Introduction to AI: Final Report

Nguyen Quang Huy
Faculty of Information Technology
Ton Duc Thang University
523H0140@student.tdtu.edu.vn

Nguyen Tran Nhat An
Faculty of Information Technology
Ton Duc Thang University
523H0115@student.tdtu.edu.vn

Nguyen Phuc Toan
Faculty of Information Technology
Ton Duc Thang University
523H0185@student.tdtu.edu.vn

Chung Quang Vu
Faculty of Information Technology
Ton Duc Thang University
523H0196@student.tdtu.edu.vn

Nguyen Van Minh Tri
Faculty of Information Technology
Ton Duc Thang University
523H0187@student.tdtu.edu.vn

Nguyen Thanh An
Lecturer
Faculty of Information Technology
Ton Duc Thang University
nguyenthanhan@tdtu.edu.vn

Ho Chi Minh City, Vietnam — 23 May 2025

***Abstract*—This paper presents a comprehensive exploration of core artificial intelligence techniques through modular and efficient implementations. It includes an object-oriented Simulated Annealing algorithm for optimizing a complex 3D function, demonstrating robust stochastic search capabilities. A heuristic-driven Alpha-Beta pruning method is designed for a 9×9 Tic-Tac-Toe game, employing advanced evaluation functions for effective adversarial decision-making. The report also details a constraint satisfaction problem modeled as a graph coloring challenge, solved via propositional logic and SAT solvers, highlighting logical reasoning and problem encoding. Finally, an object-oriented Naïve Bayes classifier is implemented using discretized quiz score data, showcasing probabilistic learning and classification. The system architectures emphasize clean, extensible code with modular class designs and integrated visualization tools to elucidate complex structures, ensuring maintainability and performance across diverse AI domains.

I. Background Knowledge

Simulated Annealing is an optimization method inspired by how metals cool and harden. It searches for the best solution by sometimes accepting worse answers at first to avoid getting stuck, then slowly focusing on better solutions.

Alpha-Beta Pruning is a way to speed up decision-making in games by ignoring parts of the game tree that won't affect the final choice. When combined with smart guesswork (heuristics), it works well even in big games like 9×9 Tic-Tac-Toe.

Constraint Satisfaction Problems (CSPs) involve finding values for variables that meet all given rules. Many of these problems can be turned into logic formulas and solved efficiently using SAT solvers, linking AI problem-solving with formal logic methods.

Naïve Bayes Classification is a simple but effective method for sorting data into categories. It assumes features are independent and works well when data can be split into clear groups.

II. Task 1: Simulated Annealing Search Optimization on 3D Surfaces

A. Problem Overview

We aim to find the global maximum of the following two-variable function:

$$f(x,y) = \sin\left(\frac{x}{8}\right)+\cos\left(\frac{y}{4}\right)-\sin\left(\frac{xy}{16}\right)+\cos\left(\frac{x^2}{16}\right)+\sin\left(\frac{y^2}{8}\right)$$

This highly non-linear and multi-modal surface poses challenges for traditional optimization techniques. We use the Simulated Annealing Search (SAS) algorithm to approximate the global maximum, starting from the origin $(0,0)$ and exploring the domain.

B. Approach

The solution is implemented in Python using an object-oriented structure with the following components:

- **Surface3:** Represents the surface function, handles symbolic and numerical evaluation, and provides 2D and 3D plotting methods for visualizing the search path and best solution.
- **Schedule:** Defines the cooling schedule $T(t)$, which controls the algorithm's exploration-exploitation balance.
- **SimulatedAnnealing:** Executes the search by generating neighbors, evaluating candidates, applying acceptance criteria, and tracking the best result.

The neighborhood of each state is defined using 8 directions uniformly spaced on the unit circle. A fixed step size of $\pi/32$ is used.

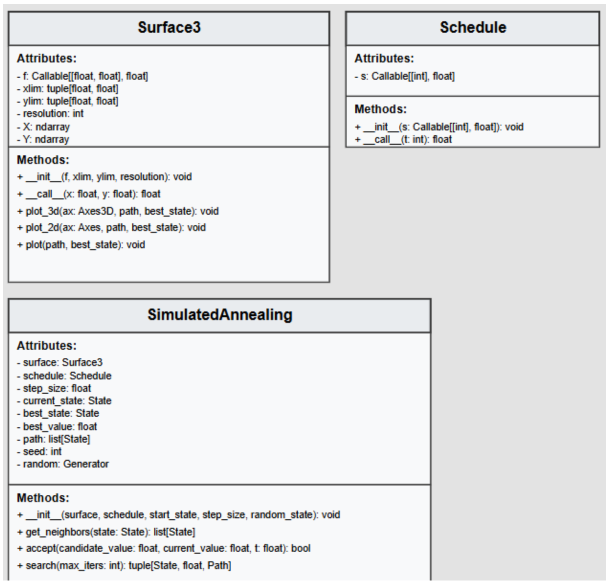

Surface3
Attributes:
- f: Callable[[float, float], float]
- xlim: tuple[float, float]
- ylim: tuple[float, float]
- resolution: int
- X: ndarray
- Y: ndarray
Methods:
+ __init__(f, xlim, ylim, resolution): void
+ __call__(x: float, y: float): float
+ plot_3d(ax: Axes3D, path, best_state): void
+ plot_2d(ax: Axes, path, best_state): void
+ plot(path, best_state): void
Schedule
Attributes:
- s: Callable[[int], float]
Methods:
+ __init__(s: Callable[[int], float]): void
+ __call__(t: int): float
SimulatedAnnealing
Attributes:
- surface: Surface3
- schedule: Schedule
- step_size: float
- current_state: State
- best_state: State
- best_value: float
- path: list[State]
- seed: int
- random: Generator
Methods:
+ __init__(surface, schedule, start_state, step_size, random_state): void
+ get_neighbors(state: State): list[State]
+ accept(candidate_value: float, current_value: float, t: float): bool
+ search(max_iters: int): tuple[State, float, Path]

Fig. 1. Class Diagram

C. *Temperature Schedule*

We used a logarithmic decay function:

$$T(t) = \max\left(10^{-4}, \frac{1}{\log(t+e)}\right)$$

This function allows significant exploration in early iterations and encourages convergence in later stages.

D. *Algorithm Overview*

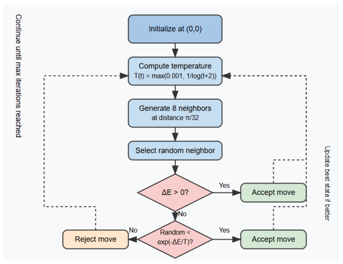

Initialize at (0,0)
Compute temperature T(t) = max(0.001, 1/log(t+2))
Generate 8 neighbors at distance π/32
Select random neighbor
ΔE > 0?
Yes
Accept move
No
Random < exp(-ΔE/T)?
Yes
Accept move
No
Reject move
Continue until max iterations reached
Update best state if better

Fig. 2. Simulated Annealing Search Algorithm Flowchart

The search algorithm proceeds as follows:

1) Initialize at $(0, 0)$ with best state set to the starting point.
2) For each iteration t:
   - Compute temperature $T(t)$ from the schedule.
   - Generate 8-directional neighbors.
   - Randomly select one as the candidate.
   - Accept the candidate based on:

$$P_{\text{accept}} = \exp\left(-\frac{\Delta f}{T(t)}\right)$$

   - If accepted, update the current state and possibly the best state.
3) Repeat for a fixed number of iterations.

E. *Visualization of the Search*

The optimization process is visualized in both 3D and 2D using the following figures:

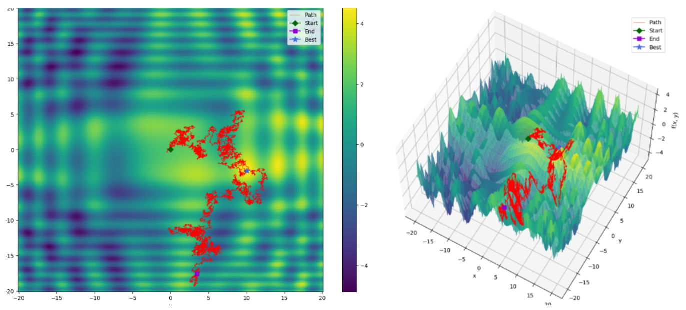


Fig. 3. 2D heatmap and 3D surface of $f(x, y)$ with search path (red line)

These visualizations illustrate how the algorithm navigates the landscape and converges near a high-value region.

F. *Experimental Results*

Using a fixed random seed for reproducibility, we ran the algorithm for 20,000 iterations. The outcomes are summarized below:

- **Random Seed:** 117844577062006770338411923264921799524
- **Best State Found:** $(x^*, y^*) \approx (9.98, -2.90)$
- **Estimated Maximum Value:** $f(x^*, y^*) \approx 4.535$

These results demonstrate the effectiveness of Simulated Annealing in identifying near-optimal solutions on complex surfaces.

III. Task 2: Heuristic Alpha-Beta Search in 9x9 Tic-Tac-Toe

A. *Problem Overview*

The game board is a 9x9 grid, and two players take turns placing their symbols. The winner is the first player to get four symbols in a row horizontally, vertically, or diagonally.

We encapsulated the game logic in object-oriented Python using the Pygame library for UI rendering. The key components are:

- **HumanPlayer**: A class that handles human interactions via mouse input.
- **AIPlayer**: An intelligent agent that chooses moves using alpha-beta pruning.
- **Game Manager**: Coordinates the game loop, board drawing, input handling, and win condition checking.

B. *Heuristic Alpha-Beta Search Design*

The AI's core decision-making leverages a modified alpha-beta pruning algorithm with a heuristic evaluation function. Due to the large branching factor in a 9x9 grid, we use a depth-limited version of the minimax algorithm.

1) Heuristic Evaluation Function: The heuristic function evaluates the board state based on:

- Number of potential lines of length 2 or 3 that can lead to a win
- Blocking opponent's 3-symbol sequences
- Prefer central positions to maximize flexibility

2) Alpha-Beta Pruning with Depth Limit: We implemented depth-limited alpha-beta pruning to enhance computational efficiency. The depth limit L was experimentally tuned to balance decision quality and performance.

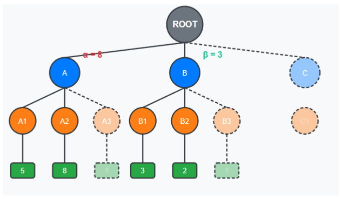


Fig. 4. Alpha-Beta Pruning

C. Game Flow Diagram

The following diagram outlines the interaction between the components in the game:

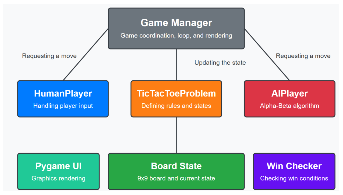


Fig. 5. Game Flow Diagram

D. Experimental Results

The 9x9 Tic-Tac-Toe implementation works as expected, providing a challenging opponent for human players.

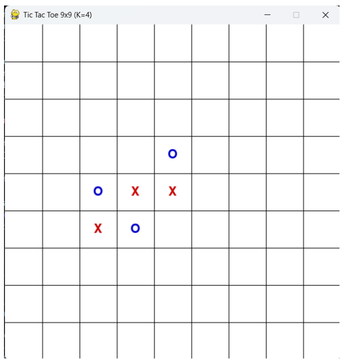


Fig. 6. Game Screen

The AI demonstrates intelligent behavior:

- Blocking the player's potential winning moves
- Setting up its own winning opportunities
- Playing strategically to control important board positions

Performance analysis shows:

- **Average decision time:** $<$ 1 second on standard hardware
- **Effective pruning:** On average, alpha-beta pruning reduces nodes examined by $\sim$70% compared to standard minimax
- **Win rate:** The AI successfully defeats naive strategies and presents a challenge to human players

IV. Task 3: Constraint Satisfaction Problem with Propositional Logic

A. Problem Overview

Given a $m \times n$ matrix, each cell either contains a non-negative integer or is blank. The goal is to color the cells using two colors: green (true) and red (false), so that for every cell containing a number, the count of adjacent green cells (including itself and its 8 neighbors) equals the number in the cell. Blank cells impose no constraints.

–	2	3	–	–	0	–	–	–	–
–	–	–	–	3	–	2	–	–	6
–	–	5	–	5	3	–	5	7	4
–	4	–	5	–	5	–	6	–	3
–	–	4	–	5	–	6	–	–	3
–	–	–	2	–	5	–	–	–	–
4	–	1	–	–	–	1	1	–	–
4	–	1	–	–	–	1	–	4	–
–	–	–	–	6	–	–	–	–	4
–	4	4	–	–	–	–	4	–	–

Fig. 7. Input matrix

B. Approach

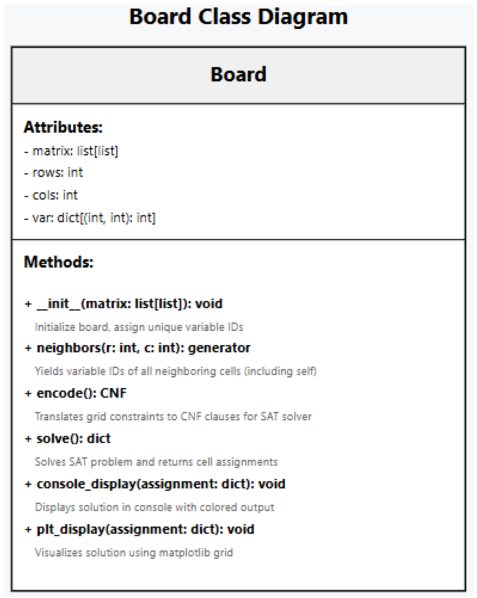
We solve this problem using propositional logic and the Glucose3 solver from the PySAT library. The main steps are:

- Assign each cell a unique propositional variable. If the variable is true, the cell is colored green; otherwise, it is red.
- For each non-blank cell, enumerate its neighbors and generate CNF clauses to enforce that exactly k of them are true, where k is the number in the cell.
- Use combinations to create upper and lower bounds on green neighbors, eliminating combinations that would violate the constraint.
- Use Glucose3 to solve the CNF and extract a satisfying assignment.
- Visualize the result using console output with ANSI colors and an optional matplotlib grid display.

C. Object-Oriented Implementation

The problem is modeled using a `Board` class with the following key methods:

- **`neighbors(r, c)`**: Returns the variable IDs of all valid neighboring cells including itself.
- **`encode()`**: Translates numeric constraints into CNF using PySAT's CNF format.
- **`solve()`**: Uses Glucose3 to find a satisfying assignment.
- **`console_display(assignment)`**: Prints the solution in the terminal using red and green background colors.
- **`plt_display(assignment)`**: Optionally renders the solution as a matplotlib plot.



Board Class Diagram

Board

Attributes:
- matrix: list[list]
- rows: int
- cols: int
- var: dict[(int, int): int]

Methods:

+ **__init__(matrix: list[list]): void**
  Initialize board, assign unique variable IDs
+ **neighbors(r: int, c: int): generator**
  Yields variable IDs of all neighboring cells (including self)
+ **encode(): CNF**
  Translates grid constraints to CNF clauses for SAT solver
+ **solve(): dict**
  Solves SAT problem and returns cell assignments
+ **console_display(assignment: dict): void**
  Displays solution in console with colored output
+ **plt_display(assignment: dict): void**
  Visualizes solution using matplotlib grid

Fig. 8. Board Class Diagram

D. Constraint Encoding

Let (r, c) be a cell containing a number k. We collect the variable IDs of its adjacent cells, and generate:

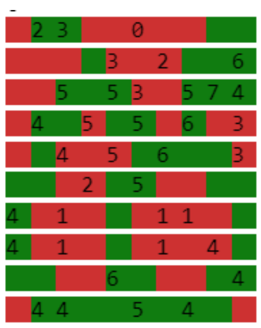
- **At most** k **true:** For all combinations of $k+1$ variables, add a clause with their negations.
- **At least** k **true:** For all combinations of $n-k+1$ variables (where n is the total neighbors), add a clause requiring at least one to be true.

```
for combo in combinations(lits, k+1):
    cnf.append([-v for v in combo])
for combo in combinations(lits, n-k+1):
    cnf.append(list(combo))
```

This ensures exactly k adjacent green cells.

E. Experimental Results

We evaluated the algorithm on a 10×10 matrix containing a combination of numeric constraints and unconstrained (blank) cells and produced a valid assignment that satisfies all given conditions. The results are displayed in two formats: a console-based textual output using ANSI color codes and a graphical visualization for easier interpretation.



```
  2 3     0         
        3   2     6 
    5   5 3   5 7 4 
  4   5   5   6   3 
    4   5   6     3 
      2   5         
4   1       1 1     
4   1       1   4   
        6         4 
  4 4     5   4     
```

Fig. 9. Text-based output of the solved matrix)

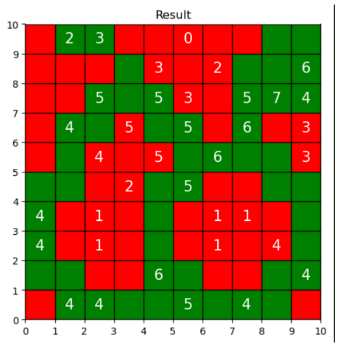


Fig. 10. Graphical output of the solved matrix

V. Task 4: Naïve Bayes Classifier

A. Problem Overview

The dataset contains students' quiz scores and whether they passed or failed. The goal is to build a model that can predict the result (Pass or Fail) based on the quiz scores.

B. Approach

We implemented the Naïve Bayes classifier from scratch using an object-oriented design in Python. The classification pipeline includes:

- Data parsing and preprocessing
- Score discretization
- Prior probability estimation
- Conditional likelihood computation with Laplace smoothing
- Posterior probability computation using log-space arithmetic
- Prediction and evaluation

C. Type Definitions

To enhance code clarity and maintainability, we defined the following type aliases:

- `Feature = Hashable` (e.g., "Low", "Average", "Good", "Great")
- `Label = Hashable` (e.g., "P", "F")
- `Input = Iterable[Feature]` (discretized quiz scores for a student)

D. Preprocessing: Discretization of Scores

As quiz scores are continuous, we first transform them into categorical bins to enable probabilistic modeling. The discretization scheme is as follows:

- **Y (Low)**: score < 5
- **TB (Average)**: $5 \leq$ score < 6
- **K (Good)**: $6 \leq$ score < 8
- **G (Great)**: score ≥ 8

E. Object-Oriented Implementation

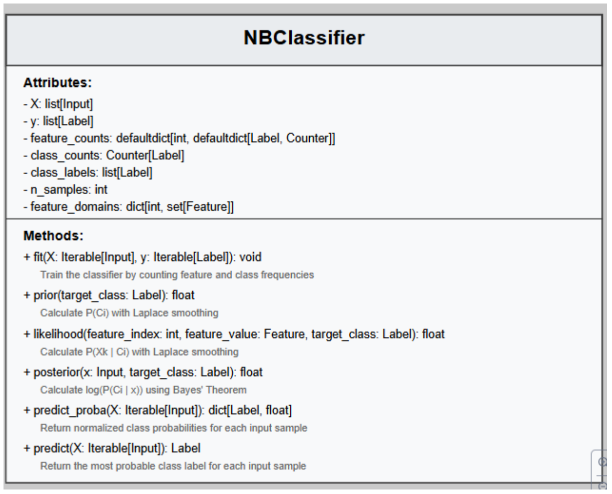
The Naïve Bayes classification system is structured using two main object-oriented components:

- `NBClassifier`: Encapsulates the core logic for training, prediction, and evaluation.

- `Preprocessor`: Handles data cleaning and discretization of continuous quiz scores into categorical bins.

Key methods implemented in `NBClassifier` include:

- `fit(X, y)`: Estimates prior probabilities for each class and conditional likelihoods for each feature given the class.
- `prior(class)`: Computes the prior probability of a given class label, adjusted with Laplace smoothing for unseen classes.
- `predict(X)`: Predicts class labels by computing the Maximum A Posteriori (MAP) estimate for each instance.
- `accuracy_score(y_true, y_pred)`: Computes the classification accuracy as the proportion of correct predictions.



NBClassifier

Attributes:
- X: list[Input]
- y: list[Label]
- feature_counts: defaultdict[int, defaultdict[Label, Counter]]
- class_counts: Counter[Label]
- class_labels: list[Label]
- n_samples: int
- feature_domains: dict[int, set[Feature]]

Methods:
+ fit(X: Iterable[Input], y: Iterable[Label]): void
  Train the classifier by counting feature and class frequencies
+ prior(target_class: Label): float
  Calculate P(Ci) with Laplace smoothing
+ likelihood(feature_index: int, feature_value: Feature, target_class: Label): float
  Calculate P(Xk | Ci) with Laplace smoothing
+ posterior(x: Input, target_class: Label): float
  Calculate log(P(Ci | x)) using Bayes' Theorem
+ predict_proba(X: Iterable[Input]): dict[Label, float]
  Return normalized class probabilities for each input sample
+ predict(X: Iterable[Input]): Label
  Return the most probable class label for each input sample

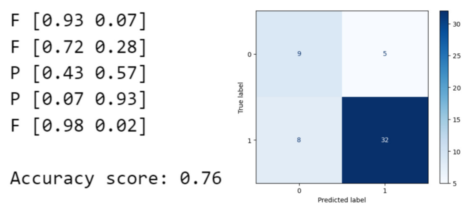
Fig. 11. NBClassifer Class Diagram

Key methods implemented in `Preprocessor` include:

- `__init__(bins, labels, fill_value)`: Constructor that initializes binning thresholds, labels, and the fill-in value for missing data.
- `transform(X)`: Applies binning and missing value replacement to an iterable of input vectors. The method returns the processed dataset ready for training or inference with the classifier.

F. Conclusion

The Naïve Bayes classifier achieved strong performance on this task. Discretization of quiz scores preserved key patterns while enabling interpretable probabilistic reasoning. The object-oriented implementation allows modular extension and reuse in future classification problems.



```
F [0.93 0.07]
F [0.72 0.28]
P [0.43 0.57]
P [0.07 0.93]
F [0.98 0.02]

Accuracy score: 0.76
```

Fig. 12. Accuracy and Confusion Matrix of Naïve Bayes Classifier

VI. Contributions

- **Nguyen Phuc Toan**: Task 1 – Simulated Annealing Search (100%); Task 4 – Naïve Bayesian Classifier(100%)
- **Nguyen Tran Nhat An**: Task 2 – 9x9 Tic-Tac-Toe with Heuristic Alpha-Beta Search (100%)
- **Nguyen Quang Huy**: Task 3 – Constraint Satisfaction Problems with Propositional Logic (100%)
- **Chung Quang Vu** and **Nguyen Van Minh Tri**: Task 5 – Report (100%)

VII. Self-Evaluation

- **Task 1:** Complete (100%).
- **Task 2:** Complete (100%).
- **Task 3:** Complete (100%).
- **Task 4:** Complete (100%).
- **Task 5:** Complete (100%).

VIII. Conclusion

This report explored four fundamental AI techniques through practical implementations. Simulated Annealing effectively optimized a complex 3D function using probabilistic search. Alpha-Beta Pruning, combined with heuristics, enabled strategic decision-making in a 9x9 Tic-Tac-Toe game. The Constraint Satisfaction Problem was successfully solved using propositional logic and a SAT solver, demonstrating logical reasoning. Finally, a Naïve Bayes classifier was built to predict student outcomes from quiz scores, showcasing the power of probabilistic learning. Each task highlighted a different AI approach, reinforcing the diversity and utility of AI methods in solving real-world problems.

References

[1] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 3rd ed. Pearson, 2010.

[2] S. J. Russell and P. Norvig, “Artificial Intelligence: A Modern Approach,” 4th ed. Pearson, 2020.

[3] E. Gansner, E. Koutsofios, S. North, and K. P. Vo, “A technique for drawing directed graphs,” IEEE Transactions on Software Engineering, vol. 19, no. 3, pp. 214-230, 1993.

[4] E. R. Gansner and S. C. North, “An open graph visualization system and its applications to software engineering,” Software: Practice and Experience, vol. 30, no. 11, pp. 1203-1233, 2000.

[5] N. J. Nilsson, “Principles of Artificial Intelligence,” Tioga Publishing Company, 1980.

[6] W. F. Clocksin and C. S. Mellish, “Programming in Prolog: Using the ISO Standard,” 5th ed. Springer, 2003.