Algorithmically Optimize Botball Competition Strategies
Yimo Xu
ITCCC - Scuderia Canotage Wanderers

Algorithmically Optimize Botball Competition Strategies

Abstract: This paper focus on finding and optimizing competition strategies in Botball tournaments by mathematically modeling the competition playground and tasks as well as adopting algorithms in computer science and statistics.

Keywords: Botball, Algorithm, Statistics, Strategy

1 Introduction

Determining a strategy in Botball Competition can be a complicated task, as a large variety of factors are to be considered, including time cost, score estimation and rate of success (robustness). As an alternative to imagining a perfect plan, computer programs can be employed to find the best solutions more scientifically, after the whole playground is modeled in a proper manner.

2 Modeling

The whole playground can be modeled as a graph for analysis. Since robots can move freely in designated areas, the graph should be cyclic. However, some tasks have prerequisites. For example, the frisbee will score higher only if it's in the right block with yellow sponge block, so the graph will be directed.

2.1 Features of the Graph

Theoretically, the graph will have n *nodes*[1], indicating n tasks[2] or waypoints (a point which the robot will pass by while no task may be done here). Among these nodes, there are m *edges* connecting these notes[1]. We name all graphs that are modeled from Botball Competition playground "Botball Graph", and in recent years, it should contain two identical directed cyclic subgraphs connected with two reversed edges between two identical nodes.

[1] Also referred as "Vertices" in Graph Theory

[2] Performing one task in different situations, e.g. carrying different numbers of poms, will be considered as performing different tasks if the score varies.

Please refer to Appendix 0 to see a small portion of this year's graph and its explanation.

2.2 Other definitions and inferences

2.2.1 Features of Nodes

We define, for every Botball Graph:

1. There is a weight, $W_{i,j}$, for every edge connecting node i and j indicating the time that will cost to travel between these two nodes.
2. There is also a weight, T_i, for every node i, indicating the time that will cost to finish the task in node i.
3. There is an income, M_i, for every node i, indicating the mean value of points the team will earn after finishing the task in this node, and finally,
4. There is a success rate P_i, for every node i, indicating the rate of successfully finishing the task, assuming that success rates are distributed binomially. Do keep in mind that we will assume that the robot cannot perform any further tasks if it failed to do one task.

Specially, we define that for all waypoint node i, $T_i = 0, M_i = 0, P_i = 1.00$, and for all task node j, $T_j, M_j > 0$ and $0 < P_i < 1$, initially.

There exists one and only one starting node and it should be a waypoint node. The robot can stop at any node, including the starting node.
A task node may be "used", means that a task may be finished. A "used" task node will become a waypoint node. A task node that is not used when the robot is currently at also behaves the same as a waypoint node.

2.2.2 Fail Rate of the Strategy

As for the fail rate for a whole strategy S, we define it as the sum of all failing possibilities. Mathematically, it is defined as:

$$\mathrm{P(S\ Fail)} = \sum_{\mathrm{i=1}}^{\mathrm{n}} P(i^{th}\ node\ fail)$$

Where:

$$P(i^{th}\ node\ fail) = (1 - \mathrm{P_i}) \prod_{c=1}^{i-1} P_c$$

And S is an ordered list such that:

$$\mathrm{S} =< \mathrm{i}, \mathrm{j} \ldots, \mathrm{k} >$$

Where $\mathrm{i}, \mathrm{j}, \ldots, \mathrm{k}$ are elements within the Botball Graph.

2.2.3 Reasons why two nodes are unreachable

Some nodes may only be reachable by some specific point due to the following reasons:

1. Prerequisites. Some tasks need prerequisites to score the indicated M_i, for example, the frisbee must be put into the area with three yellow sponge blocks in order to score 200 points.
2. Angle. For example, the most optimized way to catch the frisbee is by going through the position where the group of poms which are nearest to the starting area is.

3 Design of the algorithm

3.1 Prototype

Calculating what we need in a cyclic graph would be inefficient. The easiest main idea is using Depth-First-Search, as known as DFS. To make sure that the search does not end in a dead loop, we need to maintain a stack which pushes in the current node when we make any move. Afterwards, when the time reaches the limit (which is 120 seconds), or we find two identical nodes with all nodes between them are "used" nodes or waypoint nodes, which indicates that we entered a cycle that cannot let us gain any extra score[3], current search will be terminated.

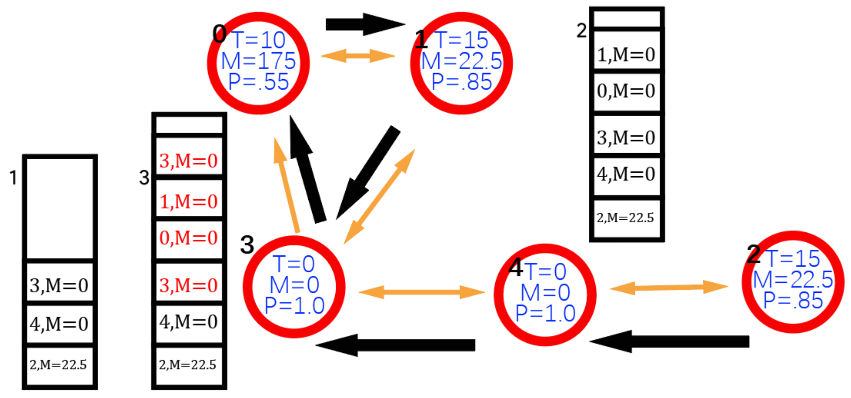


Fig 3.1 Simple demonstration of the concept

However, this method is so complex both in time and space, making it a less acceptable algorithm as we need to deal with a much complex graph, and the searching process requires too much redundant work.

[3] This seems obvious, nevertheless, to understand it mathematically, please refer to [2] J. L. Gross and J. Yellen, "Graph Theory and Its Applications, Second Edition," Taylor & Francis, 2005, pp. 197-207..

3.2 Changing the approach

Another approach may be used in the calculation as a more proper method to figure out a more realistic strategy. The program will do the following tasks:

1. Find the optimal path (shortest path) from the starting node to all tasks going through other task nodes.

The team will need to pick one end with reasonable incomes, relatively low but acceptable success rate (since it's the last one), and reasonable time needed. Afterwards, the program will:

2. Determine if there is any node within a cycle, and if tasks inside the cycle can be done, then
3. Determine which tasks among the way can be done, then find the optimal strategy

The program will calculate the success rate of this strategy afterwards. If it is too easy to fail, another strategy may be calculated by cancelling the node with lowest success rate.

3.3 Finding the shortest path

Our graph is directed and with no edge with negative weight since no time can be added to the total competition time. Among all suitable algorithms[4], Dijkstra's algorithm is the fastest with determined complexity. Since our graph has no edge with negative weight, we don't have to use an algorithm that is able to find cycles with negative weight[5].

3.3.1 Concept explanation

Dijkstra's algorithm can find the shortest path from one source to all nodes within graphs with no edge that has negative weight[4]. It's time complexity, without optimization using Fibonacci heap[6] is $O(n^2)$.

In the algorithm, two sets will be maintained: S, which contains nodes that have already know what the shortest path to the source is and U, which contains nodes that haven't. We continuously find the node with the shortest path to the source node in U, remove it and add it in S. Then, update all nodes in U the shortest distance from the source point to them using data in S until U is empty.

[4] Which include Depth-First Brute-Force search, Dijkstra, Floyd-Warshall, Bellman-Ford, SPFA, etc.

[5] For further understanding, please refer to [3] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, "Introduction to Algorithms," MIT Press, 2009, pp. 615-623.

[6] It' s unnecessary to do so. Optimized complexity is $O(E + V\log V)$.

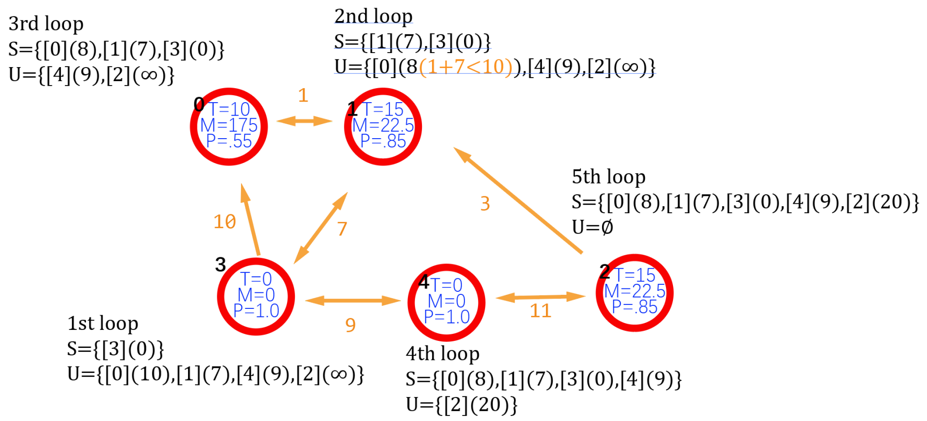


Fig 3.2 Simple demonstration on how Dijkstra's algorithm work

3.3.2 Running result

The shortest path with reasonable income is moving the black frisbee on the far side by going through all 6 groups of poms and the position that can move the trolley.

3.4 Finding loops

3.4.1 Concept Explanation

We will use Depth-First Search to find all loop that starts from any node within the nodes in the strategy, which in this case is the fastest[5]. The main idea is similar to that discussed in 3.1. The program will advance to all children of the current node until it reaches a non-starting which has been before or a node with an out-degree of 0 (which means it has no child nodes). If the program reaches the starting node or any node in the shortest-path node after the starting node, it will record this loop and terminate pop out the current stack.
Afterwards, we will re-construct the cycle $S = < N_1, N_2, \dots, N_n >$ as a node N where:

$$T_N = \sum_{i=1}^{n} T_{N_i} + \sum_{i=1}^{n-1} W_{N_i N_{i+1}}, M_N = \sum_{i=1}^{n} M_{n_i}, P_N = 1 - (\sum_{i=1}^{n} (1 - P_{N_i}) \prod_{c=1}^{i-1} P_{N_C})$$

N will have the same in-degree as T_{N_1} and the same out-degree as T_{N_n}.

To simplify the calculation, nodes will be permutated according to decreasing order of P_N, which means nodes with lower P_N will be in the back if there are multiple nodes reconstructed.
This is reasonable since in such single-direction loop, tasks represented by latter nodes will depend on any node in front of it in topological order.
See Appendix 1 for a detailed example with explanation.

3.4.2 Running result

Reasonable result includes catching all poms on the ground and putting them inside the trolley, putting Botguy inside the trolley, catching green frisbee and put it to the area where 3 yellow sponge blocks reside (assuming another robot has already moved big blocks to the designated location) and putting 3 sets of palm seeds (orange poms) inside the PVC.

3.5 Selecting tasks

After two steps described above, we will have a DAG (directed acyclic graph) containing all possible tasks that can be performed. As a common sense, when solving problems involving expectations within a DAG, dynamic programming methods can be used[6].

3.5.1 Concept Explanation

The main idea of dynamic programming in this problem is a bit like 0-1 Knapsack problem. Assume we have a queue[7] Q storing all nodes $< \mathrm{N}_1, \mathrm{N}_2, \ldots, \mathrm{N}_n >$, where N is sorted in decreasing topological order[8].
Then, we define, for a task that needs to be finished in time C, the maximum expected value in node N is $\mathrm{E}_{\mathrm{N,C}}$. Assume $\mathrm{S_N}$ is a set which contains all child of node N and $\mathrm{K_N}$ is any node such that $\mathrm{K_N} \in \mathrm{S_N}$. We apply dynamic programming's idea:
We can either choose to execute the tasks in node N, and accept a score gain $\mathrm{M_N}$ with a time lost $\mathrm{T_N}$, or not, and proceed to any of its children, then do the same calculation. Since there's no aftereffect now, indicating whether doing the previous task will have no effect on latter tasks' optimal strategy, the optimal strategy of node N will include the optimal strategy of one of its children (or its only child).
Bearing this in mind, this formula will come out:

$$\mathrm{E}_{\mathrm{N,C}} = \max\left(E_{K_N, C-W_{N,K_N}}, E_{K_N, C-W_{N,K_N}-T_N} + M_N\right) \text{ when } \mathrm{C} \neq 0, \mathrm{K_N} \neq \emptyset$$

$$\mathrm{E}_{\mathrm{N,C}} = 0 \text{ when } \mathrm{C} = 0 \text{ or } \mathrm{K_N} = \emptyset, C < T_N$$

$$\mathrm{E}_{\mathrm{N,C}} = \mathrm{M_N} \text{ when } \mathrm{K_N} = \emptyset, C \geq T_N$$

In the program, according to dependencies, we will calculate all $\mathrm{E}_{\mathrm{N,C}}$ in increasing topological order. $\mathrm{E}_{\mathrm{N_0},120}$ will be our final answer. An additional list should be maintained to store current best strategy selection.
Finally, fail rate is calculated.

[7] A First-In-First-Out(FIFO) Linear List.
[8] Which means for any $0 < \mathrm{i} < \mathrm{j} \leq \mathrm{n}, \mathrm{I}, \mathrm{j} \in \mathrm{N}^*$, $\mathrm{N_j}$ cannot be $\mathrm{N_i}$' s ancestor.

3.5.2 Running Result

The final running result is to grab 4 groups of poms and put them into the trolley, put 3 groups of orange poms into the PVC pipe, grab the green frisbee and put it in the designated area, finally put the black frisbee to the top position. The fail rate is 0.04, which is acceptable.

3.6 Alternation of the algorithm

This algorithm should be working on all valid Botball Graphs. However, due to technical limitations, some tasks may be unable to perform; High fail rate may also refute what the program solves.
The solution is to alter the graph itself, specifically by "disabling" a node, which means to set the T value of the node to an unreasonably big number so that it will not be reasonable to perform the task represented by the node, also resulting in a decrease of fail rate.
Also, since this algorithm can only process situations with only one robot, the graph can be altered to match how the playground will be after another robot has already finished its job.

4 Conclusion and Future

According to statistical analysis and few experiments made on Botball playgrounds from season 2016 to 2018 compared to the optimal strategy we figured out with our mind, the conclusion that such program can improve the final score and success rate is correct with an order of magnitude.

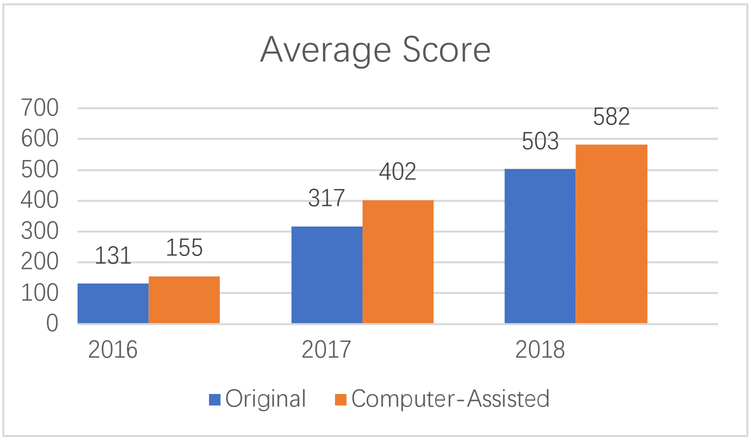


Fig 4.1 Average Score of Different Strategies

Since this algorithm will work not only on one specific Botball playground, it may be used in real-life decision-making situations, when multiple tasks are to be selected to be done or not and there is time consumption when travelling from task to task.
Nevertheless, while such algorithm can help determine optimal strategies easily, it has limitations to some degree. For example, it can only figure out the optimized strategy for one robot, and some unreasonable strategies may be hard to avoid due to modeling limitations (season 2016 in particular).

It is clear that much additional work will be required before we gain a sound and comprehensive understanding and such an algorithm is improved.

Acknowledgments

I would like to express my very great acknowledgements to several members in our school's Computer Olympiads team, especially M.S. Ding for assistance with the mathematical explanation of Dijkstra's Algorithm and Z.K. Wen for valuable discussions. I also would like to thank the following companies for their assistance with the collection of my data: X. Yu and X.Y. Liu.

Reference

[1] N. Biggs, E. K. Lloyd, and R. J. Wilson, *Graph Theory, 1736-1936*. Clarendon Press, 1986.

[2] J. L. Gross and J. Yellen, "Graph Theory and Its Applications, Second Edition," Taylor & Francis, 2005, pp. 197-207.

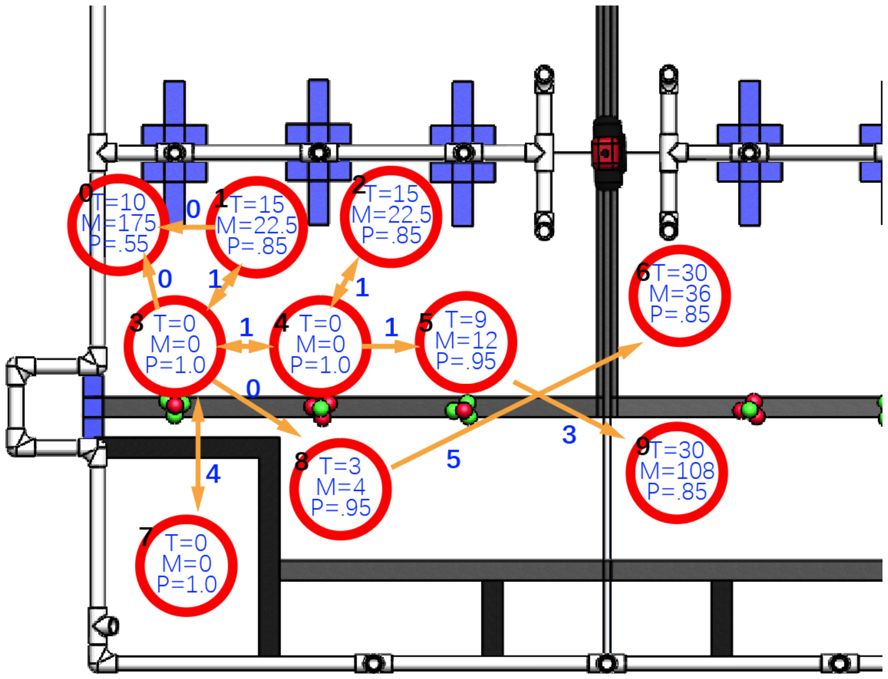
[3] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, "Introduction to Algorithms," MIT Press, 2009, pp. 615-623.

[4] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische Mathematik,* journal article vol. 1, no. 1, pp. 269-271, December 01 1959.

[5] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, "Introduction to Algorithms," MIT Press, 2009, pp. 603-615.

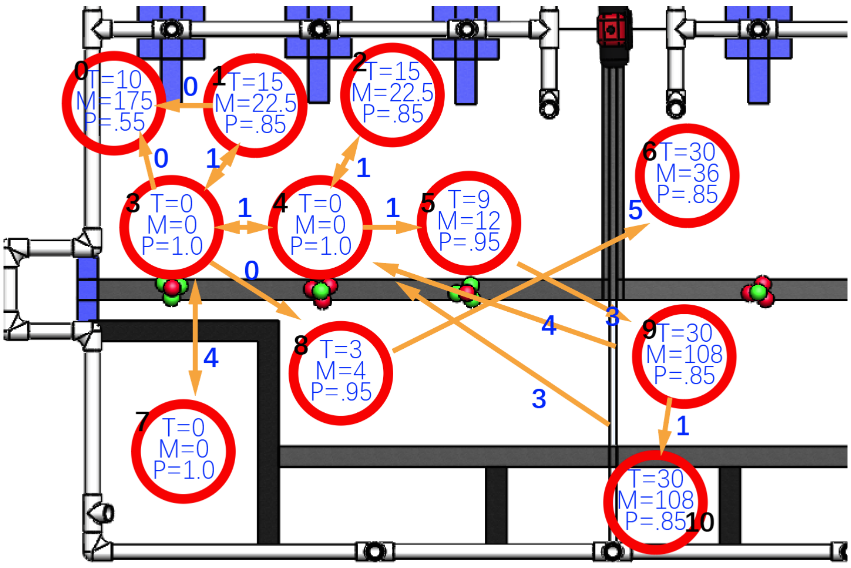
[6] D. Panigrahi, "Dynamic Programming," Available: https://www2.cs.duke.edu/courses/spring16/compsci330/Notes/dynamic.pdf

Appendix 0



0: Catching the frisbee.
1: Putting orange poms into PVC.
2: Putting orange poms into PVC.
3: Waypoint – catch one group of poms on the ground or go to column on the left.
4: Waypoint – catch all poms on the ground, go back to 3 or go to column in the middle.
5: Catching all poms on the ground.
6: Putting one group of poms, which are collected in 8, into trolley.
7: Waypoint – starting area.
8: Catching one group of poms on the ground.
9: Putting all poms, which are collected in 5, into trolley.

Appendix 1



The sample search starts from node No. 4.

Attempts (X means termination due to node repetition, O means termination due to an out-degree of 0), sets in red means plausible ones:

[4, 2, 4]	[4, 3, 1, 0, O]	[4, 3, 8, 6, O]
[4, 3, 0, O]	[4, 3, 4]	[4, 5, 9, 4]
[4, 3, 1, 3, X]	[4, 3, 7, 3, X]	[4, 5, 9, 10]