Dual-Robot Score Multi-Objective Optimization using a Two-Part Knapsack Algorithm with Mutual Exclusivity through Dynamic Programming in Python
Ruotong Gao
The Webb Schools
Tonygao@webb.org

Dual-Robot Multi-Objective Score Optimization using a Two-Part Knapsack Algorithm with Mutual Exclusivity through Dynamic Programming in Python

Abstract:

Time and score are the two imperative metrics that Botball is played around. Optimizing the maximum number of points within the allotted timeframe is crucial to a team's performance during Botball competitions and is the determining factor of success within the competition. A team's choice in the tasks they decide to pursue ultimately defines their methodology and robot design. Teams must carefully select the optimal tasks that each robot will perform, considering the score, difficulty, and time within each task. This results in balancing high-value tasks, which may be time-consuming,e against lower-value tasks, which can be quickly completed. A variation of the knapsack problem through multi-objective optimization and mutual exclusivity was constructed to address this challenge for two robots in a realistic and accurate way.

The typical knapsack problem is a well-known optimization algorithm in which the objective is to select a subset of items, to maximize the total value without exceeding a specified limit. This paper presents a research model that involves dividing tasks between the two robots with one larger task list that the algorithm self-arranges into two separate knapsacks within the larger algorithm while considering mutual exclusivity constraints [1]. Mutual exclusivity implies that both robots cannot perform specific tasks simultaneously due to physical space limitations or task dependencies. The model applies multi-objective optimization (MOO) principles to achieve the best overall performance, focusing on finding a balance between score and difficulty within a set timeframe for the two robots. MOO ensures that the selection of tasks accounts for both score and difficulty so that if two solutions have a similar score with a tolerance of 10, the one with the lower difficulty is preferred [2].

By leveraging this dual-robot optimization strategy, teams can effectively plan and execute their task allocations, ensuring both robots focus on the most valuable tasks in the available time. By applying a two-part knapsack algorithm with mutual exclusivity constraints and multi-objective optimization, teams can efficiently plan and execute an optimal direction for their game. This research highlights the practical application of advanced optimization techniques in a

competitive robotics setting, demonstrating their value in enhancing the decision-making process and overall success in Botball competitions.

Introduction:

Scoring:

Botball 2024 offers a unique game set in a lunar environment. It challenges teams with various objectives. The scoring system consists of base amounts and multipliers, as well as unique tasks that can be accomplished to multiply the existing score within a particular objective area. Each area has a theoretical maximum score, where teams can allocate the maximum requirements for scoring and any additional multipliers. The maximal score is the primary metric the algorithm was built on. The scoring sheet is as follows:

Areas	Itemized Points	Multipliers	Totals
Area 1		Cube X 5 or Botguy X 10	
Sorted Poms / Pool Noodles	#_____ X 5 = _____		
All Other Game Pieces	#_____ X 1 = _____		
	Subtotal = _____		
Small Rover Bay		Cube X 5 or Botguy X 10	
Sorted Poms / Pool Noodles	#_____ X 5 = _____		
All Other Game Pieces	#_____ X 1 = _____		
	Subtotal = _____		
Large Rover Bay		Cube X 5 or Botguy X 10	
Sorted Poms / Pool Noodles	#_____ X 5 = _____		
All Other Game Pieces	#_____ X 1 = _____		
	Subtotal = _____		
Area 2		Cube X 5 or Botguy X 10	
Sorted Poms / Pool Noodles	#_____ X 5 = _____		
All Other Game Pieces	#_____ X 1 = _____		
	Subtotal = _____		
Area 3		Cube X 5 or Botguy X 10	
Sorted Poms / Pool Noodles	#_____ X 5 = _____		
All Other Game Pieces	#_____ X 1 = _____		
	Subtotal = _____		
Area 4		Cube X 5 or Botguy X 10	
Sorted Poms / Pool Noodles	#_____ X 5 = _____		
All Other Game Pieces	#_____ X 1 = _____		
	Subtotal = _____		
Area 5		Cube X 5 or Botguy X 10	
Sorted Poms / Pool Noodles	#_____ X 5 = _____		
All Other Game Pieces	#_____ X 1 = _____		
	Subtotal = _____		
Area 6		Cube X 5 or Botguy X 10	
Sorted Poms / Pool Noodles	#_____ X 5 = _____		
All Other Game Pieces	#_____ X 1 = _____		
	Subtotal = _____		
Rock Heap		Cube X 5 or Botguy X 10	
Rocks	#_____ X 8 = _____		
All Other Game Pieces	#_____ X 1 = _____		
	Subtotal = _____		
Solar Panel		Astronaut in Solar Station X 5	
Solar Panel Flipped	#_____ X 50 = _____		
	Subtotal = _____		
Lava Tube Area		Deepest Lava Tube with Noodle (1, 2 or 3) X____	
Blue Poms	#_____ X 20 = _____		
Capped Tube/ Purple Noodle	#_____ X 25 = _____		
Purple Noodles in Tubes	#_____ X 100 = _____		
	Subtotal = _____		
Moon Base		Air Lock Closed X 3	
Air Lock Open/ Flipped Switch	#_____ X 25 = _____		
Blue Poms in Air Lock	#_____ X 25 = _____		
Sorted Blue Poms in Air Lock	#_____ X 100 = _____		
	Subtotal = _____		
Habitat Construction		#of Posts X_____	
Red or Green Noodles	#_____ X 15 = _____		
	Subtotal = _____		
Astronauts		#Stations with Astronaut X_____	
Astronauts In Stations	#_____ X 25 = _____		
Flag Raised	#_____ X 100 = _____		
	Subtotal = _____		

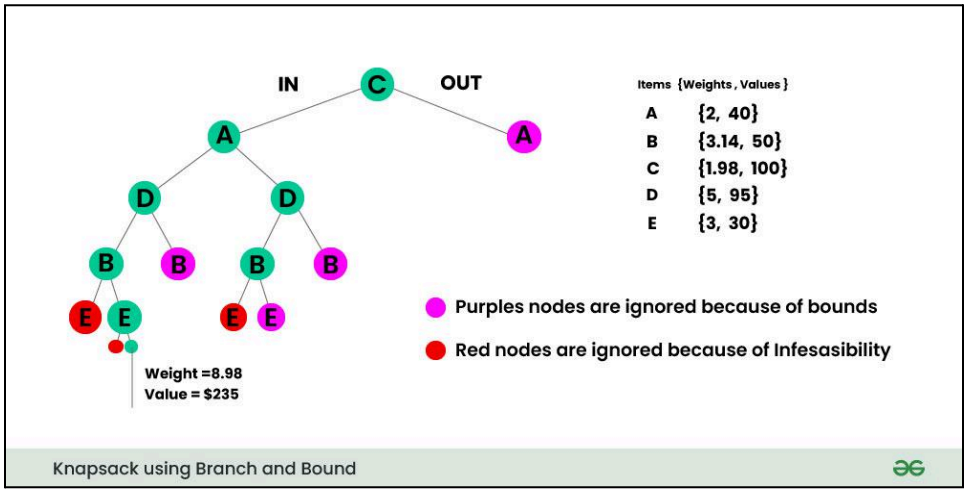
(Table 1) Scoresheet for the 2024 Botball GCER game [3]

/

Knapsack and MOO:

Knapsack problems are a fundamental optimization challenge where an algorithm selects items to maximize the total value without exceeding a specific capacity. This problem typically finds applications in computer science, resource allocation, scheduling, and finance, among many others. There are many types of knapsack problems, but the primary focus falls into three main types: Binary Knapsack Problem, items cannot be divided; it is either take it or leave it, and

Fractional Knapsack Problems: Items can be taken in fractions, allowing more flexibility in selecting items to maximize value, and Bounded Knapsack Problems: Items can only be placed in the knapsack a certain amount of times [4].



(Figure 1) Diagram for the logic of a bounded Knapsack Algorithm [5]

At its core, the knapsack problem involves selecting a subset of items from a given set, each characterized by a weight, and a value, with the goal of maximizing the total value without exceeding a maximum capacity. A further characterization by UT Dallas defines the variables

$$w_k = \text{the weight of each type-}k\text{ item, for } k = 1, 2, \ldots, N,$$

$$r_k = \text{the value associated with each type-}k\text{ item, for } k = 1, 2, \ldots, N,$$

$$c = \text{the weight capacity of the knapsack.}$$

Variables defined by UT Dallas for the Knapsack Algorithm [6]

Thus, the algorithm can be expressed mathematically as follows:

$$\text{Maximize} \quad \sum_{k=1}^{N} r_k x_k$$

$$\text{Subject to:} \quad \sum_{k=1}^{N} w_k x_k \leq c,$$

With X_n being the nonnegative integer decision variables, defined by X_k = the number of type-k items that are loaded into the knapsack. [6]

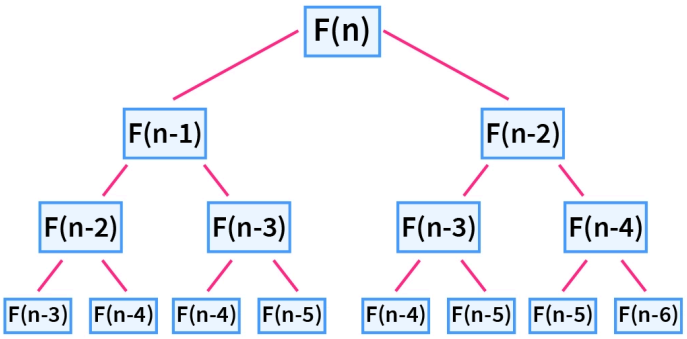
Summation equation for the Knapsack Algorithm [6]

Multi-objective optimization (MOO) involves optimizing multiple objectives simultaneously, aiming to find a set of solutions that represent trade-offs between these objectives. Unlike single-objective optimization, where a clear best solution exists, MOO deals

with multiple different variables to consider, like score and difficulty, while managing to fit within another constraint: time. This is incredibly important for the game as the difficulty/feasibility of tasks is also a crucial aspect of a team's selection of objectives. For example, Botguy tends to result in a ten-times multiplier for the particular game area he is set in. However, Botguy tends to be heavy, oddly shaped, and placed in a hard-to-get-to location, which raises its difficulty much more than most other tasks. When a team faces two tasks of similar score value and varying difficulties, an algorithm must consider this and adjust based on difficulty to make the optimal scoring more feasible/realistic.

Mutual exclusivity is also necessary within this particular game since only a set number of pieces can provide points. These pieces can be moved to different parts of the game board to score varying amounts of points with different objectives, such as placing poms into one area versus sorting them and placing them within a bin. As only a set amount of pieces can be used, some tasks are mutually exclusive with others as they each utilize the maximum amount of pieces to be most effective in their scoring. An example from this year's game is the moon base and lava tube area, which both utilize the blue poms.

All of these aspects were integrated through dynamic programming via Python. Dynamic programming (DP) is a method used by algorithms to simplify complex problems by breaking them down into smaller subproblems. This approach is beneficial in optimization problems, as it breaks the optimization problem down into a sequence of decisions to maximize the objective function [7]. The method used by dynamic programming involves solving each subproblem and storing its solution to avoid the need to recompute it every time it is needed.



[8]

Methods:

Defining Parameters:

The Botball game indirectly and directly outlines various requirements for the algorithm, such as the timeframe, scores, and difficulties for every task. KIPR clearly outlines the score and timeframe for the robots, though this does not account for many specifics and difficulties. Though technically, both robots have a time of two minutes to run, typically, there is a delay

between the commencement of both robots' algorithms, either intentionally or unintentionally. On average, teams will usually have a difference of 30 seconds between robot activation, which sets a different timeframe for the Wombat and Create. Difficulty is assigned due to three parameters: complexity, feasibility, and percentage of failure in the tasks. This was then outlined along with the score and time in a larger table (Table 2) to create the list for the algorithm to run.

WOMBAT

Name of Task	Score	Time	Difficulty
Area w/ Sorted Poms	25	10	1
Rover Bays (L and S)	10	15	1
Rock Heap	48	30	1
Solar Panel	50	25	2
Solar Panel w/ Astronauts (ME)	250	35	3
Moon Base (w/ pulled airlock)	1575	50	4
Astronauts(1 station is unreachable by link) (ME)	800	60	4
Lava Tubes Flipped Switch (enabler)	25	0	1

CREATE

Name of Task	Score	Time	Difficulty
Habitat Construction	300	25	2
Area w/ Sorted Poms	25	10	1
Fuel	15	5	1
Astronauts (one station)	100	15	2
Multipliers	MultipliableArea* 5	25	3
Lava Tubes	500	35	4

(Table 2) Table of the tasks, scores, amount of time, and difficulty for the robots

```
55  tasks = [("Area w/ sorted poms", 25, 10, 1), ("Rover bay", 15, 10, 1), ("Rock Heap", 48, 30, 2), ("Solar Panel", 50, 25, 1),
56           ("Solar Panel w/ Astronauts", 250, 35, 4), ("Moon Base w/ pulled airlock", 1575, 50, 4),
57           ("Astronauts(1 station is unreachable by link)", 800, 60, 4), ("Habitat Construction", 300, 30, 2), ("Fuel Poms", 15, 5, 1), ("Lava Tubes", 500, 50, 4)]
58
```

(Figure 3) Tasks in a list

Coding Methodology:

The methodology for the coding process is displayed below:

Objective: Select tasks to maximize the total score while minimizing the total difficulty within two different timeframes (primary and secondary).

Constraints:

- Each task has a score, time requirement, and difficulty level.
- The primary and secondary knapsacks have distinct time capacities.
- Some tasks may be mutually exclusive.

Task Definition:

- Task name, Score, Time required, Difficulty level

Initialization:

- Utilize DP to solve the knapsack problem for both primary and secondary timeframes
- Create a DP table to store the maximum score for each capacity
- Create a corresponding difficulty table (difficulty) for the total difficulty.
- Initialize an included matrix to track the tasks in the solution for each capacity.

Task Processing:

- Iterate through all tasks for maximum score to the task's time requirement:
- Calculate the new potential score and difficulty if the task is included.
- Update the DP and difficulty tables if the new score is better or similar (within the tolerance) to the current best score and has a lower difficulty.

Knapsack Optimization

- Use DP for both primary and secondary knapsack capacities.
- Identify and select tasks that yield the maximum score independently within the two timeframes.
- Determine if any mutually exclusive tasks are selected within the primary knapsack and adjust the remaining tasks for the secondary knapsack based on mutual exclusivity.
- Combine selected tasks from both primary and secondary knapsacks and calculate the total score and difficulty for the combined tasks.

Following this structured methodology, the multi-objective optimization process can be systematically implemented to balance maximizing scores and minimizing difficulties for the given tasks within the specified timeframes.

Algorithm:

double_knapsack function: This is the main function that optimizes the double knapsack. It takes the list of tasks, primary and secondary timeframes, and a score tolerance. It calls the knapsack function twice: first for the primary knapsack and then for the secondary knapsack.

```
1   def double_knapsack(tasks, primary_timeframe=120, secondary_timeframe=90, tolerance= 10):
```

(Figure 4)

Knapsack Function:

- This function solves the knapsack problem for a given set of tasks and capacity using dynamic programming to find the optimal combination of tasks to maximize the score while considering the difficulty. Inside the knapsack function, dp and difficulty are

initialized as 2D arrays to store scores and difficulties, respectively, for each capacity. It maintains two DP tables ("dp" for score and "difficulty" for difficulty) and the "included" matrix to track selected tasks.

- The for loop iterates through each task (task_name, score, time, diff) in the list "tasks". It skips the task specified by exclude_task (used for secondary knapsack when tasks from the primary are excluded). It then iterates backward through possible capacities in variable "t" from the variable "capacity" down to "time - 1".

```
def knapsack(tasks, capacity, exclude_task=None):
    n = len(tasks)
    # Initialize the DP table and the difficulty table
    dp = [[0] * (capacity + 1) for _ in range(2)]
    difficulty = [[float('inf')] * (capacity + 1) for _ in range(2)]
    included = [[False] * n for _ in range(capacity + 1)]

    for i, (task_name, score, time, diff) in enumerate(tasks):
        if exclude_task is not None and i == exclude_task:
            continue
        if time > capacity:
            continue
        for t in range(capacity, time - 1, -1):
            # Update the DP table to consider both score and difficulty
            new_score = dp[0][t - time] + score
            new_difficulty = dp[1][t - time] + diff
            if (new_score > dp[0][t]) or (abs(new_score - dp[0][t]) <= tolerance and new_difficulty < dp[1][t]):
                dp[0][t] = new_score
                dp[1][t] = new_difficulty
                included[t] = included[t - time][:]
                included[t][i] = True

    max_score = max(dp[0])
    max_time = dp[0].index(max_score)
    selected_tasks = [tasks[i] for i, included_task in enumerate(included[max_time]) if included_task]

    return selected_tasks, max_score, dp[1][max_time]
```

(Figure 5)

- The function updates the dp table for each task and each capacity. If the task is included, it calculates a new_score and new_difficulty. It updates dp[0][t] (score) and dp[1][t] (difficulty) if the new solution provides a higher score or if the score is within score_tolerance of the current best score and has a lower difficulty.
- After processing all given parameters, the function identifies the maximum score (max_score) achievable within the given capacity (capacity). It retrieves the list of selected tasks (selected_tasks) that contribute to this maximum score based on the "included" matrix.

Mutual Exclusivity and Calls:

```
    # Solve for primary knapsack
    selected_tasks_primary, max_score_primary, total_difficulty_primary = knapsack(tasks, primary_timeframe)
    print(selected_tasks_primary)
    # Determine if task is in primary knapsack
    mutually_exclusive = any(task[0] == "Moon Base w/ pulled airlock" for task in selected_tasks_primary)

    # Remove tasks already selected for primary knapsack
    remaining_tasks = [task for task in tasks if task not in selected_tasks_primary]

    # Solve for secondary knapsack, exclusivity
    exclude_task = None
    if mutually_exclusive:
        for i, (task_name, _, _, _) in enumerate(remaining_tasks):
            if task_name == "Lava Tubes":
                exclude_task = i
                break
    selected_tasks_secondary, max_score_secondary, total_difficulty_secondary = knapsack(remaining_tasks, secondary_timeframe, exclude_task)
    print(selected_tasks_secondary)
    # Combine results
    total_score = max_score_primary + max_score_secondary
    combined_selected_tasks = list(set(selected_tasks_primary + selected_tasks_secondary))
    total_difficulty = total_difficulty_primary + total_difficulty_secondary

    return combined_selected_tasks, total_score, total_difficulty
```

(Figure 6)

- Then, the double_knapsack function first calls knapsack to solve for the primary knapsack (selected_tasks_primary) and checks if the task "Moon Base w/ pulled airlock" is included in it. If true, it sets up to handle exclusivity in the secondary knapsack solution.
- It then removes tasks already selected in the primary knapsack from tasks and calls the knapsack again to solve for the secondary knapsack (selected_tasks_secondary) and excludes the task (Lava Tubes) related to the mutual exclusivity if needed
- Finally, the code returns the combined tasks, scores, and difficulties

Output:

```
[('Area w/ sorted poms', 25, 10, 1), ('Moon Base w/ pulled airlock', 1575, 50, 4
), ('Astronauts(1 station is unreachable by link)', 800, 60, 4)]
[('Solar Panel', 50, 25, 1), ('Solar Panel w/ Astronauts', 250, 35, 4), ('Habita
t Construction', 300, 30, 2)]
Selected Tasks: [('Area w/ sorted poms', 25, 10, 1), ('Solar Panel', 50, 25, 1),
 ('Astronauts(1 station is unreachable by link)', 800, 60, 4), ('Solar Panel w/
Astronauts', 250, 35, 4), ('Moon Base w/ pulled airlock', 1575, 50, 4), ('Habita
t Construction', 300, 30, 2)]
Total Score: 3000
Total Difficulty: 16
```

(Figure 7)

- Figure 7 demonstrates the function's output, with an optimal total score of 3000 and a total difficulty value of 16. The selected tasks were Habitat Construction, Astronauts(1 station is unreachable by link), Solar Panel with Astronauts, Moon Base with pulled airlock, Area with sorted poms, and Flipping the Solar Panel.
- The primary knapsack (Create Robot) performed the following tasks: Area with sorted poms, Moon Base with pulled airlock, and the Astronauts (1 station is unreachable by link).
- The secondary knapsack (Link Robot) was performed by flipping the solar panel, solar panel w/astronaut, and habitat construction.

Conclusion:

This research presents a novel approach to optimize task allocation for dual-robot teams in Botball competitions through a two-part knapsack algorithm integrated with mutual exclusivity constraints and multi-objective optimization (MOO). Considering the interplay between score, difficulty, and time, this methodology ensures that both robots in a team can efficiently maximize their contributions within the limited competition timeframe.

The dual-robot optimization uses dynamic programming to systematically simplify the complex problem of task allocation into manageable subproblems. By applying the knapsack algorithm twice—first for the primary knapsack (Create Robot) and then for the secondary knapsack (Wombat Robot)—the model accommodates each robot's capabilities and timeframes. Mutual exclusivity is incorporated to prevent overlapping tasks that could lead to inefficiencies or physical interferences.

Through this structured approach, the model successfully identifies each robot's optimal set of tasks, balancing high-value yet time-consuming tasks with more difficulty against quicker, more accessible, lower-value ones. The practical application of this model in the Botball competition setting demonstrates its effectiveness, as shown by the simulation output, achieving a total score of 3000 with a total difficulty of 16. This result highlights the value of advanced optimization techniques in enhancing strategic decision-making and overall team performance.

In conclusion, the two-part knapsack algorithm with mutual exclusivity constraints and MOO principles provides a robust framework for optimizing dual-robot task allocation in competitive robotics. This approach maximizes scoring potential and ensures a balanced distribution of task difficulty, allowing for more efficient and successful robot performances in Botball competitions. Future work could expand on this model by further incorporating real-time adjustments and machine learning techniques to refine task allocation strategies in dynamic competition environments.

References:

1. Cacchiani, Valentina, et al. “Knapsack Problems — an Overview of Recent Advances. Part I: Single Knapsack Problems.” Computers & Operations Research, Feb. 2022, p. 105692, https://doi.org/10.1016/j.cor.2021.105692.
2. K. Deb, Multi-Objective Optimization using Evolutionary Algorithms, John Wiley & Sons, Inc., 2001
3. 2024 Botball Game Review v2.0
4. Suhas, C. “A Study of Performance Analysis on Knapsack Problem.”International Journal of Computer Applications (0975 – 8887) National Conference on “Recent Trends in Information Technology” (NCRTIT-2016)
5. “Implementation of 0/1 Knapsack Using Branch and Bound.” GeeksforGeeks, 29 Apr. 2016,www.geeksforgeeks.org/implementation-of-0-1-knapsack-using-branch-and-bound/. Accessed 23 June 2024.
6. *“The Knapsack Problem”* UT Dallas
7. Sha, Huajing, et al. “Overview of Computational Intelligence for Building Energy System Design.” Renewable and Sustainable Energy Reviews, vol. 108, July 2019, pp. 76–90, https://doi.org/10.1016/j.rser.2019.03.018.
8. Dixit, Ananya. “Dynamic Programming.” Scaler Topics, 20 Dec. 2021, www.scaler.com/topics/data-structures/dynamic-programming/.