Python Code Explanation: Palletizer Pattern Generator

Viktor Laurens De Groote

December 11, 2024

1 Introduction

This document provides an explanation of a Python implementation for a palletizer pattern generator. The goal is to optimally place boxes on a pallet using a greedy search approach. The code uses modular design principles to compute and visualize the placement patterns. For examples demonstrating the program's functionality, please refer to the README file in the GitHub repository.

2 Code Breakdown

2.1 Classes for Box and Pallet

The code defines two core classes: `Box` and `Pallet`. The `Box` class represents the dimensions of a box, while the `Pallet` class represents the pallet and contains the logic for placing boxes.

Listing 1: Box and Pallet Classes

```
class Box:
    """Represents a box with dimensions width (w) and length
        (l)."""
    def __init__(self, width, length):
        self.width = width
        self.length = length

class Pallet:
    """Represents a pallet with dimensions width (W) and
       length (L)."""
    def __init__(self, width, length):
        self.width = width
        self.length = length
```

Explanation:

- The `Box` class defines the basic properties of a box: its width and length.

- The `Pallet` class initializes the dimensions of a pallet, which will be used to calculate the placement of boxes.

2.2 Greedy Search Algorithm: `place_greedy`

The `place_greedy` method is the core of the palletizer algorithm, designed to maximize the number of boxes placed on a pallet. It uses a greedy search approach to iteratively evaluate and optimize patterns for box placement.

Listing 2: Greedy Search for Box Placement

```
def place_greedy(self, box):
    """
    Place boxes on the pallet using a greedy search.

    Args:
        box (Box): The box to place.

    Returns:
        list: List of tuples representing the poses (x, y, theta).
        int: Total number of boxes placed.
    """
    patterns_to_check = []
    explored_patterns = []
    max_boxes = self.calculate_upper_bound(box)
    if max_boxes == 0:
        return [], 0

    # Initialize patterns with boxes without rotation and boxes with rotation
    # Without rotation
    n_boxes_w = self.width // box.width
    n_boxes_l = self.length // box.length
    n_boxes = int(n_boxes_w * n_boxes_l)
    pattern = [(i * box.width, j * box.length, 0) for i in range(n_boxes_w) for j in range(n_boxes_l)]
    usable_space = self.calc_usable_space(pattern, box)
    patterns_to_check.append((n_boxes, usable_space, pattern))
    if n_boxes == max_boxes:
        return pattern, max_boxes

    # With rotation
    n_boxes_w = self.width // box.length
    n_boxes_l = self.length // box.width
    n_boxes = int(n_boxes_w * n_boxes_l)
    pattern = [(i * box.length, j * box.width, 90) for i in range(n_boxes_w) for j in range(n_boxes_l)]
    usable_space = self.calc_usable_space(pattern, box)
```

```
    patterns_to_check.append((n_boxes, usable_space, pattern))
    if n_boxes == max_boxes:
        return pattern, max_boxes

    while patterns_to_check:
        # Take the pattern with the most usable space
        n_boxes, usable_space, pattern = max(patterns_to_check, key=lambda x: x[1])
        patterns_to_check.remove((n_boxes, usable_space, pattern))

        # Check for free space
        free_space_rectangles = self.calc_free_space_rectangles(pattern, box)
        # For each free rectangle, check if boxes can be placed
        usable_rectangles = [
            rect for rect in free_space_rectangles if self.check_free_space_rectangle(rect, box)
        ]
        if not usable_rectangles:
            # No more boxes can be placed
            explored_patterns.append((n_boxes, usable_space, pattern))
            continue
        else:
            # Choose biggest usable rectangle to fill
            rect_to_fill = max(usable_rectangles, key=lambda r: r[2] * r[3])
            # Fill the free space with boxes both with and without rotation and add to patterns_to_check
            x, y, w, l = rect_to_fill

            # Without rotation
            n_boxes_w = w // box.width
            n_boxes_l = l // box.length
            n_boxes_rect_1 = int(n_boxes_w * n_boxes_l)
            pattern_rect_1 = [
                (x + i * box.width, y + j * box.length, 0)
                for i in range(n_boxes_w)
                for j in range(n_boxes_l)
            ]
            usable_space = self.calc_usable_space(pattern + pattern_rect_1, box)
            patterns_to_check.append((n_boxes + n_boxes_rect_1, usable_space, pattern + pattern_rect_1))
            if n_boxes + n_boxes_rect_1 == max_boxes:
                return pattern + pattern_rect_1, max_boxes
```

```
73
74              # With rotation
75              n_boxes_w = w // box.length
76              n_boxes_l = l // box.width
77              n_boxes_rect_2 = int(n_boxes_w * n_boxes_l)
78              pattern_rect_2 = [
79                  (x + i * box.length, y + j * box.width, 90)
80                  for i in range(n_boxes_w)
81                  for j in range(n_boxes_l)
82              ]
83              usable_space = self.calc_usable_space(pattern +
                    pattern_rect_2, box)
84              patterns_to_check.append((n_boxes +
                    n_boxes_rect_2, usable_space, pattern +
                    pattern_rect_2))
85              if n_boxes + n_boxes_rect_2 == max_boxes:
86                  return pattern + pattern_rect_2, max_boxes
87
88              # Add the better pattern to patterns_to_check
89              if n_boxes_rect_1 >= n_boxes_rect_2:
90                  patterns_to_check.append((n_boxes +
                        n_boxes_rect_1, usable_space, pattern +
                        pattern_rect_1))
91              else:
92                  patterns_to_check.append((n_boxes +
                        n_boxes_rect_2, usable_space, pattern +
                        pattern_rect_2))
93
94      # Return the best pattern
95      best_pattern = max(explored_patterns, key=lambda x: x
            [0])
96      return best_pattern[2], best_pattern[0]
```

2.3 Explanation

- **Inputs:**
  - `box`: The dimensions of the box being placed on the pallet.
- **Outputs:**
  - A list of tuples (x, y, θ) representing the placement positions and orientations of the boxes.
  - The total number of boxes successfully placed.
- **Key Steps:**
  1. **Calculate Upper Bound:** The method begins by calculating the maximum number of boxes that can theoretically fit on the pallet

using `calculate_upper_bound`. If this value is 0, the method immediately returns no placement.

2. **Generate Initial Patterns:**
   - Boxes are placed on the pallet in two initial configurations:
     (a) Without rotation ($\theta = 0$).
     (b) With rotation ($\theta = 90°$).
   - For each configuration, the total number of boxes and the usable space are computed using `calc_usable_space`.
   - If either configuration matches the theoretical maximum, it is immediately returned as the optimal solution.
3. **Iterative Pattern Optimization:**
   - While there are patterns to evaluate:
     (a) Select the pattern with the highest usable space.
     (b) Identify free space on the pallet using `calc_free_space_rectangles`.
     (c) Check each free rectangle for compatibility using `check_free_space_rectangle`.
     (d) If no further boxes can be placed, mark the pattern as explored.
     (e) Otherwise, select the largest usable rectangle and attempt to fill it:
       * Place boxes without rotation.
       * Place boxes with rotation.
     (f) Evaluate the new patterns created by filling the rectangle and add them to the evaluation queue.
4. **Return the Best Pattern:** If no further optimizations are possible, the pattern that placed the most boxes is returned.

2.4 Detailed Example

Consider a pallet of size 120 cm × 100 cm and a box of size 22 cm × 17 cm.

1. **Initial Placement:**
   - Without rotation: n_boxes = 31, usable_space = 1122 cm^2.
   - With rotation: n_boxes = 32, usable_space = 1200 cm^2.
2. The rotated pattern is selected as it places more boxes.
3. Remaining free space is divided into rectangles. Additional boxes are placed iteratively until no more boxes can fit.

2.5 Use Case

The `place_greedy` method balances efficiency and accuracy, making it a practical solution for optimizing box placement on a pallet. Its modular design allows for extensions, such as incorporating constraints or additional box types.

2.6 Supporting Methods

Several helper methods are implemented to support the placement logic:

- `calculate_upper_bound`: Calculates the theoretical maximum number of boxes that can fit on the pallet.
- `calc_usable_space`: Computes the usable space for a given pattern.
- `calc_free_space_rectangles`: Identifies the largest remaining free spaces on the pallet.
- `check_free_space_rectangle`: Determines whether a box can fit into a given rectangle.

2.6.1 `calculate_upper_bound`

The `calculate_upper_bound` method computes the theoretical maximum number of boxes that can fit on the pallet based on their areas.

Listing 3: Supporting Method: Calculate Upper Bound

```
def calculate_upper_bound(self, box):
    """Calculate the upper bound for the number of boxes
       that can fit on the pallet."""
    pallet_area = self.width * self.length
    box_area = box.width * box.length
    return pallet_area // box_area
```

Explanation:

- The `pallet_area` is computed as the product of the pallet's width and length.
- The `box_area` is similarly calculated as the product of the box's width and length.
- Dividing the `pallet_area` by the `box_area` provides an estimate of how many boxes can theoretically fit on the pallet.
- The `//` operator is used for floor division, rounding down to the nearest integer. This accounts for partial boxes that cannot fully fit on the pallet, ensuring that only whole boxes are considered.

Example:

- If the pallet dimensions are 120 cm x 100 cm ($\text{pallet\_area} = 12,000\,\text{cm}^2$) and the box dimensions are 22 cm x 17 cm ($\text{box\_area} = 374\,\text{cm}^2$):

$$\text{Upper Bound} = \frac{\text{pallet\_area}}{\text{box\_area}} = \frac{12,000}{374} \approx 32.0856 \implies 32$$

- This means that a maximum of 32 boxes can theoretically be placed on the pallet in this configuration.

2.6.2 `check_free_space_rectangle`

The `check_free_space_rectangle` method determines whether a given box can fit into a specified rectangle. This method is essential for evaluating whether a box can be placed in unused areas of the pallet, either in its original orientation or rotated.

Listing 4: Supporting Method: Check Free Space Rectangle

```
def check_free_space_rectangle(self, rect, box):
    """
    Check if a box can fit in a free space rectangle.

    Args:
        rect (tuple): A rectangle as (x, y, width, length).
        box (Box): The box dimensions.

    Returns:
        bool: True if the box can fit, False otherwise.
    """
    _, _, rect_width, rect_length = rect

    # Check if the box fits in either orientation
    return (box.width <= rect_width and box.length <=
        rect_length) or \
        (box.length <= rect_width and box.width <=
            rect_length)
```

Explanation:

- `rect` represents a free space on the pallet, defined by its:
  - x, y: Bottom-left corner coordinates.
  - `rect_width`: Width of the rectangle.
  - `rect_length`: Length of the rectangle.
- The method evaluates whether the box can fit in the rectangle either:
  1. Without rotation: The box's width must be less than or equal to the rectangle's width, and its length must be less than or equal to the rectangle's length.
  2. With a 90° rotation: The box's length must fit within the rectangle's width, and its width must fit within the rectangle's length.
- The method returns `True` if the box can fit in either orientation, and `False` otherwise.

Example:

- Consider a free space rectangle with dimensions `rect_width` = 40 cm and `rect_length` = 30 cm.

- A box with dimensions `box.width` = 25 cm and `box.length` = 35 cm is evaluated:
  - Without rotation: `box.width` = $25 \leq 40$ and $30 \leq$ `box.length` = 35, so the box does not fit.
  - With rotation: `box.length` = $35 \leq 40$ and `box.width` = $25 \leq 30$, so the box fits when rotated 90°.
- The method would return `True` in this case.

Use Case: This method is critical for determining whether the algorithm can utilize free space on the pallet. It ensures that all potential placements are considered, improving the efficiency and accuracy of the packing algorithm.

2.6.3 `calc_free_space_rectangles`

The `calc_free_space_rectangles` method calculates two large rectangles representing the remaining free space on the pallet after some boxes have been placed. These rectangles are key to determining where additional boxes can potentially be placed.

Listing 5: Supporting Method: Calculate Free Space Rectangles

```
def calc_free_space_rectangles(self, pattern, box):
    """
    Calculate two large rectangles representing the free
        space in the pallet.

    Args:
        pallet (Pallet): The pallet dimensions (width,
            length).
        pattern (list): A list of placed boxes, each as (x,
            y, theta).

    Returns:
        list: Two rectangles (above and right), as (x, y,
            width, length).
    """
    # Step 1: Determine the covered area
    if not pattern:
        return []
    max_x = max(x + (box.width if theta == 0 else box.length
        ) for x, y, theta in pattern)
    max_y = max(y + (box.length if theta == 0 else box.width
        ) for x, y, theta in pattern)

    # Step 2: Calculate the remaining rectangles
    rectangles = []

    # Rectangle above the covered area
```

```
    above_rect = (0, max_y, self.width, self.length - max_y)
    rectangles.append(above_rect)

    # Rectangle to the right of the covered area
    right_rect = (max_x, 0, self.width - max_x, self.length)
    rectangles.append(right_rect)
    return rectangles
```

Explanation:

- **Inputs:**
  - `pattern`: A list of placed boxes, where each box is represented by its bottom-left corner coordinates (x, y) and its orientation ($\theta = 0°$ for no rotation or $90°$ for rotated).
  - `box`: The dimensions of the box being placed.
- **Step 1: Determine the covered area.**
  - `max_x`: The farthest horizontal extent of the covered area is calculated as $x +$ `box.width` for unrotated boxes or $x +$ `box.length` for rotated boxes.
  - `max_y`: The farthest vertical extent of the covered area is similarly calculated as $y +$ `box.length` for unrotated boxes or $y +$ `box.width` for rotated boxes.
- **Step 2: Calculate the remaining free space.**
  - Two large rectangles are identified:
    1. **Above Rectangle:** The area above the covered boxes, starting from $y =$ `max_y`, with width equal to the pallet's width and length equal to the remaining vertical space (`self.length` $-$ `max_y`).
    2. **Right Rectangle:** The area to the right of the covered boxes, starting from $x =$ `max_x`, with length equal to the pallet's length and width equal to the remaining horizontal space (`self.width`$-$ `max_x`).
  - Both rectangles are added to the `rectangles` list.
- **Output:** A list of two rectangles, each represented as $(x, y,$ `rect_width`$,$ `rect_length`$)$.

Example:

- Consider a pallet of size 120 cm $\times$ 100 cm and a pattern with three unrotated boxes:

$$\texttt{pattern} = [(0, 0, 0), (22, 0, 0), (44, 0, 0)]$$

- **Step 1:** Determine the covered area:

$$\texttt{max\_x} = 44 + 22 = 66\,\text{cm}$$
$$\texttt{max\_y} = 0 + 17 = 17\,\text{cm}$$

- **Step 2:** Calculate the free space:
  - **Above Rectangle:** $(0, 17, 120, 100 - 17 = 83)$
  - **Right Rectangle:** $(66, 0, 120 - 66 = 54, 100)$
- Output:

$$\texttt{rectangles} = [(0, 17, 120, 83), (66, 0, 54, 100)]$$

Use Case: This method identifies usable free spaces on the pallet for placing additional boxes. It simplifies the problem by approximating the free space with two large rectangles, which can be further subdivided if needed.

2.6.4 `calc_usable_space`

The `calc_usable_space` method calculates the usable space for a given packing pattern. This includes both the area occupied by boxes already placed on the pallet and the additional free space that can still be utilized by placing more boxes.

Listing 6: Supporting Method: Calculate Usable Space

```
def calc_usable_space(self, pattern, box):
    """
    Calculate the usable space for a given pattern. This
        includes both the space
    that is already taken in by boxes as well as the space
        that can still be
    taken in by boxes.

    Args:
        pattern (list): A list of placed boxes, each as (x,
            y, theta).
        box (Box): The dimensions of the box.

    Returns:
        int: Total usable space (area) for the given pattern
            .
    """
    area_box = box.width * box.length
    amount_boxes = len(pattern)
    used_space = amount_boxes * area_box
    usable_space = used_space
    free_space_rectangles = self.calc_free_space_rectangles(
        pattern, box)
```

```
    for rect in free_space_rectangles:
        _, _, w, l = rect
        if self.check_free_space_rectangle(rect, box):
            usable_space += w * l
    return usable_space
```

Explanation:

- **Inputs:**
  - `pattern`: A list of already placed boxes on the pallet. Each box is represented by its position (x, y) and orientation (θ).
  - `box`: The dimensions of the box being considered for placement.
- **Step-by-Step Functionality:**
  1. **Compute the space already used:**
     - The area of a single box is calculated as `area_box` = `box.width`$\times$ `box.length`.
     - The total used space is determined by multiplying the number of boxes in the pattern (`amount_boxes`) by the area of a single box (`used_space`).
     - Initially, the usable space (`usable_space`) is set to the total used space.
  2. **Calculate free space:**
     - The free space rectangles are determined using the `calc_free_space_rectangles` method.
     - For each free rectangle, the width (w) and length (l) are extracted.
  3. **Evaluate additional usable space:**
     - If the rectangle is large enough to fit the box (checked using `check_free_space_rectangle`), its area $(w \times l)$ is added to the `usable_space`.
  4. **Return the total usable space.**

Example:

- Consider a pallet of size 120 cm $\times$ 100 cm and a box of size 22 cm $\times$ 17 cm.
- Suppose three boxes have been placed on the pallet:

$$\texttt{pattern} = [(0, 0, 0), (22, 0, 0), (44, 0, 0)]$$

- The space already used is:

$$\texttt{used\_space} = 3 \times (22 \times 17) = 1122\,\text{cm}^2$$

- The remaining free space is divided into rectangles using `calc_free_space_rectangles`. Assume these rectangles are:

$$\texttt{free\_space\_rectangles} = [(0, 17, 120, 83), (66, 0, 54, 100)]$$

- If both rectangles can fit the box, their combined area is added to the `usable_space`.
- Total `usable_space`:

$$\texttt{usable\_space} = \texttt{used\_space} + (120 \times 83) + (54 \times 100) = 1122 + 9960 + 5400 = 16482\,\text{cm}^2$$

Use Case: This method helps assess the efficiency of a packing pattern by accounting for both utilized and potential usable areas on the pallet. It is a key metric for comparing and optimizing packing strategies.

2.7 Visualization

The code includes functions to visualize the placement of boxes on the pallet. The `visualize` function generates a 2D representation of the pallet with the placed boxes.

Listing 7: Visualization Function

```
def visualize(box, pallet, pattern, title="Palletizing Pattern"):
    """
    Visualize the placement of boxes on the pallet.
    """
    fig, ax = plt.subplots(figsize=(10, 8))
    ax.set_xlim(0, pallet.width)
    ax.set_ylim(0, pallet.length)
    ax.set_aspect('equal')
    ax.set_title(title)
    ax.set_xlabel("Width (cm)")
    ax.set_ylabel("Length (cm)")

    # Draw the pallet and boxes
    ax.add_patch(
        plt.Rectangle((0, 0), pallet.width, pallet.length,
            edgecolor='black', facecolor='lightgray', lw=2)
    )
    for x, y, theta in pattern:
        if theta == 0:
            w, l = box.width, box.length
        else:
            w, l = box.length, box.width
        ax.add_patch(
            plt.Rectangle((x, y), w, l, edgecolor='blue',
                facecolor='skyblue', lw=1)
```

```
        )
    plt.show()
```

Explanation:

- The pallet and boxes are drawn using `matplotlib`.
- Boxes are shown in their respective orientations (rotated or unrotated).
- Multiple box types can also be visualized using the `visualize_multiple` function.

2.8 Main Function

The `main` function serves as the entry point to demonstrate the palletizing algorithm. It provides a user-friendly interface to select different configurations of pallets and boxes for optimization. Additionally, it includes a bonus feature that handles two different box sizes simultaneously.

Listing 8: Main Function for Palletizing Algorithm

```
from .utils import Box, Pallet, visualize,
    visualize_multiple

def main():
    """
    Main function to showcase the palletizing algorithm.
    """
    print("\nWelcome to the Palletizer Program!\n")

    while True:
        print("Choose a configuration to optimize:")
        print("1. Pallet: 120x100 cm, Box: 22x17 cm")
        print("2. Pallet: 120x100 cm, Box: 20x10 cm")
        print("3. Pallet: 70x60 cm, Box: 40x30 cm")
        print("4. Bonus: Two different box sizes (40x30 cm
            and 22x17 cm)")
        print("0. Exit the program")

        choice = input("\nEnter your choice: ")
        print("\n------------------------------\n")

        if choice == "1":
            pallet = Pallet(120, 100)
            box = Box(22, 17)
        elif choice == "2":
            pallet = Pallet(120, 100)
            box = Box(20, 10)
        elif choice == "3":
            pallet = Pallet(70, 60)
            box = Box(40, 30)
```

```
        elif choice == "4":
            # Bonus case with two different box sizes
            ...
        elif choice == "0":
            print("Exiting the program. Goodbye!\n")
            break
        else:
            print("Invalid choice! Please try again.")
            continue

        # Solve for a single box size
        ...
```

Explanation:

- The program provides a menu to choose from three predefined configurations or a bonus configuration with two box sizes.
- Each configuration specifies the dimensions of the pallet and the box.
- For single box configurations, the `place_greedy` method of the `Pallet` class is used to find the optimal placement of boxes.
- The results are visualized using the `visualize` function.

Bonus Case: Two Box Sizes

- For the bonus configuration, the algorithm solves the placement for the bigger box size first and identifies the remaining free space.
- The second box size is then placed within the free space using the `place_greedy` method.
- The results for both box sizes are combined and visualized on a single plot using the `visualize_multiple` function.

2.8.1 Example Usage

When the program is run, the user can interactively choose configurations, view results in the terminal, and visualize the optimal box placement on the pallet. This modular design ensures flexibility and ease of extension for additional configurations or constraints. See the README file in the GitHub repository for more information about usage and examples.

3 Conclusion

This implementation provides a flexible and modular approach to palletizing boxes. The greedy algorithm efficiently utilizes space on the pallet, and the visualization functions offer an intuitive understanding of the placement patterns. The code can be extended to handle more complex palletizing scenarios, including irregular shapes and constraints.