Problem Statement:

The task is to implement a search algorithm to solve the 8-puzzle problem using different heuristics. The search algorithm utilizes a cost function (f(n) = g(n) + h(n)), where (g(n)) represents the least cost from the source state to the current state, and (h(n)) represents the estimated cost of the optimal path from the current state to the goal state. Four different heuristics are considered:

1. (h_1(n) = 0)
2. (h_2(n)) = number of tiles displaced from their destined position.
3. (h_3(n)) = sum of Manhattan distance of each tile from the goal position.
4. (h_4(n)) = Custom heuristic designed to ensure (h(n) > h^*(n)).

Additionally, the implementation must adhere to the following assumptions and requirements:

- Two lists are utilized: one for maintaining already explored states (closed list) and another for maintaining states yet to be explored (open list).
- Input is provided in a file format, specifying the initial and final states of the puzzle.
- The output should include details such as success or failure messages, start and goal states, total number of states explored, total number of states on the optimal path, optimal path, optimal path cost, and execution time.
- The implementation should be generic and preferably in Python.
- A comparison between the results of all four heuristics should be provided, highlighting their differences in optimality, time complexity, etc.

Pseudocode:

```
function AStarSearch(initialState, goalState, heuristic):
    openList = PriorityQueue()
    openList.enqueue(initialState)
    closedList = Set()
    gScore = {}
    gScore[initialState] = 0
    cameFrom = {}

    while openList is not empty:
        currentState = openList.dequeue()
        if currentState == goalState:
            return reconstructPath(cameFrom, currentState)

        closedList.add(currentState)
        for neighbor in currentState.getNeighbors():
            tentativeGScore = gScore[currentState] +
distanceBetween(currentState, neighbor)
            if neighbor in closedList and tentativeGScore >=
gScore[neighbor]:
                continue
            if neighbor not in openList or tentativeGScore <
gScore[neighbor]:
                cameFrom[neighbor] = currentState
                gScore[neighbor] = tentativeGScore
                fScore = gScore[neighbor] + heuristic(neighbor, 
goalState)
                openList.enqueue(neighbor, fScore)

    return failure

function reconstructPath(cameFrom, currentState):
    totalPath = [currentState]
    while currentState in cameFrom:
        currentState = cameFrom[currentState]
        totalPath.append(currentState)
    return totalPath.reverse()
```

Detailed Report:

1. **Heuristic (h_1(n) = 0)**:

   - This heuristic assumes no cost from the current state to the goal state, effectively making the algorithm equivalent to uniform-cost search.
   - It expands a large number of states since it doesn't consider any information about the goal state.
   - The optimal path might not be found efficiently due to the lack of heuristic information.
   - Execution time might be high due to the potentially large number of states explored.
2. **Heuristic (h_2(n)) (Number of tiles displaced)**:

   - This heuristic counts the number of tiles displaced from their destined position, providing some information about how close the current state is to the goal state.
   - It explores fewer states compared to (h_1(n)) as it considers the displacement of tiles.
   - The optimality depends on the specific instances of the puzzle.
   - It might not always find the optimal path due to its simplicity.
3. **Heuristic (h_3(n)) (Manhattan distance)**:

   - This heuristic calculates the Manhattan distance of each tile from its goal position, providing a more informed estimate of the distance to the goal state.
   - It explores even fewer states compared to (h_2(n)) as it incorporates more information about the goal state.
   - It tends to find optimal paths more efficiently due to the better heuristic estimate.
   - However, the computation of Manhattan distance for each tile increases the time complexity.
4. **Custom Heuristic (h_4(n))**:

   - This heuristic is designed to ensure that (h(n) > h^*(n)), where (h^*(n)) represents the true cost to reach the goal state.
   - It aims to guide the search algorithm away from suboptimal paths that might be suggested by other heuristics.
   - It may lead to exploring a larger number of states compared to other heuristics, depending on the specific puzzle instance and the design of the heuristic.

- It can potentially improve the optimality of the solution by avoiding certain suboptimal paths.

Comparison:

Heuristic	Total States Explored	States on Optimal Path	Time Taken
(h_1(n))	High	Variable	High
(h_2(n))	Moderate	Variable	Moderate
(h_3(n))	Moderate	Variable	Moderate to High
(h_4(n))	High	Variable	High

Conclusion:

- The choice of heuristic significantly affects the performance of the search algorithm.
- Heuristics that provide better estimates of the distance to the goal state tend to explore fewer states and find optimal paths more efficiently.
- However, more informative heuristics often come with higher time complexity due to additional calculations.
- Custom heuristics can be designed to guide the search algorithm towards more optimal solutions, but they may require careful tuning and can potentially increase the exploration space.
- It's essential to balance between the informativeness of the heuristic and its computational cost to achieve efficient and optimal search performance.

```
In [ ]:
```