either from the ai player or the opponent, we implemented a method to **update the root** of the game tree, moving to the corresponding already explored node or creating a new one if necessary. This keeps the tree synchronized with the game without having to start from a new tree from scratch at every move.

The time to compute a single playout is linear in the depth of the game tree, which in the worst case is $d = 42$, because only one move is taken at each chance point. So the time complexity is directly proportional to the number of rollouts N to be executed, or to the allowed computation time t .

The trade-off between the time limit and the number of rollouts can be seen looking at Table 6.

However, in practice, the number of nodes actually explored by the algorithm is much smaller than the about 4.5 trillions¹ possible states of Connect Four, because the algorithm only visits the most promising game configurations.

2.2 Knapsack problem

To address this problem, we selected an existing online dataset and tried two different approaches: a heuristic one, A^* (Branch & Bound), and a meta-heuristic approach, the Genetic Algorithm.

The data is structured as a list of tuples, where each tuple represents a single item characterized by a specific pair of (weight, value). Along with the items, are also provided the knapsack capacity and the optimal known solution. Since the chosen dataset has $n = 500$ items and each one can be taken or not, the search space amounts to 2^{500} possible solutions, a magnitude that makes any exhaustive search approach prohibitive.

2.2.1 Branch & Bound

Usually, A^* algorithm is used to find the best path in a graph, where the optimal path is defined as the one that consumes the least amount of resources and guarantees a solution. Exploiting this concept, we can apply this perspective to solve the **Knapsack Problem**, with the following distinctions:

¹Grid of 7×6 cells, where each one can be in 3 states, ("X", "O", empty) $\rightarrow 3^{42} \approx 10^{20}$ possible states; without considering illegal states the number "shrinks" to about 4.5×10^{12} (source: <https://oeis.org/A212693>)

1. We do not search for the path of minimal cost, but rather for the one of **maximum value**. This requires inverting the operation of the priority queue, which forms the basis of the algorithm.
2. The Knapsack Problem is a **combinatorial optimization** problem, whereas the classical A^* algorithm is typically used for pathfinding. Consequently, the problem domain is not represented by a 2D grid, but by a **decision tree**, where each node corresponds to a branching decision (include or exclude an item).

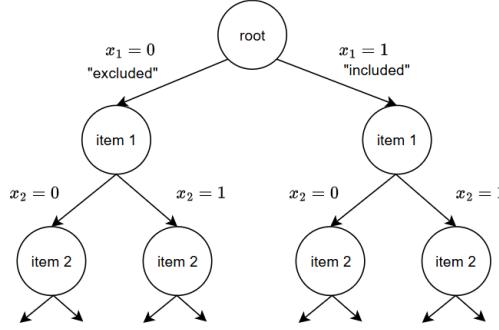


Figure 2: Knapsack problem Decision Tree

With this structure, A^* naturally turns into an informed **Branch & Bound** approach, where the branch corresponds to exploring the decision tree by choosing whether to include or exclude each item, and the bound is provided by the optimistic heuristic derived from a relaxed problem.

The algorithm applies the formula $f(n) = g(n) + h(n)$. A^* serves to explore this tree intelligently without having to check every single branch. It does this by assigning a promising score, f , to every node.

- g (Actual Value): In our case, this is not a cost but a **gain**, which is the actual value we have accumulated so far in the knapsack by following a certain path.
- h (Heuristic Estimate): This is the crucial part. h is an **optimistic estimate** of the maximum value we can still obtain with the remaining space and remaining items. In the Branch and Bound approach, this is the bound used to decide whether a branch can still lead to an optimal solution.
- f (Total Estimate): $f = g + h$. Represents the maximum potential value we could ever reach if we continue to follow this path.

The algorithm uses a priority queue to keep track of all the possible paths and, at each step, extracts and explores the one with the **highest f** (the most promising one).

Fractional Knapsack (The Heuristic) It is a "relaxed" problem where the rules are simpler. E.g. if we have 3kg of space and a 5kg item, we can "cheat" and say that we take 3/5 (60%) of the item and so 60% of its value.

To get the max value from a fractional knapsack:

- Calculate the value per weight ratio, $\frac{v_i}{w_i}$, for every remaining item.
- Sort items from highest ratio to lowest.
- Fill the remaining space taking 100% of the best items until there is one that doesn't fit.
- Take a fraction of this last item to fill the knapsack completely.

Since the heuristic is allowed to "cheat" by taking fractions, the value it calculates will always be greater than or equal to what we could obtain, because there is the constraint of rigid 0/1 rules.

This fractional relaxation provides an admissible upper bound for the 0/1 Knapsack problem, allowing the algorithm to estimate the maximum reachable value from a given node. This bound is what enables effective pruning in the Branch & Bound approach.

We modelled each **node** of the decision tree to contains the current level (index of the item we are considering), the total weight and value accumulated so far, the path (list of decisions made) and the f cost (which is computed as the sum of g (the total value accumulated) and h (the fractional knapsack heuristic)).

Starting from an empty root, in which f is given only by h ($g = 0$), the algorithm creates 2 child nodes, considering taking or not the first item and updating the corresponding values. If these nodes are still promising, which means f is greater than the best value found so far, they are added to the priority queue. Then, it extract the first node in the queue (the one with highest f) and do the same process until there are nodes to expand.

Pruning If f is already **worse** than the best solution found so far, A^* knows it is impossible for that path to lead to a better solution, so the entire branch can be safely pruned, saving an massive amount of work.

2.3 Genetic Algorithm

Genetic Algorithm does not attempt to build a single perfect solution step-by-step, like A^* . Instead, it simulates an evolutionary process on an entire population of possible solutions.

We **encoded** each solution as a binary vector where each bit indicates whether an item is taken "1" or not "0". Such a vector (e.g., $[1, 0, 0, 1, \dots]$) is also called "**individual**" or "**chromosome**". During initialization each bit is set to 1 with a certain small probability (probability to take).

Fitness Function One of the core concepts of this algorithm is the **fitness function**, which acts as the “environment” deciding survival. This function assigns a score to each individual based on a specific logic. First, it performs **validation**: it calculates the total weight of the solution. If this weight exceeds the knapsack’s capacity, the solution is deemed invalid and assigned a fitness score of 0. This effectively kills off the individual, ensuring its genes are not propagated. Otherwise, if the weight is valid, the function proceeds to **evaluation** by calculating the total value of the items included. This value becomes the individual’s fitness score, creating strong **selection pressure** that favors valid, high-value solutions.

Crossover and Mutation The evolution toward better solutions relies on two primary mechanisms to create offsprings. The first is **Crossover**, which mimics sexual reproduction and serves as the engine of **exploitation**. We implemented a **1-point crossover** with a **random cut off** position along the chromosome. For example, if Parent 1 has a good combination at the start ($[1, 1, 0 \dots]$) and Parent 2 has a good one at the end ($\dots 0, 1, 1]$), their child might inherit both segments ($[1, 1, 0 \dots 0, 1, 1]$), combining the successes of existing solutions. Moreover, crossover is applied with a probability (**crossover rate**) because always mixing parents might break good solutions, whereas occasionally copying parents unchanged helps preserve good genetic structures.

The second mechanism is **Mutation**, a random event that powers **exploration**. We implemented Mutation as a simply **binary bit-flip**: for every gene in a child’s chromosome, there is a tiny probability (**mutation rate**) that a 1 will flip to 0 or vice versa. This introduces new genetic random changes that let the algorithm explore new possibilities and prevents the algorithm from becoming “lazy” or getting stuck in a local optimum where everyone looks the same.

Parent Selection Parent selection determines which individuals get to reproduce, guiding the genetic algorithm toward better solutions while preserving enough diversity. We implemented 2 types of Parent Selection:

- **Roulette Wheel Selection**: a probabilistic method that calculates the total fitness of the population and conceptually maps it to a roulette wheel, where each individual gets a slice proportional to their fitness. The algorithm then “spins the wheel” by generating a random number to select a parent. This method strikes a perfect balance between exploitation and exploration; it gives high-fitness individuals a much higher chance of selection without guaranteeing it, while still offering valid but “mediocre” individuals a small chance to survive.

- **Tournament selection:** a method that simulates a series of small, local competitions rather than a global lottery based on total fitness. To select a parent, the algorithm randomly picks a small subset of k individuals from the general population, which is the tournament size, and compares their fitness scores. The individual with the highest score within this specific random group is declared the winner and selected as a parent.

Additionally, we also included **elitism**, ensuring that the best individual from the current generation is always carried over to the next.

The process of the Genetic Algorithm through new generations can be summarized by the following diagram:

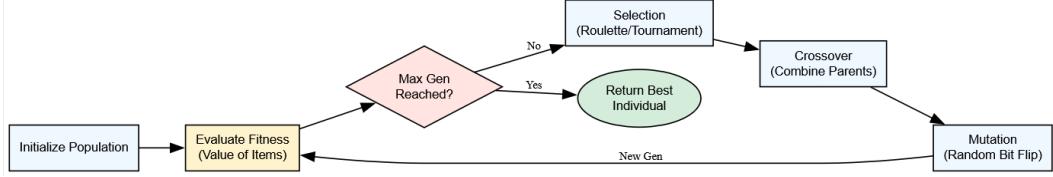


Figure 3: Genetic Algorithm

3 Results Symbolic AI

RQ 1: Between Minimax and Monte Carlo Tree Search, which algorithm achieves a higher win rate in the game Connect 4 under the same computational time?

Considering the same computational time, we have chosen max depth $d = 6$ for Minimax ($\alpha - \beta$ Pruning + Depth limited + Heuristic) as the best trade off between average reasoning time (≈ 10 s across all the game) and move quality, and therefore we have considered Monte Carlo Tree Search with time limit $t = 10$ s.

Under the same average computational budget, Minimax consistently defeated MCTS, showing stronger decision making. We also tried to increase t of MCTS to check if with more time, and so more simulations, it would be able to challenge Minimax. As the following table shows, still Minimax always beats MCTS, although giving it more time t it is able to push the game further (more plies) before losing.

We evaluated each configuration over 10 games, alternating which player moved first:

Metric	$t = 1$	$t = 5$	$t = 10$	$t = 30$	$t = 60$	$t = 180$	$t = 240$
Minimax winrate (%)	100	100	100	100	100	100	100
avg num. plies	12.3	15.5	20.3	21.10	22.7	23.1	25

Table 1: Minimax depth 6 vs. different time limit t of Monte Carlo Tree Search

Minimax consistently outperformed Monte Carlo Tree Search likely because the heuristic evaluation provides strong domain knowledge while MCTS only relies on many simulations. This means that Minimax even with limited depth can make highly informed decisions, whereas MCTS without domain knowledge need a lot of simulations, and therefore a lot of computational time. However, it becomes unfeasible for a real game to wait such a long time for only a single ply.

RQ 2: How do the informed search approach of A* algorithm and the population-based approach of the Genetic Algorithm compare in terms of solution quality for the 0/1 Knapsack problem?

Knapsack problem data instance:

- **Problem Dimension (n):** 500 total items.

- **Knapsack Capacity (W):** 2517 weight units.
- **Target (Global Optimum):** The maximum obtainable value is known and is equal to 7117.

Experimental Configuration of the Genetic Algorithm First of all, after some manual attempts, we empirically decided to fix probability to take to 0.015 (1.5%), doing a random sparse initialization, since items weight are high while the max capacity of the knapsack is low, starting with too many items in the knapsack would immediately invalidate every solution and the algorithm would be stuck. Then, to find the optimal setup we performed a **Grid Search** to test different values of the following hyper-parameters: **crossover rate** (0.6, 0.9), **mutation rate** (0.001, 0.01), **selection method** ("Roulette", "Tournament"), **tournament k** (if selection method = "Tournament") (3, 5, 10). We ran each configuration for 600 generations, and then we averaged over 3 repetitions, to ensure statistical reliability.

Prob. to take	Cross. rate	Mut. rate	Pop size	Sel. method	Gen.
0.015	0.6	0.001	250	Roulette	600

Table 2: Best configuration of hyper-parameters of the Genetic Algorithm

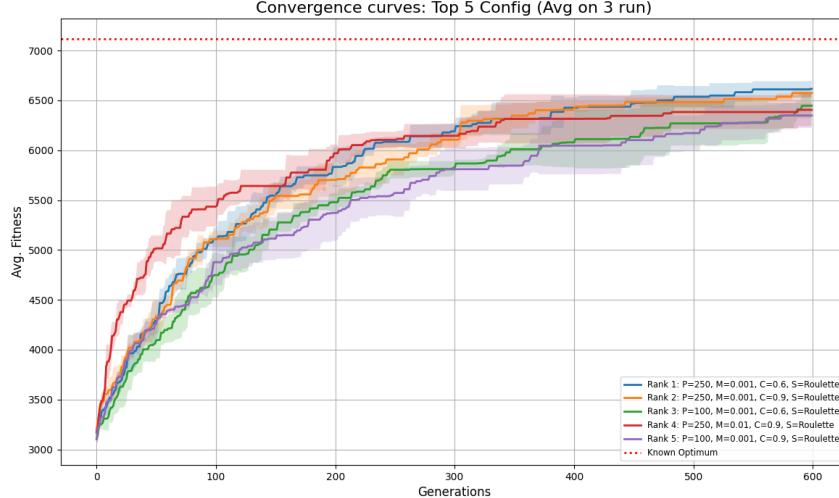


Figure 4: Genetic Algorithm Top 5 best configurations convergence curves

Unlike the Genetic Algorithm, which has a stochastic nature, A* (Branch & Bound) is deterministic and consistently reached the global optimum in every single run. Furthermore, the effective pruning of the search space allows A* to solve the problem efficiently, with an avg. runtime of 7 seconds, faster than the avg. 20 seconds required by the Genetic Algorithm.

We can conclude that the Genetic Algorithm still achieved competitive results despite being a generic meta-heuristic, demonstrating that problem agnostic approaches can efficiently approximate solutions without the domain-specific tailoring (the admissible heuristic function) required by A*.

4 Symbolic AI vs. Large Language Models

In this section we compare the performance of the implemented symbolic AI techniques with the latest Large Language Models, addressing the same tasks. To enable automatic and reproducible interaction with the LLMs, we developed a dedicated script that queries the LLMs using carefully designed **prompts** (Connect Four: Table 7; Knapsack problem: Table 8), with dynamic placeholders to include, respectively, the current game state in Connect Four, and the problem instance data for the Knapsack problem.

To standardize access to different LLM providers, we relied on **OpenRouter** as the unified interface through which all model calls were issued. In particular, we selected one of the latest reasoning models from each of the leading companies: **o4 mini high** from OpenAI, **Claude 3.7 thinking** from Anthropic and **Gemini 3 pro** from Google.

RQ 3: How does Minimax, the winner among the symbolic AI approaches, compare to the latest Large Language Models in terms of win rate and efficiency?

We let Minimax ($d = 6$) face each LLM over 5 games, alternating which player moved first:

Metric	o4 mini high	Claude 3.7 thinking	Gemini 3 pro
Minimax winrate (%)	100	100	80
avg move time (s)	74.41	74.86	104.46
avg move cost (\$)	0.034	0.106	0.117
avg total cost (\$)	0.30	1.04	1.70
avg num. plies	18.3	16.6	25
invalid moves	0	0	0

Table 3: Minmax vs. LLMs (5 games)

The results demonstrate a clear strategic gap between the symbolic AI approach and the current LLMs. In fact, Minimax achieved a dominant performance (100 % win rate) against both o4 mini high and Claude 3.7 thinking. In terms of computational efficiency, while Minimax ($d = 6$) required an avg. of only 10 seconds to compute the optimal move, the LLMs were significantly slower.

Interestingly, Gemini 3 Pro was the only LLM capable of challenging Minimax, and it managed to win a close game (38 plies, almost a full board!), despite being the most expensive, the one that reasoned the longest and the latest to be released.

RQ 4: How do symbolic AI optimization methods compare to the latest Large Language Models in terms of solution quality, validity, and time required to solve the 0/1 Knapsack problem?

The following table shows the results averaged over 5 runs:

Metric	o4 mini high	Claude 3.7 thinking	Gemini 3 pro	Gen. Alg.	A*
avg solution	7077	5767	7117	6694.4	7117
avg time (s)	131.29	248.2	157.94	19.49	7.22
avg cost (\$)	0.08	0.34	0.22	-	-
invalid sol.	0	1	0	0	0

Table 4: Knapsack Problem: Symbolic AI vs. LLMs

- **A* (Branch & Bound)** is the clear winner, finding the global optimum very efficiently.
- **Genetic Algorithm** despite not exploiting any problem specific heuristics, it achieved competitive performance, highlighting its robustness as a general purpose optimization algorithm.
- **o4-mini high** achieved near-optimal results (99.4% accuracy) and was the fastest among LLMs, although still significantly slower than A*.
- **Claude 3.7 thinking** struggled with the strict constraint, resulting in one invalid solution, and with the lowest avg. score despite the longest computation time and highest cost.
- **Gemini 3 Pro** successfully matched the optimal solution every run, demonstrating strong reasoning, but was in avg. $\approx 10\times$ slower than the symbolic AI approaches.

5 Conclusion

The results show that traditional symbolic AI techniques continue to have a major impact in well structured domains, where explicit state representation and systematic exploration of the solution space enable high performance, short timescales, and fully deterministic behaviour.

Although LLMs still underperform vs. symbolic AI techniques in strict decision-making tasks, the rapid progress of new models such as Gemini 3 Pro, which was release only some weeks ago (Nov 18, 2025), suggests that this gap could potentially be reduced in the future as the technology continues to advance.

6 Appendix

Game phase	Depth	Minimax		Minimax + $\alpha\text{-}\beta$	
		Time (s)	Nodes	Time (s)	Nodes
Start (0 ply)	1	0.000	56	0.000	56
	2	0.015	399	0.010	278
	4	0.906	19607	1.406	6292
	6	89.468	960792	37.265	108342
	8	~	~	431.343	1648893
Mid (15 plies)	1	0.000	56	0.015	46
	2	0.031	398	0.031	216
	4	1.562	17907	1.109	3748
	6	195.750	731255	9.765	46307
	8	~	~	38.500	472015
Late (32 plies)	1	0.000	46	0.000	23
	2	0.062	251	0.000	88
	4	1.046	5352	0.046	1051
	6	9.296	67318	0.265	6723
	8	47.093	417816	1.640	31684

Table 5: Minimax and Minimax + $\alpha\text{-}\beta$ results across different game phases and search depths.
~ means that it would require a lot of time to be computed.

Time (s)	Num. rollouts
1	3068
5	28588
10	62568
30	222619
60	329912
120	927735

Table 6: Monte Carlo Tree Seach: Time and Number of Rollouts

System Prompt	User Prompt
<p>You are a Connect Four expert. The board is a 6x7 list of lists made of: ' ' (empty cell), 'X' and 'O'. The last list corresponds to the bottom of the game board. You are playing as <code>{state.player2}</code>. Choose the next best move. Answer in the following structured format: <code>{'column': <number>, 'reason': '<brief explanation>'}</code> The column number must be between 0 and 6. The reason should be a short explanation of why you chose that move.</p>	<p>Current board state: <code>{state.board}</code>. Your turn to play as '<code>{state.to_play}</code>'.</p>

Table 7: Connect Four prompt

System Prompt	User Prompt
<p>You are an expert in combinatorial optimization. Solve the 0-1 knapsack problem exactly. Answer in the following structured format: <code>{'solution': <integer>}</code>.</p>	<p>Knapsack capacity: <code>{capacity}</code>. Items: <code>{Item {i}: weight={w}, value={v}, }</code>.</p>

Table 8: Knapsack problem prompt