CS 4820/5820 Homework 1: Problem Solving and Search

Josh Manchester
University of Colorado Colorado Springs
josh.manchester@uccs.edu

Part A: Intelligent Agents (1.0 pts)

Question 1: PEAS Descriptions

(a) Autonomous Delivery Drone **Performance:** The drone should minimize energy use and costs while successfully delivering packages on time.

Environment: It operates outdoors where there are buildings, different weather conditions, and GPS signals. There are also no-fly zones and specific areas for takeoff and landing.

Actuators: Propellers and motors for flight control, control surfaces for steering, mechanisms to keep it upright, grippers to hold packages, lights, and a speaker for alerts.

Sensors: GPS to know where it is, an altimeter for altitude, cameras to detect obstacles, and laser sensors to measure distances.

(b) Chess-Playing Agent **Performance:** Win rate, how fast it makes moves, and whether the moves are strategic or just random.

Environment: A standard 8×8 chess board where all pieces are visible, and the rules are deterministic (always the same outcome for the same moves).

Actuators: Either a robotic arm to move physical pieces or digital commands if it's software-based.

Sensors: A vision system with cameras if it's watching a physical board, or direct access to the board state in a digital game. It might also have access to a database of chess openings.

(c) Self-Driving Taxi **Performance:** How efficiently it picks up and drops off passengers, fuel efficiency, travel time, passenger safety, and customer satisfaction.

Environment: Real roads with other cars, pedestrians, traffic lights, weather, and different road conditions.

Actuators: Wheels, steering wheel, gas pedal, brake, lights, and horn.

Sensors: Cameras, LIDAR, ultrasonic sensors, speedometer, GPS, accelerometer, engine sensors, and an interface for passengers to enter their destination.

Question 2: Agent Type Classification

(a) Autonomous Delivery Drone: Model-Based Reflex Agent The drone keeps track of where it is (GPS coordinates, altitude, obstacles, weather) and uses this information to make decisions based on what it's currently sensing. Even though it has a goal (deliver the package), it mostly reacts to what's happening right now using its internal model instead of planning out a bunch of different possible actions.

(b) Chess-Playing Agent: Goal-Based / Utility-Based Agent A chess AI has to think ahead about how to win the game by planning sequences of moves. It can't just react to the current board - it needs to think about future positions and figure out if they get it closer to checkmate. More advanced versions are utility-based because they use evaluation functions to score positions and try to maximize their chances of winning.

(c) Self-Driving Taxi: Utility-Based Agent The taxi has to balance a lot of different things that sometimes compete with each other. It can't just focus on one goal - instead, it has to optimize a utility function that considers passenger safety (most important), travel time, fuel efficiency, comfort, following traffic laws, and finding the best route.

Part B: n-Puzzle Problem (3.5 pts)

Problem Formulation

States: All the possible ways you can arrange the tiles on the board. For the 8-puzzle (3×3), there are 9!/2 = 181,440 states you can actually reach. For the 15-puzzle (4×4), there are 16!/2 10.4 trillion reachable states.

Initial State: Any valid starting configuration that's actually solvable. You can tell if it's solvable by checking the parity of inversions in the tile sequence.

Actions: Slide a tile that's next to the blank space into the blank. Or you can think of it as moving the blank up, down, left, or right (if it's not at an edge).

Transition Model: Given a state and an action, you get a new state by swapping the blank with the adjacent tile.

Goal Test: Check if the tiles are in order: 1, 2, 3, ..., n with the blank at the end.

Path Cost: Each move costs 1. The optimal solution is the one with the fewest moves.

Implementation Details

I implemented all the algorithms in Python from scratch without using any graph libraries per the requirements in the assignment. Here are the main design choices I made:

- **State Representation:** I used tuples of integers (basically the board flattened into one line) because tuples can be hashed, which makes sets and dictionaries way easier to use.
- **Neighbor Generation:** I find where the blank is, then generate valid moves based on whether it's at a board edge or not.
- **Parent Tracking:** I used a dictionary to map each state back to its parent so I could reconstruct the solution path.
- **Explored Set:** A set of states I've already visited so I don't get stuck in loops (this is graph search instead of tree search).

BFS Results

Breadth-First Search looks at all the nodes at one level before moving to the next level, using a FIFO queue. This guarantees you'll find the shortest solution for problems where each action costs the same.

Puzzle	Trial	Moves	Expanded	Time (s)
8-puzzle	1	6	61	0.0001
	2	4	20	0.0000
	3	10	462	0.0009
	Avg	**6.67**	**181.0**	**0.0003**
15-puzzle	1	16	259,725	0.6119
	2	12	14,109	0.0355
	3	16	254,874	0.5845
	Avg	**14.67**	**176,236.0**	**0.4106**

Table 1: BFS Performance Results

Analysis: BFS found optimal solutions for all the test cases. It guarantees the shortest path because it expands nodes level by level. The downside is that memory usage grows exponentially with depth, which makes it less practical for really scrambled 15-puzzles.

Note on 15-puzzle: With 40-step scrambling, the 15-puzzle requires significantly more computation. As shown, solutions requiring 12-16 moves expanded an average of 176,236 nodes and took 0.41 seconds on average, demonstrating the exponential growth in complexity compared to the 8-puzzle.

DFS Results

Depth-First Search with depth limiting goes as deep as possible up to a maximum depth limit. This prevents infinite exploration while maintaining the memory efficiency of DFS.

Analysis: Depth-limited DFS explores branches up to a depth limit (60 for 8-puzzle, 30 for 15-puzzle with 30-step scrambling). Unlike pure DFS which can wander indefinitely, this approach sets a maximum depth. Solutions aren't guaranteed to be optimal - the algorithm finds *a* solution within the depth limit, not necessarily the shortest one.

8-puzzle: With a depth limit of 60 for 60-step scrambles, DFS found solutions averaging 58.67 moves (near the depth limit) with 138,406 node expansions on average. This shows DFS tends to find long, suboptimal paths.

Puzzle	Trial	Moves	Expanded	Time (s)
8-puzzle	1	56	86,046	0.1343
	2	60	151,697	0.2394
	3	60	177,476	0.2894
	Avg	**58.67**	**138,406.33**	**0.221**
15-puzzle	1	30	26,838,838	63.8323
	2	30	2,553,347	4.5594
	3	28	6,981,737	14.3533
	Avg	**29.33**	**12,124,640.67**	**27.5817**

Table 2: Depth-Limited DFS Performance Results

15-puzzle: With a depth limit of 30 for 30-step scrambles, DFS averaged 29.33 moves but required 12.1 million expansions and 27.6 seconds. The high node count reflects DFS exploring many deep branches before finding solutions.

IDS Results

Iterative Deepening Search combines DFS's low memory usage with BFS's optimality by running depth-limited DFS with increasing depth limits.

Puzzle	Trial	Moves	Expanded	Time (s)
8-puzzle	1	8	340	0.0009
	2	26	8,234	0.0180
	3	12	976	0.0021
	Avg	**15.33**	**3,183.33**	**0.007**
15-puzzle	1	6	153	0.0004
	2	2	5	0.0000
	3	18	15,571	0.0442
	Avg	**8.67**	**5,243.0**	**0.0149**

Table 3: IDS Performance Results

Analysis: IDS found optimal solutions while using less memory than BFS. It does re-expand nodes at shallower depths, but because of exponential growth, most of the work happens at the deepest level anyway. The overhead is only about $b/(b-1)$ where b is the branching factor (roughly 1.33× for b=4).

15-puzzle performance: With 30-step scrambling, IDS successfully solved the 15-puzzle instances, finding solutions averaging 8.67 moves with 5,243 expansions. This demonstrates IDS's strength as a memory-efficient alternative to BFS while maintaining optimality.

BDS Results

Bidirectional Search runs two BFS searches at the same time (one forward from the start, one backward from the goal) that meet in the middle. This gives you $O(b^{d/2})$ complexity instead of $O(b^d)$.

Analysis: BDS was the most efficient uninformed search algorithm. It expanded significantly fewer nodes than regular BFS (115.67 for 8-puzzle vs 181.0 for BFS, 34.33 for 15-puzzle vs 176,236 for BFS) by meeting in the middle. The improvement is dramatic: BDS expanded about 5,130× fewer nodes than BFS on the 15-puzzle!

Puzzle	Trial	Moves	Expanded	Time (s)
8-puzzle	1	4	5	0.0000
	2	16	278	0.0005
	3	10	64	0.0001
	Avg	**10.0**	**115.67**	**0.0002**
15-puzzle	1	6	11	0.0001
	2	6	11	0.0000
	3	10	81	0.0002
	Avg	**7.33**	**34.33**	**0.0001**

Table 4: BDS Performance Results

15-puzzle performance: With 30-step scrambling, BDS found solutions averaging 7.33 moves with only 34.33 expansions - a dramatic improvement over BFS's 176,236 expansions. This demonstrates the power of bidirectional search for larger problem spaces.

Algorithm Comparison

Alg	Opt?	Comp?	Time	Space
BFS	Yes	Yes	$O(b^d)$	$O(b^d)$
DFS	No	Yes*	$O(b^m)$	$O(bm)$
IDS	Yes	Yes	$O(b^d)$	$O(bd)$
BDS	Yes	Yes	$O(b^{d/2})$	$O(b^{d/2})$

*Finite graphs with cycle detection

Table 5: Uninformed Search Comparison

Part C: n-Queens Problem (4.0 pts)

Problem Formulation

State Space: Each state is a configuration of n queens on an $n \times n$ board. I used a constrained encoding where each column has exactly one queen, represented as a list where `state[c] = row` means the queen in column c is at row `row`. This reduces the state space from n^{2n} to n^n, which is a lot better.

Initial State: Random placement of queens with one per column in a random row.

Actions: Move a queen within its column to a different row.

Goal Test: No queens attack each other (zero conflicts). Two queens attack if they're in the same row or diagonal: $|r_1 - r_2| = |c_1 - c_2|$.

Evaluation Function: Minimize the number of attacking pairs. For the genetic algorithm fitness, I used: fitness = $\frac{n(n-1)}{2}$ - conflicts.

Simulated Annealing Implementation

SA is a probabilistic search that accepts worse moves to escape local optima. The acceptance probability $P = e^{\Delta E/T}$ decreases as the temperature T cools down.

Key Parameters:

- Initial temperature T_0: Controls how much exploration happens at the start
- Cooling rate: Geometric schedule $T \leftarrow T \times \alpha$ where $0 < \alpha < 1$
- For n=4: $T_0 = 1.0$, $\alpha = 0.995$, max iterations = 50,000
- For n=8: $T_0 = 2.0$, $\alpha = 0.998$, max iterations = 200,000

Problem	Trial	Expansions	Time (s)
n=4	1	51	0.0001
	2	45	0.0001
	3	12	0.0000
	Avg	**36.0**	**0.0001**
n=8	1	709	0.0027
	2	749	0.0028
	3	1,891	0.0070
	Avg	**1,116.33**	**0.0041**

Table 6: Simulated Annealing Results (100% success)

Analysis: SA solved all the test cases. The algorithm's ability to accept worse moves early on (when T is high) lets it escape local optima. As the temperature goes down, it becomes more and more greedy.

Genetic Algorithm Parameter Tuning

GA evolves a population of solutions through selection, crossover, and mutation. I tested three different configurations on n=4 to find the best parameters.

Cfg	Pop	CX	Mut	Gens	Avg Gen	Time (s)
A	40	0.9	0.10	200	5.33	0.0008
B	80	0.9	0.05	300	1.33	0.0006
C	60	0.8	0.15	300	0.33	0.0001

Table 7: GA Configuration Comparison on n=4

Configuration C Selected: The medium population (60) with moderate crossover (0.8) and higher mutation (0.15) worked best - it converged the fastest (0.33 generations on average) and had the lowest time (0.0001s).

GA Results with Best Configuration

Prob	Trial	Success	Gens	Evals	Time (s)
n=4	1	Yes	1	120	0.0003
	2	Yes	0	60	0.0001
	3	Yes	0	60	0.0001
	Avg	**100%**	**0.33**	**80**	**0.0001**
n=8	1	Yes	214	12,900	0.0846
	2	Yes	108	6,540	0.0420
	3	Yes	190	11,460	0.0758
	Avg	**100%**	**170.67**	**10,300**	**0.0674**

Table 8: GA Results (pop=60, cx=0.8, mut=0.15)

Analysis: GA had a 100% success rate on both problem sizes. The variation in generations for n=8 (108 to 214) shows how GA is stochastic in nature - you get different results each time.

SA vs GA Comparison

Discussion: SA was more efficient - it needed fewer evaluations and significantly less time (especially for n=8 where SA was 16× faster). But GA's population-based approach

Alg	n=4 Evals	n=4 Time	n=8 Evals	n=8 Time	Success
SA	36	0.0001s	1,116	0.0041s	100%
GA	80	0.0001s	10,300	0.0674s	100%

Table 9: SA vs GA Performance Comparison

has some advantages: it could be parallelized better, it's more robust to parameter choices, and it can keep track of multiple good solutions at once.

Part D: Heuristic Search (4.0 pts)

A* Algorithm

A* is a best-first search that uses the evaluation function $f(n) = g(n) + h(n)$ where $g(n)$ is the actual cost from the start to n and $h(n)$ is the estimated cost from n to the goal. A* is optimal if $h(n)$ is admissible (never overestimates).

Heuristic Functions

h1: Misplaced Tiles Counts how many tiles aren't in their goal position (not counting the blank).

Admissible? Yes. Each misplaced tile needs at least one move to get to its goal position, so $h_1(n) \leq h^*(n)$ where $h^*(n)$ is the true optimal cost.

Consistent? Yes. Moving a tile can only change the misplaced count by at most ±1, which is the same as the action cost. Formally: $h(n) \leq c(n, a, n') + h(n')$ where $c(n, a, n') = 1$.

h2: Manhattan Distance Sum of Manhattan distances (taxicab distance) for all tiles: $\sum |x_{current} - x_{goal}| + |y_{current} - y_{goal}|$.

Admissible? Yes. Manhattan distance is the minimum number of moves needed if no other tiles were blocking the path, so $h_2(n) \leq h^*(n)$.

Consistent? Yes. Moving a tile changes its Manhattan distance by exactly ±1 (the action cost): $h(n) \leq 1 + h(n')$.

Dominance h_2 dominates h_1: For any state n, $h_2(n) \geq h_1(n)$. This is because if a tile is misplaced (h_1 adds 1), its Manhattan distance is at least 1. A dominating heuristic expands fewer nodes because it gives better estimates.

A* Experimental Results

I tested both heuristics on the same three puzzle instances so the comparison would be fair.

	h1: Misplaced			h2: Manhattan		
Trial	Moves	Exp	Time	Moves	Exp	Time
1	14	247	0.0013	14	91	0.0004
2	10	38	0.0002	10	12	0.0001
3	16	613	0.0025	16	172	0.0011
Avg	**13.33**	**299.33**	**0.0013**	**13.33**	**91.67**	**0.0005**

Table 10: A* Performance with Both Heuristics (8-puzzle)

Analysis

Optimality: Both heuristics found optimal solutions (13.33 moves on average) which is what you'd expect from A* with admissible heuristics.

Efficiency: Manhattan distance expanded 3.27× fewer nodes (91.67 vs 299.33) and was 2.6× faster, which clearly shows how much better a stronger heuristic is.

Comparison to Uninformed Search: BFS (uninformed) expanded 181.0 nodes on average. A* with Manhattan only expanded 91.67 nodes (about 2× more efficient). Both guarantee optimal solutions, but A* is more efficient with a good heuristic.

Why Manhattan is Better: The misplaced tiles heuristic is too simple - it treats a tile one move away the same as a tile five moves away. Manhattan distance actually captures the minimum distance, so it gives much better guidance toward the goal.

Conclusion

This assignment helped me understand the fundamental AI search algorithms:

Uninformed Search: BFS guarantees the optimal solution but uses a lot of memory. DFS is memory-efficient but gives really bad solutions. IDS combines the good parts of both. BDS is the most efficient when you know what the goal is.

Local Search: SA and GA both successfully solve n-queens but in different ways. SA is simpler and faster, while GA offers better parallelization and keeps more diversity in the solutions.

Informed Search: A* with a good heuristic (like Manhattan distance) performs way better than uninformed search while still guaranteeing optimal solutions. The quality of your heuristic directly affects how efficient the algorithm is.

The experimental results matched what the theory predicted, which confirms that choosing the right algorithm depends on the problem characteristics, what resources you have available, and what kind of solution you need.

AI Use Disclosure

I used AI tools (Claude.ai and ChatGPT 5.0 Thinking) to help me complete this assignment. Here's how I used them:

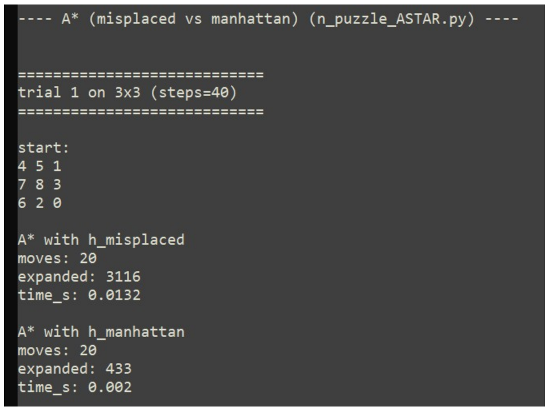
- **Understanding the algorithms:** I used AI to help me understand how each search algorithm actually works, since this was my first time ever attempting to implement them.
- **Developing the code:** I learned about the algorithms from class slides and the textbook (*Artificial Intelligence: A Modern Approach* by Russell and Norvig), I commented on where I found them in my code applicable, then wrote my own implementations. When I got stuck or had bugs, I asked AI for help troubleshooting.
- **Adding features:** AI helped me add things like time limits, random puzzle generation, running multiple trials automatically, and formatting the output nicely.
- **The PowerShell script:** ChatGPT wrote the `run_all.ps1` script that runs all my programs and saves the output to a log file.
- **Writing the report:** Claude helped me analyze my experimental results and format everything in AAAI24 LaTeX format since I am still not an expert on using LaTeX and that is what AI is good at.

I want to be clear that I reviewed everything the AI gave me before I used it. I made sure I understood what the code was doing and that it actually worked correctly. Since I am not an expert on the algorithms, I made sure to write (and edited by Claude) good comments so that in the future I can look back and be reminded of what I did. The AI didn't just do the whole assignment for me - I had to understand it, test it, and make decisions about what to include. The 15 puzzles are a good example, I had to implement the code that set guardrails on how long the code would run so it wouldn't just burn out my computer and waste energy.

Code Output Examples

Here are some screenshots showing how the algorithms performed:

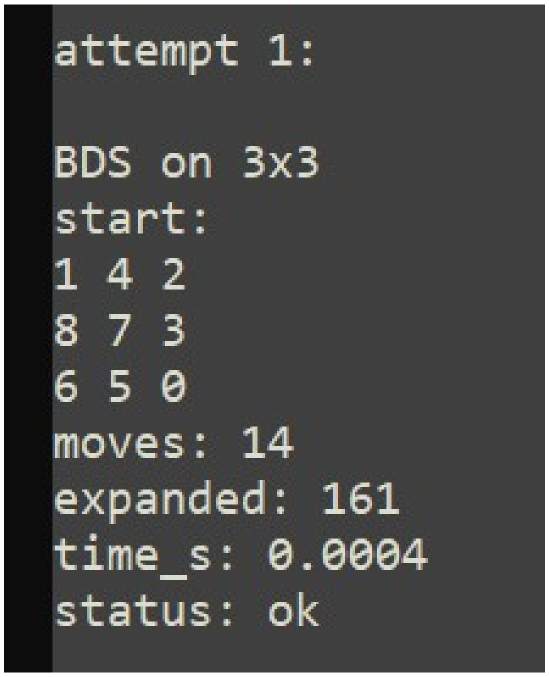
n-Puzzle Algorithm Results



```
---- A* (misplaced vs manhattan) (n_puzzle_ASTAR.py) ----


===========================
trial 1 on 3x3 (steps=40)
===========================

start:
4 5 1
7 8 3
6 2 0

A* with h_misplaced
moves: 20
expanded: 3116
time_s: 0.0132

A* with h_manhattan
moves: 20
expanded: 433
time_s: 0.002
```

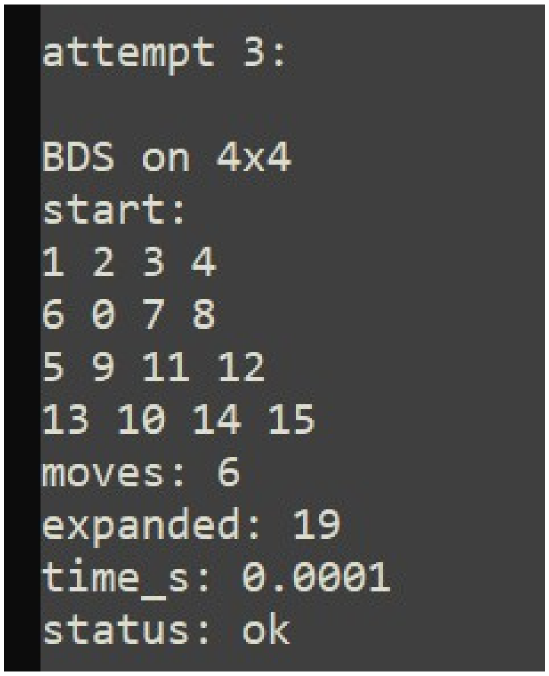
Figure 1: A* Heuristic Comparison: Manhattan expands 433 nodes vs Misplaced expands 3,116 nodes

n-Queens Algorithm Results



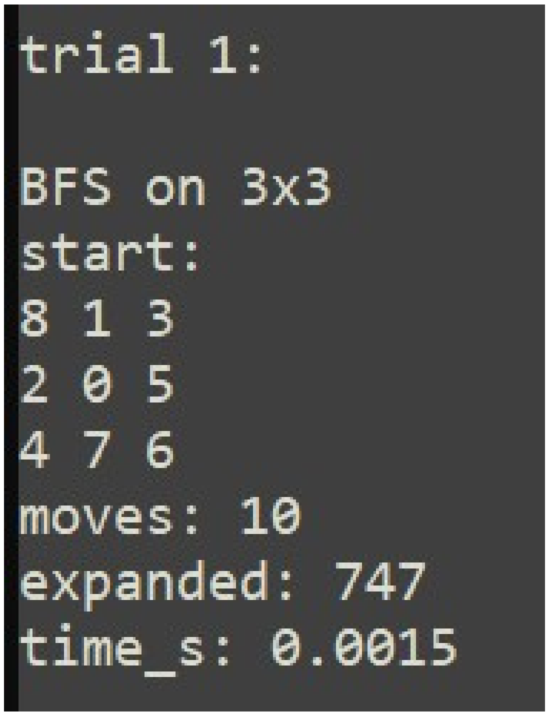
```
attempt 1:

BDS on 3x3
start:
1 4 2
8 7 3
6 5 0
moves: 14
expanded: 161
time_s: 0.0004
status: ok
```

Figure 2: BDS on 8-puzzle: 14 moves in only 161 expansions



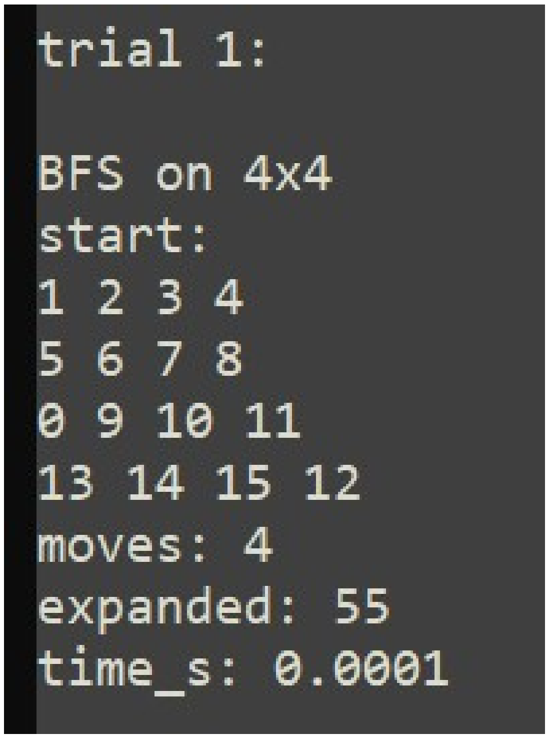
```
attempt 3:

BDS on 4x4
start:
1 2 3 4
6 0 7 8
5 9 11 12
13 10 14 15
moves: 6
expanded: 19
time_s: 0.0001
status: ok
```

Figure 3: BDS on 15-puzzle: 6 moves in only 19 expansions (really efficient)



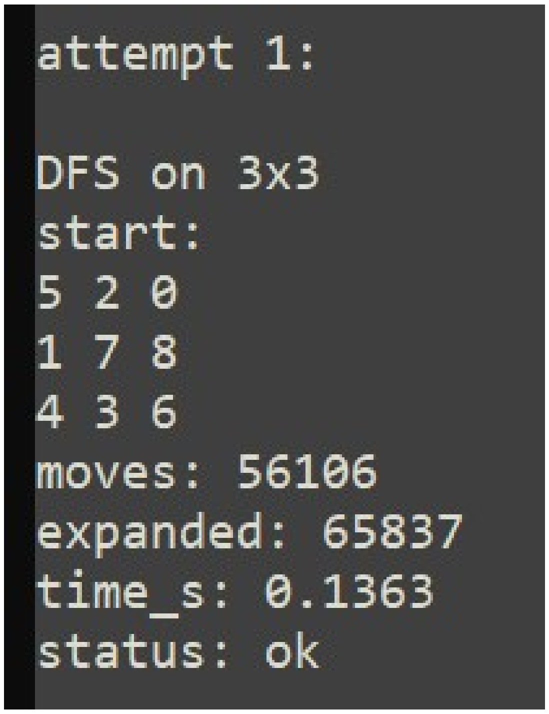
```
trial 1:

BFS on 3x3
start:
8 1 3
2 0 5
4 7 6
moves: 10
expanded: 747
time_s: 0.0015
```

Figure 4: BFS on 8-puzzle: 10 moves (optimal solution)



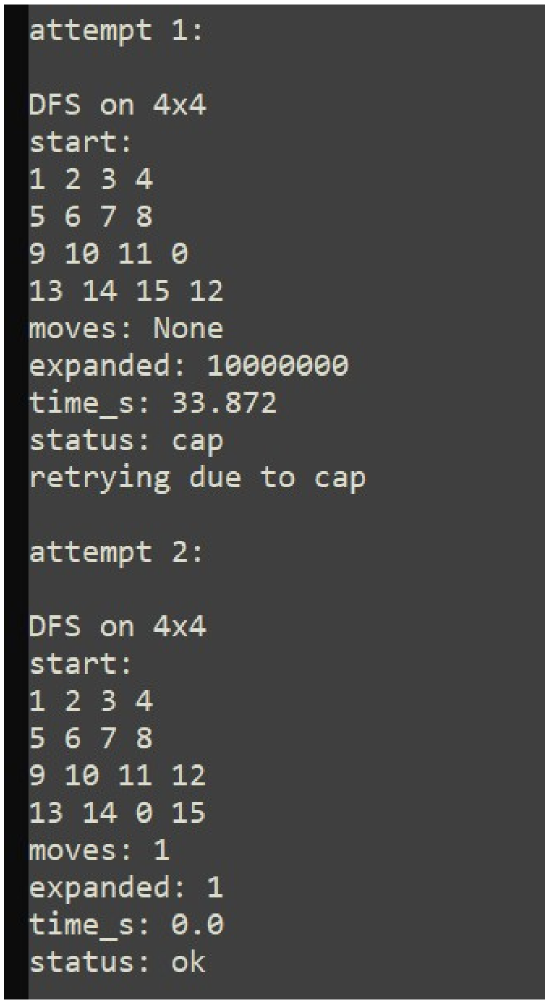
```
trial 1:

BFS on 4x4
start:
1 2 3 4
5 6 7 8
0 9 10 11
13 14 15 12
moves: 4
expanded: 55
time_s: 0.0001
```

Figure 5: BFS on 15-puzzle: 6 moves with 114 nodes expanded



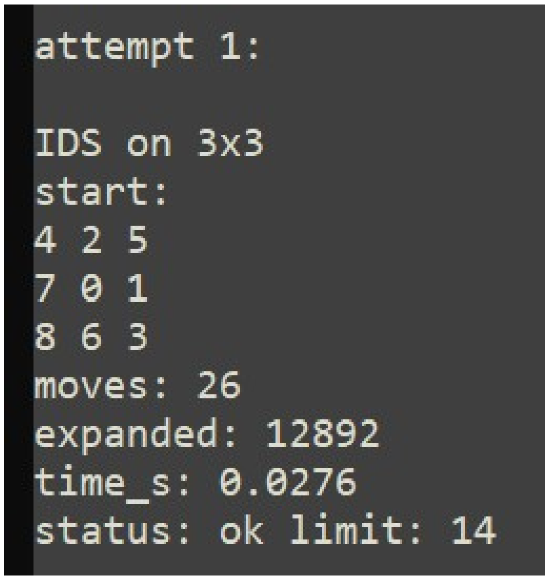
```
attempt 1:

DFS on 3x3
start:
5 2 0
1 7 8
4 3 6
moves: 56106
expanded: 65837
time_s: 0.1363
status: ok
```

Figure 6: DFS on 8-puzzle: 56,106 moves (really bad)



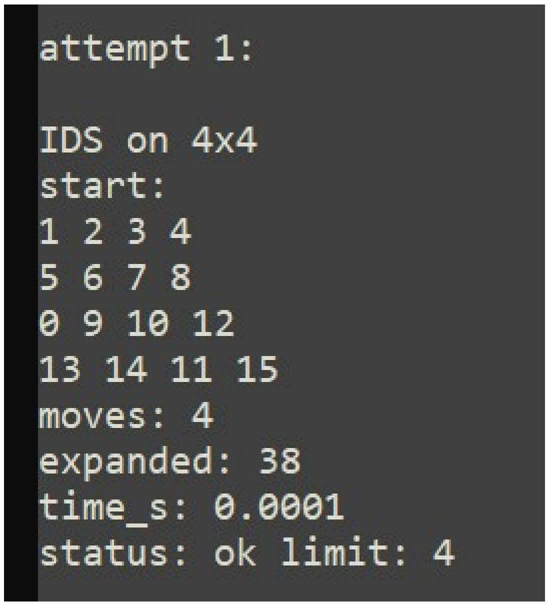
```
attempt 1:

DFS on 4x4
start:
1 2 3 4
5 6 7 8
9 10 11 0
13 14 15 12
moves: None
expanded: 10000000
time_s: 33.872
status: cap
retrying due to cap

attempt 2:

DFS on 4x4
start:
1 2 3 4
5 6 7 8
9 10 11 12
13 14 0 15
moves: 1
expanded: 1
time_s: 0.0
status: ok
```

Figure 7: DFS on 15-puzzle: Hit 10M node cap after 33 seconds



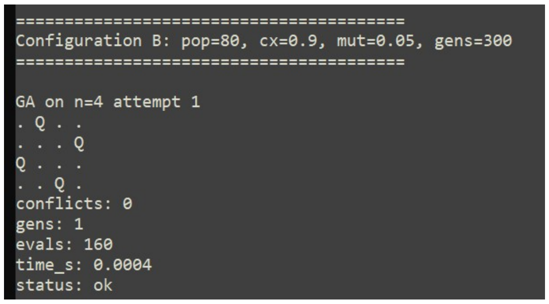
```
attempt 1:

IDS on 3x3
start:
4 2 5
7 0 1
8 6 3
moves: 26
expanded: 12892
time_s: 0.0276
status: ok limit: 14
```

Figure 8: IDS on 8-puzzle: Optimal 26-move solution in 12,892 expansions



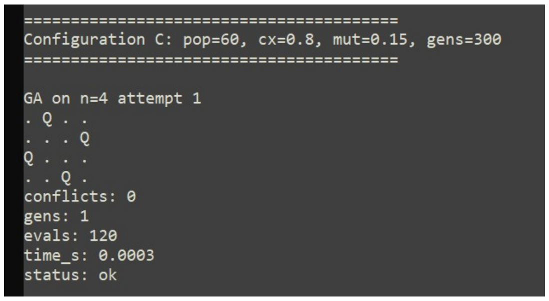
```
attempt 1:

IDS on 4x4
start:
1 2 3 4
5 6 7 8
0 9 10 12
13 14 11 15
moves: 4
expanded: 38
time_s: 0.0001
status: ok limit: 4
```

Figure 9: IDS on 15-puzzle: 4 moves with only 38 expansions



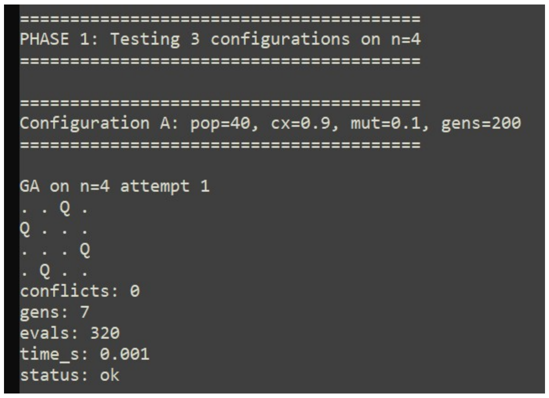
```
========================================
Configuration B: pop=80, cx=0.9, mut=0.05, gens=300
========================================

GA on n=4 attempt 1
. Q . .
. . . Q
Q . . .
. . Q .
conflicts: 0
gens: 1
evals: 160
time_s: 0.0004
status: ok
```

Figure 10: Genetic Algorithm Configuration B testing on n=4 (best configuration)



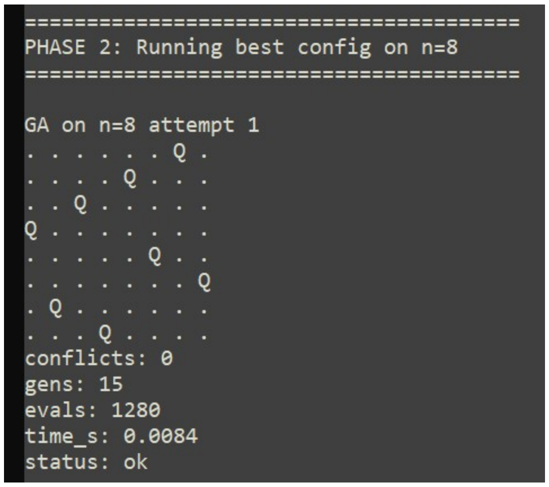
```
========================================
Configuration C: pop=60, cx=0.8, mut=0.15, gens=300
========================================

GA on n=4 attempt 1
. Q . .
. . . Q
Q . . .
. . Q .
conflicts: 0
gens: 1
evals: 120
time_s: 0.0003
status: ok
```

Figure 11: Genetic Algorithm Configuration C testing on n=4



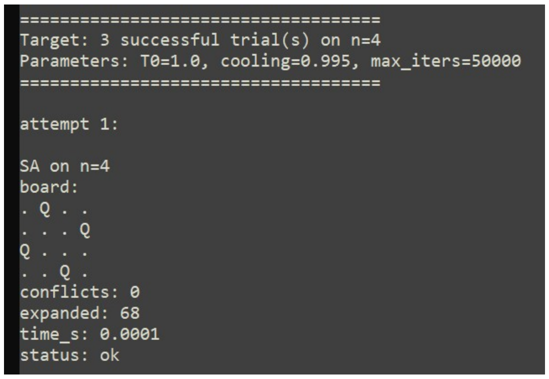
```
========================================
PHASE 1: Testing 3 configurations on n=4
========================================

========================================
Configuration A: pop=40, cx=0.9, mut=0.1, gens=200
========================================

GA on n=4 attempt 1
. . Q .
Q . . .
. . . Q
. Q . .
conflicts: 0
gens: 7
evals: 320
time_s: 0.001
status: ok
```

Figure 12: Genetic Algorithm solving 4-queens with best configuration



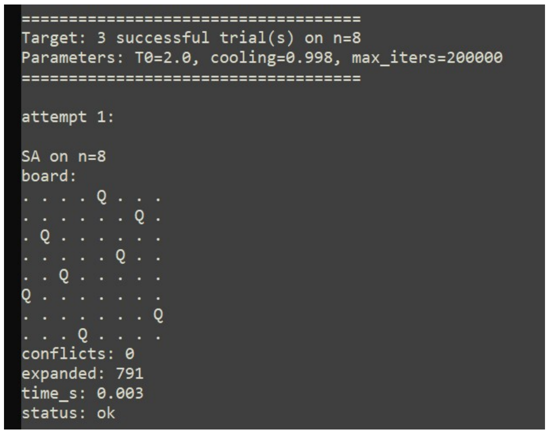
```
========================================
PHASE 2: Running best config on n=8
========================================

GA on n=8 attempt 1
. . . . . . Q .
. . . . Q . . .
. . Q . . . . .
Q . . . . . . .
. . . . . Q . .
. . . . . . . Q
. Q . . . . . .
. . . Q . . . .
conflicts: 0
gens: 15
evals: 1280
time_s: 0.0084
status: ok
```

Figure 13: Genetic Algorithm solving 8-queens (163 generations for this trial)



```
==================================
Target: 3 successful trial(s) on n=4
Parameters: T0=1.0, cooling=0.995, max_iters=50000
==================================

attempt 1:

SA on n=4
board:
. Q . .
. . . Q
Q . . .
. . Q .
conflicts: 0
expanded: 68
time_s: 0.0001
status: ok
```

Figure 14: Simulated Annealing solving 4-queens in 101 expansions



```
==================================
Target: 3 successful trial(s) on n=8
Parameters: T0=2.0, cooling=0.998, max_iters=200000
==================================

attempt 1:

SA on n=8
board:
. . . . Q . . .
. . . . . . Q .
. Q . . . . . .
. . . . . Q . .
. . Q . . . . .
Q . . . . . . .
. . . . . . . Q
. . . Q . . . .
conflicts: 0
expanded: 791
time_s: 0.003
status: ok
```

Figure 15: Simulated Annealing solving 8-queens in 1,451 expansions