Neural Computing and Applications
https://doi.org/10.1007/s00521-020-04764-3

S.I. : GREEN AND HUMAN INFORMATION TECHNOLOGY 2019

Adaptive path finding algorithm in dynamic environment for warehouse robot

Mun-Kit Ng[1] · Yung-Wey Chong[2] · Kwang-man Ko[3] · Young-Hoon Park[4] · Yu-Beng Leau[5]

Received: 2 April 2019 / Accepted: 24 January 2020

Abstract
Warehouse robots have been widely used by manufacturers and online retailer to automate good delivery process. One of the fundamental components when designing a warehouse robot is path finding algorithm. In the past, many path finding algorithms had been proposed to identify the optimal path and improve the efficiency in different conditions. For example, A* path finding algorithm is developed to obtain the shortest path, while D* obtains a complete coverage path from source to destination. Although these algorithms improved the efficiency in path finding, dynamic obstacle that may exist in warehouse environment was not considered. This paper presents AD* algorithm, a path finding algorithm that works in dynamic environment for warehouse robot. AD* algorithm is able to detect not only static obstacle but also dynamic obstacles while operating in warehouse environment. In dynamic obstacle path prediction, image of the warehouse environment is processed to identify and track obstacles in the path. The image is pre-processed using perspective transformation, dilation and erosion. Once obstacle has been identified using background subtraction, the server will track and predict future path of the dynamic object to avoid the obstacle.

Keywords Path finding · Dynamic obstacle avoidance · Warehouse robot

1 Introduction

In recent years, warehouse robots have been used as e-commerce bloom. The state-of-the-art innovation has reduced the number of workforce required and increased the efficiency of warehouse management system. E-commerce companies such as Amazon and Alibaba utilised warehouse robot to automate the process of picking, sorting and navigation assistance of goods. Thus, the robots have replaced human workloads especially in performing repetitive tasks. According to Dubois and Hamilton [1], the demand of warehouse robot is increasing and expect to growth 12% in 2018 in the USA. These warehouse robots helped to pick and pack USD 394.8 billion worth of goods in 2017. Markets and Markets [2] projected that the value will further increase to USD4.44 billion by 2022.

To achieve the tasks required in logistic arrangement, warehouse robots need to navigate autonomously to reach destination without predefined path. Unlike other industrial robots that work in fixed locations or follow specific line only, warehouse robots need to rely on navigation algorithm to manoeuvre in the warehouse. This is due to the

✉ Yung-Wey Chong
chong@usm.my

Mun-Kit Ng
ngmunkit93@student.usm.my

Kwang-man Ko
kkman@sangji.ac.kr

Young-Hoon Park
yh.park@sm.ac.kr

Yu-Beng Leau
lybeng@ums.edu.my

[1] School of Electrical and Electronic Engineering, Universiti Sains Malaysia, Nibong Tebal, Malaysia

[2] National Advanced IPv6 Centre, Universiti Sains Malaysia, George Town, Malaysia

[3] Department of Computer Science and Engineering, Sangji University, Wonju, South Korea

[4] Division of Computer Science, Sookmyung Women's University, Seoul, South Korea

[5] Faculty of Computing and Informatics, Universiti Malaysia Sabah, Kota Kinabalu, Malaysia

Published online: 04 February 2020

uncertainty in warehouse caused by moving obstacle or moving human. As such, it is important that the warehouse robot should have the capability of moving from one place to another with optimisation, shorter time and avoid dynamic obstacle. The navigation of warehouse robot need to provide collision free path to the mobility of the robot while satisfying certain optimisation conditions such as energy consumption, processing time and communication delay [3].

Path finding algorithm is a sequence of actions that transform initial state to desired goal state with an associated cost. An optimal path refers to a path which has the minimal sum of its transition costs among all possible paths [4]. The accuracy of path finding ensures the safety of autonomous warehouse robots. A variety of path finding algorithms and approaches have been introduced consistently with the valuable market trend in mobility robots. Most path finding approaches are static path findings, where information is retrieved from predefined environment. One of the concerns of path finding method is the responsiveness of mobility robots with the dynamic information in real scenario. An ideal path planner must be able to handle or response with these uncertainties, preventing any undesired accidents occurred [5]. This can be accomplished only if path planner provides updated path to mobility robots while the system receiving dynamic information on the movement of obstacles.

The most common techniques used in path computation are deterministic, heuristic-based algorithms [4]. Heuristic search algorithms guide the search trajectory within the search space using information from the problem. With the use of certain functions, heuristic search determines the cost of the current state to the goal state, which can reduce the computational work [6]. Therefore, heuristic search focuses on time reduction in path finding. Heuristic-based methods with additional modifications can overcome dynamic real-world problem, but it may fail in uncertain environment as well [7]. The imperfection of heuristic search can be resolved using vision and sensor technologies.

2 Related work

There are several criteria that need to be considered before deploying path finding algorithm, namely [8]:

- Completeness: the ability of path finding algorithm to find the path from source to destination.
- Optimality: finding the path with the lowest cost.
- Time complexity: time taken for the path finding algorithm to obtain the path.
- Space complexity: total memory consumed to obtain the optimal path.

Path finding algorithms can be broadly categorised as uninformed search and informed search as shown in Fig. 1. In uninformed search, the algorithm does not have the concept of location of the destination relative to existing location [5]. Examples of uninformed search are depth-first search and breadth-first search. Unlike uninformed search, informed search will search for optimal path using the location of destination relative to existing location [8]. The most popular informed search algorithms are greedy search, Dijkstra's algorithm, A*, lifelong planning A* and D* Lite.

2.1 Uninformed search

Depth-first search is an algorithm for traversing a finite graph. It visits the child vertices before visiting the sibling vertices. Stack is usually used in depth-first search algorithm, in which the most research node is chosen for expansion. In the algorithm, the beginning vertex will be marked as start. It will iterate from current position to adjacent vertex and checks whether adjacent vertex has been visited. The search algorithm will continue to explore the path and backtrack to previous vertex after finishing exploration of a path [8]. One of the advantages of depth-first search is the ability to find the solution without examining all the vertex.

Breadth-first search is an algorithm for traversing or searching tree or graph. It is based on queuing concept, which is first-in first-out basis. Breadth-first search is accomplished by enqueueing each level of tree sequentially. It will start with the first-level search which is the adjacent vertex of start vertex. It will check the child vertex of the level-one vertex. It continues to explore until the destination is reached [13]. Since breadth-first search explores level by level, the search algorithm may consume more memory and CPU usage. Figure 2 shows the comparison between depth-first search and breadth-first search.

2.2 Informed search

Unlike uninformed search, informed search utilises the concept of location of destination relative to existing location. It is a guided search with information. Greedy search is one of the informed searches where it seeks to minimise the estimated cost to reach a destination node [9]. Greedy search always takes the closest node towards the destination of search. The function used to estimate the operating cost is called heuristic function. Greedy searches are useful because they often find destination states quickly although the route taken to reach the destination may not

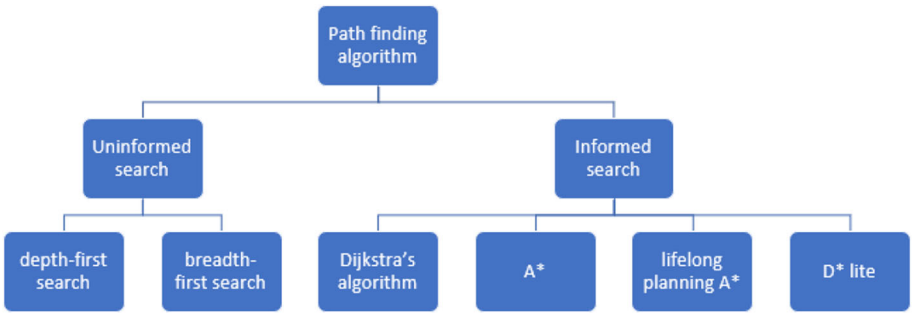


Fig. 1 Path finding algorithms

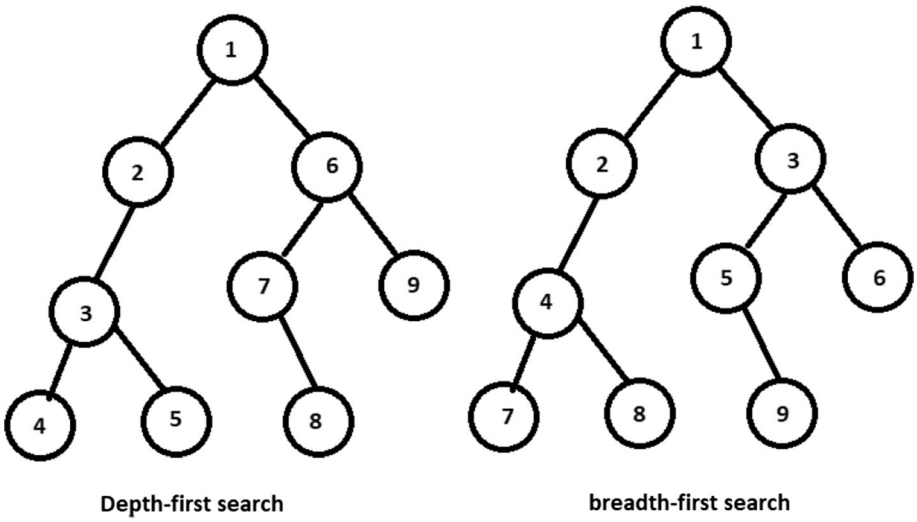


Fig. 2 Comparison between depth-first search and breadth-first search

be optimised in terms of path cost. One of the assumptions in greedy search algorithm is that the destination node will eventually be found. The algorithm continues if destination node is not found. The best cost means the lowest cost from neighbour node to destination node. The lowest cost node will record and allow current node to traverse to it. Once the best cost node is found, a back pointer is set to track back the previous node and ready to move on to another iteration [10]. Figure 3 shows the greedy search in a grid map. Greedy search will approach the destination as primary objective. For the case where no obstacle is detected in the grid map, greedy search always can directly find the shortest path from start node to destination node.

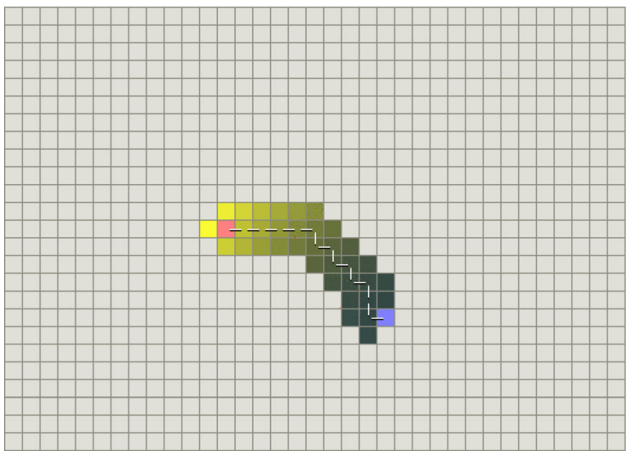


Fig. 3 Greedy search [10]

Dijkstra's algorithm is one of the earliest algorithms that fulfil search criteria [11]. In Dijkstra's algorithm, the cost to travel from start node to destination for each node n is obtained. There are two set of lists that are captured during the search, namely CLOSED and OPEN lists as shown in Algorithm 1. The cost of node in CLOSED list is considered final, or the node that has been explored will put into CLOSED list. The node that has not been explored will remain in OPEN list with a cost of infinity, except for the starting node $\text{node}_{\text{start}}$ which is assigned a path cost $g(\text{node}_{\text{start}}) = 0$.

During the node expansion phase, the four (4) neighbour nodes are evaluated. The node with the lowest path cost, $g(n)$, is removed from the OPEN list, and its cost is propagated to its neighbours. The cost of neighbour of node n, $g(n')$, is calculated by $g(n)$ + cost travel from n to n', $c(n, n')$. If the new cost is smaller than $g(n')$, then n will set as its predecessor. The whole process is referring as node expansion. Once the node expansion has occurred, n is added into the CLOSED list as it has been explored. Then,

another least path cost is chosen as new n if $n = \text{node}_{\text{goal}}$ or no node is OPEN list. The algorithm will be terminated during the absence of node in OPEN list. In this case, the algorithm is unable to find the solution for current situation [12].

Algorithm 1 Dijkstra's algorithm [11]

```
function NodeExpansion(n,n')
    if g(n) + c(n,n') < g(n') then
        g(n') = g(n) + c(n,n')
        Pred(n') = n
        OPEN.insert(n',g(n'))
    end if
end function
function Dijkstra(node_start, node_goal)
    g(node_start) = 0
    Pred(node_start) = node_start
    OPEN = 0
    OPEN.insert(node_start, g(node_start))
    CLOSE = 0
    while OPEN ≠ 0 do
        N = OPEN.pop()
        CLOSED.insert(n)
        if n = n_goal then
            Break
        end if
        for each n' in neighbour do
            if n'not in CLOSE then
                if n' not in OPEN then
                    g(n') = infinity
                end if
                NodeExpansion(n,n')
            end if
        end for
        Path = {node_goal}
        n = nodegoal
    end while
    while n ≠ node_start do
        Path = path U {n}
        n = pred(n)
    end while
end function
```

If the destination is found, path extraction phase will be conducted. This phase starts from destination node and back to start node by selecting predecessor of current node. This process occurs recursively until start node is found. Figure 4 illustrates the path extraction phase conducted. Once the algorithm is iterated in the map, the shortest path will be determined if the destination is located in a traversal place. The algorithm guarantees the shortest path using this method. However, the drawback of Dijkstra algorithm is that the memory and number of iterations required may be large and long.

A* is a best-first search algorithm that combines Dijkstra's algorithm with a heuristic function [13]. Dijkstra's algorithm, $g(n)$, represents the exact cost of the path from the starting point to any vertex n, whereas heuristic, $h(n)$, estimates the costs of travelling from n to the goal node. A* inherited both characteristics by forming a new function $f(n) = g(n) + h(n)$. A* algorithm is a compact and efficient algorithm as compared to other artificial intelligence algorithm [14]. It has the advantage of shorter running time and easier to implement on system as compared to traditional artificial intelligence system. Many applications have adopted A* algorithm for path finding in the past [15]. Unlike Dijkstra's algorithm that focuses on path cost $g(n)$, A* prioritises the nodes by $f(n)$. Since heuristic function is considered in $f(n)$, the time complexity during search improved dramatically.

A* is restricted by eight connectivity, and it could not perform every angle search. [16] modified A* algorithm so that it is used in a mobile robot with shorter computational time and path. The modified A* is able to achieve any angle search shown in Fig. 5. One of the drawbacks of modified A* is that dynamic environment and dynamic obstacle are not taken into consideration. In addition, some application such as warehouse robot may not require the feature of any angle because item in warehouse must be organised.

Abiyev [17] improved A* algorithm to avoid static obstacle. Even though it is able to navigate to reach the

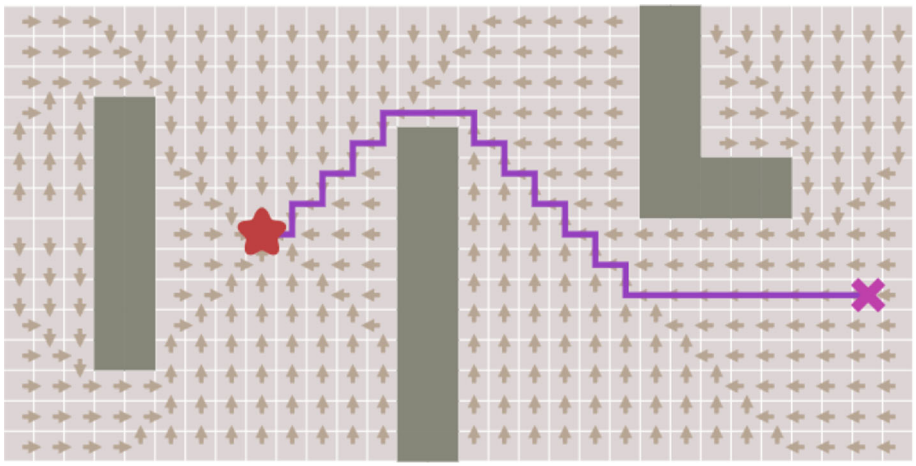


Fig. 4 Path extraction [10]

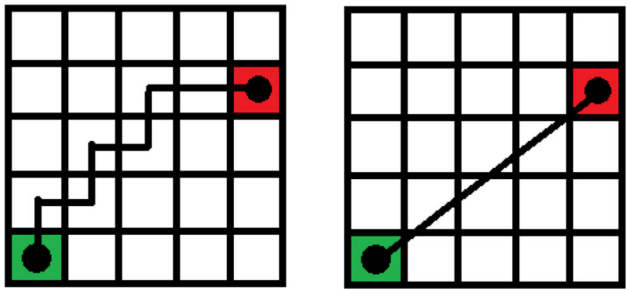


Fig. 5 A* algorithm versus any angle search

destination, the algorithm is unable to detect and avoid dynamic object. This is because A* does not have the ability to re-plan the path when the environment changes and avoid dynamic obstacle.

D* Lite is designed to perform dynamic path re-planning in unknown terrain [18]. It reverses the order and re-plans the path from destination to start node. This allows the start point moving from one block to another. When the robot is moving, the start point is updated with the current location of the node. Hence, the travelled node is excluded from the re-planning. As a result, only a small section of node is needed to re-plan and it is more efficient. Since D* Lite starts from destination node, the successor node becomes predecessor node and vice versa.

To avoid reorder priority queue, heuristic function must satisfy the following:

$$h(n,n') \leq h(n,n') + h(n,n'') \tag{1}$$

$$h(n,n') \ \leq \ c*(n,n') \tag{2}$$

$$h(n,n') \geq 0 \tag{3}$$

where n, n', n // $\in$ nodes and c^* (n, n') is the shortest path between nodes n and n'. D* Lite uses annealing approach to identify the moving obstacle. Ganeshmurthy [3] evaluated D* Lite, and it is undeniable that D* Lite has an advantage over A* as it is dynamic version of A*.

3 Adaptive dynamic path finding algorithm (AD*)

Adaptive dynamic path finding algorithm, AD*, is proposed to find optimal path that can avoid dynamic obstacle in warehouse robot. AD* algorithm enhanced from D* Lite algorithm so that warehouse robot can work not only in dynamic environment but also when dynamic obstacle exists. AD* has additional advantage as compared to D* Lite because it can predict location of dynamic obstacle and avoid dynamic obstacle. There are several processes involved in finding the optimal path for the warehouse robot as shown in Fig. 6, namely:

- Construction of world map
- Robot detection
- Offline path planning
- Online path planning
- Robot movement correction

3.1 Construction of world map

Before the image of the warehouse can be processed, the world map needs to be constructed using image processing. Perspective transformation is used to transfer different perspectives image into top view of world map. The perspective transformation formula is applied to acquire transformation metric with output image denoted as dst(i) and source image denoted as src(i).

$$\begin{bmatrix} t_i\ x_i \\ t_i y_i \\ t_i \end{bmatrix} = \text{map}_{\text{matric}} \begin{bmatrix} x_i \\ y_i \\ 1 \end{bmatrix} \tag{4}$$

where

$\text{dst}(i) = (x_i',\ y_i'),\ \text{src}(i) = (x_i,\ y_i),\quad i = 0,\ 1,\ 2,\ 3.$

The world map is then separated into x and y coordinates as shown in Fig. 7. Since AD* algorithm is tested in 85 × 65 pixels environment, the map is divided into 10 × 10 grid with each block consists of 85 × 65 pixels.

3.2 Robot detection

Before path planning algorithm is executed, the location of the robot (or in short ADSeekerbot) needs to be identified. The top of ADSeekerbot is labelled with "R1" as the identity of robot as shown in Fig. 8. If multiple robots exist in the environment, ADSeekerbot will identified based on the label ID on top of the robot using machine vision. The location of ADSeekerbot is detected using speeded up robust features (SURF) and fast library for approximate nearest neighbours (FLANN) matcher. SURF is a patented local feature detector and descriptor that used for object recognition [19]. The algorithm started with detection. A blob detector based on Hessian matrix is used for point of interest detection. The Hessian matrix $H(x, \sigma)$ is given in Eq. 5:

$$H(x,\sigma) = \begin{bmatrix} L_{xx}(x,\sigma) & L_{xy}(x,\sigma) \\ L_{xy}(x,\sigma) & L_{yy}(x,\sigma) \end{bmatrix} \tag{5}$$

where $L_{xx}(x,\ \sigma)$ is the convolution of the Gaussian second order with image I in point x.

An image of ADSeekerbot and image of ADSeekerbot in world map are provided to the detector so that keypoint of both images is computed. FLANN used nearest neighbour search technique that will retrieve the nearest data according to input [20]. It is used to compute the nearest

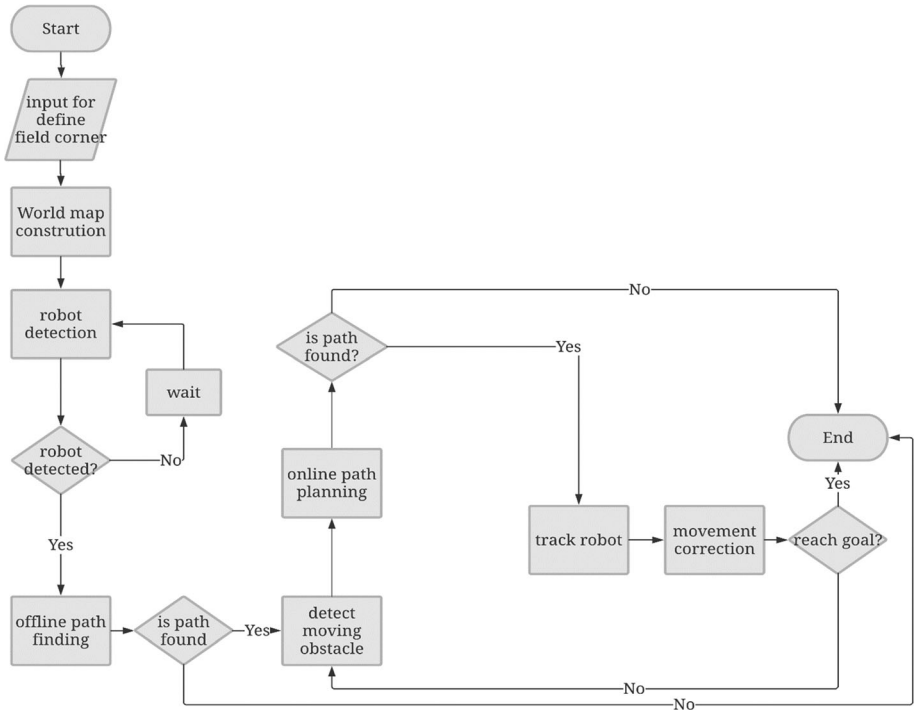


Fig. 6 AD* algorithm

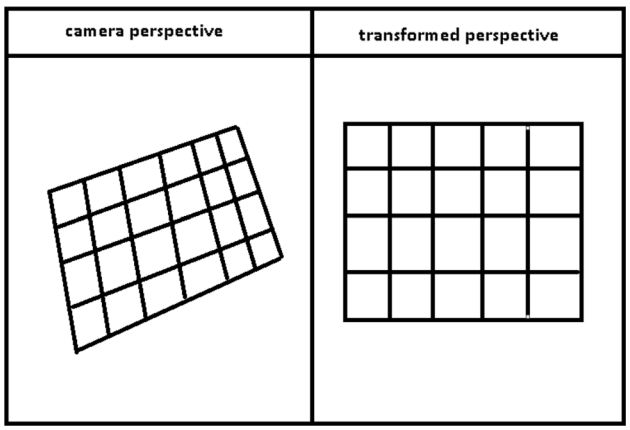


Fig. 7 Perspective transformation

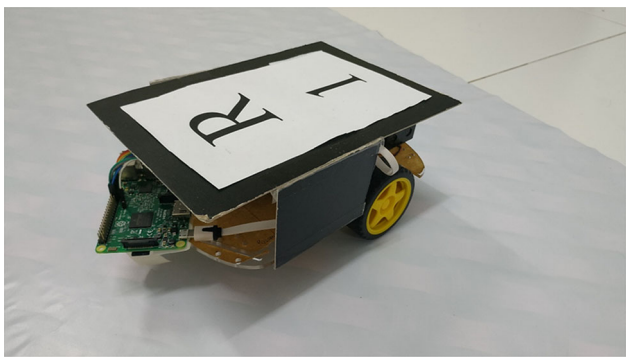


Fig. 8 ADSeekerbot label

keypoint of both image and map using the template image in world map.

3.3 Offline path finding

In order to navigate the warehouse robot without hitting any obstacle, AD* should be able to detect static obstacle. Offline path planning is an initial path planning for the ADSeekerbot to reach destination from the current position. The offline path planning focuses on detecting static obstacle when re-planning the path. There are two components in offline re-planning algorithm, namely static obstacle detection and path finding. For the static object detection, adaptive Gaussian threshold is used to segment between static object and floor. The threshold value $T(x, y)$ is a weighted sum of block size neighbourhood of (x, y) minus constant, C, and it is denoted as:

$$T(x,y) = \sum_{u=-k}^{k} \sum_{v=-k}^{k} G(u,v)P(x+u,y+v) - C \qquad (6)$$

where K is the block size of neighbourhood of pixel and P function is the pixel value. $T(x, y)$ is obtained by convolution between the Gaussian Kernel.

$$G_i = \alpha * e^{-(i-(k-1)/2)^2/2*\text{signma}^2} \qquad (7)$$

where $i = 0 \ldots k - 1$ and α is the scale factor chosen so that $\sum G_i = 1$.

Adaptive thresholding is used because it can recover image with a strong illumination gradient. Adaptive

Gaussian thresholding produces lesser noise and better-quality image as compared to other method [21]. Hence, it is used to segment between static obstacle and floor. After image segmentation of the static obstacle and floor, noise filtering is used to filter the noise in the image. Firstly, erosion is used to remove small particles in the image. Erosion will "grow" the darker part by shrinking the white dot in the image using Eq. 8:

$$\mathrm{dst}(x,y) = \mathbf{min}_{(x',y'):\mathrm{element}(x',y)\neq \mathbf{0}} \mathrm{src}(x+x', y+y') \tag{8}$$

Once the small particles are removed, dilation connects the component that is connected by "shrink" the darker part and connects the nearby bright component together using Eq. 9:

$$\mathrm{dst}(x,y) = \mathbf{max}_{(x',y'):\mathrm{element}(x',y)\neq \mathbf{0}} \mathrm{src}(x+x', y+y') \tag{9}$$

After the detection of static objects, the cost map is constructed based on heuristic function and operation cost function.

$$f(n) = g(n) + h(n) \tag{10}$$

where $g(n)$ is the cost of the path from the start node to n and $h(n)$ is a heuristic function that estimated the cost of the cheapest path from n to destination node [12]. The node expansion is used to expand from current node to find its predecessor from its eight neighbourhoods. The algorithm is terminated once ADSeekerbot reaches destination node.

3.4 Online path finding

Once static object has detected, ADSeekerbot starts to move towards destination node and trigger online path finding. Online path finding is designed for dynamic environment and dynamic obstacle avoidance. Before new path is determined, surrounding information is needed such as moving obstacle or change in destination node. There are two types of moving obstacle, either a moving object that does not exist in the world map in the beginning or a static obstacle that starts moving from its original position to another position. Moving obstacles are given an ID, and its movement is tracked. The centre of the obstacle is used as reference for the location.

In order to detect dynamic object, background subtraction is used. Current frames are subtracted from previous frame to indicate the movement of any object. Gaussian mixture-based background segmentation algorithm is used for dynamic object detection [22], and it is denoted as:

$$p(x_N) = \sum_{j=1}^{K} w_j \eta(x;\theta) \tag{11}$$

The threshold T is the minimum fraction of the background model. The Gaussian component that matches the test value will be updated using Eq. 16.

$$w_k^{N+1} = (1-\alpha) w_k^N + \alpha p(w_k|x_{N+1}) \tag{12}$$

$$\mu_k^{N+1} = (1-\alpha)\mu_k^N + \rho x_(N+1) \tag{13}$$

$$\sum_k^{N+1} = (1-\alpha)\sum_k^N + \rho(x_{N+1}-\mu_k^{N+1})(x_{N+1}-\mu_k^{N+1})^T \tag{14}$$

$$\rho = \alpha\eta\left(x_{N+1};\mu_k^N,\sum_k^N\right) \tag{15}$$

$$p(\omega_k|x_{N+1}) = \begin{cases} 1, & \text{if } w_k \text{ is the first match Gaussian component} \\ 0, & \text{otherwise} \end{cases} \tag{16}$$

where ω_k is the kth Gaussian component. $1/\alpha$ defines the time constant which determines change. Background subtraction is crucial as it can detect only new dynamic object but also static object that starts moving during the path planning process. Kernelised correlation filter (KCF) is used to track the location for both ADSeekerbot and dynamic obstacle. KCF is separated into three steps [23]. In the first step, blocks are built to find minimum squared error over sample using linear regression.

$$\min_w \sum_i (f(x_i) - y_i)^2 + \vartheta||w||^2 \tag{17}$$

$$w = [w_i, \ldots, w_N]^{\mathrm{T}} \tag{18}$$

where ϑ is the regularisation parameter

In the second step, KCF is cyclic shift to calculate the position shift of the object. The full set of shifted signals is obtained from Eq. 19. It is a permutation of an identity matrix with last row starts with 1 and ends with zero.

$$\left\{P^u \middle| u = 0, \ldots n-1\right\} \tag{19}$$

In the third step, KCF applies linear regression together with cyclic shifts to find error over sample.

$$X^H X = F\mathrm{diag}(\hat{x}^*)F^H F\mathrm{diag}(\hat{x})F^H \tag{20}$$

For warehouse robot, it is crucial to predict the future location of the moving object. If the location of obstacle is intersected with robot or the shorter path than before is found, the path is needed to be re-planned to avoid collision. Line of best fit is used to predict future movement of the detected obstacle. The movement prediction utilises minimum three to five history points of the detected obstacle to plot line of best fit of x- and y-axes with respect to time, t. The history point is acquired from each frame. The time taken between each frame is consistent; it is used as time t. Line of best fit is calculated using least square

regression method to find the relationship of x-axis or y-axis of moving object to time, t.

$$m = \frac{(N\sum tu - \sum t \sum u)}{N(\sum t^2) - (\sum t)^2} \tag{21}$$

$$b = \frac{\sum u - m\sum t}{N} \tag{22}$$

$$u = mt + b \tag{23}$$

where N is the number of points. u represents x and y. Assume $t = 0$ as present location. To calculate future five points, t is iterated from $t = 0$ with increment by 1 until $t = 5$ to predict five points ahead of current position.

Based on the prediction information, AD* may need to re-plan the path so that it can avoid the static and dynamic obstacle. The node need to be updated with new G-cost. G-cost represents the cost for the robot to travel from one point to another. RHS is a value that is used to determine whether G-cost must be updated. All RHS values must satisfy the following relationship:

$$\text{rhs}(n) = \min\left(g(n') + c(n', n)\right) \tag{24}$$

If the cost of the node changed, the update node function will recalculate the G-cost and RHS cost. If there are no changes in G-cost and RHS cost, recalculation is not required. If the both costs are inconsistent, recalculation is required. The path from start node to destination node becomes worse or intraversal when the RHS cost is higher. If RHS cost is lower, the path is easier to travel.

Algorithm 2 AD* algorithm

```
function UPDATENODE(u)
    if u ≠ nodestart then
        rhs(u) = mins∈pred(u)(g(n′) + c(n′, u))
    end if
    if u subset in U then
        U.Remove(u)
    end if
    if g(u) ≠ rhs(u) then
        U.Insert(u, CalculateKey(u))
    end if
end function
function CALCULATEKEY(n)
    Return [min(g(n), rhs(n)) + h(n); min(g(n), rhs(n))]
end function
function COMPUTESHORTESTPATH(node_start, node_goal)
    while U.TopKey() < CalculateKey(node_goal) OR rhs(node_goal) ≠ g(node_goal) do
        k_old = U.TopKey()
        u = U.Pop()
        if k_old < CalculateKey(u) then
            U.Insert(u, CalculateKey(u))
        end if
        if g(u) > rhs(u) then
            g(u) = rhs(u)
            for all s ∈ pred(u) do
                UpdateNode(n)
            end for
        else
            g(u) = ∞
            for all s ∈ pred(u) ∪ {u} do
                UpdateNode(n)
            end for
        end if
    end while
end function
```

Once the G-cost is determined, AD* will extract the path and calculate the shortest path. COMPUTESHORTESTPATH function will identify the shortest path and reflect the RHS value to G-cost to identify whether the cell getting worse will initiate it as non-travel cell with infinity cost.

Both algorithms UPDATENODE and COMPUTESHORTESTPATH allow ADSeekerbot to navigate in dynamic environment and avoid dynamic obstacles. Dynamic obstacle is avoided using obstacle movement prediction to determine intersection point and perform path re-planning.

3.5 Robot movement correction

After path planning is carried out, robot is tracked and navigated according to the planned path with movement correction. For the robot movement correction, PD controller is used for position correction. PD controller is a control system with feedback. Based on current state error, ADSeekerbot corrects its course accordingly. Figure 9 shows the control system of ADSeekerbot.

The controller gain is summation of proportional error and derivative error.

$$\text{Proportional error} = K_p\ e(t) \tag{25}$$

$$\text{Derivative error} = K_d \frac{\mathrm{d}e(t)}{\mathrm{d}t} \tag{26}$$

$$\text{controller gain} = \text{Proportional error} + \text{Derivative error} \tag{27}$$

The determined path from path planning is as a setpoint to ADSeekerbot current location. The error of position is obtained with the difference of current position and expected position. The error is used to navigate the left and right motor of the ADSeekerbot.

$$\text{left motor} = \text{base speed} + \text{controller gain} \tag{28}$$

$$\text{right motor} = \text{base speed} - \text{controller gain} \tag{29}$$

The ADSeekerbot can correct its error in position by varying the left–right motor speed. If car current position equals to destination, the car will stop. The direction of moving robot is used to compute position error. A circle is drawn surrounding of robot, and a small dot is used to indicate the direction of robot. When the robot is moving, a blue indicator with green arrow will indicate the direction of moving robot as shown in Fig. 10. This information is provided to controller system of current system direction and location.

The direction indicator is calculated using dot product of vector with itself. The vector is obtained from previous frame of centre of circle to current frame of centre of circle. Previous location of ADSeekerbot is denoted as O, current indicator is denoted as A, and current location of ADSeekerbot is denoted as k. By using the dot product with itself from centre of circle to circumference,

$$\mathbf{OA} \bullet \mathbf{OA} = |OA||OA| \cos\theta \tag{30}$$

$\theta = 0$, hence $\cos\theta = 1$

$$\mathbf{OA} \bullet \mathbf{OA} = |OA||OA|$$
$$\mathbf{OA} = t \bullet \mathbf{Ok}$$

Assuming **Ok** is unit vector,

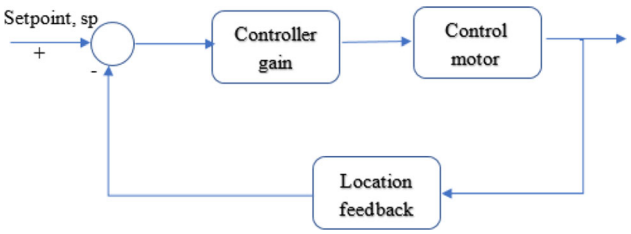


Fig. 9 Robot control system

$$\begin{aligned} t \bullet \mathbf{Ok} \bullet t \bullet \mathbf{Ok} &= r^2 \\ t^2\left((k_x - O_x)^2 + (k_y - O_y)^2\right) &= r^2 \\ t^2 &= \frac{r^2}{\left((k_x - O_x)^2 + (k_y - O_y)^2\right)} \end{aligned} \tag{31}$$

Solve

$$A = t \bullet \mathbf{Ok} + O \tag{32}$$

The direction indicator is calculated by solving the vector problem. Once the location and direction of robot are determined, error of position is calculated from current location and desire location. The location error is the shortest distance between direction indicator of robot and the shortest path. The red line in Fig. 11 shows the shortest distance between the line and robot centroid. The distance between line and ADSeekerbot is found using Eq. 33 with the given point of each line segment:

$$\text{Distance} = \frac{(y_2 - y_1)x_o - (x_2 - x_1)y_o + x_2y_1 - y_2x_1}{\sqrt{(y_2 - y_1)^2 + (x_2 - x_1)^2}} \tag{33}$$

The line passing through $P_1(x_1, y_1)$ and $P_2(x_2, y_2)$ and the centroid of ADSeekerbot is (x_0, y_0)

The robot movement correction calculates the direction of following point with positive and negative sign, which will feedback to PD controller to correct its location. In a path containing many lines segment, a separate line (blue line) is drawn to separate each line region as shown in Fig. 11. It allows ADSeekerbot to determine which path segment to refer from one segment to another. The blue line, which is the line segment, is determined by the resultant vector three points (green line) and gradient is perpendicular with the resultant vector. The blue line acts as a borderline between each path segment. If the ADSeekerbot's position is detected before the blue line, ADSeekerbot will reference to the path segment before the blue line. The direction determination equation is as follows:

$$\text{direction} = ax_o + by_0 + c \tag{34}$$

Given ADSeekerbot is (x_0, y_0) and a is x constant, b is y constant and c is y-intercept of blue line equation. The

positive sign of direction indicates that ADSeekerbot is in front of the line and negative sign of direction indicates that it is behind the line. The ADSeekerbot will stop moving if direction indicator is within 40-pixel diameter of destination node.

4 Results and discussion

AD* algorithm is tested in four different environments, and the performance is compared with A* and D* Lite. The four test cases are:

1. environment where static obstacle exists only
2. environment where dynamic obstacle is present without intercepting with ADSeekerbot
3. environment where dynamic obstacle intercepts with ADSeekerbot
4. environment where static obstacle exists in and the dynamic obstacle intercepts with ADSeekerbot

The dynamic obstacle that was used is a round plate with the radius 6 cm moving at 5 m/s. The aim of setting up the four test cases is to test the performance of the proposed AD* algorithm to fulfil the completeness, optimality, time complexity and space complexity in different environment [8]. Since the path taken for every run is the same, these experiments were only conducted in single run.

4.1 Test case 1: static obstacle only

In test case 1, a static or non-moving obstacle is placed on the world map. As shown in Fig. 12, the obstacle is placed along the original path of ADSeekerbot (blue line).

During path finding process, A*, D* and AD* algorithms can detect the object and reconstructed a new shortest path (pink line) to reach destination without hitting the obstacle. The distance from start node to destination and time taken to reach from start node to destination for each algorithm are shown in Table 1.

As shown in Table 1, the distance from start to destination for all the algorithms is the same. This is expected

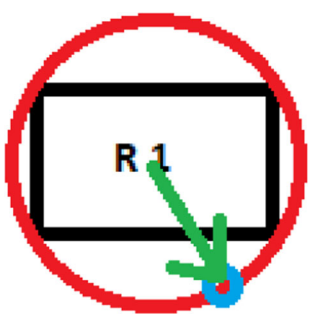


Fig. 10 Indicator for direction of robot

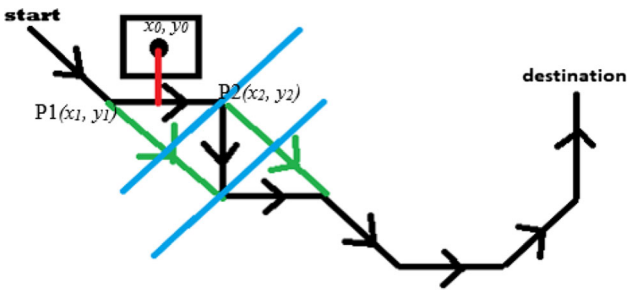


Fig. 11 Path following

because all three algorithms implement the same cost map; thus, the path is the same when static obstacle is detected. There is 0.1 second difference for the time taken by ADSeekerbot to reach destination from the start. Although the path is the same, the speed of ADSeekerbot relies on the power source, in this case, the Li-ion battery. The battery level may vary from time to time, causing inconsistent motor speed. As such, the time to reach the destination may vary even when the path is the same. Nevertheless, the 2% time difference does not impact the behaviour as none of the robot hit the obstacle when a different algorithm is used. The path before grid coordinate $x = 4$, while x starts from 0, the path length of pink path is 4.2426, and the path length of blue is 3.4142. The blue path is shorter than pink path in 24.26%. However, after $x = 4$ the pink path started to pick up blue path and eventually they are having the same path length to reach goal. Table 2 shows the initial cost map of all three algorithms. Zero cost in the map indicates destination.

After taking in consideration of static obstacle, the cost map is shown in Table 3 where all algorithm produces same final cost map for test case 1. Inf indicated the static obstacle. Because three of the algorithms have same cost map, the shortest path found is the same and having almost the same time taken. This shows three algorithms performing the same under simple static obstacle environment.

4.2 Test case 2: dynamic obstacle—non-intercept

In Test case 2, there is a dynamic obstacle across the world map. The dynamic obstacle is not blocking ADSeekerbot to move forward with the shortest path. Figure 13 shows the ADSeekerbot moving from start to destination with the presence of dynamic obstacle. Before dynamic obstacle is detected in the environment, the path length is 8.2426 which is same as the path length in static obstacle. The result of dynamic obstacle is tabulated in Table 4.

As shown in Table 4, the distance from start to destination for all the algorithms is the same. This is expected because all the algorithms implemented the same cost map

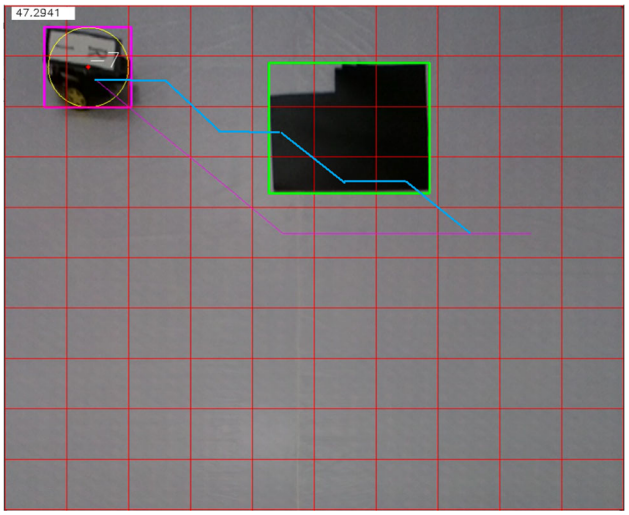


Fig. 12 Static obstacle only

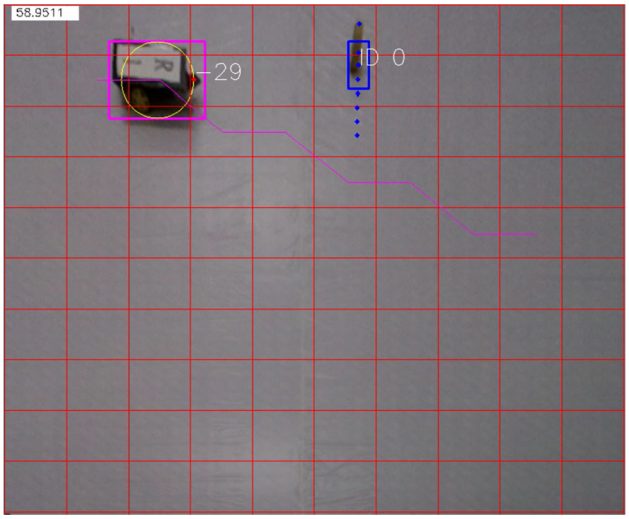


Fig. 13 Dynamic obstacle (non-intercept)

before dynamic obstacle appears. There is 0.1 s difference for the time taken by ADSeekerbot to reach destination from the start. Although the path is the same, the speed of ADSeekerbot relies on the power source, in this case, the Li-ion battery. The battery level may vary from time to time, causing inconsistent motor speed. As such, the time to reach the destination may vary even when the path is the same. Nevertheless, the difference is too small and does not impact the behaviour as none of the robot hit the obstacle when a different algorithm is used.

When dynamic obstacle appears on the path, A* algorithm did not show any difference even when dynamic obstacle intercepts with its future path as shown in Fig. 14. However, D* algorithm will instruct ADSeekerbot to take different paths as compared to A* algorithm, if dynamic obstacle intercepts with its future path. It will re-plan the path according to the dynamic obstacle and always route over the obstacle. In another hand, AD* algorithm predicted that the dynamic obstacle may intercept with its future path. As such, AD* has better prediction and

Table 1 Result for test case 1

	A* algorithm	D* algorithm	AD* algorithm
Distance from start to destination (grid unit)	8.2426	8.2426	8.2426
Time taken to reach destination from start	4.35 s	4.45 s	4.40 s
Hit obstacle	No	No	No

Table 2 Initial cost map

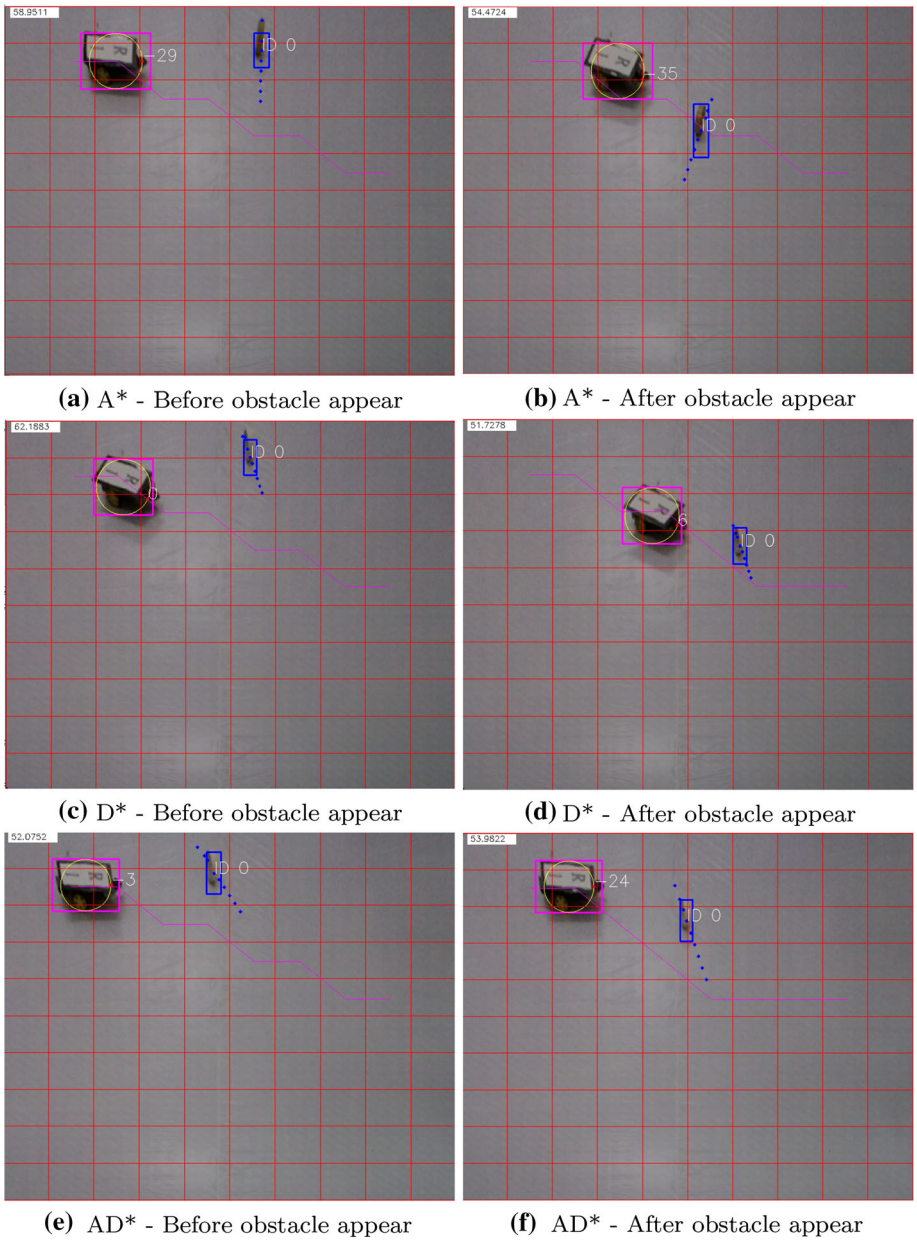
10.0710	9.0710	8.0710	7.0710	6.6568	6.2426	5.8284	5.4141	5	5.4142
9.6568	8.6568	7.6568	6.6568	5.6568	5.2426	4.8284	4.4142	4	4.4142
9.2426	8.2426	7.2426	6.2426	5.2426	4.2426	3.8284	3.4142	3	3.4142
8.8284	7.8284	6.8284	5.8284	4.8284	3.8284	2.8284	2.4142	2	2.4142
8.4142	7.4142	6.4142	5.4142	4.4142	3.414	2.4142	1.4142	1	1.4142
8	7	6	5	4	3	2	1	0	1

Table 3 Test case 1: final cost map

10.0710	9.6568	9.2426	8.8284	7.8284	6.8284	5.8284	5.4142	5	5.4142
9.6568	8.6568	8.2426	7.8284	Inf	Inf	Inf	4.4142	4	4.4142
9.2426	8.2426	7.2426	6.8284	Inf	Inf	Inf	3.4142	3	3.4142
8.8284	7.8284	6.8284	5.8284	Inf	Inf	Inf	2.4142	2	2.4142
8.4142	7.4142	6.4142	5.414	4.4142	3.4142	2.4142	1.4142	1	1.4142
8	7	6	5	4	3	2	1	0	1

Table 4 Result for test case 2

	A* algorithm	D* algorithm	AD* algorithm
Distance from start to destination (grid unit)	8.2426	8.2426	8.2426
Time taken to reach destination from start	4.25 s	4.28 s	4.30 s
Hit obstacle	No	No	No



(a) A* - Before obstacle appear **(b)** A* - After obstacle appear

(c) D* - Before obstacle appear **(d)** D* - After obstacle appear

(e) AD* - Before obstacle appear **(f)** AD* - After obstacle appear

Fig. 14 Dynamic obstacle that does not intercept with ADSeekerbot

Table 5 A* final cost for test case 2

10.0710	9.0710	8.0710	7.0710	6.6568	6.2426	5.8284	5.4141	5	5.4142
9.6568	8.6568	7.6568	6.6568	5.6568	5.2426	4.8284	4.4142	4	4.4142
9.2426	8.2426	7.2426	6.2426	5.2426	4.2426	3.8284	3.4142	3	3.4142
8.8284	7.8284	6.8284	5.8284	4.8284	3.8284	2.8284	2.4142	2	2.4142
8.4142	7.4142	6.4142	5.4142	4.4142	3.414	2.4142	1.4142	1	1.4142
8	7	6	5	4	3	2	1	0	1

Table 6 D* Lite final cost for test case 2

10.0710	9.0710	8.0710	7.0710	6.6568	6.2426	5.8284	5.4141	5	5.4142
9.6568	8.6568	7.6568	6.6568	6.2426	5.2426	4.8284	4.4142	4	4.4142
9.2426	8.2426	7.2426	6.2426	5.2426	4.8284	3.8284	3.4142	3	3.4142
8.8284	7.8284	6.8284	5.8284	4.8284	3.8284	Inf	2.4142	2	2.4142
8.4142	7.4142	6.4142	5.4142	4.4142	3.414	2.4142	1.4142	1	1.4142
8	7	6	5	4	3	2	1	0	1

Table 7 AD* final cost for test case 2

10.0710	9.0710	8.0710	7.0710	6.6568	6.2426	5.8284	5.4141	5	5.4142
9.6568	8.6568	7.6568	6.6568	5.6568	5.2426	4.8284	4.4142	4	4.4142
9.2426	8.2426	7.2426	6.2426	Inf	4.2426	3.8284	3.4142	3	3.4142
8.8284	7.8284	6.8284	5.8284	4.8284	Inf	2.8284	2.4142	2	2.4142
8.4142	7.4142	6.4142	5.4142	4.4142	3.414	2.4142	1.4142	1	1.4142
8	7	6	5	4	3	2	1	0	1

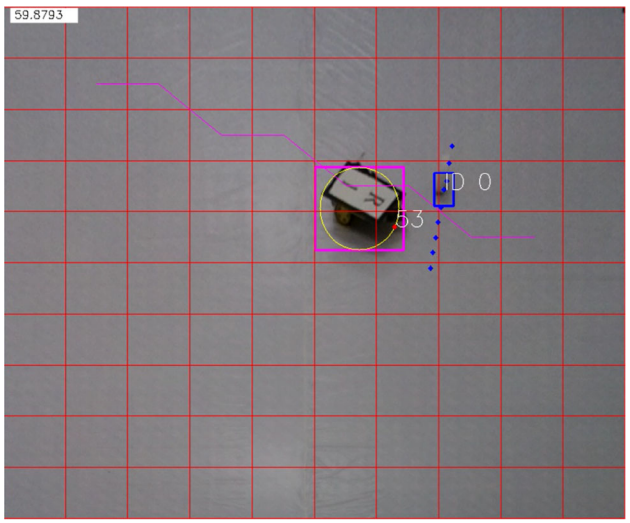


Fig. 15 Dynamic obstacle (intercept)

estimation about future path if obstacle intercepts with ADSeekerbot.

Tables 5, 6 and 7 show the final cost map of each algorithm. In Table 5, the cost map did not change from case 1 to case 2 as compared with Table 3. Since the cost map of A* algorithm did not change, A* algorithm will react towards dynamic obstacle. On the other hand, D* Lite and AD* show different cost maps as compared with A* once dynamic obstacle is detected. D* Lite and AD* will re-route the path as illustrated in Fig. 14. Both D* Lite and AD* show same path length after re-routing. Therefore, D* Lite and AD* algorithms are better than A* algorithm when dynamic obstacle presents in the environment.

4.3 Test case 3: dynamic obstacle—intercept

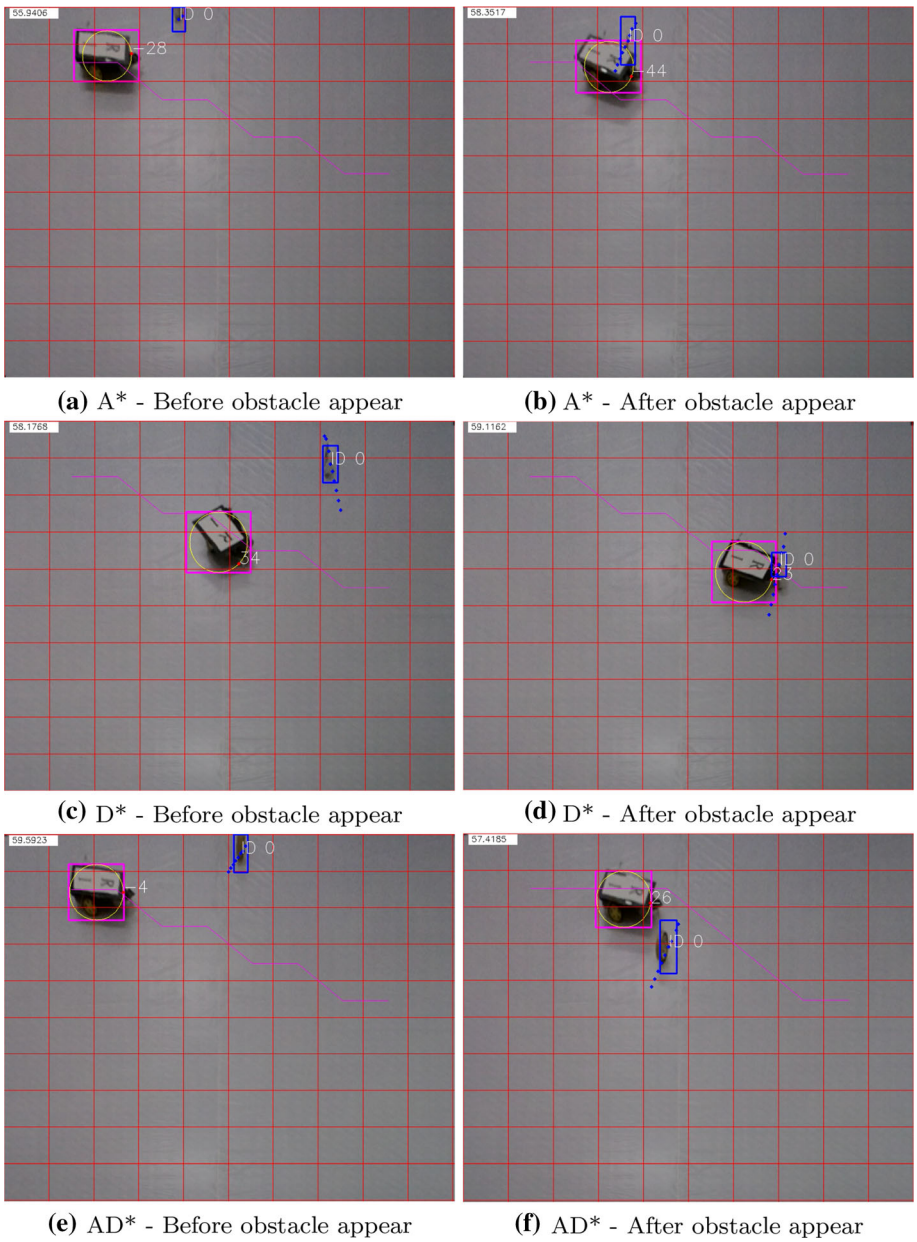
In test case 3, a dynamic obstacle moves across the world map with the aim to collide with ADSeekerbot as shown in Fig. 15. Before the presence of dynamic obstacle, the path length that is calculated from path finding algorithm is 8.2363, which is same as the path length in static obstacle.

As shown in Table 8, the distance from start to destination for all the algorithms is the same, which is 8.2426. This is expected because all three algorithms implement the same cost map. When dynamic obstacle is detected, a new path is reconstructed when AD* algorithm is used. The path is almost the same when A* and D* algorithms are used. However, the robot hit the obstacle, causing delay or slowdown to reach the destination. AD* did not hit the obstacle; therefore, it is the fastest to reach destination. Compared with other method, AD* is faster than other for 8% for one dynamic obstacle. If more obstacles are detected, there will be further delay for the robot to reach destination. When dynamic obstacle is detected, the path changed at location where both dynamic obstacle and ADSeekerbot might intercept as shown in Fig. 16.

There is no path re-planning in A* algorithm even when dynamic obstacle intercepts. As such, when A* algorithm is used, ADSeekerbot will hit the dynamic obstacle. On the other hand, D* algorithm will re-plan the path when dynamic obstacle is detected. However, since there is no

Table 8 Results of test case 3

	A* algorithm	D* algorithm	AD* algorithm
Distance from start to destination(grid unit)	8.2426	8.2426	8.2426
Time taken to reach destination from start	4.85 s	4.91 s	4.5 s
Hit obstacle	Yes	Yes	No



(a) A* - Before obstacle appear **(b)** A* - After obstacle appear

(c) D* - Before obstacle appear **(d)** D* - After obstacle appear

(e) AD* - Before obstacle appear **(f)** AD* - After obstacle appear

Fig. 16 Dynamic obstacle intercepts with ADSeekerbot

Table 9 A* final cost for test case 3

10.0710	9.0710	8.0710	7.0710	6.6568	6.2426	5.8284	5.4141	5	5.4142
9.6568	8.6568	7.6568	6.6568	5.6568	5.2426	4.8284	4.4142	4	4.4142
9.2426	8.2426	7.2426	6.2426	5.2426	4.2426	3.8284	3.4142	3	3.4142
8.8284	7.8284	6.8284	5.8284	4.8284	3.8284	2.8284	2.4142	2	2.4142
8.4142	7.4142	6.4142	5.4142	4.4142	3.414	2.4142	1.4142	1	1.4142
8	7	6	5	4	3	2	1	0	1

prediction about the future path of the dynamic obstacle during the path planning process, ADSeekerbot will hit the obstacle. In another hand, AD* algorithm shows better reaction as it will direct the ADSeekerbot to manoeuvre at different paths in order to avoid the dynamic obstacle. Tables 9, 10 and 11 show the final cost map of all three algorithms. A* algorithm did not show any changes on the cost map even when dynamic obstacle approached the robot. On the other hand, D* Lite and AD* cost map changed when dynamic obstacle approached the robot. Since the cost map of A* did not change, A* algorithm cannot avoid dynamic obstacle. For D* Lite algorithm,

Table 10 D* Lite final cost for test case 3

10.0710	9.0710	8.0710	7.0710	6.6568	6.2426	5.8284	5.4141	5	5.4142
9.6568	8.6568	7.6568	6.6568	6.2426	5.2426	4.8284	4.4142	4	4.4142
9.2426	8.2426	7.2426	6.2426	5.2426	4.8284	3.8284	3.4142	3	3.4142
8.8284	7.8284	6.8284	5.8284	4.8284	3.8284	2.8284	Inf	2	2.4142
8.4142	7.4142	6.4142	5.4142	4.4142	3.414	2.4142	1.4142	1	1.4142
8	7	6	5	4	3	2	1	0	1

Table 11 AD* final cost for test case 3

10.0710	9.6667	8.0710	7.0710	6.6568	6.2426	5.8284	5.4141	5	5.4142
9.6568	8.6568	8.2426	6.6568	5.6568	5.2426	4.8284	4.4142	4	4.4142
9.2426	8.2426	7.2426	6.8284	Inf	4.2426	3.8284	3.4142	3	3.4142
8.8284	7.8284	6.8284	5.8284	Inf	3.8284	2.8284	2.4142	2	2.4142
8.4142	7.4142	6.4142	5.4142	4.4142	3.414	2.4142	1.4142	1	1.4142
8	7	6	5	4	3	2	1	0	1

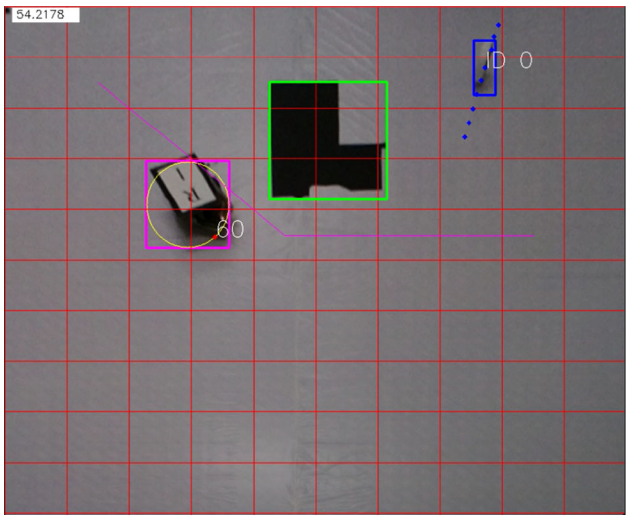


Fig. 17 Dynamic and static obstacle

although the cost map changed, the changes were small. In addition, the re-routing plan is slow, making it unable to avoid obstacle. AD* cost map changes using dynamic obstacle future path information and re-routes correctly to avoid dynamic obstacle.

4.4 Test case 4: dynamic and static obstacle

In test case 4, a static obstacle and a dynamic obstacle move across the world map. The dynamic obstacle is meant to intercept with ADSeekerbot as shown in Fig. 17.

As shown in Table 12, the distance from start to destination for all the algorithms is the same before dynamic obstacle appears. This is expected because all three algorithms implement the same cost map; thus, the path is the same before dynamic obstacle appears. After dynamic obstacle appears, with the consideration of dynamic obstacle AD* changed the path to avoid dynamic obstacle. The path for AD* is slightly longer than D* and A* about 0.8284 or 9.1%, but there is no much difference for the time taken by ADSeekerbot to reach destination from the source between different algorithms. Although the path of AD* algorithm is longer than A* and D*, time taken by ADSeekerbot to reach destination node using A* and D* is longer because the robot hit the dynamic object as shown in Fig. 18 (Tables 13, 14).

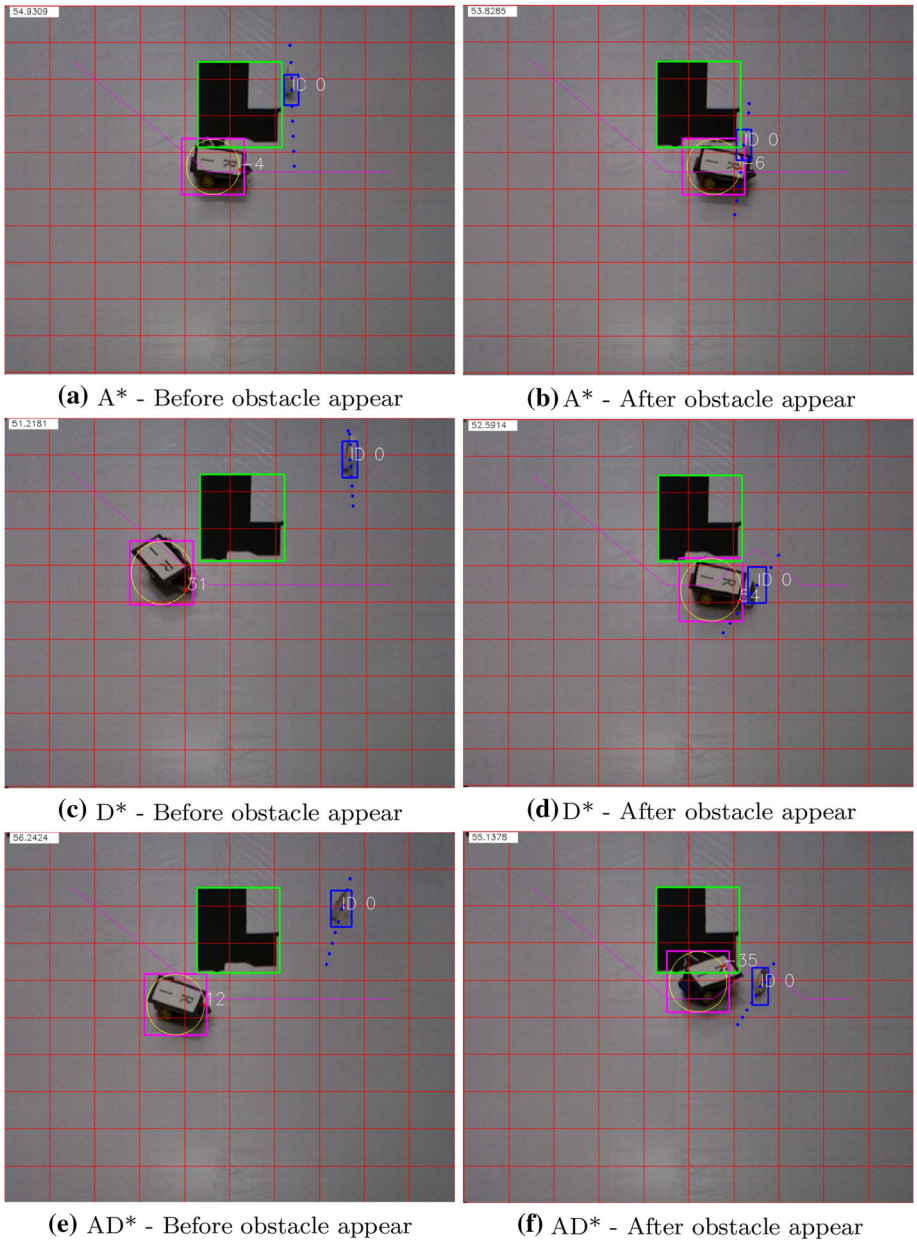
As shown in Table 15, when AD* is used, the cost map changes accordingly to the location of dynamic obstacle and its future path. Due to the dynamic obstacle, AD* re-routes longer path to avoid dynamic obstacle. The changes in cost map influence the shortest path to destination and hence avoid dynamic obstacle. Since A* cost map did not change from beginning to end, ADSeekerbot is unable to avoid dynamic obstacle. Although the cost map and path for D* algorithm changed, the changes are too late to allow robot react and avoid the dynamic obstacle.

5 Conclusion

In a nut shell, AD* algorithm works perfectly in both static and dynamic environment. In addition, AD* is able to avoid hitting the dynamic obstacle due to the online path

Table 12 Test result for test case 4

	A* algorithm	D* algorithm	AD* algorithm
Distance from start to destination (grid unit)	8.2426	8.2426	9.071
Time taken to reach destination from start	4.88 s	4.95 s	4.93 s
Hit obstacle	Yes	Yes	No



(a) A* - Before obstacle appear **(b)** A* - After obstacle appear

(c) D* - Before obstacle appear **(d)** D* - After obstacle appear

(e) AD* - Before obstacle appear **(f)** AD* - After obstacle appear

Fig. 18 Dynamic and static obstacle result

Table 13 A* final cost for test case 4

10.0710	9.6568	9.2426	8.8284	7.8284	6.82843	5.8284	5.4142	5	5.4142
9.6568	8.6568	8.2426	7.8284	Inf	Inf	4.828	4.4142	4	4.4142
9.2426	8.2426	7.2426	6.8284	Inf	Inf	3.828	3.4142	3	3.4142
8.8284	7.8284	6.8284	5.8284	Inf	Inf	2.828	2.4142	2	2.4142
8.4142	7.4142	6.4142	5.4142	4.4142	3.4142	2.4142	1.4142	1	1.4142
8	7	6	5	4	3	2	1	0	1

planning capability. The future location of obstacle provides useful information to AD* algorithm so that it can avoid the dynamic obstacle. The existing AD* algorithm depends on the cameras for re-planning process; thus, blind spot issue may occur and affect the path planning process. Nevertheless, the algorithm can be a base in developing new algorithm that can avoid dynamic obstacle to obtain the shortest path. Thus, it is suitable to be used in warehouse robot.

Table 14 D* Lite final cost for test case 4

10.0710	9.6568	9.2426	8.8284	7.8284	6.8284	5.8284	5.4142	5	5.4142
9.6568	8.6568	8.2426	7.8284	Inf	Inf	4.8284	4.4142	4	4.4142
9.2426	8.2426	7.2426	6.8284	Inf	Inf	3.8284	3.4142	3	3.4142
8.8284	7.8284	6.8284	5.8284	Inf	Inf	2.8284	2.4142	2	2.4142
8.4142	7.4142	6.4142	5.4142	4.4142	3.4142	Inf	1.4142	1	1.4142
8	7	6	5	4	3	2	1	0	1

Table 15 AD* final cost for test case 4

10.0710	9.6568	8.0710	7.6568	6.6568	6.2426	5.8284	5.4141	5	5.4142
10.485	8.6568	8.2426	7.8284	Inf	Inf	4.8284	4.4142	4	4.4142
10.0710	9.071	7.2426	6.8284	Inf	Inf	3.8284	3.4142	3	3.4142
9.6568	8.6568	7.6568	5.8284	Inf	Inf	2.8284	2.4142	2	2.4142
9.2426	8.2426	7.2426	6.2426	4.4142	3.4142	Inf	1.4142	1	1.4142
8.8284	7.8284	6.8284	5.8284	4.8284	Inf	2	1	0	1

Acknowledgements This research was supported by Publication Fund under Research Creativity and Management Office, Universiti Sains Malaysia.

Compliance with ethical standards

Conflict of interest The authors declare that they have no conflict of interest.

References

1. Dubois J, Hamilton R (2017) The team who created Amazon's warehouse robots returns with a new robot named Chuck. Retrieved from https://www.cnbc.com/2017/07/26/6-river-systems-former-kiva-execs-build-warehouse-robot.html
2. Markets and Markets (2017) Warehouse robotics market by type mobile, articulated, cylindrical, SCARA, parallel & Cartesian. Retrieved from https://www.marketsandmarkets.com/Market-Reports/warehouse-robotic-market-128876258.html
3. Ganeshmurthy M, Suresh G (2017) Path planning algorithm for autonomous mobile robot in dynamic environment. In: 2015 3rd International conference in signal processing, communication and networking (ICSCN), pp 1–6
4. Ferguson D, Likhachev M, Stentz A (2005) A guide to heuristic-based path planning. In: Proceedings of the international workshop on planning under uncertainty for autonomous systems at international conference on automated planning and scheduling (ICAPS), pp 9–18
5. Sariff N, Buniyamin N (2006) An overview of autonomous mobile robot path planning algorithms. In: 2006 4th student conference on research and development, pp 183–188
6. Silva JB, Siebra CA, do Nascimento TP (2015) A new cost function heuristic applied to A* based path planning in static and dynamic environments. In: 2015 12th Latin American robotics symposium and 2015 3rd Brazilian symposium on robotics (LARS-SBR), pp 37–42
7. Atyabi A (2013) Powers DM (2013) Review of classical and heuristic-based navigation and path planning approaches. Int J Adv Comput Technol 5(14):1
8. Russell SJ, Norvig P (2009) Artificial intelligence: a modern approach. Pearson Education Limited, New Delhi
9. Otte MW (2015) A survey of machine learning approaches to robotic path-planning. University of Colorado at Boulder, Boulder
10. Games RB (2014) Introduction to A*. Retrieved from https://www.redblobgames.com/pathfinding/a-star/introduction.html
11. Dijkstra EW (1959) A note on two problems in connexion with graphs. Numer Math 1(1):269–271
12. Reddy H (2013) Path finding—Dijkstra's and A* algorithm's. Int J IT Eng 1–15. Retrieved from http://cs.indstate.edu/hgopireddy/algor.pdf
13. Hart PE, Nilsson NJ, Raphael B (1968) A formal basis for the heuristic determination of minimum cost paths. IEEE Trans Syst Sci Cybern 4(2):100–107
14. Wang W, Ye S (2000) Principle and application of artificial intelligence. Electronic Industries Publishing Company
15. Li L, Tao Y, Langton C (2002) Current situation and future of researching move robot technology. Robot 24(5):475–480
16. Duchon F, Babinec A, Kajan M, Beno P, Florek M, Fico T, Jurišica L (2014) Path planning with modified a star algorithm for a mobile robot. Procedia Eng 96:59–69
17. Abiyev RH, Arslan M, Gunsel I, Cagman A (2017) Robot pathfinding using vision based obstacle detection. In: 3rd IEEE international conference on cybernetics (CYBCONF), pp 1–6
18. Koenig S, Likhachev M (2002) D* Lite. In: Eighteenth national conference on artificial intelligence, pp 476–483
19. Bay H, Tuytelaars T, Van Gool L (2006) SURF: speeded up robust features. In: European conference on computer vision, pp 404–417
20. Muja M, Lowe DG (2009) Fast approximate nearest neighbors with automatic algorithm configuration. In: International conference on computer vision theory and applications (VISAPP'09), pp 331–340
21. OpenCV (2017) Thresholding with OpenCV. Retrieved from https://docs.opencv.org/3.4.0/d7/d4d/tutorial_py_thresholding.html
22. KaewTraKulPong P, Bowden R (2001) An improved adaptive background mixture model for real-time tracking with shadow detection. In: 2nd European workshop on advanced video based surveillance systems (AVBS01), pp 1–5
23. Henriques JF, Caseiro R, Martins P, Batista J (2014) High-speed tracking with kernelized correlation filters. IEEE Trans Pattern Anal Mach Intell 37(3):583–596