Real-Time Search-based Planning in Structure Environments

Than Le [1]

[1]Universty of Bordeaux

October 30, 2023

Abstract

In this paper, we address the data sending and visualization in search-based planning using the open source software based on motion planning problems. First, we explore the computing architecture of software where we can communicate with other devices or sensors. It also is to understand the finding path problem by using the A-Start algorithm. By the way, it is

integrated to ROS (Robot Operation System) and implemented

in Nao Humanoid Robot based on solving the optimize the

trajectories.

Real-Time Search-based Planning in Structure Environments

[1,2]Than D. Le
University of Bordeaux
Bordeaux, France
Email: than.ld@ieee.org

[2]Faculty of Information Technology
Ton Duc Thang University
Ho Chi Minh City, Vietnam

***Abstract*—In this paper, we address the data sending and visualization in search-based planning using the open source software based on motion planning problems. First, we explore the computing architecture of software where we can communicate with other devices or sensors. It also is to understand the finding path problem by using the A-Start algorithm. By the way, it is integrated to ROS (Robot Operation System) and implemented in Nao Humanoid Robot based on solving the optimize the trajectories.**

***Index Terms*—motion planning, swarm mobile robotics, obstacle avoidance, bug algorithms**

I. Introduction

Search-based planning is recently implements in [1], [3], [4]. Moreover, Footstep search-based planning [5] was used to integrate in ROS (Robot Operating System) based on understanding theoretical concepts.

In real-time applications of robotics and autonomous systems [2], there are precisely understanding the software cognitive architect and linked sensor perception systems in order to build a complete AI system.

In this paper, we propose the context of navigation for software architecture and applied search-based planning for solving the real-time by using humanoid robots.

II. The architecture of ROS at file-system level

A. Environment

The organization of ROS files is structured on the hard disk in a certain way as an operating system. In this level, we can see the structure of ROS data folder on the disk.

III. Path-Finding in Search-based Planning

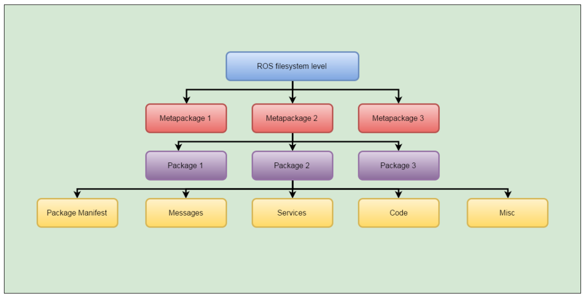
One of the most basic requirement when operating a mobile or humanoid robot is to be able to successfully navigate in the map environment. The navigation problem includes 2 parts: locomotion and path-finding. While locomotion is concerned with the physical motion of a robot such as: control the joints, the way robots interact with the environment ect., path-finding is concerned with finding a valid route for the robot to go from location to other location. In short, the path-finding is a method of search that finds a route between two points in an environment. There are many methods to find the valid route: BellmanFord algorithm, Dijkstra's algorithm, Floyd-Warshall algorithm, etc. In this thesis, the A* Shortest Path Finding Algorithm is used. In this chapter, we am going to talk about some path-finding algorithms and the reason why we chose A* Algorithms:

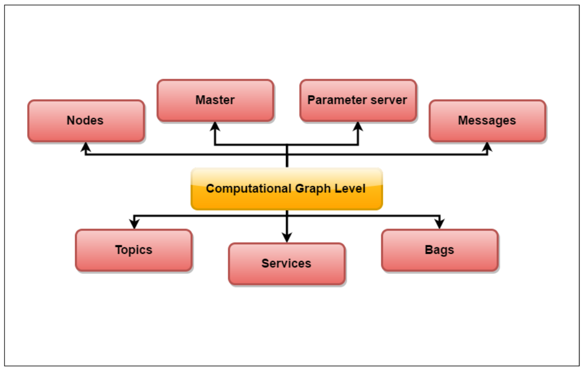
A. Dijkstra's Algorithm and Best-First-Search

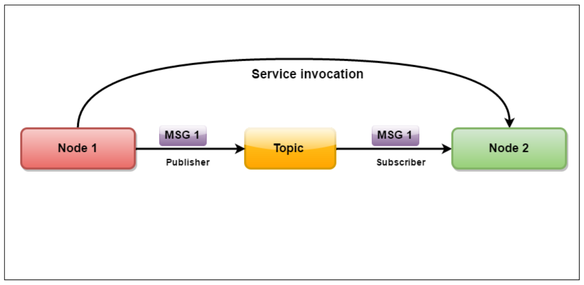
A common example of a graph-based path-finding algorithm is Dijkstra's algorithm. Dijkstra's algorithm is a shortest path-finding algorithm conceived by computer scientist Edsger W. Dijkstra in 1956. It works by visiting a set of open nodes in the graph starting with the starting node. It then repeatedly examines the closest node with the lowest distance cost that have not been examined , adding its to the set of Closed node (nodes that have been examined). It expands outwards from the starting point until it reaches the goal. If there are no negative edge node (node with the negative distance cost), Dijkstra's algorithm is guaranteed to find a shortest path from the starting point to the goal, since the lowest distance nodes are examined first. In the following map, the star is the starting point, the "X" is the goal, the white path is the calculated path and the blue and area inside it is the areas Dijkstra's algorithm have scanned.

First, we create an open list and closed list. The open node list start with the start node and contains all nodes that have not yet been checked. The closed node list stores all node that have been visited.(moved from the open node list). The algorithm works by maintaining these two lists. The core loop of the algorithm selects a node from the open list with the lowest estimated cost (f) to reach the goal. If one of the selected node is the goal, the search will be stopped. Else, it calculate the node cost then push all the valid direction nodes(8 nodes around the current node) into the open list. Then the checked node is moved to the closed node list.The process repeats until the path is generated.

A* Shortest Path Finding Algorithm Peter Hart was first described by Nils Nilsson and Bertram Raphael of Stanford Research Institute (now SRI International) in 1968. It is an extension of Edsger Dijkstra's 1959 algorithm. Since then, it has become the leading path-finding algorithm. A* Algorithm is widely used in map navigation and graph traversal, the process of plotting an efficiently traversable path between multiple nodes. A* is a best-first search algorithms, meaning that it will choose the path considered as the best solution (least distance traveler, shortest time, etc.) by searching among all possible







(a) A tiger

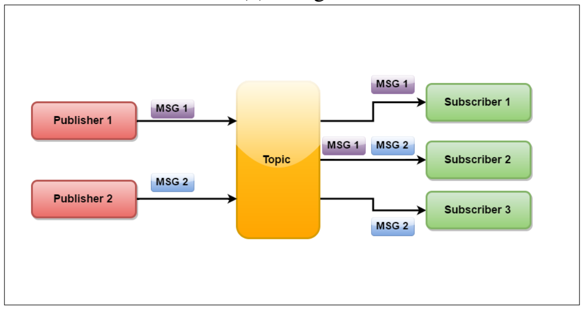


Fig. 1: [3] Left: Bug 1; Center: Bug 2; Right: Targent Bug;

paths to the target. As we have mentioned before, Dijkstra's Algorithm is accuracy to find the shortest path, but it wastes time exploring in directions that aren't promising while Greedy Best First Search explores in promising directions but it may return the longer path result. The A* algorithm calculates both the actual distance from the start and the estimated distance to the goal so it can guarantee to find shortest path while taking much less time Dijkstra's Algorithm. First, let's define the cost function:

First, we create an open list and closed list. The open node list start with the start node and contains all nodes that have not yet been checked. The closed node list stores all node that have been visited.(moved from the open node list). The algorithm works by maintaining these two lists. The core loop of the algorithm selects a node from the open list with the lowest estimated cost (f) to reach the goal. If one of the selected node is the goal, the search will be stopped. Else, it calculate the node cost then push all the valid direction nodes(8 nodes around the current node) into the open list. Then the checked node is moved to the closed node list.The process repeats until the path is generated.

- **Node**: Each node has their own position on the map and also has 3 cost values associated with it. A* Algorithms will take these 3 cost values to decide which node to consider first
  - **g score:** The g score is the distance cost of moving from the start node to this node.
  - **H score - the heuristic:** this is an estimate cost of the distance between each node and the goal. The method we use to calculate H score is very important, it decides how "good" your algorithm is. The implementation of the H score can vary depending on the situation you are working on, here are some common heuristics.
    - ∗ Manhattan distance: In Manhattan, the shortest possible path between two points is not a straight line. The green line is diagonal, straight-line distance. The red/blue/yellow lines are Manhattan distances.

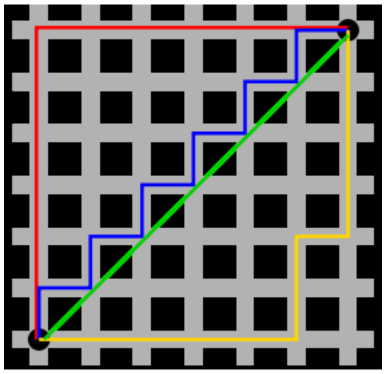


Fig. 2: Rviz shows only black blocks

    - ∗ Euclidean distance: The Euclidean distance between points a and b is the length of the line segment connecting them.
  - **f score:** The f score is calculated by the equation:

$$f(n) = g(n) + h(n) \tag{1}$$

This equation represents the total path distance cost of the current node.

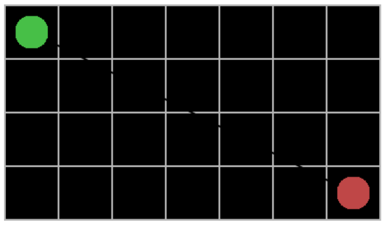


Fig. 3: Rviz shows only black blocks

Algorithm 1: AStar($s_{\text{start}}, s_{\text{goal}}$)

Input : start node s_{start}, goal node s_{goal}
Output: shortest path P from s_{start} to s_{goal}.

```
begin
    $G = \emptyset$ ;      /* G-score look-up table */
    $T = \emptyset$ ;      /* Parent reference list */
    $F = \emptyset$ ;    /* F-score priority queue */
    $G[s_{start}] = 0$;
    $T[s_{start}] = NULL$;
    $F[s_{start}] = h(s_{start}, s_{goal}) + G[s_{start}]$;
    $s = s_{start}$;
    while $F \neq \emptyset$ or $s \neq s_{goal}$ do
        $s = F.ExtractMin()$;
        for node $n$ in $Neighbor(s)$ do
            $g_{new} = G[u] + cost(s, n)$;
            if $n \notin T$ or $g_{new} < G[n]$ then
                $T[n] = s$;
                $G[n] = g_{new}$;
                $F[n] = h(n, s_{goal}) + G[n]$;
    $P = \emptyset$;
    $s = s_{goal}$;
    $P.append(s)$;
    while $s \neq s_{start}$ do
        $s = T[s]$;
        $P.append(s)$;
    $P.reverse()$;
    return $P$;
```

IV. Implementation

A. Displaying NAO in RVIZ

After bring up all Nao drivers, we can display Nao in RVIZ (Fig: **??**) on MATLAB simulation engine according to the diagram below. Our implementation depends heavily on Curve Intersecting library, which is the underlying mechanism of detecting discontinuities when objects present ahead in simulation environment. At first, the simulator initialize the map according to user input about position of obstacles, initial robot pose and goal pose. After generating map and displaying it, our program creates Range Sensor instance that returns sensor lines data and analyses these data to gain discontinuities. After that, these discontinuities are pushed into Tangent Bug Algorithm to generate movement decision until the goal is reached, the algorithm terminate. Otherwise, it will report failure to reach goal.

We have experimented the simulation with different scenarios, all the results are success that the algorithm terminates correctly when the robot reaches the goal.

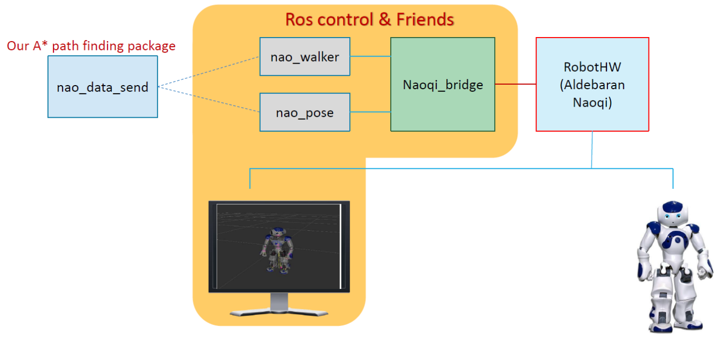


Fig. 4: Overview of Architecture System

The following content is the describtion of 3 metapackages

1) The nao_robot metapackage:: Contains the core functionality libraries. With 3 packages inside and many useful nodes to intergrate with Nao robot:

- *nao_bringup*: includes launch and configuration files as a single-entry point for nao. We we execute the nao_bringup launch files, all basic actuators and sensor publishers in the robot specific configurations will start to active.
- *nao_description*: Store the urdf model file of Nao robot which contains all joints and links according to the documentation by Aldebaran Robotics for V3 and V4 Naos. We can use robot_state_publisher launch file to display the robot's state of joint angles.
- *nao_apps*: This package stores a variety of application nodes for NAO using the NAOqi API, included: nao_alife, nao_behavior, nao_diagnostic_updater, nao_footsteps, nao_leds, nao_speech, nao_tactile, nao_walker (illustrated Fig. 4).

2) The naoqi_bridge metapackage: The nao extra metapackage: This metapackage provides extra tools for the Nao robot: teleoperation with a gamepad and path following.

- naoqi_driver_py: This pakage is a python implementation of the driver package for the Nao robot. It provides access to walking commands, joint angles, and sensor data (odometry, IMU,).
- naoqi_driver: It's the same with the former python edition naoqi driver py but It is the C++ implementation.
- naoqi_pose: This package contains nodes for managing Nao's poses, invloved: pose_manager and pose_controller.
- naoqi_sensors_py: includes ROS driver for camera, sonar sensors and microphone on NAO.
- naoqi_tools: Set of tools provided by Aldebaran to convert Aldebaran files (URDF, blender...)

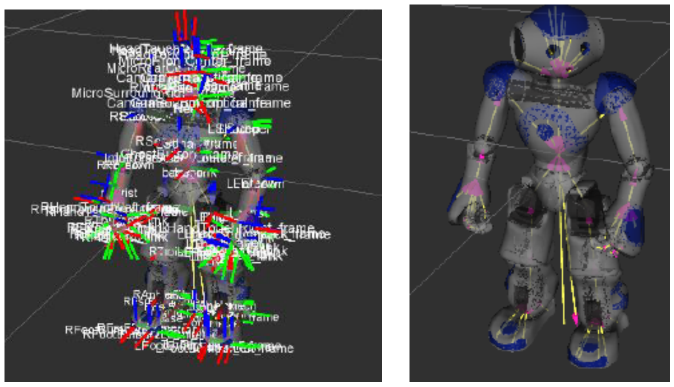


Fig. 5: Left: Simulated Nao on Rvizy; Right: Simulated Nao on Rviz with extra axe and name shown

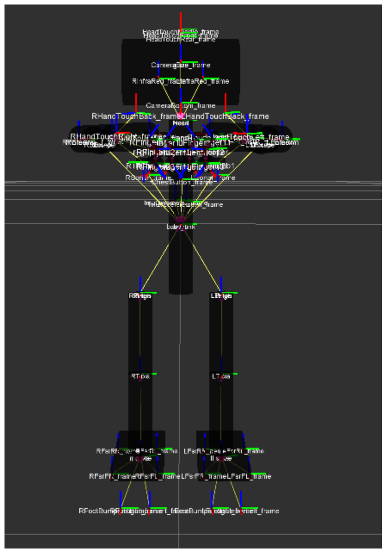
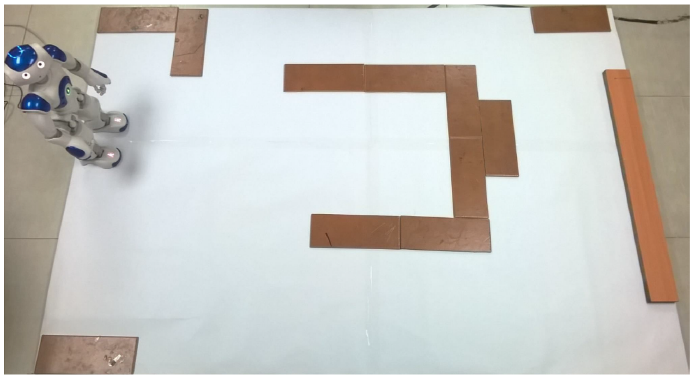


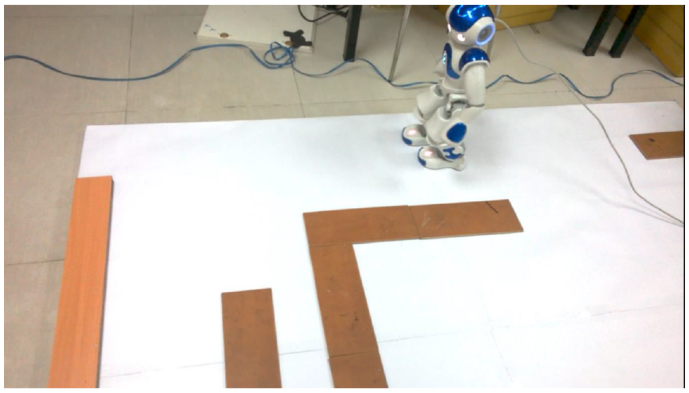
Fig. 6: Rviz shows only black blocks

3) The nao_extra metapackage: This metapackage provides extra tools for the Nao robot: teleoperation with a gamepad and path following.

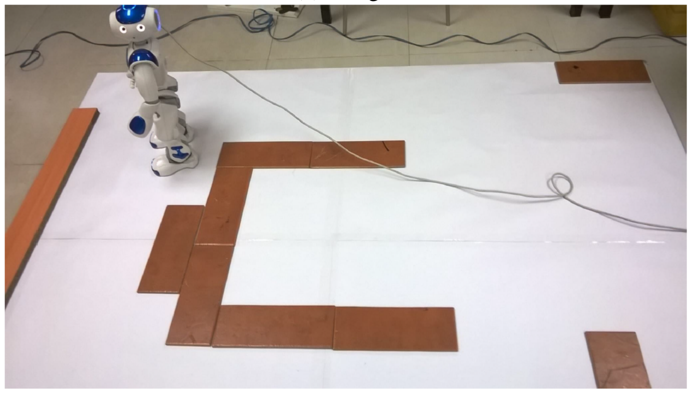
- nao_path_follower: It enables a Nao Robot to either walk to a target location (with localization feedback), or follow a planned 2D path closely. The method is to send naoqi msgs to the nao walker node in nao apps. However, this node is not fully maintain and developed. No launch file and config file is provided.
- nao_teleop: This package allows you to teleoperate Nao using any joystick or gamepad configured in ROS with omnidirectional velocities. You need to have nao_walker and nao_controller node activated, which is impossible to work with ROS Indigo. Simply because nao controller node is the very old node, and not supported or included on our driver package anymore. However, you can still run this nao teleop node together with nao walker node to control Nao walking direction.

B. Result





(a) A tiger



(b) A mouse

Fig. 7: Top: Visualize the real-time environment - (1) m-line: is the linear and shortest distance between start point and final goal point as the assumption. (2) Bottom:

The problem we faced while trying to show Nao on Rviz is the Naop model not showing 3D meshes but only show black blocks. The reason that lead to this problem is the Nao 3D model is not installed probably. The solution is simple, just type the following command, accept the licence argreement and the problem will be solved.

In our case, the problem is solve, but not completely. We can see the Nao meshes instead of the black blocks now (shown 6). However, the Nao meshes is on wrong position, it collides with each others and cant receive the pose from the real Nao.

After searching the error, we found out that the problem is cause because of the ROS installation was not setup carefully. The solution for this problem is to setup ROS again, you can see (ROS first-time installation). After that, catkin make your catkin package again. Now the problem is gone.

V. Conclusion

This paper has successfully design and build a navigation software. It's the result of understanding ROS concepts, Nao Robot hardware and based on search-based planning, we also applied the Path Finding Algorithms. The Robot can follow the 2D path generated by the software simulation in structure environments, but somewhat inaccurate due to lack of feed back from the robot.

In the future, these improvements can be made: (1) Applied SLAM(Simultaneous Localization and Mapping) method so we can scan the environment and build a map for Nao. (2) Design a 3D map using Rviz or Webot. (3) Cross-compile ROS directly on Nao, so we can control it using wifi. (4) Design a feedback system for better control

References

[1] Khiem N. Doan, An T. Le, Than D. Le, Nauth Peter, *"Swarm Robots' Communication and Cooperation in Motion Planning"*, Springer International Publishing, Page 191–205, 2017.

[2] Than D. Le, Le T. An, Duy T. Nguyen, *Model-based Q-learning for humanoid robots*, 18th International Conference on Advanced Robotics (ICAR), 2018.

[3] An T. Le and Than D. Le (December 20th 2017). *Search-Based Planning and Replanning in Robotics and Autonomous Systems, Advanced Path Planning for Mobile Entities,* Rastislav Róka, IntechOpen, DOI: 10.5772/intechopen.71663. Available from: https://www.intechopen.com/books/advanced-path-planning-for-mobile-entities/search-based-planning-and-replanning-in-robotics-and-autonomous-systems

[4] An T. Le, Minh Q. Bui, Than D. Le, Nauth Peter, *D* Lite with Reset: Improved Version of D* Lite for Complex Environment*, The First IEEE International Conference on Robotic Computing (IRC). 2017.

[5] Armin Hornung, Andrew Dornbush, Maxim Likhachev, Maren Bennewitz, *Anytime search-based footstep planning with suboptimality bounds*