CSE3013 (컴퓨터공학 설계 및 실험 I)
PRJ-1 테트리스 프로젝트 3주차 결과 보고서

서강대학교 컴퓨터공학과 박수현 (20181634)

서강대학교 컴퓨터공학과

1 목적

실습 시간에 작성한 알고리즘과 자료구조를 요약해 기술한다. 모든 경우를 고려하는 트리 구조와 가지치기를 한 트리 구조를 비교한다.

2 추천 알고리즘

현재 상태를 루트로 갖는 트리를 만든다. 현재 블록과 다음 블록을 이용해 첫 번째 블록을 놓는 모든 위치와 회전 상태에 대해 자식 노드들을 만든다. 그 자식 노드들에서 두 번째 블록을 놓는 모든 위치와 회전 상태에 대해 자식 노드를 만든다. 그리고 트리를 탐색하면서 가장 높은 점수를 도출하는 위치에 다음 블럭을 놓는다.

간단히 말하면, 추천 알고리즘은 가능한 미래의 상태 공간을 결정 트리decision tree로 만들어 완전 탐색해 최적의 위치를 찾아 블록을 놓는다.

추천 위치를 계산하는 로직은 다음과 같이 세분화된다.

- 결정 트리를 만드는 로직
- 보드 상태 점수를 계산하는 로직
- 결정 트리를 최적화하는 로직

현재 상태를 루트라고 하면, 다음 n개 블록에 대한 높이 $n+1$의 결정 트리를 만들 수 있다. 결정 트리의 각 노드에는 보드 상태와 점수가 포함된다.

공간 복잡도의 개선을 위해 탐색과 가지치기를 진행하면서 결정 트리를 만들어나간다. 가지치기를 하기 위해서는 노드를 정렬할 필요가 있다. 아래는 n개의 노드를 amortized $\mathcal{O}(n \log n)$의 시간에 정렬하는 메서드이다. Swap의 의사 코드는 trivial하므로 기술하지 않는다.

QUICKSORT(*array*, *start*, *end*)

```
1  if start ≥ end
2      return
3  // Set pivot to first element
4  i = start, j = end, p = start
5  while i < j
6      while array[i] ≤ array[p] and i < end
7          i = i+1
8      while array[j] > array[p]
9          j = j+1
10     if i < j
11         Swap array[i] and array[j]
12 Swap array[p] and array[j]
13 QUICKSORT(array, start, j−1)
14 QUICKSORT(array, j+1, end)
```

결정 트리를 만들 때, 블록을 놓을 수 있는 위치를 모두 계산해야 한다. 단순히 각 x좌표마다 블록이 놓일 수 있는 최대 y좌표를 계산하는 것은 블록을 좌우 구멍으로 끼워넣어 효율적인 상황을 만들 수 있음에도 불구하고 끼워넣지 않는 비효율적인 상황만을 탐색하게 되므로, 이런 상황과 같이 조각을 밀어넣어 필드를 더 효율적으로 만들 수 있는 경우도 놓치지 않고 탐색하기 위해 BFS를 사용한다.

FINDAVAILABLESPOTS(*field*, *nextBlock*)

```
1  Allocate new queue q, avilable_spots
2  for rot = 0 to 4
3      for x = −4 to WIDTH+4
4          if CHECKTOMOVE(field, nextBlock, rot, 1, x)) == TRUE
5              Emplace (5, x+4, rot) to q
6  while q.size > 0
7      (y+4, x+4, r) = q.front
8      Pop q
9      for all neighboring cell (ny, nx) to (y, x)
10         for current rotation state and next rotation state nr
11             if state (ny, nx, nr) not checked
12                 if CHECKTOMOVE(field, nextBlock, nr, ny, nx)) == TRUE
13                     if ny == y+1 and nr == r
14                         Emplace (y+4, x+4, r) to avilable_spots
15                     else
16                         Emplace (ny+4, nx+4, nr) to q
17
18 return avilable_spots
```

위의 메서드는 블록을 필드 또는 바닥에 닿게 하면서 사용자가 키보드를 조작해 블록을 놓을 수 있는 모든 위치를 BFS로 찾아 준다. 전체 필드 공간을 3-tuple을 이용한 BFS로 탐색하면서 CHECKTOMOVE를 호출하므로 시간 복잡도는 $\mathcal{O}(WIDTH \times HEIGHT)$인데, 이는 각 x좌표마다 블록이 놓일 수 있는 최대 y좌표를 계산하는 시간 복잡도와 동일하다.

결정 트리를 만드는 것과 최적화하는 것은 동시에 이루어진다. RECOMMEND-BFS는 한 레벨의 노드들의 리스트를 받아 다음 노드들을 만든 후, 정렬하고, 상위 몇 개 노드들을 다시 RECOMMEND-BFS로 보내 주는 재귀적 메서드이다.

이 메서드는 끝 레벨이 아닐 경우 매번 호출될 때마다 *nodes*의 크기만큼 FINDAVAILABLESPOTS를 호출하며 $(nodes \times children)$의 크기만큼 BOARD-SCORE를 호출한다. *nodes*는 위의 레벨에서 가지치기 상한선 개만큼 들어올 수 있으며 FINDAVAILABLESPOTS의 공간 복잡도는 $\mathcal{O}(WIDTH \times HEIGHT)$이므로 *children*의 공간 복잡도도 그러하고, BOARD-SCORE의 시간 복잡도는 $\mathcal{O}\left(WIDTH \times HEIGHT^2\right)$이므로 노드 하나에 대해 *childs*를 만드는 시간은 $\mathcal{O}\left(WIDTH^2 \times HEIGHT^3\right)$이다.

한편 가지치기 상한선을 m이라 하면, 생길 수 있는 *childs*의 개수는 *node*마다 $\mathcal{O}(WIDTH \times HEIGHT)$개씩이고 이후에 amortized $\mathcal{O}(n \log n)$의 시간에 정렬되므로, 이를 수행하는 데 걸리는 시간은

$$\mathcal{O}(m \times WIDTH \times HEIGHT \times \log(m \times WIDTH \times HEIGHT))$$

이다. 따라서 이 메서드가 한 번 호출되려면

$$\mathcal{O}\left(m \times WIDTH^2 \times HEIGHT^3 + m \times WIDTH \times HEIGHT \times \log(m \times WIDTH \times HEIGHT)\right)$$

의 시간이 필요하며, 이는 $\mathcal{O}\left(m \times WIDTH^2 \times HEIGHT^3\right)$와 같다.

이 작업을 트리의 높이만큼 재귀적으로 반복하므로 다음 블록의 개수를 k라 하면 전체 시간 복잡도는

$$\mathcal{O}\left(k \times m \times WIDTH^2 \times HEIGHT^3\right)$$

이다.

RECOMMEND-BFS(*level*, *nodes*)

```
1  if level == number of future blocks
2      max_score = −∞
3      max_node = NULL
4      for node in nodes
5          if max_score > node.score
6              max_score = node.score
7              max_node = node
8      if max_node == NULL
9          return NULL
10     while max_node.parent ≠ NULL and max_node.parent.parent ≠ NULL
11         max_node = max_node.parent
12     return max_node
13 Allocate new list childs
14 for node in nodes
15     children = FINDAVAILABLESPOTS(node.field, nextBlock[level])
16     while children.size > 0
17         Allocate new node newChild
18         Copy node.field to newChild.field
19         newChild.curBlockID = nextBlock[level]
20         (y+4, x+4, r) = children.front
21         newChild.recBlockX = x, newChild.recBlockY = y, newChild.recBlockRotate = r
22         Pop children
23         board_score = BOARD-SCORE(newChild.field, nextBlock[level])
24         newChild.parent = node, newChild.score = board_score
25         Add newChild to childs
26 QUICKSORT(childs, 0, count of childs − 1)
27 pruned_size = min(count of childs, tree prune limit)
28 Allocate new list pruned
29 for i = 0 to pruned_size
30     Add childs[i] to pruned
31 return RECOMMEND-BFS(level + 1, pruned)
```

3 가지치기의 효율성

추천 알고리즘의 복잡도와 효율성을 검증하기 위해 다음과 같은 평가를 수행하였다.

- 다른 조건은 그대로 두고, 미래에 보는 블록 수에 따라 시뮬레이션을 12번 진행해 추천 알고리즘의 누적 계산 시간과 누적 사용 공간, 점수의 평균과 표준편차를 각각 구한다.
- 위 평가를 트리 가지치기를 하지 않고 진행한다.
- 다른 조건은 그대로 두고, 트리 가지치기 이후에 남기는 노드의 상한에 따라 시뮬레이션을 12번 진행해 추천 알고리즘의 누적 계산 시간과 누적 사용 공간, 점수의 평균과 표준편차를 각각 구한다.

단, 프로그램이 쉽게 게임 오버되지 않는 점을 고려해 1,000번째 블록을 떨어뜨리는 시점에서 강제로 게임이 종료되도록 한다. 또한 1,000번째 블록 이전에서 게임 오버를 당할 경우 1,000블록 당 시간, 공간, 점수를 계산한다.

1번째 평가의 결과는 다음과 같았다. 남기는 노드의 상한은 모든 경우에서 32였다.

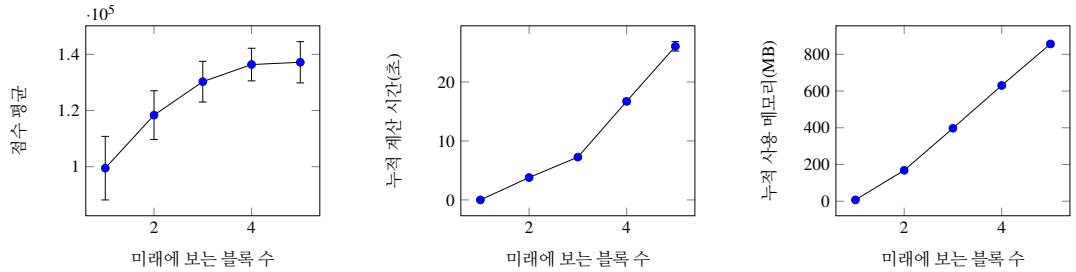
미래에 보는 블록 수	1*	2	3	4	5
점수 평균	99,470	118,352	130,272	136,363	137,178
점수 표준편차	11,317	8,675	7,253	5,806	7,357
누적 계산 시간 평균 (초)	< 0.01	3.80	7.26	16.71	26.04
누적 계산 시간 표준편차 (초)	< 0.01	0.14	0.25	0.43	0.80
1초당 점수	∞	31,145	17,944	8,161	5,268
누적 사용 메모리 평균 (MB)	7.29	168.31	397.50	630.41	856.28
누적 사용 메모리 표준편차 (MB)	0.22	3.82	4.97	12.44	12.78
1MB당 점수	13,645	703	327	216	160

* 미래에 1블록만 보는 경우는 평균적으로 456번째 블록이 내려온 직후 게임 오버 당했다. 나머지 경우에서는 1,000번째 블록이 내려올 때까지 게임 오버를 당하지 않았다.

1초당 점수와 1MB당 점수는 미래에 보는 블록 수가 증가할수록 오히려 감소했지만, 미래에 1블록만 보는 경우에서 1,000번째 블록이 내려올 때까지 게임 오버를 당하지 않은 경우가 없었던 것으로 미루어 보아 미래에 많은 블록을 볼수록 안정성이 높아진다고 추측할 수 있다. 따라서 안정성과 효율의 균형이 잘 맞는 블록 수를 선택해야 할 것이다.

그래프는 다음과 같았다.

점수는 미래에 보는 블록 수가 증가할수록 어떤 값에 수렴하는 형태를 보였으며, 계산 시간은 1–3블록, 3–5블록은 각각 선형의 경향성을 보이고 있다. 3블록을 기점으로 누적 계산 시간이 꺾였는데,



(a) 점수 평균 (b) 계산 시간 평균 (c) 사용 메모리 평균

요인은 잦은 캐시 미스 혹은 소팅 시간의 증가라고 추측된다. 사용 메모리는 선형의 경향성을 보이고 있다. 이는 가지치기를 했을 때 공간 복잡도가 미래에 보는 블록의 수에 비례함을 뒷받침한다.

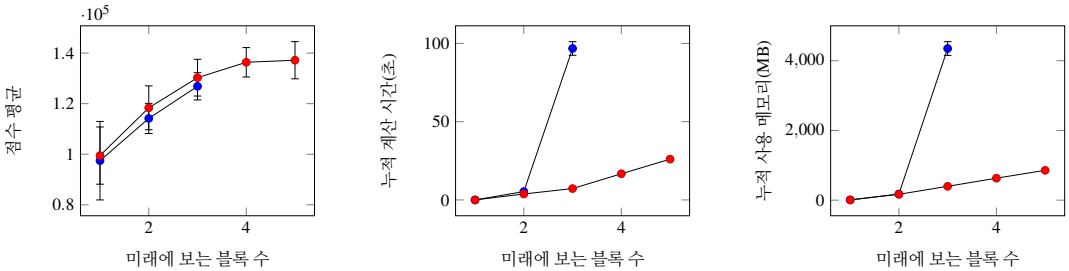
2번째 평가의 결과는 다음과 같았다.

미래에 보는 블록 수	1*	2	3	4**	5**
점수 평균	97,456	114,163	126,863	–	–
점수 표준편차	15,523	5,958	5,383	–	–
누적 계산 시간 평균 (초)	< 0.01	5.33	96.78	–	–
누적 계산 시간 표준편차 (초)	< 0.01	0.22	4.36	–	–
1초당 점수	∞	21,479	1,311	–	–
누적 사용 메모리 평균 (MB)	7.40	176.72	4349.17	–	–
누적 사용 메모리 표준편차 (MB)	0.17	3.52	197.27	–	–
1MB당 점수	13,170	646	29	–	–

* 미래에 1블록만 보는 경우는 평균적으로 477번째 블록이 내려온 직후 게임 오버 당했다. 나머지 경우에서는 1,000번째 블록이 내려올 때까지 게임 오버를 당하지 않았다.

** 미래에 4블록, 5블록을 보는 경우는 1,000번째 블록이 떨어질 때까지의 시간이 너무 오래 걸려 테스트하지 못했다.

가지치기를 하지 않을 경우 사용하는 시간과 공간이 기하급수적으로 늘어남을 알 수 있다. 그래프는 다음과 같았다. 붉은색 점은 가지치기를 했을 때의 결과이다.



(a) 점수 평균 (b) 계산 시간 평균 (c) 사용 메모리 평균

점수 평균에서는 가지치기를 했을 때의 결과가 약간 더 좋았으나 의미 있는 차이가 보이지 않았다. 또한 가지치기를 하지 않았을 때 미래에 3블록을 보는 경우 기하급수적으로 계산 시간과 사용 메모리가 늘어났음을 확인할 수 있다. 종합해 보면 가지치기는 시간, 공간적 측면 모두에서 결정 트리를 효율적으로 사용할 수 있게 해 주는 수단이라는 것을 확인할 수 있다.

가지치기를 했을 때의 단점은 최선의 상태를 버릴 수 있다는 가능성이 존재하는 것이었으나, 실제 점수를 비교했을 때 유의미한 차이가 없었던 것으로 미루어 보아 가지치기를 했을 때 얻을 수 있는 장점이 단점보다 훨씬 크다고 판단된다.

4 토의

추천 알고리즘은 클래식 테트리스를 바탕으로 개발되었다. 클래식 테트리스는 모던 테트리스에 있는 홀드, T-스핀, 벽에서의 회전 등의 기능이 없다. 기회가 된다면 모던 테트리스 규칙으로 추천 알고리즘을 작성해 보고 싶다.