8PUZZLE 탐색 알고리즘 구현

컴퓨터인공지능학부 202323007 이진선

1. 구현 설명 및 수행 결과

- 각 탐색 알고리즘의 구현 방식에 대한 설명 및 수행 시에 나온 결과 정리

2. 수행 결과 비교 및 분석

- 각 수행 결과에 대한 알고리즘의 성능 비교 및 분석

3. 여러 상황에서의 비교

1. 이동시킨 횟수에 따른 비교 분석
2. 차이나는 퍼즐의 개수에 따른 비교 분석
3. 퍼즐의 크기를 증가시켰을 경우에 변화되는 결과에 대한 비교 분석

4. 알고리즘의 문제점 분석 및 해결방안, 그에 따른 실험 결과

- 각 알고리즘의 문제점에 대해서 분석하고 알고리즘이 가장 최적의 방향으로 실행될 수 있는 방법에 대한 해결방안 제시
- 해결 방안을 직접 적용시키고 그에 따른 실험 결과에 대해서 분석 및 가장 최적의 탐색 알고리즘에 대해서 결론 도출

5. 결론

- 비교 분석 정리

1. 구현 설명 및 수행 결과

1. DFS

```python
def depth_first_search(problem):
    start = problem.getStartState()
    node = [(start, "", 0)]
    frontier = [node]
    explored = set()

    while frontier:
        node = frontier.pop()
        state = node[-1][0]
        if problem.isGoalState(state):
            return [x[1] for x in node][1:]
        if state not in explored:
            explored.add(state)
            for successor in problem.getSuccessors(state):
                if successor[0] not in explored:
                    new = node[:]
                    new.append(successor)
                    frontier.append(new)
    return []
```

스택을 활용하여 깊은 경로부터 탐색하는 방법으로 구현하였다. 경로가 탐색될 때까지 탐색을 진행하기 때문에 explored 를 통해서 처음 탐색을 진행한 경우는 set()에 넣어두고 이미 탐색을 한 곳은 pop()을 하고 다음 경우로 넘어가도록 구현하였다. 탐색을 진행하다가 결과에 도달했을 때, 해당 깊이를 출력하는 방식으로 최단 경로를 보장할 수 없다.

2. BFS

```python
def breadth_first_search(problem):
    start = problem.getStartState()
    node = [(start, "", 0)]
    frontier = deque([node])
    explored = set()

    while frontier:
        node = frontier.popleft()
        state = node[-1][0]
        if problem.isGoalState(state):
            return [x[1] for x in node][1:]
        if state not in explored:
            explored.add(state)
            for successor in problem.getSuccessors(state):
                if successor[0] not in explored:
                    new = node[:]
                    new.append(successor)
                    frontier.append(new)
    return []
```

큐를 활용하여 가장 얕은 경로부터 탐색하는 방법으로 구현하였다. 탐색한 경로의 순서대로 해당 경로에서 가능한 방향을 분석하여 탐색한다. Explored 를 통해서 이미 탐색한 경로인지를 확인한다.

거리의 가중치가 있는 경우에는 가장 짧은 거리를 탐색할 수 없지만, 8 퍼즐의 경우 거리의 가중치가 탐색한 횟수(1)로 모두 동일하기 때문에 최단경로를 보장한다.

3. Uniform cost search

```
138  def uniform_cost_search(problem):
139      start = problem.getStartState()
140      node = [(start, "", 0)]
141      frontier = []
142      breaker = 0
143      heapq.heappush(frontier, (0, breaker, node))
144
145      explored = {}
146
147      while frontier:
148          cost, unuse , node = heapq.heappop(frontier)
149          state = node[-1][0]
150          if problem.isGoalState(state):
151              return [x[1] for x in node][1:]
152          if state not in explored or explored[state] > cost :
153              explored[state] = cost
154              for successor in problem.getSuccessors(state):
155                  unuse2, unuse3, next_cost = successor
156                  breaker += 1
157                  new = node[:]
158                  new.append(successor)
159                  print(next_cost)
160                  heapq.heappush(frontier, (cost + next_cost, breaker,  new))
161      return []
```

우선순위 큐를 활용하여 BFS 의 탐색 방식에서 탐색해야 하는 노드들 중 누적 비용이 가장 적은 경로가 먼저 탐색되는 방식으로 작성하였다.

Heapq.heappush(frontier, (cost + next_cost, new))라고 작성할 경우, cost+next_cost 가 동일할 경우에는 new 의 대소비교를 통해서 우선순위를 정하게 되기 때문에 해당 경우를 방지하기 위해서 breaker 를 작성하여 cost 가 동일하더라도 breaker 를 통해서 우선순위를 정할 수 하여 new 에서 우선순위 조건을 탐색하지 않도록 처리하였다.

4. Heuristic

```python
def heuristic(state, problem=None):
    cnt = 0
    if state.cells[0][0] != 1: cnt += 1
    if state.cells[0][1] != 2: cnt += 1
    if state.cells[0][2] != 3: cnt += 1
    if state.cells[1][0] != 4: cnt += 1
    if state.cells[1][1] != 5: cnt += 1
    if state.cells[1][2] != 6: cnt += 1
    if state.cells[2][0] != 7: cnt += 1
    if state.cells[2][1] != 8: cnt += 1
    if state.cells[2][2] != 0: cnt += 1
    return cnt
```

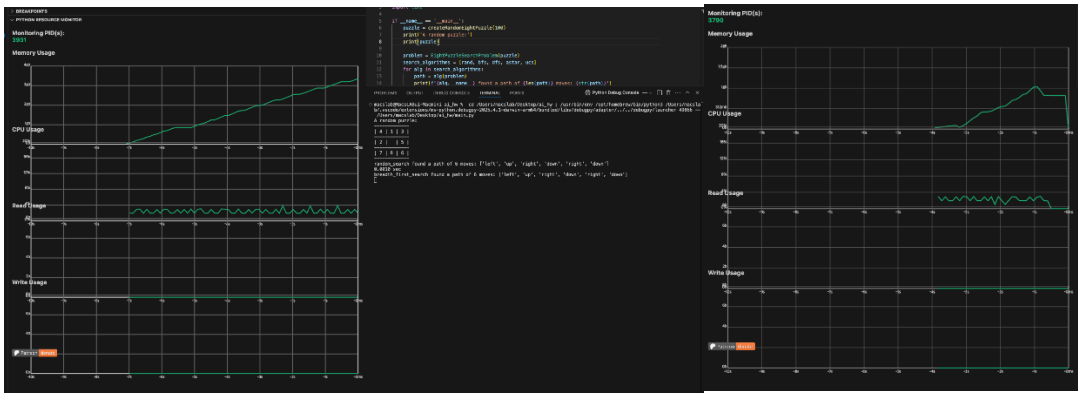
잘못 배치되어 있는 타일의 개수를 통하여 휴리스틱 함수를 구현하였다. 각 타일이 정답위치와 다를 경우에는 개수만큼 cnt 를 1 씩 증가시키도록 만들었다.

5. aStar_search

```python
def aStar_search(problem, heuristic=heuristic):
    start = problem.getStartState()
    node = [(start, "", 0)]
    frontier = []
    breaker = 0

    heuristic_g = 0
    heuristic_f = heuristic_g + heuristic(start,problem)
    heapq.heappush(frontier, (heuristic_f, heuristic_g, breaker, node))
    explored = {}

    while frontier:
        cost_f, cost_g , unuse , node = heapq.heappop(frontier)
        state = node[-1][0]
        if problem.isGoalState(state):
            return [x[1] for x in node][1:]
        if state not in explored or explored[state] > cost_g :
            explored[state] = cost_g
            for successor in problem.getSuccessors(state):
                unuse2, unuse3, next_cost = successor
                new_g = cost_g + next_cost
                new_h = heuristic(unuse2, problem)
                new_f = new_g + new_h

                breaker += 1
                new = node[:]
                new.append(successor)
                heapq.heappush(frontier, (new_f, new_g, breaker,  new))
    return []
```

A*알고리즘은 현재까지 온 거리와 결과에 도달하기 위해서 소요되는 예상 비용(휴리스틱 함수의 결과_잘못 배치되어 있는 타일의 개수)를 비교하여 먼저 탐색할 노드를 선택한다. 우선순위 큐를 활용하여 f(n)이 낮은 경로부터 연산을 하여 최적으로 예상되는 경로부터 탐색을 진행하게 되며, h(n)이 적절할 경우에는 최단 경로를 출력하게 된다.

2. 수행 결과 비교 및 분석



dfs	2.1695	18040
	89.5525	74152
	421.0253	101700
	0.8212	11542
bfs	0.1023	16
	0.0447	14
	0.0010	6
	0.5173	20
Ucs	0.1069	16
	0.0474	14
	0.0010	6
	0.6393	20
A*	0.0064	16
	0.0025	14
	0.0011	6
	0.0736	20

BFS, UCS, A* 알고리즘의 연산속도는 0.1 초 내외로 연산속도가 매우 빠름에 반해 DFS 는 연산속도가 낮았다. 메모리적인 측면에 있어서 두 사진을 참고하였을 때에 DFS 를 연산을 시작하게 되니 메모리 사용량이 급격히 증가함을 파악할 수 있다. 이를 통해서 DFS 가 깊이 우선탐색을 하기 때문에 탐색해야하는 경우의 수를 스택에 저장하여야하고 이로 인해 메모리적인 측면에서 비효율적이라는 점을 파악할 수 있다. DFS 가 연산이 완료되고 난 후 메모리 사용량이 급격하게 줄어들면서 DFS 와 다른 탐색 알고리즘의 메모리 사용량의 차이를 비교할 수 있었다. BFS/ UCS 의 경우, UCS 는 거리가 모두 1 임에도 불구하고 우선순위 큐를 통해서 연산 순서을 정해야 한다. 따라서 BFS 에 비해 연산속도가 느렸다. BFS 의 경우 거리가 1 로 모두 동일하다는 요소로 인해서 언제나 최단경로를 보장하고 있다. 실험 결과, BFS, UCS, A*

알고리즘은 최단 경로를 산출해내어 최단경로를 보장한다. A*알고리즘이 최단 경로를 잘 산출해낸다는 점에 있어서 휴리스틱 함수가 적절하다라고 볼 수 있다. A* 알고리즘은 휴리스틱 함수를 통하여 가장 최단경로로 예상되는 경로부터 탐색하기 때문에 다른 알고리즘보다 좋은 연산속도를 보이고 있다.

3. 여러 상황에서의 비교

1. 이동시킨 횟수에 따른 비교 분석

본 실험은 moves 수가 10, 100, 1000,10000 일 경우로 나누어 성능을 비교하였다.

Moves	Algorithm	avg_time	avg_result	bfs_result	result_error
10	A*	0.00016	5.6	5.6	0
	BFS	0.00108	5.6	5.6	0
	DFS	0.00806	548.4	5.6	542.8
	UCS	0.00092	5.6	5.6	0
100	A*	0.0209	14	14	0
	BFS	0.166325	14	14	0
	DFS	128.392125	51358.5	14	51344.5
	UCS	0.19865	14	14	0
1000	A*	0.4587	23	23	0
	BFS	1.23815	23	23	0
	DFS	45.397325	62499.5	23	62476.5
	UCS	1.81915	23	23	0
10000	A*	0.1061	21.5	21.5	0
	BFS	0.932525	21.5	21.5	0
	DFS	91.470225	55364	21.5	55342.5
	UCS	1.25865	21.5	21.5	0

각 결과에 대해서 분석해보았을 때, 다른 알고리즘에 비해 DFS 가 시간 복잡도가 매우 높게 측정되었다. 경로로 출력된 결과도 다른 알고리즘에 비해 매우 큰 편차를 보이고 있다. 10, 100, 1000 으로 증가할수록 DFS 의 결과와 실제 값에 대한 편차가 더욱 커지는 경향을 확인하였다.

다만, 10000 일 경우에 편차가 하락세를 보이고 있어 편차와 시간의 증가가 moves 의 크기에만 의존하진 않고 실제 퍼즐 상태와 탐색해야 하는 방향에 따라서 요소도 영향을 미친다는 것을 파악할 수 있었다.

UCS 는 BFS 와 비교하였을 때에 moves 수가 증가할수록 시간 복잡도가 급격히 증가하며 BFS 의 탐색 속도와 차이가 벌어진다. 따라서, 우선순위 큐를 통해서 누적 비용을 비교하여 탐색 순서를 정하는 연산이 moves 수의 영향을 받음을 알 수 있었다.

전반적으로 DFS 를 제외한 알고리즘은 최단경로를 잘 탐색할 수 있다는 것을 알 수 있었다.

- 해당 도표는 14 페이지의 결과표에서 계산한 도표이다.

2. 차이나는 퍼즐의 개수에 따른 비교 분석

차이나는 퍼즐의 개수에 따라서 성능이 다르게 분석되는지를 파악하기 위해서 타일의 위치를 임의로 변화시켜 탐색을 진행하였다.

다만, inversion 수가 홀수일 경우에는 본래자리로의 탐색이 불가능하기 때문에 inversion 수가 짝수인 경우에 한하여 실행하였다. 탐색이 불가능한 경우의 예로는 [1,2,3,4,6,5,7,8,0]이 있다. 차이나는 퍼즐의 개수에는 0 도 포함한다.

1) Misplaced tile = 2

[1, 2, 3, 4,5, 6, 7,0,8]	시간	결과
dfs	0.0000	1
bfs	0.0003	1
ucs	0.0000	1
A*	0.0000	1
[1, 2, 3, 0, 5, 6 ,7, 8, 4]	시간	결과
dfs	0.0109	893
bfs	0.0096	11
ucs	0.0102	11
A*	0.0011	11

2) Misplaced tile = 4

[1, 2, 6, 4, 5, 3, 0, 8, 7]	시간	결과
dfs	6.0844	30394
bfs	0.4190	20
ucs	0.5039	20
A*	0.0432	20

[3, 2, 1, 4, 0, 6, 7, 8, 5]	시간	결과
dfs	268.6194	101352
bfs	0.5798	20
ucs	0.6488	20
A*	0.0622	20

2) Misplaced tile = 5

[1, 2, 3, 5, 4, 6, 8, 0, 7]	시간	결과
dfs	24.8526	54523
bfs	0.0276	13
ucs	0.0313	13
A*	0.0022	13

3) misplaced tile = 6

[1, 2, 3, 6, 4, 5, 0, 7, 8]	시간	결과
dfs	2.3385	19992
bfs	0.1002	16
ucs	0.1075	16
A*	0.0072	16
[1, 2, 6, 4, 3, 5, 0, 7, 8]	시간	결과
dfs	11.9874	38174
bfs	0.0142	12
ucs	0.0113	12
A*	0.0011	12

4) misplaced tile = 8

[8, 7, 6, 5, 4, 3, 2, 1, 0]	시간	결과
dfs	32.5819	58288
bfs	2.2348	30
ucs	3.4906	30
A*	2.4851	30
[6, 5, 4, 3, 2, 1, 0, 8, 7]	시간	결과
dfs	28.6073	54832
bfs	2.2182	30
ucs	3.6814	30
A*	2.4456	30

분석 결과

DFS)

실험 결과, 잘못 배치된 타일의 수가 적을 수록 DFS 와 다른 알고리즘의 편차가 줄어드는 걸 파악할 수 있다. 잘못 배치된 타일의 수가 많으면 많을 수록 DFS 의 시간과 경로의 거리(0.0109sec, 893 회 -> 28.6073sec, 54832 회)이 급격히 증가하는 추세를 보이고 있다.

UCS 와 BFS)

잘못 배치된 타일의 수가 많으면 많을수록 BFS 와 UCS 의 연산속도차가 벌어지는 모습을 볼 수 있다. 하지만, [1, 2, 6, 4, 3, 5, 0, 7, 8](잘못 배치된 타일의 수 = 6)의 경우에 반례가 존재했다. 이는 초기에 잘못 배치된 타일의 수 이외에 결과에 영향을 미치는 요소가 있다는 걸 의미한다. 출력된 최단경로 결과와 비교하였을 때, 최단 경로가 커지면 커질수록 연산속도 차가 커진다. 이를 통해, UCS 와 BFS 의 격차가 벌어지는 이유는 잘못 배치된 타일 개수 뿐만 아니라 최단경로 수의 증가가 탐색 속도의 저하에 영향을 미침을 파악할 수 있다.

A*)

A*알고리즘은 잘못된 타일의 수가 많으면 많을 수록 성능이 하락함을 파악할 수 있었다. 이는 초기에 잘못된 타일의 수가 많아 휴리스틱 함수의 영향력이 낮아지기 때문이다. 그에 대한 증거로 잘못 배치된 타일의 수가 적을 경우에 A*알고리즘의 성능속도가 명백하게 빠르다는 것을 파악할 수 있다.

결론)

각 알고리즘이 어떠한 방식으로 작동되는지에 따라서 성능에 영향을 크게 미치는 요소가 다르다는 것을 파악할 수 있다.

3. 퍼즐의 크기를 증가시켰을 경우에 변화되는 결과에 대한 비교 분석

8 퍼즐을 15 퍼즐로 증가시켜 비교 분석을 진행한다.

15 퍼즐로 진행할 경우에 연산 속도가 오래 걸리고 메모리 사용공간이 급격하게 증가하기 때문에 moves 수의 단위를 10, 20 으로 낮추어 진행하였다.

이동할 수 있는 면적이 증가하기 때문에 동일 이동 횟수로 설정하더라도 시간과 메모리 부분에서 큰 차이를 보인다. Moves 수가 10 일 때에는 이동횟수가 적기 때문에 모든 알고리즘이 빠르게 탐색을 완료할 수 있었다. 다만 DFS 에서 값에 따른 편차가 매우 컸다. DFS 는 moves 수에 따른 편차가 매우 컸다. 대표적인 예시로 moves 수가 50 일 때에 5 분이 지나도 탐색을 제대로 하지 못하는 문제가 발생했다. UCS 는 8 퍼즐에서보다 15 퍼즐에서 좋은 성능을 보였다. 규모가 클수록 더욱 좋은 성능을 보이는 것으로 파악되었다. A* 알고리즘은 휴리스틱을 활용해서 최적해를 더욱 빠르게 탐색을 하게 되므로 8 퍼즐보다 경우의 수가 많은 15 퍼즐의 경우 더욱더 좋은 성능을 보였다.

Moves	탐색 종류	시간	결과
10	dfs	0.0000	2
	bfs	0.0007	2
	ucs	0.0002	2
	A*	0.0001	2
20	dfs	16.7440	40030
	bfs	0.0174	8
	ucs	0.0140	8
	A*	0.0002	8

4. 알고리즘의 문제점 분석 및 해결방안, 그에 따른 실험 결과

1. 각 알고리즘의 문제점 분석

- **DFS (Depth-First Search)**

  DFS 는 깊이 우선 탐색 방식이여서 규모가 크고, 잘못 배치되어 있는 이동해야 하는 거리가 멀수록 기하급수적으로 안 좋은 성능을 보였다. 많이 탐색할 수록 메모리에 저장되는 경로도 많아지기 때문에 시간적 측면과 공간적 측면 모두 비효율적이다.

- **BFS (Breadth-First Search)**

  BFS 는 8 퍼즐의 경우 거리가 1 로 모두 동일하기 때문에 최단 경로는 보장한다는 데에 있어 큰 장점을 보였다. 다만, 8 퍼즐에서 15 퍼즐로 증가할 때에 더 많은 경우의 수를 탐색해야 하기 때문에 시간적 측면과 공간적 측면에서 급격히 성능이 낮아졌다.

- **UCS (Uniform Cost Search)**

  UCS 의 경우 비용이 적은 경로부터 탐색하게 되는데 8 퍼즐의 경우 모든 거리가 1 로 동일하기 때문에 성능의 변화에 큰 영향을 미치지 못했다. 더하여 비용이 적은 경로를 먼저 탐색하도록 하는 연산으로 인해서 오히려 BFS 보다 시간적 측면에서 오래 걸렸다.

- **A* (A-Star Search)**

  A* 알고리즘은 휴리스틱 함수(잘못 배치된 타일의 개수)를 통하여 예상비용을 확인하여 연산할 수 있어 BFS 보다 좋은 성능을 보였다. 특히, 잘못 배치된 타일의 개수는 규모가 커질수록 영향력이 커지는데 15 퍼즐에서 유독 좋은 성능을 보였다. 하지만, 잘못 배치된 하나의 타일이 먼 위치에 존재하게 된다면 이동 거리에 대해서 파악할 수 없기 때문에 정확도가 낮아질 수 있다.

2. **해결 방안과 성능 변화**

- **DFS 개선**: 탐색 깊이를 moves 수로 제한

  DFS_limited 를 통해서 moves 수를 제한했을 경우, 연산 속도와 산출되는 결과에서 DFS 에 비해 좋은 성능을 보였다. 특히나 적게는 1/8 에서 많게는 1/20 의 연산속도로 인해 제한을 걸어둘 때에 경로 탐색에서 헤매는 것을 방지할 수 있었다. Limit 에 걸려 나오게 되면 더 많은 경로를 탐색해야 한다는 우려와 달리, 메모리적 측면에서 효율적인 모습을 보여 연산 속도도 빨라 졌음을 파악할 수 있었다. 경로에 대해서도 DFS 는 99092 라고 산출된 데에 반해 1000 이라고 최대 연산에 대해서는 방지할 수 있음을 알 수 있었다.

- **A* 개선**: misplaced tile 이 아닌 현재의 위치와 정답 위치의 거리 차로 설정할 경우

  현재의 위치와 정답 위치의 거리차를 활용하여 휴리스틱 함수를 작성했을 경우, 오히려 좋지 못한 성능을 보였다. 특히 이동 거리가 많을수록 이 편차는 더욱 컸다. 원인에 대해서 생각해보았을 때에, 거리의 차를 연산하는 과정을 통해서 연산속도가 저하됐음을 분석할 수 있었다. 다만, 이동해야 하는 거리를 연산을 통해서 최단경로에 대해서 잘 보장할 수 있을 것으로

예상된다. 다만 이럴 경우에도 하나의 잘못 배치된 타일의 영향력이 크다는 경우를 고려했을 때에 무엇이 더 좋은 효과를 놓을 수 있을지는 분석해야 하는 타일에 따라서 변화될 것으로 파악된다.

Moves = 100)

탐색 종류	시간	결과	시간	결과	시간	결과
dfs	0.2688	6590	8.9002	35128	2.4959	20666
Dfs – limited	0.0249	100	1.1632	100	0.1710	96
Bfs	0.0583	14	1.6893	24	0.3403	18
Ucs	0.0644	14	2.5792	24	0.3910	18
A* – mahatten	0.0030	14	0.3137	24	0.0198	18
A*	0.0010	14	0.0273	24	0.0046	18

Moves = 1000)

탐색 종류	시간	결과	시간	결과	시간	결과
dfs	20.6321	52564	307.9677	99092	3.8761	24178
Dfs – limited	1.9587	484	0.8201	1000	0.2595	1000
Bfs	1.6704	24	0.3370	18	0.5608	20
Ucs	2.2516	24	0.3939	18	0.6574	20
A* – mahatten	0.3312	24	0.0209	18	0.0338	20
A*	0.0329	24	0.0049	18	0.0028	20

5. 결론

다양한 실험을 통해서 각 알고리즘의 성능과 영향을 미치는 요소를 파악하고, 알고리즘의 단점을 파악하여 이를 개선할 수 있는 방안과 실제 적용했을 때의 장점과 문제점에 대해서 살펴보았다.

DFS 는 가능한 경우가 있다면 계속해서 깊게 탐색하는 단순한 알고리즘이지만, 깊이 제한을 두지 않아 연산이 언제까지 진행될지 모른다는 문제점이 있었다. 이 문제점으로 인해서 연산속도가 저하되고 메모리 사용량이 급격하게 증가한다는 문제점이 발생했다. 이는 잘못 배치된 타일이 많거나 움직인 이동량이 많을수록 커졌다. 따라서, 깊이의 제한을 두어 진행해보았을 때, 시간과 메모리적인 측면에서 많이 개선되었다.

BFS 는 최단 경로를 무조건적으로 보장한다는 데에 있어서 좋은 성능을 보였다. 다만, 15 퍼즐로 증가할 경우에 메모리 사용량과 연산 시간이 급격하게 증가한다는 데에 있어서 단점이 존재했다.

UCS 는 우선순위 큐를 활용하여서 현재에서 최단 경로를 먼저 탐색한다는 점에 있어서 장점을 보였지만, 거리의 가중치나 이동 횟수에 대한 가중치가 존재하지 않는다는 점으로 인해서 우선순위 큐를 활용하는 데에 장점이 보이지 않았다. 더하여 우선순위 큐를 사용하는 연산을 더하여야 하기 때문에 오히려 BFS 보다 성능이 낮아졌다.

A* 알고리즘은 잘못 배치된 타일의 수(휴리스틱 함수)를 활용하여서 연산을 하였을 때, 최단 경로를 예상하면서 탐색하기 때문에 다른 알고리즘에 비해서 연산속도 및 메모리 사용량에 있어서 효율적인 모습을 보였다. 이는 규모가 커질 때에 더욱더 효과적인 모습을 보였다. 더 효율적인 연산을 위해서 배치되어야 하는 위치까지의 거리를 합하여 연산하여 성능을 개선하려 해보았지만, 잘못 배치된 타일의 수보다 성능 향상된 모습을 보이지는 못했다.

이와 같이 각 알고리즘은 연산 되는 방식에 따라서 특정 상황에서 장점과 단점을 가진다. 현재의 8 퍼즐에서는 A* 알고리즘이 성능과 정확도 부분을 합했을 때, 가장 좋은 탐색 결과를 도출해냈다.

Moves	Search	Time	Result
10	dfs	0.0120	836
		0.0079	580
		0.0114	830
		0.004	32
		0.0050	464
	bfs	0.0010	6
		0.0002	4
		0.0013	6
		0.0003	4
		0.0026	8
	Ucs	0.0007	6
		0.0002	4
		0.0010	6
		0.0003	4
		0.0024	8
	A*	0.0002	6
		0.0001	4
		0.0002	6
		0.000	4
		0.0003	8
100	dfs	2.1695	18040
		89.5525	74152
		421.0253	101700
		0.8212	11542
	bfs	0.1023	16
		0.0447	14
		0.0010	6
		0.5173	20
	Ucs	0.1069	16
		0.0474	14
		0.0010	6
		0.6393	20
	A*	0.0064	16
		0.0025	14
		0.0011	6
		0.0736	20
1000	dfs	27.2735	54468
		54.4890	71344
		77.3338	72652
		22.4930	51534
	bfs	0.5417	20
		1.9961	26
		2.2117	28

		0.2031	18
	Ucs	0.7228	20
		2.9632	26
		3.3341	28
		0.2565	18
	A*	0.0327	20
		0.5664	26
		1.2213	28
		0.0144	18
10000	dfs	0.5236	9658
		14.9228	45326
		285.1257	95910
		65.3088	70562
	bfs	1.5910	24
		1.0810	22
		0.5730	20
		0.4851	20
	Ucs	2.2042	24
		1.4619	22
		0.7821	20
		0.5864	20
	A*	0.2444	24
		0.0975	22
		0.0462	20
		0.0363	20