선박 블록 최적 배치 알고리즘 프로젝트

프로젝트 개요

이 프로젝트는 선박 건조 과정에서 블록을 최적으로 배치하는 알고리즘을 구현한 것입니다. 2.5D 복셀화 표현을 사용하여 블록을 모델링하고, 백트래킹 알고리즘을 통해 최적의 배치 방법을 찾습니다.

주요 기능

- 2.5D 복셀화 표현을 사용한 블록 데이터 구조
- 블록 배치 영역 관리
- 백트래킹 기반 배치 알고리즘
- 블록 배치 후보 위치 생성기
- 블록 배치 시각화 도구
- 임의 블록 생성기

배치 알고리즘 설명

Bin Packing Problem과의 연관성

이 프로젝트는 2D Bin Packing Problem(2차원 빈 패킹 문제)의 개념을 응용하고 있습니다. Bin Packing Problem은 크기가 다른 여러 아이템을 최소한의 빈(컨테이너)에 효율적으로 배치하는 문제입니다. 이 프로젝트에서는 다양한 크기와 모양의 블록을 제한된 배치 영역에 최대한 많이 배치하는 문제로 변형되었습니다.

주요 연관성:

- **Bottom-Left 전략**: 2D Bin Packing에서 자주 사용되는 전략으로, 아이템을 가능한 한 왼쪽 아래에 배치합니다. 이 프로젝트에서는 X축 우선 배치와 왼쪽 정렬 휴리스틱으로 구현되었습니다.
- **First Fit Decreasing**: 아이템을 크기 순으로 정렬 후 첫 번째 가능한 위치에 배치하는 전략으로, 이 프로젝트에서는 블록을 너비, 면적, 밀도 순으로 정렬하는 방식으로 응용되었습니다.
- **Best Fit**: 가장 적합한 위치에 배치하는 전략으로, 이 프로젝트에서는 여러 휴리스틱 기준에 따라 최적 위치를 선정하는 방식으로 구현되었습니다.

차이점:

- 전통적인 Bin Packing은 사용되는 빈의 수를 최소화하는 것이 목적이지만, 이 프로젝트는 주어진 영역 내에 배치되는 블록의 수와 공간 활용도를 최대화하는 것이 목적입니다.
- 많은 Bin Packing 알고리즘은 그리디 휴리스틱을 사용하지만, 이 프로젝트는 휴리스틱 기반 백트래킹을 사용하여 더 광범위한 탐색을 수행합니다.
- 전통적인 2D Bin Packing은 주로 직사각형 아이템을 다루지만, 이 프로젝트는 복셀 기반의 다양한 형태의 블록을 다룹니다.

휴리스틱 기반 백트래킹 알고리즘

이 프로젝트에서는 전통적인 백트래킹 알고리즘에 휴리스틱 요소를 추가한 "휴리스틱 백트래킹" 또는 "정보 기반 백트래킹(informed backtracking)" 방식을 사용합니다. 이는 백트래킹의 완전 탐색 특성을 유지하면서도, 휴리스틱을 통해 탐색 순서를 최적화하여 더 빠르게 좋은 해를 찾을 수 있도록 합니다.

알고리즘의 주요 단계는 다음과 같습니다:

1. **휴리스틱 기반 블록 정렬**: 블록을 너비, 면적, 밀도 순으로 내림차순 정렬하여 X축 방향으로 넓은 블록부터 배치합니다. 이는 큰 블록을 먼저 배치하여 공간 활용도를 높이는 휴리스틱 전략입니다.
2. **휴리스틱 기반 후보 위치 생성**: 각 블록에 대해 가능한 배치 위치를 생성하고, 여러 휴리스틱 기준(X축 우선 배치, 왼쪽 정렬, 인접성 등)에 따라 점수를 계산하여 유망한 위치부터 시도합니다.
3. **재귀적 탐색**: 현재 블록을 배치한 후 다음 블록으로 재귀적으로 진행하며, 더 좋은 해를 찾지 못하면 현재 블록을 제거하고 다른 위치에 배치합니다. 이는 전통적인 백트래킹의 특성입니다.
4. **휴리스틱 기반 최적해 평가**: 배치 점수(배치된 블록 비율과 공간 활용률)를 계산하여 최적해를 선택합니다.

```python
def _backtrack(self, current_area, sorted_blocks, depth):
    """
    재귀적 백트래킹 함수
    """
    # 시간 제한 확인
    if time.time() - self.start_time > self.max_time:
        return

    # 현재 배치 상태의 점수 계산
    current_score = current_area.get_placement_score()

    # 최적해 업데이트
    if current_score > self.best_score:
        self.best_score = current_score
        self.best_solution = copy.deepcopy(current_area)

    # 모든 블록이 배치된 경우
    if depth >= len(sorted_blocks):
        return

    # 현재 배치할 블록
    current_block = sorted_blocks[depth]

    # 후보 위치 생성
    candidates = self.candidate_generator.generate_candidates(current_block)

    # 각 후보 위치에 대해 시도
    for pos_x, pos_y, rotation, _ in candidates:
        # 원본 회전 상태 저장
        original_rotation = current_block.rotation

        # 블록 회전 설정
        if current_block.rotation != rotation:
            current_block.rotate()

        # 블록 배치 시도
        if current_area.place_block(current_block, pos_x, pos_y):
            # 다음 블록으로 진행
            self._backtrack(current_area, sorted_blocks, depth + 1)

            # 백트래킹: 블록 제거
            current_area.remove_block(current_block.id)
```

```
            # 블록 회전 복원 (원래 상태로)
            while current_block.rotation != original_rotation:
                current_block.rotate()

        # 현재 블록을 배치하지 않고 다음 블록으로 진행
        self._backtrack(current_area, sorted_blocks, depth + 1)
```

휴리스틱 기반 후보 위치 생성 (Bin Packing 응용)

후보 위치 생성기는 Bin Packing Problem의 휴리스틱 전략을 응용하여 각 블록에 대해 가능한 배치 위치를 생성하고 평가합니다. 이는 단순히 가능한 모든 위치를 무작위로 시도하는 것이 아니라, 유망한 위치부터 시도하여 탐색 효율을 높이는 방식입니다.

주요 휴리스틱 기준은 다음과 같습니다:

1. **X축 우선 배치 (Bottom-Left 전략)**: Y값이 작을수록 높은 점수를 부여하여 X축 방향으로 먼저 채우도록 유도합니다. 이는 2D Bin Packing의 Bottom-Left 전략과 유사합니다.
2. **왼쪽 정렬 (Left Justification)**: X값이 작을수록 높은 점수를 부여하여 왼쪽부터 채우도록 유도합니다. 이는 공간을 효율적으로 활용하기 위한 전략입니다.
3. **인접성 (Adjacency)**: 다른 블록과 인접할수록 높은 점수를 부여하여 블록 간 빈 공간을 최소화합니다. 이는 Bin Packing에서 빈 공간을 최소화하는 전략과 유사합니다.
4. **경계 활용 (Boundary Utilization)**: 배치 영역의 경계에 인접할수록 높은 점수를 부여하여 경계를 효율적으로 활용합니다. 이는 Bin Packing에서 모서리와 경계를 우선적으로 활용하는 전략과 유사합니다.
5. **공간 효율성 (Density)**: 블록이 차지하는 공간이 조밀할수록 높은 점수를 부여합니다. 이는 Bin Packing에서 밀도를 고려하는 전략과 유사합니다.

```python
def _calculate_heuristic_score(self, block, pos_x, pos_y):
    """
    휴리스틱 점수 계산
    """
    # 1. X축 우선 배치 점수
    # X축 방향으로 먼저 채우기 위해 Y값이 작을수록 높은 점수
    x_first_score = 1.0 - (pos_y / self.placement_area.height)

    # 2. 왼쪽 정렬 점수 (Left 전략)
    # 왼쪽에 가까울수록 높은 점수
    left_alignment_score = 1.0 - (pos_x / self.placement_area.width)

    # 3. 인접성 점수 (Adjacent 전략)
    # 다른 블록과 인접할수록 높은 점수
    adjacency_score = self._calculate_adjacency_score(block, pos_x, pos_y)

    # 4. 면적 활용 점수
    # 블록의 면적이 클수록 높은 점수
    area_score = block.get_area() / (self.placement_area.width * self.placement_area.height)

    # 5. 경계 활용 점수
    # 배치 영역의 경계에 인접할수록 높은 점수
    boundary_score = self._calculate_boundary_score(block, pos_x, pos_y)

    # 6. 공간 효율성 점수
    # 블록이 차지하는 공간이 조밀할수록 높은 점수
    density_score = block.get_area() / (block.width * block.height)

    # 가중치를 적용한 종합 점수
    score = (
        0.4 * x_first_score +      # X축 우선 배치에 높은 가중치
        0.2 * left_alignment_score +
        0.2 * adjacency_score +
        0.1 * area_score +
        0.05 * boundary_score +
        0.05 * density_score
    )

    return score
```

블록 모델링

블록은 2.5D 복셀화 표현을 사용하여 모델링됩니다. 각 복셀은 (x, y, [empty_below, filled, empty_above]) 형태로 표현됩니다.

```
class VoxelBlock:
    """
    복셀 기반 블록 클래스

    Attributes:
        id (str): 블록 ID
        voxel_data (list): 복셀 데이터 [(x, y, [empty_below, filled, empty_above]), ...]
        min_x (int): 최소 x 좌표
        min_y (int): 최소 y 좌표
        max_x (int): 최대 x 좌표
        max_y (int): 최대 y 좌표
        width (int): 블록 너비
        height (int): 블록 높이
        position (tuple): 배치 위치 (x, y)
        rotation (int): 회전 각도 (0 또는 180)
    """
```

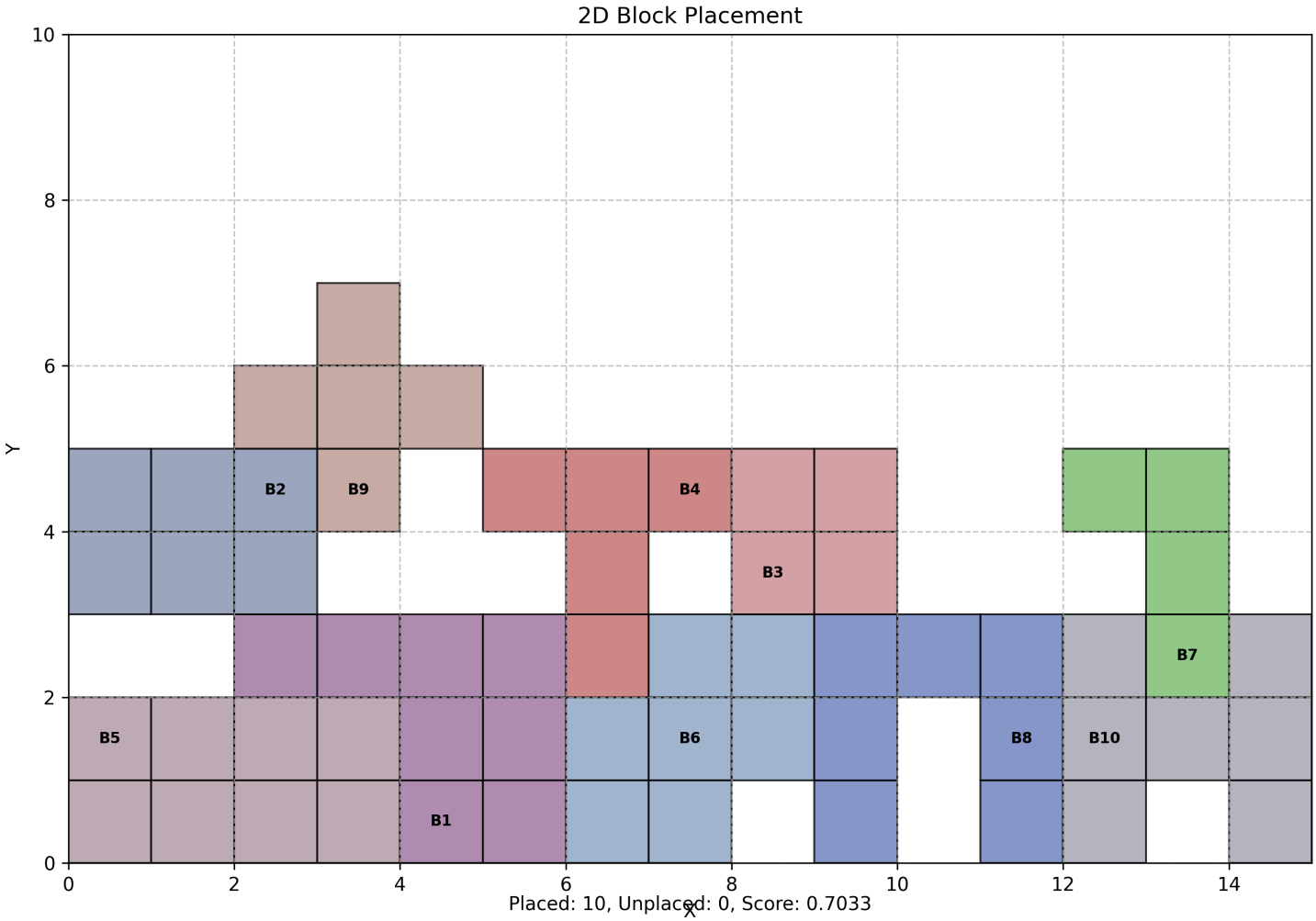
배치 영역 관리

배치 영역은 2D 그리드로 표현되며, 각 셀은 블록 ID 또는 None(빈 공간)을 저장합니다.

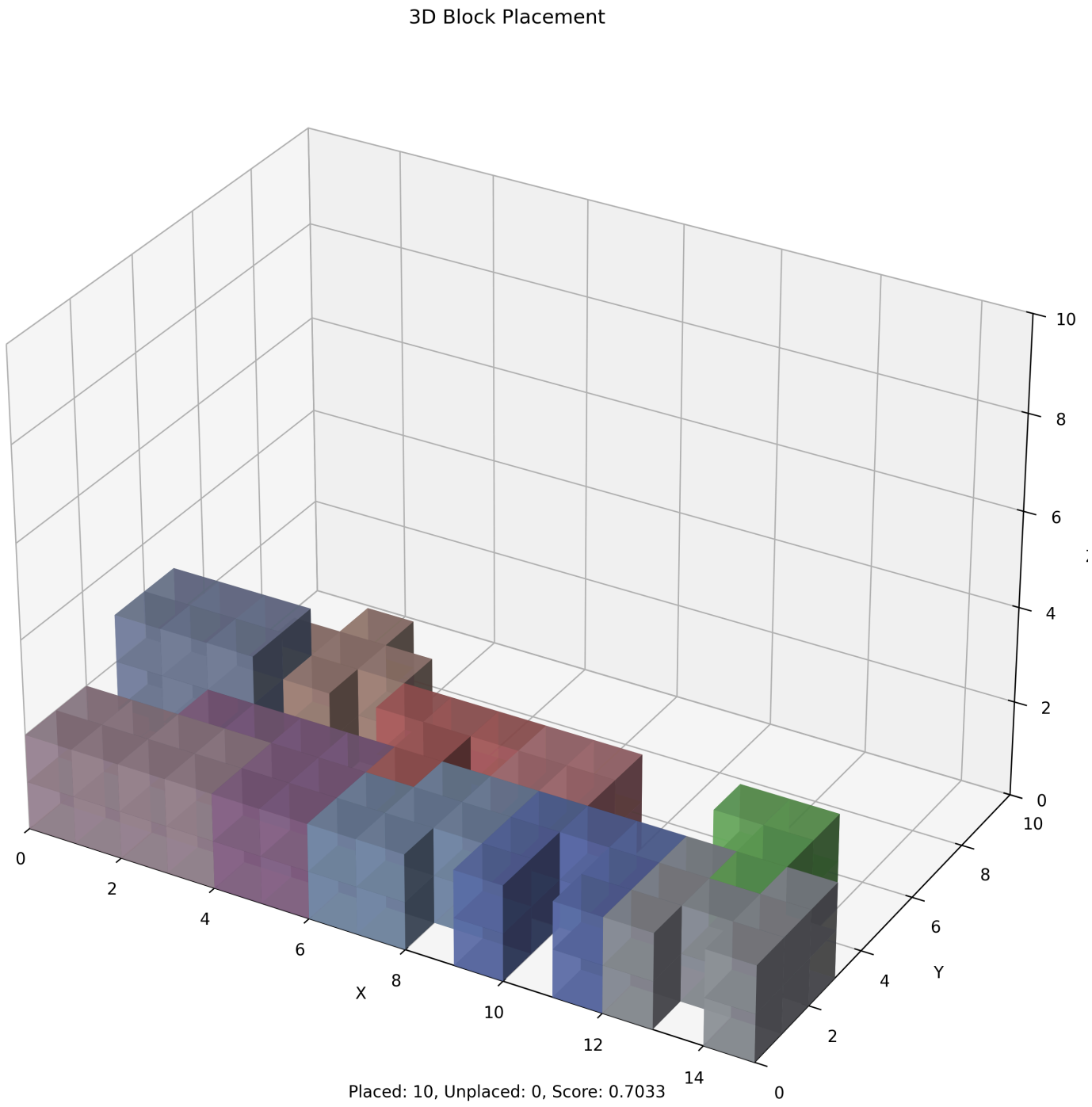
```
class PlacementArea:
    """
    블록 배치 영역 클래스

    Attributes:
        width (int): 배치 영역 너비
        height (int): 배치 영역 높이
        grid (numpy.ndarray): 배치 영역 그리드
        placed_blocks (dict): 배치된 블록 목록 {block_id: block}
        unplaced_blocks (dict): 미배치 블록 목록 {block_id: block}
    """
```

시각화 결과

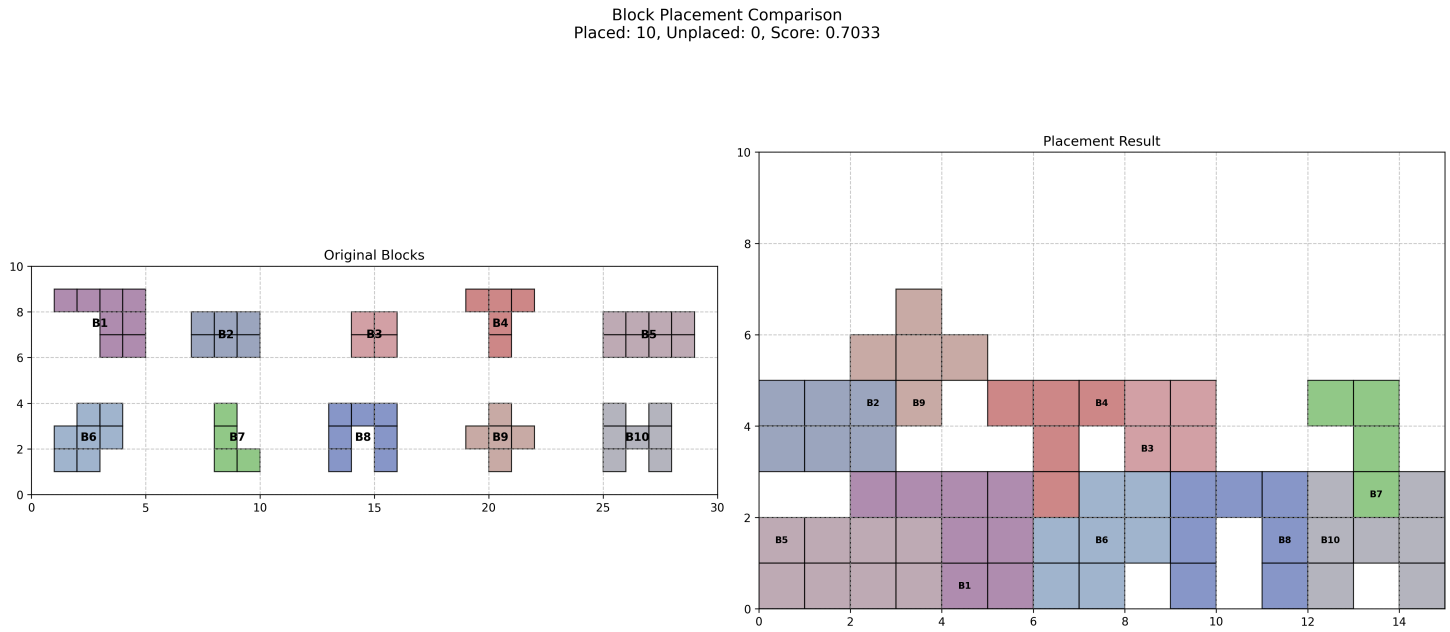
2D 배치도


2D Block Placement
B2
B9
B4
B3
B7
B5
B6
B8
B10
B1
X
Y
Placed: 10, Unplaced: 0, Score: 0.7033

3D 배치 결과


3D Block Placement
0
2
4
6
8
10
12
14
X
0
2
4
6
8
10
Y
0
2
4
6
8
10
Placed: 10, Unplaced: 0, Score: 0.7033

원본 블록과 배치 결과 비교



사용 방법

미리 정의된 블록 세트 사용

```
python main.py --use-predefined --max-time 10
```

랜덤 블록 생성

```
python main.py --block-count 10 --max-block-size 8 --complexity 1.0 --large-block-bias 0.7
```

옵션 설명

- `--width` : 배치 영역 너비 (기본값: 15)
- `--height` : 배치 영역 높이 (기본값: 10)
- `--block-count` : 블록 수 (기본값: 10)
- `--max-block-size` : 최대 블록 크기 (기본값: 8)
- `--complexity` : 블록 복잡도 (0.0~2.0, 기본값: 1.0)
- `--large-block-bias` : 대형 블록 생성 비율 (0.0~1.0, 기본값: 0.7)
- `--max-time` : 최대 실행 시간 (초, 기본값: 60.0)
- `--use-predefined` : 미리 정의된 블록 세트 사용 여부
- `--output-dir` : 결과 저장 디렉토리 (기본값: results)

- `--no-visualization` : 시각화 비활성화

프로젝트 구조

```
.
├── algorithms/
│   ├── backtracking_placer.py  # 백트래킹 기반 배치 알고리즘
│   └── candidate_generator.py  # 후보 위치 생성기
├── models/
│   ├── placement_area.py       # 배치 영역 클래스
│   └── voxel_block.py          # 복셀 기반 블록 클래스
├── utils/
│   ├── random_block_generator.py  # 임의 블록 생성기
│   └── visualizer.py           # 시각화 도구
├── main.py                     # 메인 실행 파일
└── results/                    # 결과 저장 디렉토리
```

결론

이 프로젝트는 선박 블록 배치 문제를 해결하기 위한 알고리즘을 구현했습니다. 2D Bin Packing Problem의 개념을 응용하여 휴리스틱 기반 백트래킹 알고리즘을 개발했으며, 이를 통해 최적의 배치 방법을 찾을 수 있습니다.

주요 성과:

- Bin Packing Problem의 휴리스틱 전략(Bottom-Left, First Fit Decreasing, Best Fit 등)을 응용하여 효율적인 배치 알고리즘 구현
- 백트래킹 알고리즘과 휴리스틱을 결합하여 탐색 효율성 향상
- X축 방향으로 먼저 채우고, 더 배치하지 못하면 Y축 방향으로 배치하는 전략을 통한 공간 활용도 최대화
- 2D 및 3D 시각화를 통한 직관적인 결과 확인

이 프로젝트는 선박 건조 과정에서의 블록 배치 최적화뿐만 아니라, 다양한 2D Bin Packing 문제에도 응용할 수 있는 가능성을 보여줍니다.