Robotics Report

January 2022

UP940148
University of Portsmouth
School of Computing

***Abstract*—This report will look at improving the functionality of the KUKA YouBot to retrieve items from shelves in a warehouse through the use of sensory intelligence and the Multi-Fragment heuristic approximation algorithm for the Travelling Salesman Problem.**

I. Problem Outline

The problem that this report will look into is gathering items in a distribution warehouse environment. This problem was chosen as it is an area in which robots have large potential.

This problem involves locating and collecting items in a warehouse, and delivering them to a collection point, where they would then be processed further for distribution.

II. Approach

A. Assumptions

For the purpose of this solution, some assumptions have been made. These are listed below.

- Shelves are organised neatly with even spacing between items.
- Only one type of item will be stored on each shelf.
- YouBot will be the only robot operating in the warehouse. Furthermore, there will be no people present in the warehouse whilst YouBot is operating.
- All items to be collected have identical dimensions and weight.
- There is a main computer which sends orders to YouBot and keeps updated records of warehouse layout.

B. Overview

This report aims to build upon the pre-existing system in which YouBot solves the Hanoi Tower Problem, which is included in V-REP [1], to allow YouBot to accept a list of boxes which have been requested, all of which are identified by unique colours, and locate then deliver them to a conveyer belt which would take the boxes away from the warehouse floor.

The warehouse floor will have nodes on the floor at shelves and intersections to mark out locations that YouBot can navigate to. These will help YouBot plan what route to take when collecting the items, as each node forms a vertex in a graph, and a shortest path-finding algorithm can then be applied to said graph.

C. Object Collection and Output

When collecting the items. YouBot will know which shelf the item should be stored on, however it will not know the exact locations of the items on their respective shelves. In order to detect the items and pick them up, some form of sensor will be required. A pressure sensor could be installed into each shelf which would send the location information to YouBot, a ray proximity sensor could be installed at the end of each shelf, facing inwards, which would be able to tell YouBot how far along the shelf the item is, or YouBot could be fitted with a vision sensor which it could use to locate the items itself.

For this report, a vision sensor will be attached to YouBot's gripper, which will then be used to determine whether the gripper is currently pointed at an object. A vision sensor was chosen for this over a proximity sensor because it will allow objects to be identified by multiple factors, such as colour and shape. Colour and shape detection are considered to be outside the scope of this report and as such, they shall not be applied here, however they could be additions made in future reports.

YouBot will have some modifications made to its platform in the form of short wall like structures. These will allow YouBot to navigate the warehouse without the risk of items falling off the platform. It also keeps them in positions that are easy to reach when the items are then removed from the platform later.

When delivering items to the output conveyor, a proximity sensor will be added just under YouBot's arm, facing towards the mounted platform. This sensor will then be used to detect objects currently being stored so that they can be easily located and removed from the platform, and then placed on the output conveyor (Fig. 15).

D. Network Creation

To begin creating the network, it had to be decided where each of the shelves within the warehouse would be positioned. For this report a simple setup was chosen, in which the warehouse was given eight shelves set up in a grid of two columns and four rows. This means that the network will need a total of twelve nodes (Fig. 3). One at each shelf, and one at each intersection, as shown in Fig. 11.

The output conveyor belt will not be given it's own node, as YouBot will have direct access to the output from node $\boldsymbol{A}$, and once output of the items has been completed, YouBot will then return to node $\boldsymbol{A}$ and await new instructions.

E. Route Calculation

This report will apply the Multi-Fragment heuristic approximation algorithm (MF algorithm), first defined in 1992 by John Bentley [2], to calculate the best route for YouBot to take in order to gather all the requested items in the shortest time.

The MF algorithm is well suited to this scenario as the overall network graph can be taken, and a sub-graph can be created out of all the nodes that YouBot needs to visit, which will be the shelves that the requested items are stored on. Then the problem of visiting all the nodes in the sub-graph can be solved in the same way that one would solve the Travelling Salesman Problem. For a small network, this solution may be unnecessarily complex; however, one of the main benefits of using the MF algorithm to solve this problem is that the results of this report can be scaled up to a warehouse of any size and complexity, and the process followed by YouBot will be the same.

Because MF is a heuristic approximation, increasing the number of nodes (n) being evaluated will not cause as large an increase in time taken as a more accurate approach would, such as a brute-force algorithm. The worst-case performance time complexity of MF is $\mathcal{O}(n^2 \log n)$[2] versus a brute-force algorithm, which would have to search all $n!$ permutations, which has a time complexity of $\mathcal{O}(n!)$[3].

1) Setup: The MF algorithm builds on Prim's algorithm [4] and therefore can only be applied to a connected network, i.e. one which can have a minimum spanning tree (MST) constructed out of it's nodes. Whilst our main network is connected, it is not guaranteed that any sub-graph we create will also be connected. If the request requires us to visit nodes ***B***, ***C***, ***E***, and ***F***, there are no connections between those nodes in our current matrix (Fig. 5) and therefore the created sub-graph would be disconnected.

To overcome this issue, we must build upon our distance matrix so that any sub-graph created will be connected. To do this, we can use the Floyd-Warshall algorithm [5], with path reconstruction [6], to calculate the shortest distance and path between every pair of nodes in the network (Algorithm. 1).

2) Calculation: Once the distance matrix and route matrix have been created, YouBot will need to calculate the optimal route to take. It is assumed that both the distance and route matrices will be generated outside of YouBot's programming and will provided to YouBot when needed. So when simulating this in V-REP, these matrices will be assigned as global variables and will not be calculated at run-time.

To calculate the optimal route, YouBot will need to read through the requested items and determine which unique shelves it needs to visit, as well as how many items are required from each shelf. To do this we just read through the request and create a new pair of lists, one keeping track of unique shelf indexes, and the other counting how many items are needed from each shelf (Algorithm. 2). Once the request has been broken down into these lists, we can apply the MF algorithm on the sub-graph created from the unique nodes.

Algorithm 1 Floyd-Warshall algorithm with path reconstruction [7], [6]

```
let dist be a |V| × |V| array of minimum distances initialized to ∞ (infinity)
let next be a |V| × |V| array of vertex indices initialised to null
for each edge (u, v) do
    dist[u][v] ← w(u, v) // The weight of the edge (u, v)
    next[u][v] ← v
end for
for each vertex v do
    dist[v][v] ← 0
    next[v][v] ← v
end for
for k from 1 to |V| do
    for i from 1 to |V| do
        for j from 1 to |V| do
            if dist[i][j] > dist[i][k] + dist[k][j] then
                dist[i][j] ← dist[i][k] + dist[k][j]
                next[i][j] ← next[i][k]
            end if
        end for
    end for
end for
```

Algorithm 2 Sort request into unique nodes and item count per shelf

```
let order be a list containing the required nodes // Required nodes should be assigned based on item name
let nodes be an empty list
let visits be an empty list
for each item in order do
    index ← nodes[item] // Index of nodes in which item appears, or -1 if not in list
    if index = -1 then
        nodes.push(item)
        visits.push(1)
    else
        visits[index] ++
    end if
end for
```

The MF algorithm takes a list of all relevant edges from the distance matrix, where a relevant edge is any edge connected to any required node, organised by weight in ascending order, and goes through each edge to check whether it can be used in the final route (Algorithm. 3).

The rules used to create the path are as follows [2]:

1) If one of the edge's nodes appears in the middle of a fragment, the edge is discarded.
2) If neither of the edge's nodes appear in any existing fragments, then a new fragment is created from that edge.
3) If one of the edge's nodes appears at the end of a fragment (A) and doesn't appear on the end of any other fragment, then the edge gets added to fragment A.

4) If one of the edge's nodes appears at the end of a fragment (A) and the other node appears at the end of a fragment (B), then the fragments and edge are joined together to make one larger fragment (fragment A + edge + fragment B), providing the larger fragment does not form a closed tour which doesn't already include all the nodes required.

The first edge will always be used as it complies with rule 2. From there you check every edge in order until one fragment contains all the required nodes, the fragment is then considered to be a full tour and the algorithm is finished.

Once the MF algorithm has finished finding the best route, the output will be in the form of a list containing every node, in the order that they need to be visited. This can then be returned and the final stage of route calculation can begin.

Algorithm 3 Multi-Fragment algorithm as described by Krari [8, p. 6]

Require: Sorted set of all edges of the problem *E*.
Ensure: A tour *T*.
  for each *e* in *E* **do**
    if (*e* is closing *T* and size(*T*) ¡ *n*) or (*e* has a city already connected to two others) **then**
      go to next edge
    end if
    if *e* is closing *T* and size(*T*) = *n* **then**
      add *e* to *T* **return** *T*
    end if
    add *e* to *T*
  end for

  return *T*

If we were to put in a request to go to nodes ***B***, ***L***, and ***F***, then using these three algorithms, we would end up with an output list *[**B**, **F**, **L**]* which whilst it does form a closed tour, there's no direct connections between those nodes. So we have to use the path reconstruction section of the Floyd-Warshall algorithm in order to get a traversable route. The path reconstruction is a relatively simple algorithm with no intensive calculations.

For each node in the list, we need to find the specific path to take to get to it. Let's use the list above again, *[**B**, **F**, **L**]*.

To find the path from ***B*** $\Rightarrow$ ***F*** we look at the route matrix (Fig. 6). We start by putting ***F*** in an empty list. Then look at row **B**, column **F**. We can see in this position we have ***D***, so we add it to the list and next we look at row **B**, column **D**, where we can see we have ***A***, which we add to the list as well, and now finally we look at row **B**, column **A** and we see we have ***A*** again, because we have now found ***A*** twice in a row, we know there are no more nodes to visit before this one, and so we can add ***B*** to the list and reverse it to give us *[**B**, **A**, **D**, **F**]*. This is then the path to take from ***B*** $\Rightarrow$ ***F*** (Algorithm. 4) [6, p. 76].

This process can be applied for each item in the list returned from the MF algorithm, to give a detailed route for YouBot to travel.

Algorithm 4 Floyd-Warshall path reconstruction as described on Wikipedia [7]

procedure $path$(u, v)
  if next[u][v] = null **then**
    return []
  end if
  path = [u]
  while u $\neq$ v **do**
    u $\leftarrow$ next[u][v]
    path.append(u)
  end while
  return path
end procedure

III. Amendments made in development

Throughout the development process, changes may need to be made to the approach for many reasons. Large changes to the established approach will be documented in this section.

A. Network model change

When implementing the path-finding algorithm on the given network shown in Fig. 3, YouBot would stop on every node as it passed them. This meant that a journey from node ***A*** to node ***J*** would pause at nodes ***D*** & ***G***. These pauses added unnecessary lengths of time to the overall process, and so a solution had to be found to prevent this from happening.

I looked at programming a short algorithm which would analyse a given route and eliminate unnecessary nodes, so that if we took the journey from node ***A*** to node ***L***, the algorithm would receive the route ***A*** $\Rightarrow$ ***D*** $\Rightarrow$ ***G*** $\Rightarrow$ ***J*** $\Rightarrow$ ***L***, and would realise that ***A***, ***D***, ***G***, and ***J*** all lay on the same straight line, with no breaks in direction between them. And as such the route would be modified to remove the unnecessary nodes and would simply return the route ***A*** $\Rightarrow$ ***J*** $\Rightarrow$ ***L***.

Implementing such an algorithm ended up being an impractical solution as there was no way of determining what a straight line was in the network without either making the system far more complex by running larger algorithms, or losing system scalability due to hard-coded and network specific functions and values.

In the end, I decided to remodel the whole network to account for these skippable nodes, by adding extra connections between nodes ***A*** & ***G***, ***A*** & ***J***, ***B*** & ***C***, ***D*** & ***J***, ***E*** & ***F***, ***H*** & ***I***, and ***K*** & ***L***, as shown in Fig. 7. This network change allows YouBot to travel across nodes without stopping on them and wasting additional time.

As a result of the altered network structure, the initial distance matrix (Fig.4 had to be changed, also the matrix after the Floyd-Warshall algorithm had been applied (Fig. 5), and the final route matrix (Fig. 6) both had to be re-calculated to reflect the alterations. The updated versions are shown in Fig. 8, 9, and 10 respectively.

IV. Testing

As with any project, the features implemented in this report must be tested to ensure they consistently work as expected. This section will cover the testing process and results.

A. Network navigation

The values for $startNode$ and $endNode$ were generated by rolling a D12 die for each required value.

1) Test 1: First test of $getRoute(startNode, endNode)$ function (before making changes to the network documented in $III.A$) Results shown below in Fig. 1.

Input	Expected Output	Actual output	Pass or Fail
8, 7	7	7	Pass
5, 11	4, 7, 10, 11	4, 7, 10, 11	Pass
12, 10	10	10	Pass
4, 12	7, 10, 12	7, 10, 12	Pass
2, 8	1, 4, 7, 8	1, 4, 7, 8	Pass
3, 12	1, 4, 7, 10, 12	1, 4, 7, 10, 12	Pass
5, 7	4, 7	4, 7	Pass
11, 5	10, 7, 4, 5	110, 5	Fail
1, 6	4, 6	4, 6	Pass
4, 5	5	5	Pass
3, 11	1, 4, 7, 10, 11	1, 4, 7, 10, 11	Pass
12, 6	10, 7, 4, 6	10, 6	Fail

Fig. 1: Results from first test of $getRoute()$ function

2) Test 2: Second test of $getRoute(startNode, endNode)$ function (after making changes to the network documented in $III.A$) Results shown below in Fig. 2.

Input	Expected Output	Actual output	Pass or Fail
12, 6	10, 4, 6	10, 4, 6	Pass
5, 2	4, 1, 2	4, 1, 2	Pass
1, 6	4, 6	4, 6	Pass
5, 11	4, 10, 11	4, 10, 11	Pass
12, 10	10	10	Pass
5, 7	4, 7	4, 7	Pass
10, 2	1, 2	1, 2	Pass
4, 5	5	5	Pass
1, 7	7	7	Pass
1, 10	10	10	Pass

Fig. 2: Results from second test of $getRoute()$ function

B. Item Collection and Output

1) Test 1:

Expectation:

When $request$ is [$'red'$, $'red'$, $'green'$, $'red'$], YouBot should navigate to the red shelf and pick up three items, then navigate to the green shelf and pick up one item, or vice versa. After collecting the items, all four items should be taken to the output conveyor belt.

Result: Pass

YouBot navigated to the red shelf, picked up three items (Fig. 12), then navigated to the green shelf, picked up one item, then went to the output conveyor and successfully output all four items (Fig. 13).

2) Test 2:

Expectation:

When $request$ is [$'red'$, $'green'$, $'blue'$, $'cyan'$, $'magenta'$, $'yellow'$, $'black'$], YouBot should navigate to each specified shelf in turn and pick up one item from each shelf. Once it reaches maximum capacity (three items on platform and one in hand), YouBot should navigate to the output conveyor and deposit all items, then it should return to work and collect the final three items. Then it should go to the output conveyor one more time and output the remaining items.

Result: Pass

YouBot successfully navigated to every shelf, and after the first four, made its way to the conveyor to output everything before continuing (Fig. 14 & 16).

3) Test 3:

Expectation:

When $request$ is [$'red'$, $'red'$, $'red'$, $'red'$, $'red'$], YouBot should navigate to the red shelf, pick up four red items and take them to the output. Then it should return to the red shelf and attempt to get a fifth item, and upon failure to detect one, it should retreat back to the start and end process.

Result: Fail

YouBot successfully retrieved the first four items and delivered them to the output. However when it returned for the fifth item, it search along the shelf, but didn't give up when it got to the end. Instead it continued trying to move until it was stuck on the shelf and couldn't move (Fig. 17).

V. Results Analysis

A. Network navigation

1) Test 1: After getting the results from the first test of my $getRoute()$ function shown in Fig. 1 I realised something was clearly wrong with my algorithm. When $startNode$ was a higher number than $endNode$, the algorithm couldn't properly figure out what route to take. After some investigating, I believe the problem was caused because my route matrix isn't diagonally symmetrical. So I altered the function slightly so that if $startNode > endNode$ they get switched around, and then the resulting list just doesn't get reversed at the end.

Whilst watching YouBot move, I also noticed it will unnecessarily stop on nodes when it could just go straight over them. I decided to tweak my network to stop this issue. Changes documented in more detail in section $III.A$.

2) Test 2: Looking at the results from the second test of $getRoute()$, it looks as though the function is now working as intended in all situations, and is now ready for deployment into the final solution.

B. Item Collection and Output

Overall, the results for the $ItemCollectionandOutput$ tests (Section $IV.B$) were successful. YouBot always collected all required items and delivered them, with the exception of $IV.B.3)Test1$ in which YouBot became stuck on the shelf because there was no item to locate.

Preventing YouBot from getting stuck is a relatively simple fix. If it counts how many spaces its already searched on the shelf, then when it gets to the end of a shelf without finding the item it can just retreat and assume the item isn't there. Alternatively some additional sensors could be used here. If a proximity sensor were mounted to the front of YouBot, facing outwards and towards the sides, then they could feed information back to YouBot's navigation and tell it when to stop moving.

VI. Discussion & Conclusions

Overall the chosen approach has been a success. The navigation system works without any faults in testing, with the exception of $IV.A.1)Test1$, however that bug has been fixed and no other bugs have been found. Efficient route planning is an essential component in any warehouse system that utilises robots and using the Multi-Fragment heuristic approximation algorithm is a good method of planning any route. Whilst it may be an over-engineered solution for this scenario, it's important when developing any system to remember to leave room for it to expand wherever possible. Providing the distance and route matrices are updated whenever a warehouse layout changes, the MF algorithm will scale without failing.

Implementing some more sensory intelligence into the path-finding process to allow for collision detection and avoidance would have been a good avenue to explore as this would allow multiple robots, and even people, to work in one warehouse without collisions. Exploring the Largest Convex Polygon concept, as has been done in a paper by Bo Yang *et al.* [9] is something that could be exciting to explore for future reports.

A. Future Considerations

- Whilst this approach has been good for one robot in a warehouse, allowing for multiple robots and people is more realistic for a practical application of the system, so collision detection and avoidance would be a high priority consideration.
- The MF algorithm does a good job of finding an efficient path to take, however so far the robot doesn't take into account whether it has enough space to hold all the items its heading to. Applying some logic to make it check how many items it needs from the next shelf and then using that to maybe decide to go back to the output conveyor before visiting said shelf.
  - Furthermore, applying logic to the initial path-finding algorithm so that it can attempt to balance shortest distance and least return trips in one journey. Potentially through applying a weight to each node in the network based on how many items are needed from the shelf, and then attempting to collect them together into groups of four at a time.
- There's a lot of room for more sensory intelligence in this solution. Not all objects are the exact same shape and size, so adding a vision sensor to scan for the object needed and calculating the best place to hold it to lift it up would be a very useful feature to implement.

References

[1] Coppelia Robotics AG, *V-rep pro edu*, version 3.6.2, Jun. 25, 2019. [Online]. Available: https : / / www . coppeliarobotics.com/.

[2] J. J. Bentley, "Fast algorithms for geometric traveling salesman problems," *ORSA Journal on Computing*, vol. 4, no. 4, pp. 390–392, 1992. DOI: 10.1287/ijoc.4.4.387.

[3] C. Chase, H. Chen, A. Neoh, and M. Wilder-Smith, "An evaluation of the traveling salesman problem," California State University, 2020. [Online]. Available: http://hdl.handle.net/20.500.12680/8g84mp499.

[4] R. C. Prim, "Shortest connection networks and some generalizations," *The Bell System Technical Journal*, vol. 36, no. 6, pp. 1389–1401, 1957. DOI: 10.1002/j.1538-7305.1957.tb01515.x.

[5] R. W. Floyd, "Algorithm 97: Shortest path," *Communications of the ACM*, vol. 5, no. 6, p. 345, Jun. 1962, ISSN: 0001-0782. DOI: 10.1145/367766.368168.

[6] H. Smith, S. Jameson, P. Sherran, and K. Pledger, *Edexcel AS and A level Further Mathematics Decision Mathematics 1*. Pearson Education Limited, 2017, pp. 73–77, ISBN: 9781292183299.

[7] Wikipedia contributors. "Floyd–warshall algorithm — Wikipedia, the free encyclopedia." (2022), [Online]. Available: https://en.wikipedia.org/wiki/Floyd%5C%E2%5C%80%5C%93Warshall_algorithm (visited on 01/31/2022).

[8] M. l Krari, B. Ahiod, and B. El Benani, "An empirical study of the multi-fragment tour construction algorithm for the travelling salesman problem," in *roceedings of the 16th International Conference on Hybrid Intelligent Systems (HIS 2016)*, A. Abraham, A. Haqiq, A. M. Alimi, G. Mezzour, N. Rokbani, and A. K. Muda, Eds., Springer International Publishing, 2017, pp. 278–287, ISBN: 978-3-319-52940-0.

[9] B. Yang, W. Li, J. Wang, J. Yang, T. Wang, and X. Liu, "A novel path planning algorithm for warehouse robots based on a two-dimensional grid model," *IEEE Access*, vol. 8, pp. 80 347–80 357, 2020. DOI: 10.1109/ACCESS.2020.2991076.

FIGURES

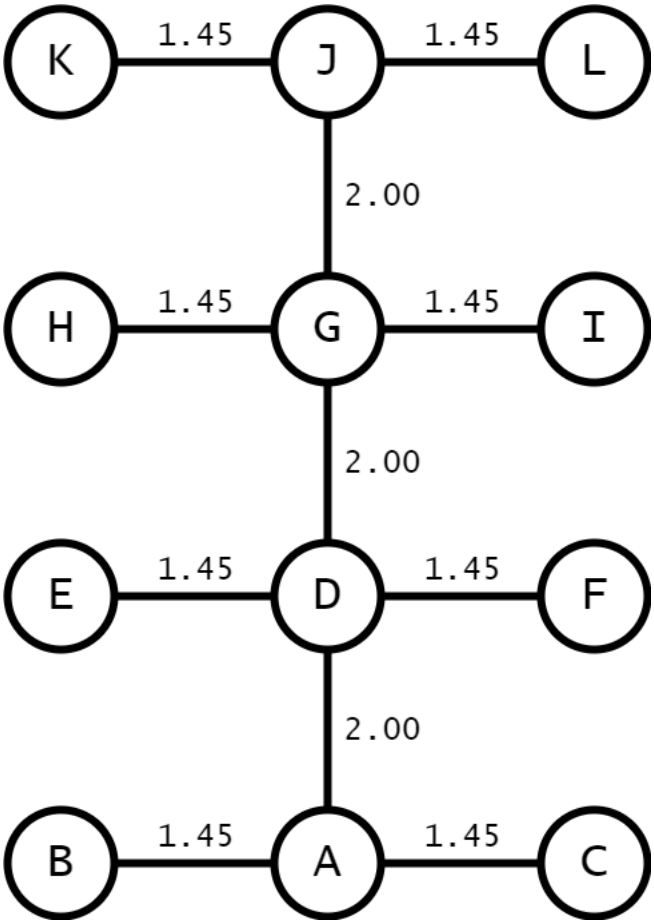

Fig. 3: Graph network (Version 1).

	A	B	C	D	E	F	G	H	I	J	K	L
A	0.00	1.45	1.45	2.00	∞	∞	∞	∞	∞	∞	∞	∞
B	1.45	0.00	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
C	1.45	∞	0.00	∞	∞	∞	∞	∞	∞	∞	∞	∞
D	2.00	∞	∞	0.00	1.45	1.45	2.00	∞	∞	∞	∞	∞
E	∞	∞	∞	1.45	0.00	∞	∞	∞	∞	∞	∞	∞
F	∞	∞	∞	1.45	∞	0.00	∞	∞	∞	∞	∞	∞
G	∞	∞	∞	2.00	∞	∞	0.00	1.45	1.45	2.00	∞	∞
H	∞	∞	∞	∞	∞	∞	1.45	0.00	∞	∞	∞	∞
I	∞	∞	∞	∞	∞	∞	1.45	∞	0.00	∞	∞	∞
J	∞	∞	∞	∞	∞	∞	2.00	∞	∞	0.00	1.45	1.45
K	∞	∞	∞	∞	∞	∞	∞	∞	∞	1.45	0.00	∞
L	∞	∞	∞	∞	∞	∞	∞	∞	∞	1.45	∞	0.00

Fig. 4: Initial distance matrix (Version 1).

	A	B	C	D	E	F	G	H	I	J	K	L
A	0.00	1.45	1.45	2.00	3.45	3.45	4.00	5.45	5.45	6.00	7.45	7.45
B	1.45	0.00	2.90	3.45	4.90	4.90	5.45	6.90	6.90	7.45	8.90	8.90
C	1.45	2.90	0.00	3.45	4.90	4.90	5.45	6.90	6.90	7.45	8.90	8.90
D	2.00	3.45	3.45	0.00	1.45	1.45	2.00	3.45	3.45	4.00	5.45	5.45
E	3.45	4.90	4.90	1.45	0.00	2.90	3.45	4.90	4.90	5.45	6.90	6.90
F	3.45	4.90	4.90	1.45	2.90	0.00	3.45	4.90	4.90	5.45	6.90	6.90
G	4.00	5.45	5.45	2.00	3.45	3.45	0.00	1.45	1.45	2.00	3.45	3.45
H	5.45	6.90	6.90	3.45	4.90	4.90	1.45	0.00	2.90	3.45	4.90	4.90
I	5.45	6.90	6.90	3.45	4.90	4.90	1.45	2.90	0.00	3.45	4.90	4.90
J	6.00	7.45	7.45	4.00	5.45	5.45	2.00	3.45	3.45	0.00	1.45	1.45
K	7.45	8.90	8.90	5.45	6.90	6.90	3.45	4.90	4.90	1.45	0.00	2.90
L	7.45	8.90	8.90	5.45	6.90	6.90	3.45	4.90	4.90	1.45	2.90	0.00

Fig. 5: Distance matrix after Floyd-Warshall algorithm applied (Version 1).

	A	B	C	D	E	F	G	H	I	J	K	L
A	A	B	C	D	D	D	D	G	G	G	J	J
B	A	B	A	A	D	D	D	G	G	G	J	J
C	A	A	C	A	D	D	D	G	G	G	J	J
D	A	A	A	D	E	F	G	G	G	G	J	J
E	D	D	D	D	E	D	D	G	G	G	J	J
F	D	D	D	D	D	F	D	G	G	G	J	J
G	D	D	D	D	D	D	G	H	I	J	J	J
H	G	G	G	G	G	G	G	H	G	G	J	J
I	G	G	G	G	G	G	G	G	I	G	J	J
J	G	G	G	G	G	G	G	G	G	J	K	L
K	J	J	J	J	J	J	J	J	J	J	K	J
L	J	J	J	J	J	J	J	J	J	J	J	L

Fig. 6: Final route matrix (Version 1).

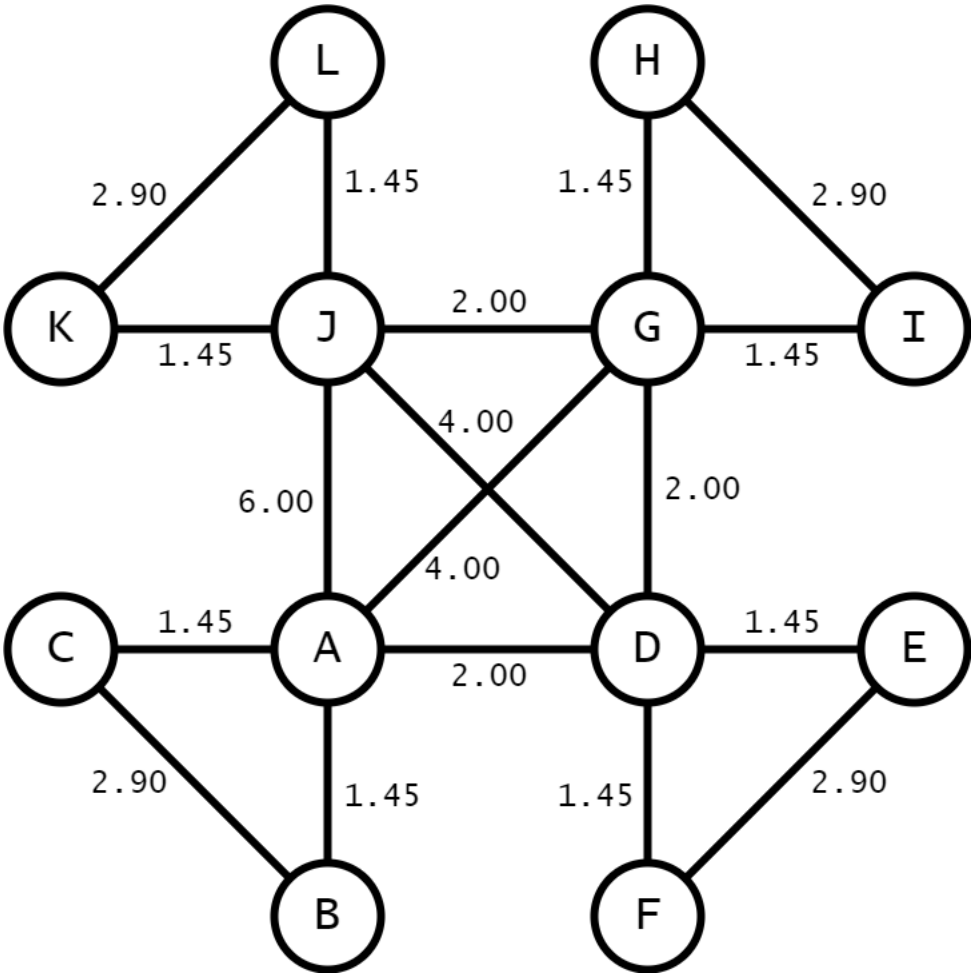

Fig. 7: Graph network (Version 2).

	A	B	C	D	E	F	G	H	I	J	K	L
A	0.00	1.45	1.45	2.00	∞	∞	4.00	∞	∞	6.00	∞	∞
B	1.45	0.00	2.90	∞	∞	∞	∞	∞	∞	∞	∞	∞
C	1.45	2.90	0.00	∞	∞	∞	∞	∞	∞	∞	∞	∞
D	2.00	∞	∞	0.00	1.45	1.45	2.00	∞	∞	4.00	∞	∞
E	∞	∞	∞	1.45	0.00	2.90	∞	∞	∞	∞	∞	∞
F	∞	∞	∞	1.45	2.90	0.00	∞	∞	∞	∞	∞	∞
G	4.00	∞	∞	2.00	∞	∞	0.00	1.45	1.45	2.00	∞	∞
H	∞	∞	∞	∞	∞	∞	1.45	0.00	2.90	∞	∞	∞
I	∞	∞	∞	∞	∞	∞	1.45	2.90	0.00	∞	∞	∞
J	6.00	∞	∞	4.00	∞	∞	2.00	∞	∞	0.00	1.45	1.45
K	∞	∞	∞	∞	∞	∞	∞	∞	∞	1.45	0.00	2.90
L	∞	∞	∞	∞	∞	∞	∞	∞	∞	1.45	2.90	0.00

Fig. 8: Initial distance matrix (Version 2).

	A	B	C	D	E	F	G	H	I	J	K	L
A	0.00	1.45	1.45	2.00	3.45	3.45	4.00	5.45	5.45	6.00	7.45	7.45
B	1.45	0.00	2.90	3.45	4.90	4.90	5.45	6.90	6.90	7.45	8.90	8.90
C	1.45	2.90	0.00	3.45	4.90	4.90	5.45	6.90	6.90	7.45	8.90	8.90
D	2.00	3.45	3.45	0.00	1.45	1.45	2.00	3.45	3.45	4.00	5.45	5.45
E	3.45	4.90	4.90	1.45	0.00	2.90	3.45	4.90	4.90	5.45	6.90	6.90
F	3.45	4.90	4.90	1.45	2.90	0.00	3.45	4.90	4.90	5.45	6.90	6.90
G	4.00	5.45	5.45	2.00	3.45	3.45	0.00	1.45	1.45	2.00	3.45	3.45
H	5.45	6.90	6.90	3.45	4.90	4.90	1.45	0.00	2.90	3.45	4.90	4.90
I	5.45	6.90	6.90	3.45	4.90	4.90	1.45	2.90	0.00	3.45	4.90	4.90
J	6.00	7.45	7.45	4.00	5.45	5.45	2.00	3.45	3.45	0.00	1.45	1.45
K	7.45	8.90	8.90	5.45	6.90	6.90	3.45	4.90	4.90	1.45	0.00	2.90
L	7.45	8.90	8.90	5.45	6.90	6.90	3.45	4.90	4.90	1.45	2.90	0.00

Fig. 9: Distance matrix after Floyd-Warshall algorithm applied (Version 2).

	A	B	C	D	E	F	G	H	I	J	K	L
A	A	B	C	D	D	D	G	G	G	J	J	J
B	A	B	C	A	D	D	A	G	G	A	J	J
C	A	B	C	A	D	D	A	G	G	A	J	J
D	A	A	A	D	E	F	A	G	G	J	J	J
E	D	D	D	D	E	F	D	G	G	D	J	J
F	D	D	D	D	E	F	D	G	G	D	J	J
G	A	A	A	D	D	D	G	H	I	J	J	J
H	G	G	G	G	G	G	G	H	I	G	J	J
I	G	G	G	G	G	G	G	H	I	G	J	J
J	A	A	A	D	D	D	G	G	G	J	K	L
K	J	J	J	J	J	J	J	J	J	J	K	L
L	J	J	J	J	J	J	J	J	J	J	K	L

Fig. 10: Final route matrix (Version 2).

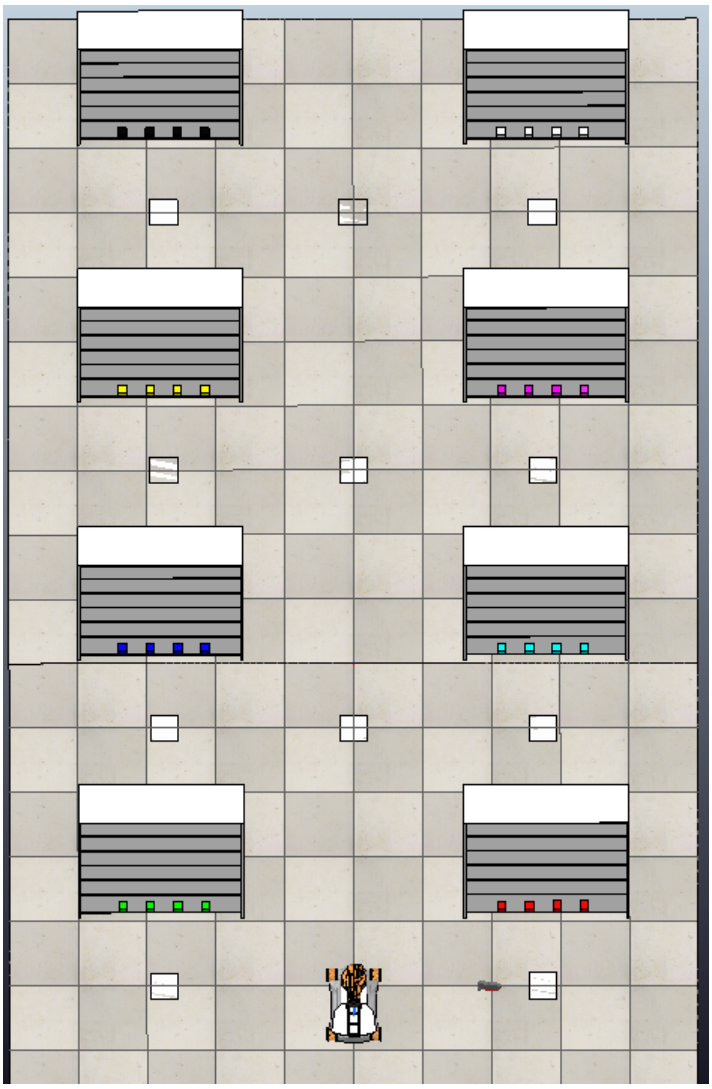

Fig. 11: Graph network applied to warehouse.

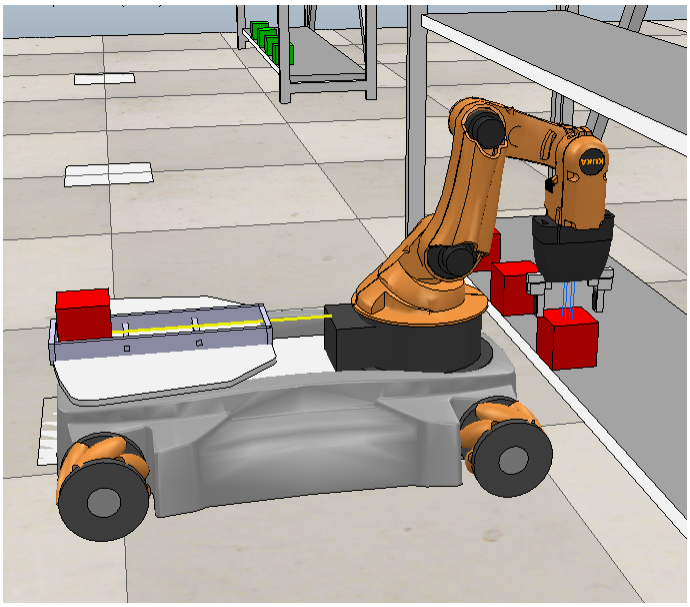

Fig. 12: YouBot aligning vision sensor with red item before picking up.

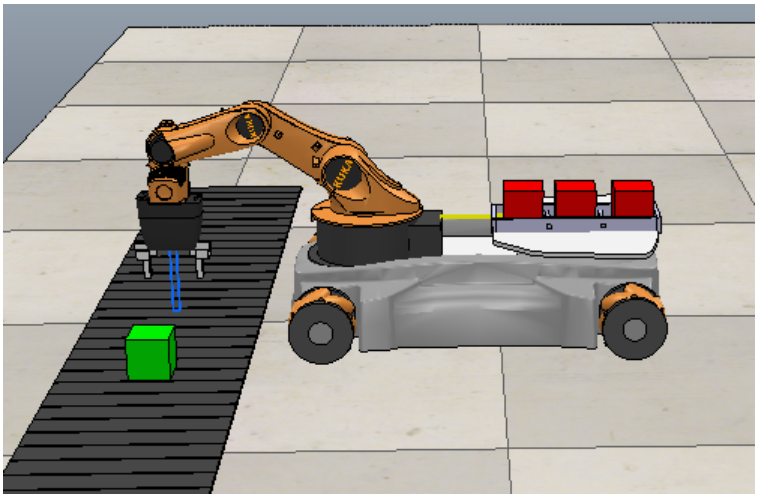

Fig. 13: YouBot successfully placing green item in centre of conveyor belt.

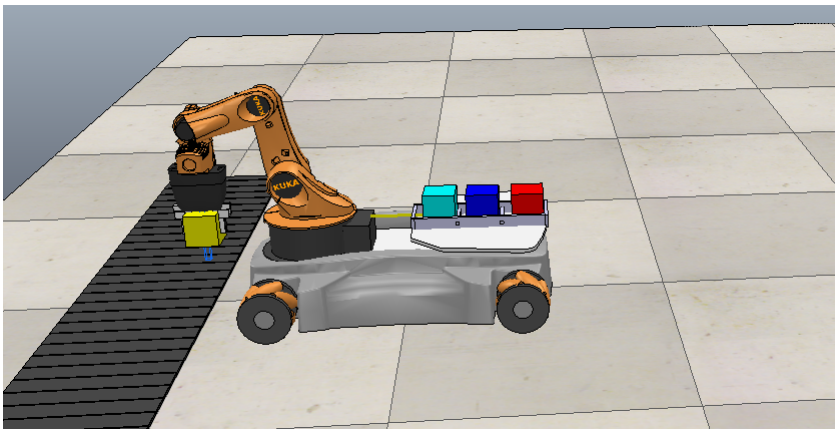

Fig. 14: YouBot returning to conveyor to output current items before collecting more.

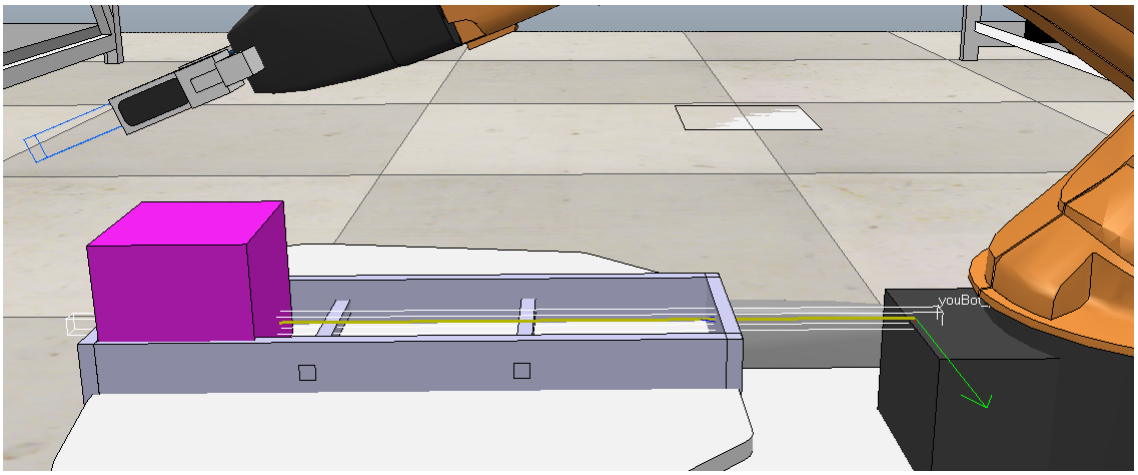

Fig. 15: Proximity sensor detecting item on platform.

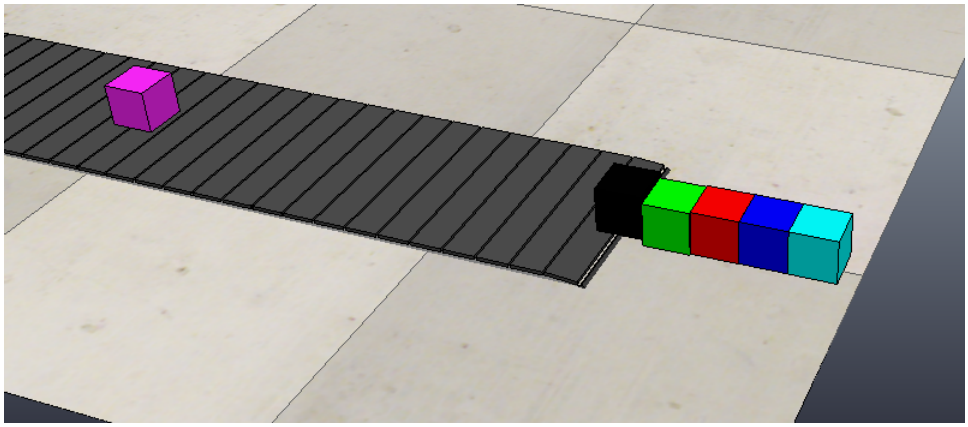

Fig. 16: Selection of boxes lined up at output.

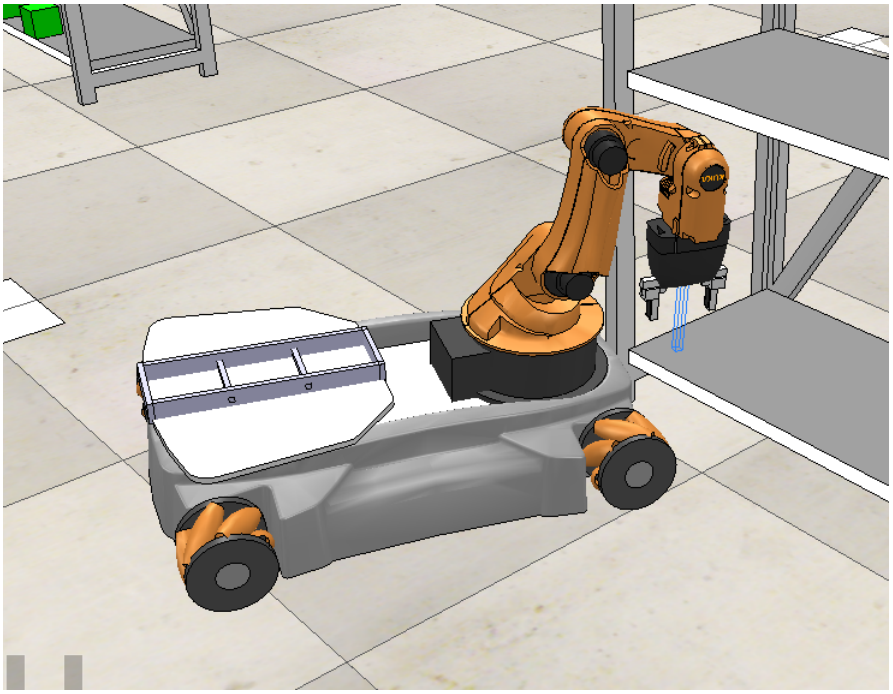

Fig. 17: YouBot stuck on shelf.