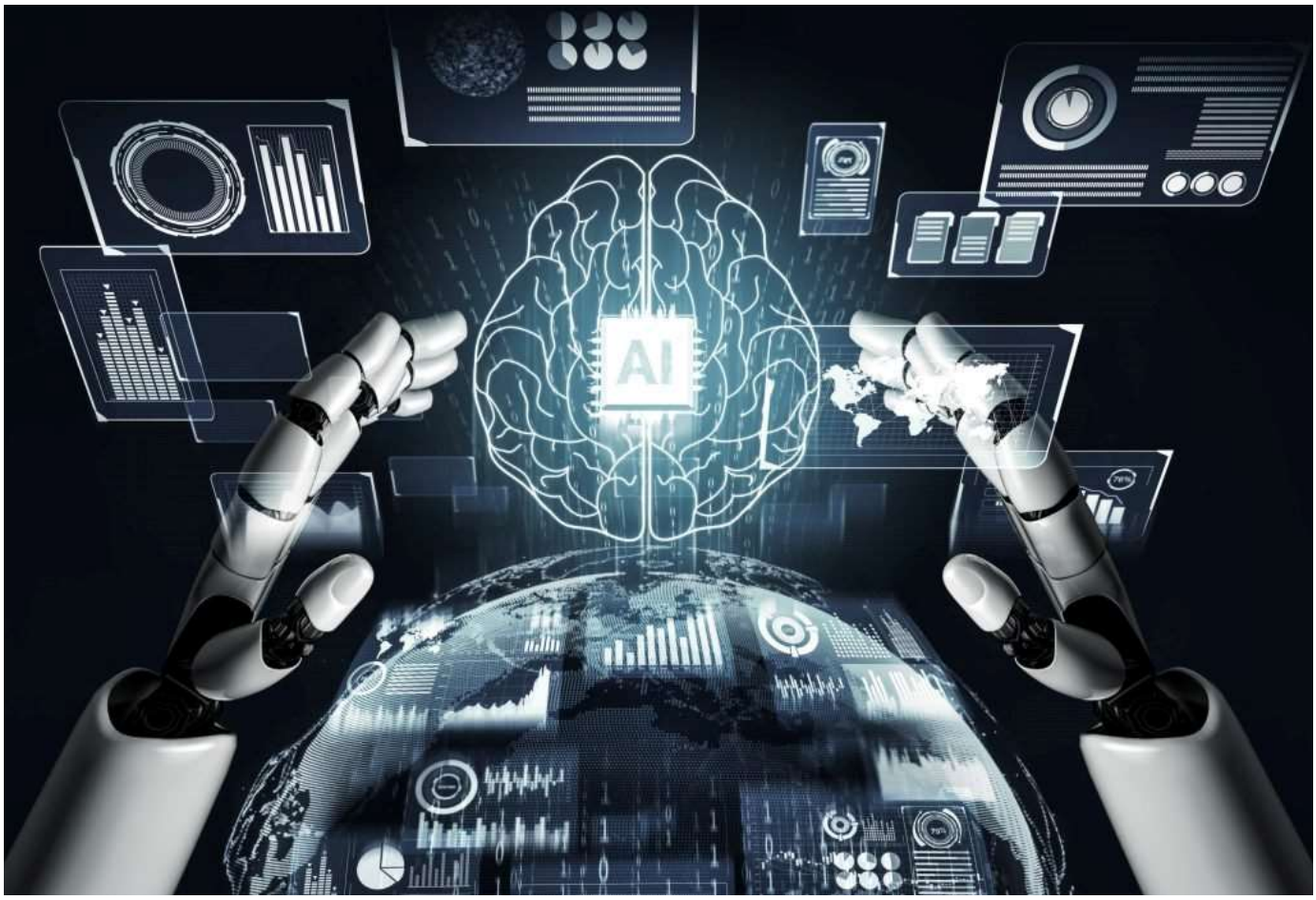
N-QUEENS



Artificial Intelligence

Contents:

1. Introduction and Overview
    i. Project Idea
    ii. Problem Description
2. Proposed Solution & System Features
    i. Main Functionalities
    ii. Use-Case Diagram
3. Applied Algorithms
    i. Backtracking
    ii. Best-First
    iii. Hill-Climbing
    iv. Cultural
    v. Heuristic Functions
4. Results and Analysis
    i. Results and Comparison
    ii. Analysis and Discussion

Introduction and Overview

1. Project Idea:

The N-Queens project aims to design an implement a program capable of solving the N-Queen problem for different board sizes.

The value of N is selected by user, allowing the program to work with small and large boards alike.

To deal with this challenge, the project explores 4 different Ai search techniques:

- Backtracking
- Best-First
- Hill-Climbing
- Cultural

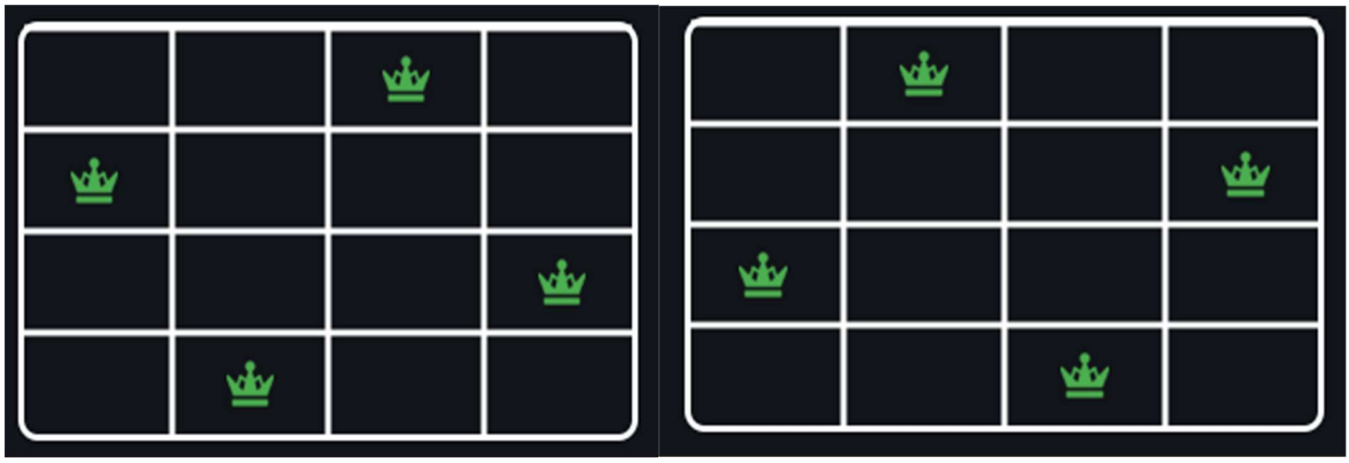
Using multiple approaches allow comparison of performance, effectiveness, and search behavior.

2. Problem Description:

In the N-Queen problem, the task is to arrange N queens on a NxN board so that no queen is attacking any other queen.

So to satisfy that **no** 2 queens must share the same row, column, or diagonal

As N increases, the number of possible board states grows exponentially, making the problem good for comparing search algorithms



Possible solutions for 4-Queens

Proposed Solution & System Features

1. Main Responsibilities:

The N-Queens program provides a set of core functionalities that allow user to explore and compare different algorithms for solving the N-Queens problem

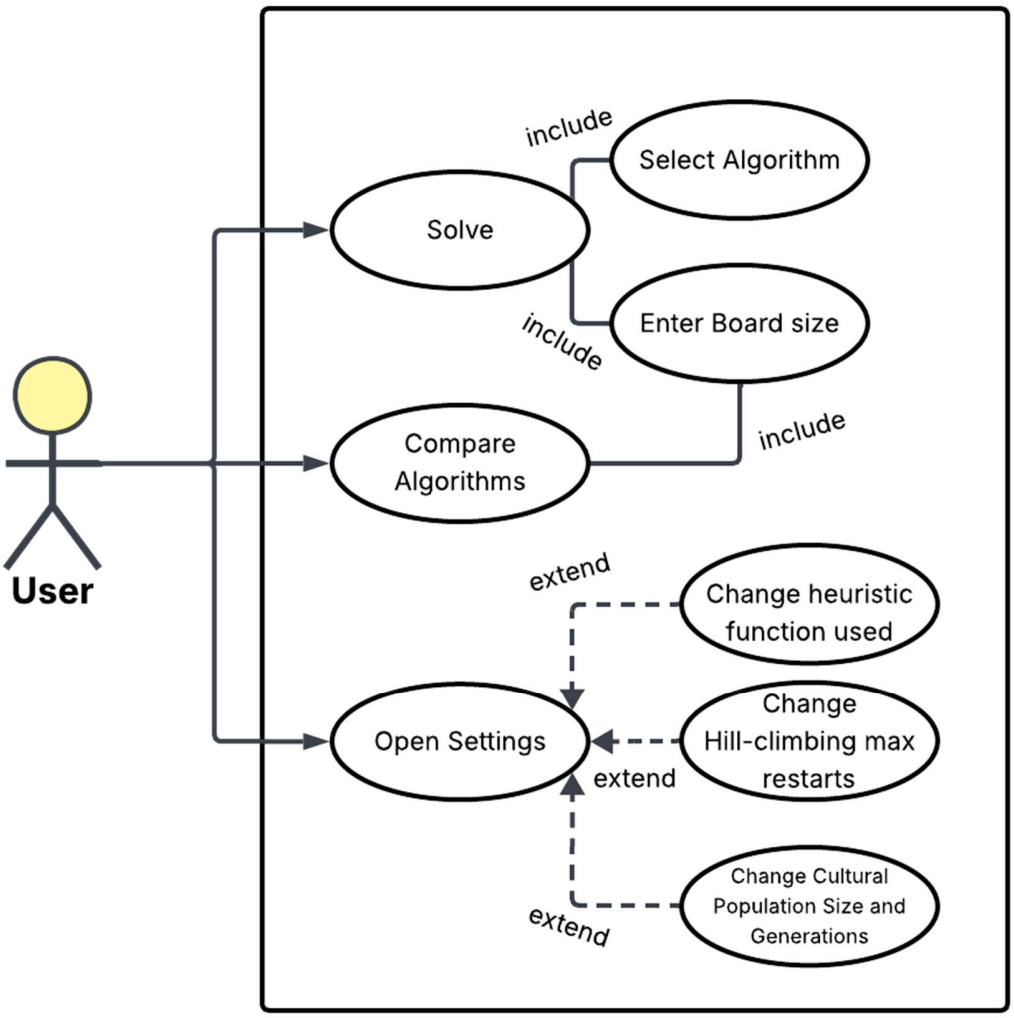
- User Input for Board Size:
    - Allow user to enter value of N which determine board size
    - Validates input
- Solving using the 4 algorithms individually:
    - **Backtracking**: Systematically explores all valid queen placements

    - **Best-First**: Uses heuristic function to guide search by selecting the most promising path first and aiming to find valid solution faster by reducing amount of blind exploration
    - **Hill-Climbing**: Start with initial state and iteratively improves by reducing conflicts, it also preforms local moves to reach better states
    - **Cultural**: Uses evolutionary strategies combined with belief space, it updates both population and belief space to accelerate convergence and generate improved candidate solutions over multiple generations

- User Interface:
    - Display board showing position of queens

- Allow user to view solution of selected algorithm or all of them
- Display time taken to find solution
- Allow user to compare all algorithms

2. Use-Case:



3. Development Tools:

- Programming Language: Python
- IDE: PyCharm, VSC
- Libraries: flet->GUI
  heapq->Priority queue
  random->Random initials
  time->Record time taken
- Version Control: GitHub

Applied Algorithms

1. Backtracking:

Backtracking is a **systemic search technique** that tries to reach solution step by step and **backtracks** choices when they lead to a conflict

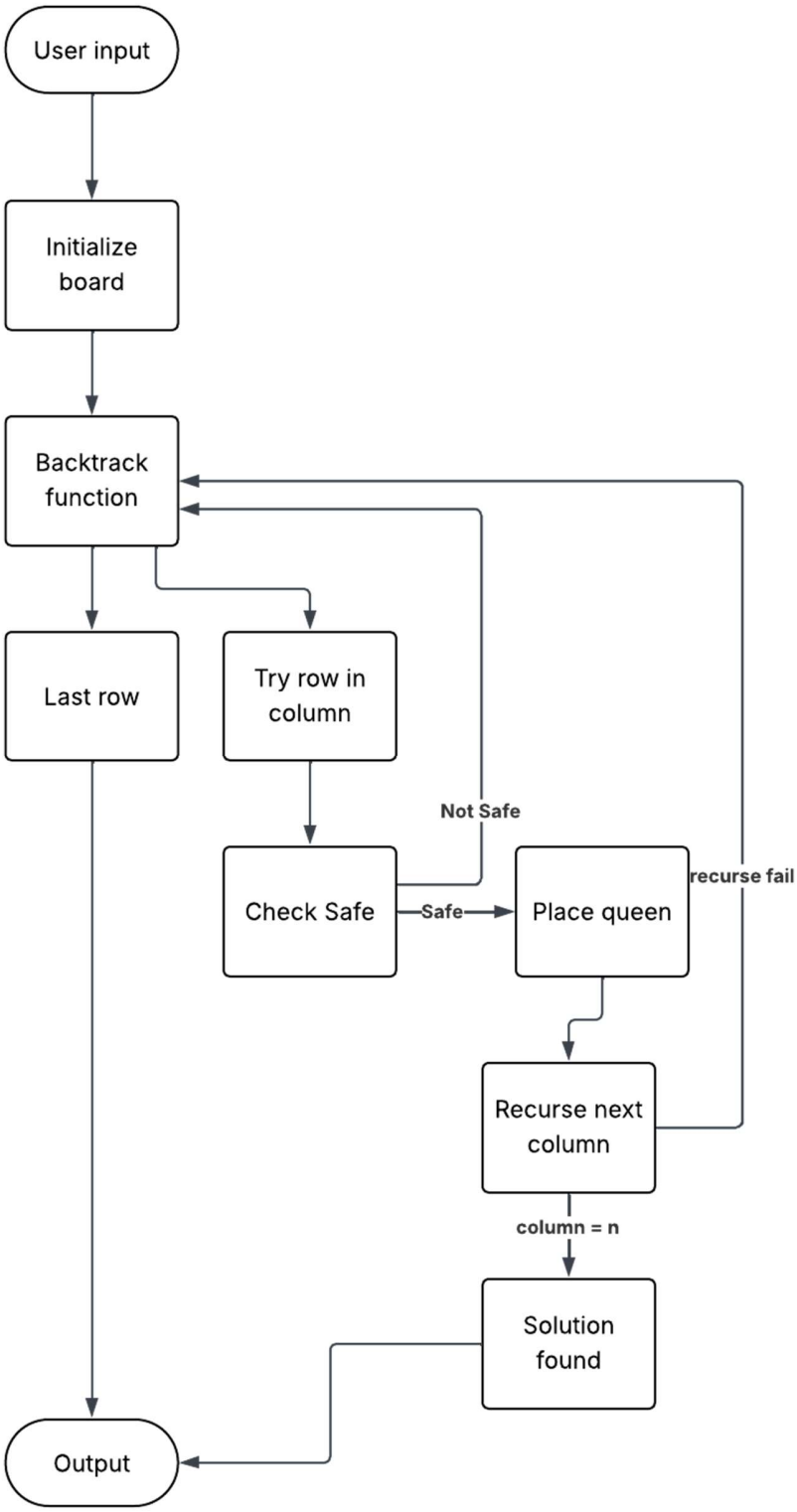
In N-Queens, it places 1 queen per column and explore all possible placements

- When board is empty start at first column
- Try placing a queen in a row, if row 0 cause conflict, try row 1 and if it also causes conflict try row 2 and etc. until a safe move is found
- If safe move is found, place queen in row and move to next column (Recursive call)
- If safe move isn't found, backtrack by returning to previous column and

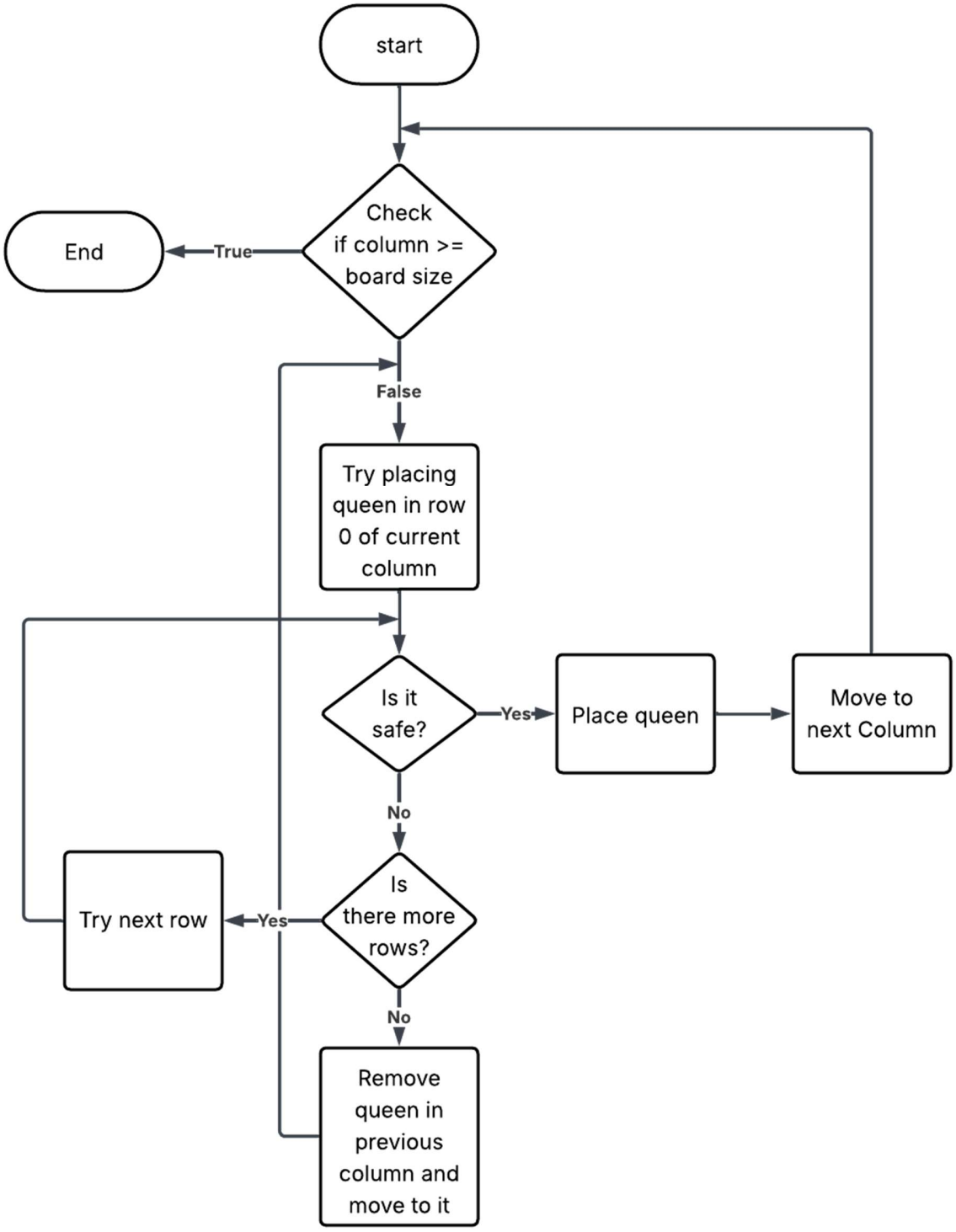
remove the queen there and try the next row in that old column

- Continue doing this process until column ==N which would means that N queens have been placed and a solution has been found
- Algorithm ends when it finds solution or tried all valid ways of placing queen
- **Backtracking guarantees correct solution, avoid exploring invalid branches, and efficient for small values of N**

Block Diagram:



Flowchart:


start
Check if column >= board size
True
End
False
Try placing queen in row 0 of current column
Is it safe?
Yes
Place queen
Move to next Column
No
Is there more rows?
Yes
Try next row
No
Remove queen in previous column and move to it

- ## Literature Review:

  Backtracking has long been a core method for solving constraint satisfaction problems, and extensive research has examined both its theory and the factors behind its effectiveness. Although many variants exist, such as Forward Checking, Backjumping, and Conflict-Directed Backjumping. The papers shows that they can all be understood through a shared set of underlying principles. Two major research directions shaped this modern view: a unified interpretation of backtracking based on no-goods, and formal analyses of the search trees and consistency checks that different algorithms necessarily explore.

- A central insight is that these algorithms differ not in their fundamental operations, but in how they record and exploit information about inconsistent partial assignments. The concept of a no-good an assignment combination that cannot appear in any solution, provides a common framework for understanding all backtracking methods. Within this perspective, forward-looking techniques and backward-looking techniques are simply different strategies for discovering, storing, and reusing no-goods. The pruning strength of an algorithm depends on the types of no-goods it learns, how detailed they are, and how long they are retained.
- The papers further clarify how these algorithms traverse the search space. Formal characterisations identify exactly which nodes must be visited or avoided by each method, grounding claims about pruning power. For instance, algorithms enforcing stronger consistency, such as Forward Checking, skip many nodes that simpler chronological backtracking must explore. Conflict-based methods go further by identifying the deeper sources of inconsistencies and skipping whole portions of the search tree.
- It also helped clarify correctness and complexity, especially for early conflict-directed methods like backjumping and its extensions, which originally lacked clear formal justification

- **References:**

[1] F. Bacchus, “A uniform view of backtracking,” *Principles and Practice of Constraint Programming (CP)*, pp. 232–247, 2001.

[2] G. Kondrak and P. van Beek, “A theoretical evaluation of selected backtracking algorithms,” *Artificial Intelligence*, vol. 89, no. 1–2, pp. 365–387, 1997.

2. Best-First:

Overview

1. The `best_first` function implements a **Best-First Search** strategy to solve the **N-Queens problem**.
2. It attempts to find a board configuration where no queens attack each other by repeatedly exploring the most promising states first, using a heuristic score as the priority.

The algorithm uses:

- A **priority queue** (`heapq`) to always expand the state with the lowest heuristic value.
- A **heuristic function** `Heuristics.current` to evaluate states.
- A **visited set** to avoid revisiting previously seen configurations.

Function Signature

```
def best_first (board):
```

Parameters

- **board**: An object representing the N-Queens board.
  - `board.N`: number of queens (and board size)
  - `board.start`: initial state — a list where `state[col] = row` of a queen
  - `board.place_queen(row, col)`: places a queen on the final board

How the Algorithm Works :

1. Initialization

```
n = board.N
start = board.start.copy()
heuristic = Heuristics.current
pq = [(heuristic(start, n), start)]
visited = set()
```

- `start` is the initial configuration.
- The priority queue contains `(heuristic_value, state)`.
- The lower the heuristic, the more promising the state.
- `visited` stores states already expanded.

---

2. Main Loop: Priority-Driven State Expansion

```
while pq:
    h, state = heapq.heappop(pq)
```

- The queue always pops the state with the **lowest heuristic**.
- `h` is the number of conflicts
- `state` is a list of queen positions

---

3. Goal Test

```
if h == 0:
    for col in range(n):
        board.place_queen(state[col], col)
    return True
```

- If heuristic = 0 → no queens attack each other → solution found.
- The queens are placed onto the board.

---

4. Mark State → Visited

```
visited.add(tuple(state))
```

- States are converted to tuples because lists are not hashable.
- Prevents infinite loops or redundant computation.

---

5. Generate Neighbor States

```
for i in range(n):
    for j in range(n):
        if j != state[i]:
            new_state = state.copy()
            new_state[i] = j
            ...
```

For each column `i`:

- Try placing the queen in every other row `j`
- Each new configuration is a neighbor state differing by one queen move.

---

6. Push Valid New States to the Priority Queue

```
if tuple(new_state) not in visited:
    heapq.heappush(pq, (heuristic(new_state, n), new_state))
```

- Only unvisited states are added.
- Priority is determined again by the heuristic value.

---

7. Failure Case

```
return False
```

If all states are exhausted and no solution has heuristic 0, the algorithm reports failure.

Algorithm Characteristics

Search Strategy

- **Greedy Best-First Search**: always expands the state with minimum heuristic.
- **Not guaranteed optimal**, but often fast.

Heuristic

- Depends on `Heuristics.current`.
- Usually→ number of attacking queen pairs.

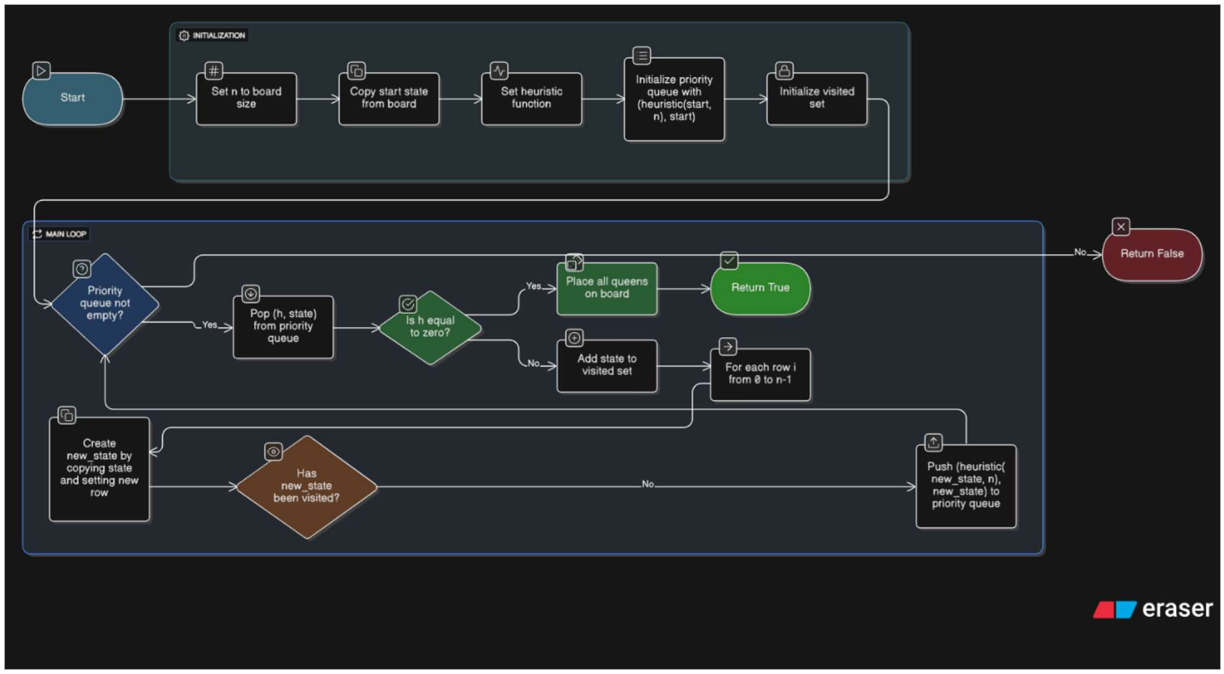
State Space

- Each state is an array of size `n`

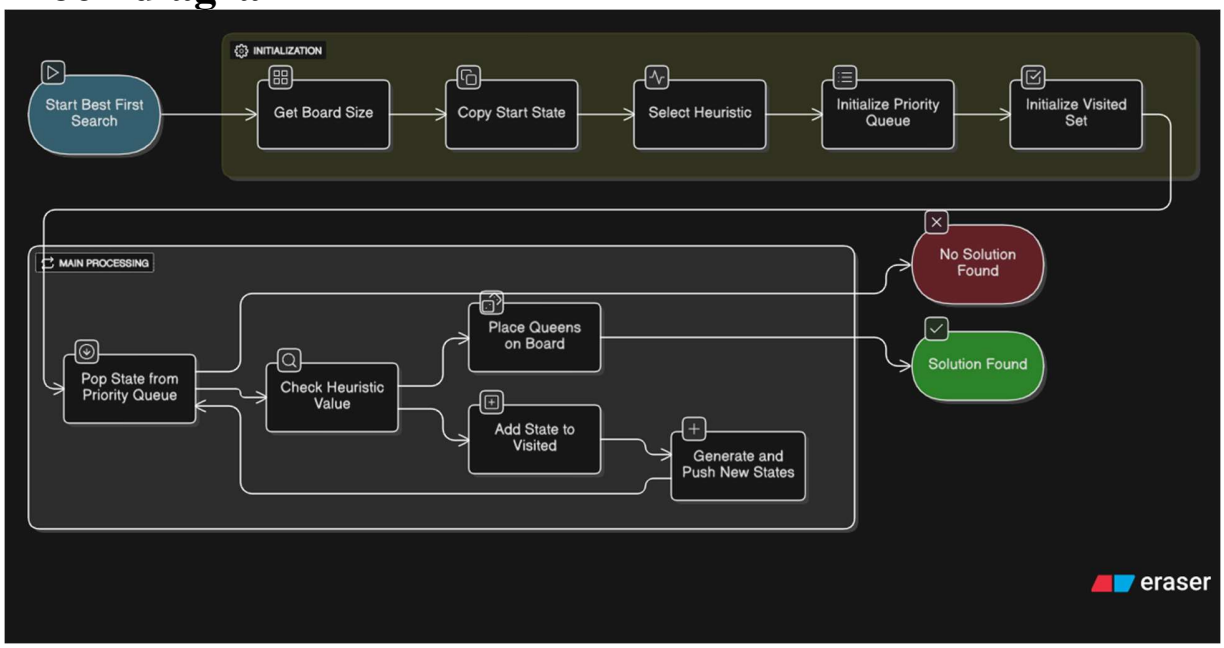
Complexity

- Worst-case: exponential (as with most N-Queens search algorithms)

Flowchart



Block diagram



Paper Summary

This paper generalizes the theory of **best-first search** by studying evaluation functions that depend on the *entire path* rather than only the standard A* components g(n)g(n)g(n) and h(n)h(n)h(n). The authors analyze how well-known properties of A*—such as admissibility, termination, node-expansion conditions, and optimality—extend to this broader class of **generalized best-first (GBF)** strategies.

Key Contributions

1. A general framework for path-dependent evaluation functions

The paper introduces a generalized formulation where the evaluation function

f(n)=f(s,n1,n2,…,n)f(n) = f(s, n_1, n_2,\dots ,n)f(n)=f(s,n1,n2,…,n)

may depend on the entire path, not just g(n)+h(n)g(n)+h(n)g(n)+h(n). This includes nonlinear, nonadditive, and path-dependent heuristics.

They show that many properties of A* extend naturally when fff is *order-preserving* (i.e., expanding a path cannot reverse earlier merit comparisons).

---

2. Termination, completeness, and solution-quality results

The authors define a critical value called **M** (a "minimax saddle value"), based on the maximum fff-value along solution paths.

They prove:

- **The algorithm always terminates** if fff is unbounded on infinite paths.
- **The returned solution has cost ≤ $\varphi^{-1}(M)$** for monotonic fff.
  This produces bounds for **suboptimal but non-admissible heuristics**—including weighted A* variants and dynamically weighted schemes.

---

3. Generalized node-expansion conditions

They generalize the classic A* expansion condition:

f(n)≤C∗f(n) \le C^*f(n)≤C∗

to arbitrary GBF strategies. They provide:

- **Necessary and sufficient conditions** for when a node must be expanded.
- Specialized, exact conditions for **tree search**, independent of knowing the final solution path.

These conditions support performance analysis of GBF under broader classes of heuristics.

---

4. Computational optimality of A*

The central contribution is a rigorous examination of A*'s optimality among **equally informed** algorithms—those given the same heuristic hhh.

They define a **hierarchy of optimality notions (Type 0–3)** and evaluate how A* compares to other admissible algorithms under several domains:

- admissible heuristics ($h \leq h^*$)
- consistent heuristics
- pathological vs. non-pathological cases
- best-first vs. more general strategies

Major findings

- *A is not the most efficient admissible algorithm in general.**
  There exist admissible algorithms that expand fewer nodes than A* for some admissible heuristics.
- **No admissible algorithm is globally optimal** under arbitrary admissible (possibly inconsistent) heuristics.
- *A is optimal when heuristics are consistent.**
  Under consistency, every admissible algorithm must expand all nodes A* must expand—establishing **computational optimality**.
- A* is also optimal within the class of **best-first algorithms guided by path-dependent evaluation functions**, assuming consistency.

Thus, A*'s optimality is **conditional**, not universal.

References Mentioned in the Paper

1. Bagchi, A., and Mahanti, A. *Search Algorithms Under Different Kinds of Heuristics.*
2. Dechter, R., and Pearl, J. *Generalized Best-First Search Strategies and the Optimality of A** (the present paper).
3. Bellman, R. *Dynamic Programming.*
4. Gelperin, D. *On the Optimality of A**.
5. Pohl, I. *Heuristic Search Viewed as Path Finding in a Graph.*
6. Hart, P. E., Nilsson, N. J., and Raphael, B. *A Formal Basis for the Heuristic Determination of Minimum Cost Paths.*
7. Hart, P. E., Nilsson, N. J., and Raphael, B. *Correction to "A Formal Basis..."*
8. Gaschnig, J. *Heuristic Search Algorithms.*
9. Karp, R. *Probabilistic Analysis of Search Algorithms* (relates to Appendix).
10. Lawler, E. L., and Wood, D. E. *Branch-and-Bound Methods.*

11. Pearl, J. *Heuristics: Intelligent Search Strategies for Computer Problem Solving.*
12. Miro, R. *On the Optimality of A*.*
13. Nilsson, N. J. *Principles of Artificial Intelligence.*
14. Nilsson, N. J. *Problem-Solving Methods in Artificial Intelligence.*

3. Hill-Climbing:

4. Cultural:

5. Heuristic Function:

Heuristic function is a **problem specific function** used in search algorithms to **estimate how close are you to reach goal state from current state**

It's a **guiding measure** to prioritize certain path over others

It's commonly referred to as **h(n)**, where n is current state

Use:

- Guide Search algorithms
- Reduce search space
- Provide way to score potential solutions

1)Heuristic1

```python
def heuristic1(state, n):
    conflicts = 0
    for i in range(n):
        for j in range(i + 1, n):
            if state[i] == state[j] or abs(state[i] - state[j]) == abs(i - j):
                conflicts += 1
    return conflicts
```

h(n)= Number of attacking pairs

Logic:

- Iterate over all pairs of queens (i,j) with i<j
- Check if they are in same row or diagonal
- If either condition is true, increment conflicts
- Return number of total conflicting pairs

It counts pairs so each conflict is counted once

It has ***Time Complexity O(n^2)*** because of nested loop, so it's only good for small n

It has ***Space Complexity O(n)***

2)Heuristic2

```python
def heuristic2(state, n):
    row = [0] * n
    d1 = [0] * (2*n)
    d2 = [0] * (2*n)
    for c in range(n):
        r = state[c]
        row[r] += 1
        d1[c + r] += 1
        d2[c - r + n] += 1
    conflicts = 0
    for c in range(n):
        r = state[c]
        conflicts += (row[r] - 1)
        conflicts += (d1[c + r] - 1)
        conflicts += (d2[c - r + n] - 1)
    return conflicts
```

h(n)= Total number of conflicts for all queens

Logic:

- Initialize array to count, row[r] for number of queens in row r, d1[c+r] for queens in same major diagonal (top-left and bottom-left) and d1[c-r+n] for minor diagonal
- Loop over all columns c and update count for queens
- Loop again to get conflicts for each queen
- Return number of total conflicts

Optimized version by using arrays instead of nested loop

It has ***Time Complexity O(n)***

It has ***Space Complexity O(n)***

3)Heuristic3

```python
def heuristic3(state, n):
    conflicts = 0
    for c in range(n):
        r = state[c]
        for c2 in range(n):
            if c == c2:
                continue
            r2 = state[c2]
            if r == r2:
                conflicts += 1
            if abs(r - r2) == abs(c - c2):
                conflicts += 1
    return conflicts // 2
```

h(n)= Number of attacking pairs of queens

Logic:

- Loop each queen c and compare with every other queen c2
- If c==c2 then continue
- Check row for conflicts r==r2
- Check diagonals for conflicts abs(r-r2) == abs(c-c2)

- Divide conflicts by 2 at end so each conflict is only counted once

More readable and easier to comprehend

It has ***Time Complexity O(n^2)*** because of nested loop, so it's only good for small n

It has ***Space Complexity O(n)***

Comparison

All 3 heuristic aim to measure how bad board state is

Lower value-> fewer conflicts-> closer to solution

In terms of speed Heuristic2 is the fastest followed by Heuristic1 and Heuristic3

In terms of strength (how accurately it estimates true cost from current state to goal) Heuristic3 is the strongest as it counts exact tiles and is more accurate followed by Heuristic2 then Heuristic1

Results and Analysis

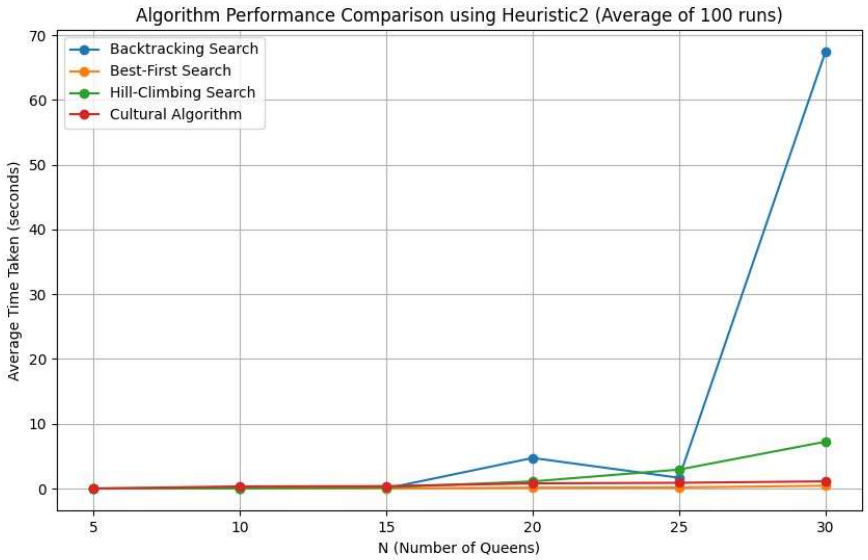
1. Results and Comparison

- To keep comparison fair, we used the same start state and same board size
- We ran each algorithm several times to get average
- For N=5

Algorithm	Avg Time	Success
Backtracking	0.000132	100%
Best-First	0.000302	100%
Hill-Climbing	0.000561	100%
Cultural	0.001600	100%

- For N=20

Algorithm	Avg Time	Success
Backtracking	4.754722	100%
Best-First	0.104976	100%
Hill-Climbing	1.153362	95%
Cultural	0.612077	100%



2. Analysis and Discussion

1)Backtracking:

- Always find solution if it exists
- Very efficient for small number of N
- Execution time increase exponentially as N increase because it searches exhaustively
- It's doesn't use heuristic so no guidance
- Not practical for large N

2)Best-First:

- Faster than Backtracking
- Explore less states due to using heuristic function to guide to the most promising node

- It depends on the heuristic function strength

3)Hill-Climbing:

- Works better with small N and higher success rate
- For large N it can get stuck in local minima
- Restarting improves success rate

4)Cultural:

- More stable than Hill-Climbing
- Faster than others for large N
- Performance improves as belief space increase
- Doesn't fall into local minima

Algorithm	Advantage	Disadvantage
Backtrack	Guaranteed solution	Slowest and exhaustive
Best-First	Fast	Depend on Heuristic
Hill-climbing	Faster than backtrack	Unreliable
Cultural	Fastest	Depend on population size and number of generations