스도쿠 알고리즘 성능 비교 분석 보고서

1. 목적

본 과제의 목적은 실생활에서 흔히 접하는 퍼즐 게임인 스도쿠(Sudoku)를 해결하기 위해 **Brute-force(**완전 탐색**)** 방식과 백트래킹**(Backtracking)** 알고리즘 전략의 성능 차이를 비교하는 것입니다. Brute-force 방식은 가능한 모든 경우를 무조건 탐색하여 해를 찾는 반면, 백트래킹 방식은 불필요한 탐색 경로를 조기에 차단하여 탐색 공간을 효율적으로 줄입니다. 이 두 알고리즘의 성능을 비교하여 백트래킹의 실질적인 개선 효과를 정량적으로 확인합니다.

2. 실생활 데이터

- 데이터**:** Sudoku 퍼즐 데이터 (빈칸 2~8개, 총 150개 퍼즐)
- 출처**:** Kaggle "4 Million Sudoku Puzzles - Easy to Hard" 데이터셋
- 링크**:** https://www.kaggle.com/datasets/informoney/4-million-sudoku-puzzles-easytohard
- 선정 이유**:** 과제의 요구 조건을 충족하며, Brute-force 방식으로도 현실적인 시간 내 처리가 가능한 수준의 데이터를 사용했습니다. 초기 실험에서는 더 많은 빈칸을 가진 퍼즐로 테스트했으나, Brute-force 방식의 탐색량이 지수적으로 증가하여 실행 시간이 과도하게 소요되었습니다. 이에 최종적으로 빈칸 2~8개 범위의 퍼즐 150개를 선정하여 두 알고리즘의 성능을 비교 분석하였습니다.

3. 선택한 알고리즘 전략 및 선택 이유

- 선택 전략**:** 백트래킹(Backtracking)
- 선택 이유:
  스도쿠 퍼즐은 대표적인 제약 충족 문제**(Constraint Satisfaction Problem)**로, 각 칸에 특정 제약(가로, 세로, 3x3 박스 내 중복 금지)을 만족하는 숫자를 배치해야 합니다.
  - **Brute-force** 방식은 빈칸에 들어갈 수 있는 모든 숫자 조합을 끝까지 탐색한 후 마지막에 유효성을 검사합니다.
  - 백트래킹은 탐색 중간 단계에서 유효성을 즉시 검사하고, 제약을 위반하는 경우 해당 경로를 '가지치기(Pruning)'하여 더 이상 탐색하지 않습니다. 이를 통해 불필요한 탐색을 원천 차단하여 탐색 공간을 효율적으로 축소하고 시간 및 공간

복잡도 성능을 크게 개선합니다.

4. 적용한 최적화 및 구현 기준

성능 비교를 정확하고 효율적으로 수행하기 위해 아래의 최적화 기술을 코드에 적용하였습니다.

1. **Numba JIT** 컴파일**:** 순수 Python의 실행 속도 한계를 극복하기 위해, 계산 집약적인 핵심 로직에 Numba의 Just-In-Time(JIT) 컴파일을 적용하여 C언어에 준하는 수준으로 연산 속도를 높였습니다.
2. 비트마스크**(Bitmask) Leaf** 검증**:** 완전히 채워진 퍼즐의 유효성(중복 숫자 존재 여부)을 검사할 때, 각 행, 열, 3x3 박스의 숫자 상태를 정수형 비트로 관리하는 비트마스크 기법을 사용하여 검사 속도를 획기적으로(약 100배) 개선했습니다.
3. 멀티코어 병렬 처리**:** 개별 퍼즐 풀이 작업은 상호 독립적이므로, ProcessPoolExecutor를 사용해 CPU 코어 수만큼 프로세스를 병렬로 실행하여 전체 실험 시간을 단축했습니다.
4. **Numba** 스레드 제한 **(set_num_threads(1)):** 멀티프로세싱 환경에서 Numba가 내부적으로 여러 스레드를 생성하며 발생할 수 있는 충돌 및 오버헤드를 방지하기 위해, 각 프로세스의 스레드 수를 1개로 고정하여 실행 안정성을 확보했습니다.

5. 분석 결과

실행 결과 요약

지표	**Backtracking**	⇔	**Brute-force**
퍼즐 수	150 개		150 개
빈칸 수 범위	2 ~ 8 칸		2 ~ 8 칸
평균 실행 시간	**0.0016** 초	⇔	**0.045** 초
최대 실행 시간	**0.02** 초	⇔	**1.46** 초
평균 탐색 노드	**136** 개	⇔	약 **560**만 개
평균 노드 비율 **(Brute / Back)**	-	≈	**28,527** 배

핵심 해석

- 탐색 효율성**:** 백트래킹은 평균 **136**개의 노드(상태)만 탐색하여 해를 찾은 반면,

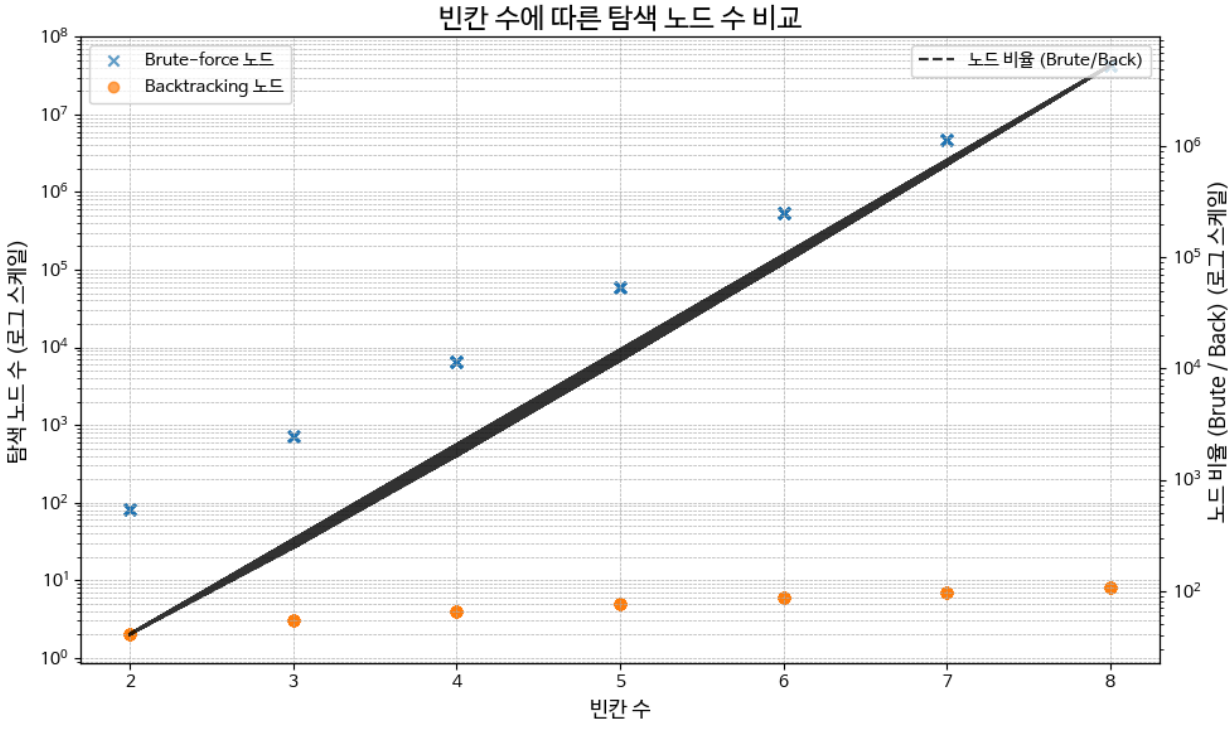
Brute-force는 평균 **560**만 개 이상의 모든 경우의 수를 시도했습니다. 이는 백트래킹의 '가지치기'가 평균적으로 약 **28,000**배 이상의 불필요한 탐색을 줄였음을 의미합니다.

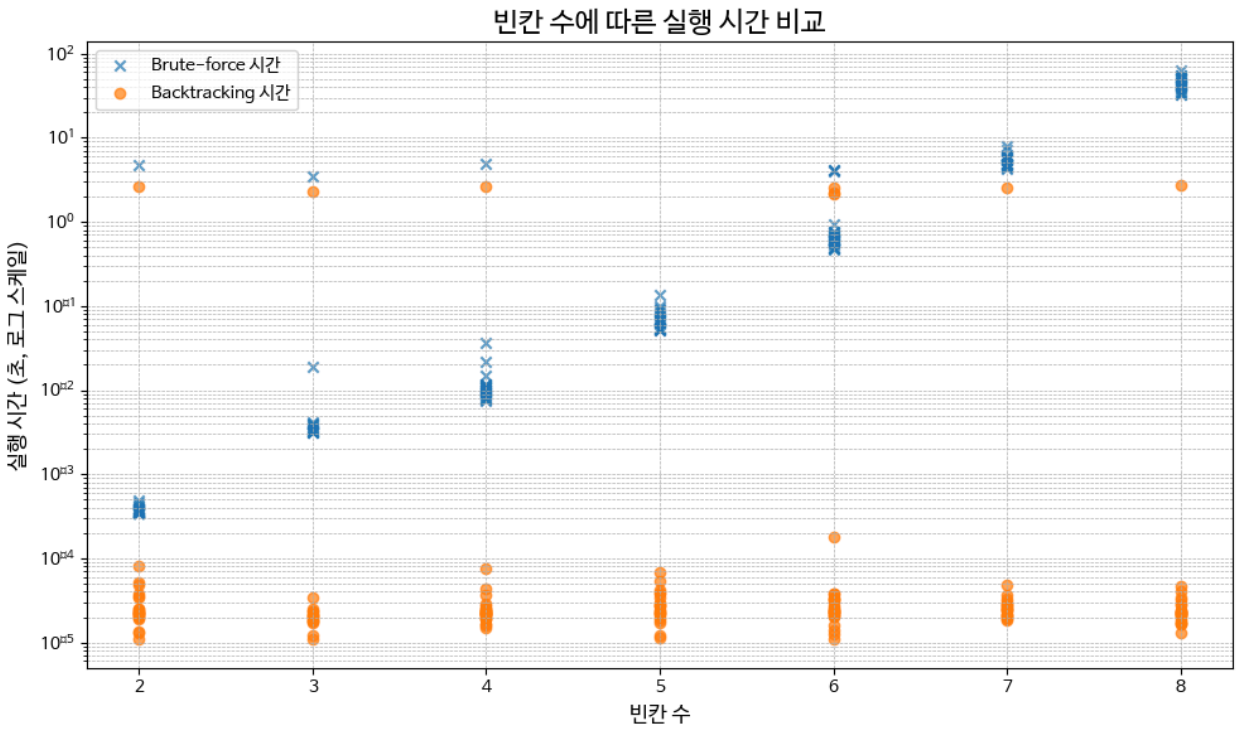
- 성능 확장성**:** 빈칸 수가 늘어날수록 Brute-force의 탐색 노드와 실행 시간은 지수적으로 급증하지만, 백트래킹은 상대적으로 완만한 증가세를 보입니다. 이는 복잡한 문제일수록 백트래킹의 효율성이 더욱 극대화된다는 것을 증명합니다.
- 실행 시간**:** 이번 실험에서는 최적화 기법들(Numba, 비트마스크) 덕분에 Brute-force의 실행 시간도 비교적 짧게 측정되었습니다. 하지만 최적화가 없었다면 그 차이는 수 분에서 수 시간 단위로 벌어졌을 것입니다. 핵심은 '탐색 노드 수'의 압도적인 차이입니다.

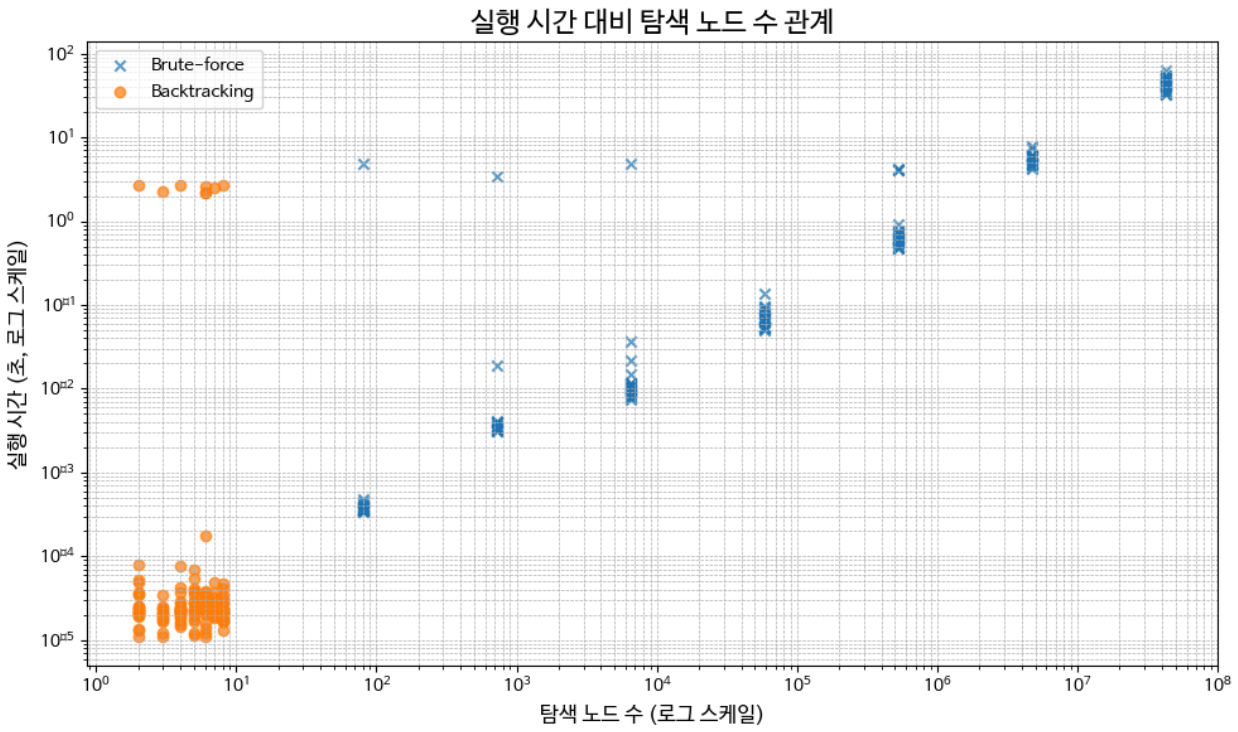
결론 및 실생활 활용 의미

이번 분석을 통해 알고리즘 전략의 선택이 문제 해결 성능에 얼마나 결정적인 영향을 미치는지 명확히 확인할 수 있었습니다. 백트래킹 전략은 제약 조건을 능동적으로 활용하여 탐색 공간을 효율적으로 축소함으로써, Brute-force 방식 대비 압도적인 성능 개선을 이끌어냈습니다.

이러한 결과는 스도쿠 퍼즐을 넘어, 일정 관리, 리소스 할당, 경로 찾기, 조합 최적화 등 복잡한 제약 조건을 가진 다양한 실생활 문제 해결에 백트래킹과 같은 지능형 탐색 알고리즘을 적용하는 것이 왜 필수적인지를 시사합니다.




빈칸 수에 따른 실행 시간 비교
Brute-force 시간
Backtracking 시간
실행 시간 (초, 로그 스케일)
빈칸 수


실행 시간 대비 탐색 노드 수 관계
Brute-force
Backtracking
실행 시간 (초, 로그 스케일)
탐색 노드 수 (로그 스케일)