Writing Global Path Planners Plugins in ROS: A Tutorial

Maram Alajlan and Anis Koubâa

Abstract In this tutorial chapter, we demonstrate how to integrate a new planner into ROS and present their benefits. Extensive experimentations are performed to show the effectiveness of the newly integrated planners as compared to Robot Operating System (ROS) default planners. The navigation stack of the ROS open-source middleware incorporates both global and local path planners to support ROS-enabled robot navigation. Only basic algorithms are defined for the global path planner including Dijkstra, A*, and carrot planners. However, more intelligent global planners have been defined in the literature but were not integrated in ROS distributions. This tutorial was developed under Ubuntu 12.4 and for ROS Hydro version. However, it is expected to also work with Groovy (not tested). A repository of the new path planner is available at https://github.com/coins-lab/relaxed_astar. A video tutorial also available at https://www.youtube.com/playlist?list=PL8UbFU8tzwRjkxccq2zLkmTkOOYela5fu.

Keywords ROS · Global path planner · Navigation stack

M. Alajlan (✉) · A. Koubâa
Cooperative Networked Intelligent Systems (COINS) Research Group,
Riyadh, Saudi Arabia
e-mail: maram.ajlan@coins-lab.org

A. Koubâa
College of Computer and Information Sciences, Prince Sultan University,
Rafha Street, Riyadh 11586, Saudi Arabia
e-mail: akoubaa@coins-lab.org

M. Alajlan
College of Computer and Information Sciences, King Saud University,
Riyadh 11683, Saudi Arabia

A. Koubâa
CISTER/INESC-TEC, ISEP, Polytechnic Institute of Porto, Porto, Portugal

1 Introduction

Mobile robot path planning is a hot research area. Indeed, the robot should have the ability to autonomously generate collision free path between any two positions in its environment. The path planning problem can be formulated as follows: given a mobile robot and a model of the environment, find the optimal path between a start position and a final position without colliding with obstacles. Designing an efficient path planning algorithm is an essential issue in mobile robot navigation since path quality influences the efficiency of the entire application. The constructed path must satisfy a set of optimization criteria including the traveled distance, the processing time, and the energy consumption.

In the literature, the path planning problem is influenced by two factors: (1) the environment, which can be static or dynamic, (2) the knowledge that the robot has about the environment; if the robot has a complete knowledge about the environment, this problem is known as global path planning. On the other hand, if the robot has an incomplete knowledge, this problem is classified as local path planning.

The navigation stack of the Robot Operating System (ROS) open-source middleware incorporates both global and local path planners to support ROS-enabled robot navigation. However, only basic algorithms are defined for the global path planner including Dijkstra, A*, and carrot planners.

In this tutorial, we present the steps for integrating a new global path planner into the ROS navigation system. The new path planner is based on a relaxed version of the A* (RA*) algorithm, and it can be found in [1]. Also, we compare the performance of the RA* with ROS default planner.

The rest of this tutorial is organized as follow.

- Section 2 introduces ROS and its navigation system.
- Section 3 introduces the relaxed A* algorithm.
- In Sect. 4, we present the steps of integrating a new global path planner into the ROS navigation system.
- Section 5 presents the ROS environment configuration.
- In Sect. 6, we conduct the experimental evaluation study to compare the performance of RA* and ROS default planner.

2 ROS

ROS (Robot Operating System) [2] has been developed by Willow Garage [3] and Stanford University as a part of STAIR [4] project, as a free and open-source robotic middleware for the large-scale development of complex robotic systems.

ROS acts as a meta-operating system for robots as it provides hardware abstraction, low-level device control, inter-processes message-passing and package management. It also provides tools and libraries for obtaining, building, writing, and running code across multiple computers. The main advantage of ROS is that it allows manipulating sensor data of the robot as a labeled abstract data stream, called topic, without having to deal with hardware drivers. This makes the programming of robots much easier for software developers as they do not have to deal with hardware drivers and interfaces. Also, ROS provides many high-level applications such as arm controllers, face tracking, mapping, localization, and path planning. This allow the researchers to focus on specific research problems rather than on implementing the many necessary, but unrelated parts of the system.

Mobile robot navigation generally requires solutions for three different problems: mapping, localization, and path planning. In ROS, the *Navigation Stack* plays such a role to integrate together all the functions necessary for autonomous navigation.

2.1 ROS Navigation Stack

In order to achieve the navigation task, the *Navigation Stack* [5] is used to integrate the mapping, localization, and path planning together. It takes in information from odometry, sensor streams, and the goal position to produce safe velocity commands and send it to the mobile base (Fig. 1). The odometry comes through `nav_msgs/Odometry` message over ROS which stores an estimate of the position and velocity of a robot in free space to determine the robot's location. The sensor information comes through either `sensor_msgs/LaserScan` or `sensor_msgs/PointCloud` messages over ROS to avoid any obstacles. The goal is sent to the navigation stack by `geometry_msgs/PoseStamped` message. The navigation stack sends the velocity commands through `geometry_msgs/Twist` message on `/cmd_vel` topic. The `Twist` message composed of two sub-messages:

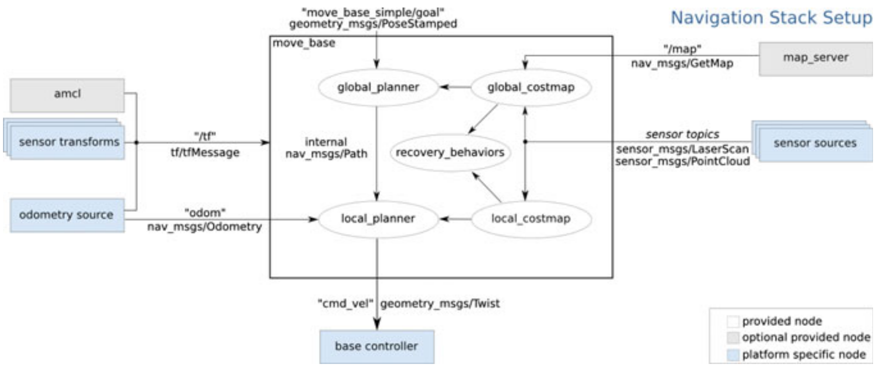

Fig. 1 ROS navigation stack [2]

```
geometry_msgs/Vector3 linear
    float64 x
    float64 y
    float64 z
geometry_msgs/Vector3 angular
    float64 x
    float64 y
    float64 z
```

`linear` sub-message is used for the x, y and z linear velocity components in meters per second and `angular` sub-message is used for the x, y and z angular velocity components in radians per second. For example the following `Twist` message:

```
linear: {x: 0.2, y: 0, z: 0}, angular: {x: 0, y: 0, z: 0}
```

will tell the robot to move with a speed of 0.2 m/s straight ahead. The base controller is responsible for converting `Twist` messages to "motor signals"which will actually move the robot's wheels [6].

The navigation stack does not require a prior static map to start with. Actually it could be initialized with or without a map. When initialized without a prior map, the robot will know about the obstacles detected by its sensors only and will be able to avoid the seen obstacles so far. For the unknown areas, the robot will generate an optimistic global path which may hit unseen obstacles. The robot will be able to re-plan its path when it receives more information by the sensors about these unknown areas. Instead, when the navigation stack initialized with a static map for the environment, the robot will be able to generate an informed plans to its goal using the map as prior obstacle information. Starting with a prior map will have significant benefits on the performance [5].

To build a map using ROS, ROS provides a wrapper for OpenSlam's Gmapping [7]. A particle filter-based mapping approach [8] is used by the `gmapping` package to build an occupancy grid map. Then a package named `map_server` could be used to save that map. The maps are stored in a pair of files: `YAML` file and image file. The `YAML` file describes the map meta-data, and names the image file. The image file encodes the occupancy data. The localization part is solved in the `amcl` package using an Adaptive Monte Carlo Localization [9] which is also based on particle filters. It is used to track the position of a robot against a known map. The path planning part is performed in the `move_base` package, and is divided into *global* and *local* planning modules which is a common strategy to deal with the complex planning problem.

The *global path planner* searches for a shortest path to the goal and the *local path planner* (also called the *controller*), incorporating current sensor readings, issues the actual commands to follow the global path while avoiding obstacles. More details about the global and local planners in ROS can be found in the next sections.

The *move_base* package also maintains two costmaps, *global_costmap* and *local_costmap* to be used with the global and local planners respectively. The costmap used to store and maintain information in the form of occupancy grid about the obstacles in the environment and where the robot should navigate. The costmap

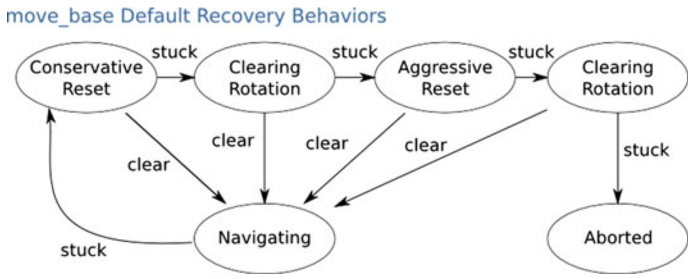

Fig. 2 Recovery behaviors [2]

initialized with prior static map if available, then it will be updated using sensor data to maintain the information about obstacles in the world. Besides that, the *move_base* may optionally perform some previously defined recovery behaviors (Fig. 2) when it fails to find a valid plan.

One reason of failure, is when the robot find itself surrounded with obstacles and cannot find a way to its goal. The recovery behaviors will be performed in some order (defined by the user), and after performing one recovery, the *move_base* will try to find a valid plan, if it succeeds, it will proceed its normal operation. Otherwise if it fails, it will perform the next recovery behavior. If it fails after performing all the recovery behaviors, the goal will be considered infeasible, and it will be aborted. The default recovery behaviors order is presented in Fig. 2 and it is in increasingly aggressive order to attempt to clear out the robot space. First recovery behavior is clearing all the obstacles outside a specific area from the robot's map. Next, an in-place rotation will be performed if possible to clear the space. Next, in case this too fails, more aggressively clearing for the map will be performed, to remove all the obstacles outside of the rectangular area in which the robot can make an in-place rotation. Next, another in-place rotation will be performed. If all this fails, the goal will be aborted.

Therefore, in each execution cycle of the *move_base*, one of three main states should be performed:

- Planning state: run the *global path planner*.
- Controlling state: run the *local path planner* and move the robot.
- Clearing state: run recovery behavior in case the robot stuck.

There are some predefined parameters in ROS navigation stack that are used to control the execution of the states, which are:

- *planner_frequency*: to determine how often the *global path planner* should be called, and is expressed in Hz. When it is set to zero, the global plan will be computed only once for each goal received.
- *controller_frequency*: to determine how often the *local path planner* or *controller* should be called, and also expressed in Hz.

For any global or local planner or recovery behavior to be used with the *move_base* it must be first adhere to some interfaces defined in *nav_core* package, which contains key interfaces for the navigation stack, then it must be added as a plugin to ROS. We developed a tutorial on how to add a new global planner as a plugin to ROS navigation stack, available at [10] and [11].

2.1.1 Global Planner

The *global path planner* in ROS operates on the *global_costmap*, which generally initialized from a prior static map, then it could be updated frequently based on the value of *update_frequency* parameter. The *global path planner* is responsible for generating a long-term plan from the start or current position to the goal position before the robot starts moving. It will be seeded with the costmap, and the start and goal positions. These start and goal positions are expressed by their x and y coordinates. A grid-based global planner that can use Dijkstra's algorithm [12] or A* algorithm to compute shortest collision free path for a robot is obtained in `global_planner` package. Also, ROS provide another global planner named `carrot_planner`, which is a simple planner that attempts to move the robot as close to its goal as possible even when that goal is in an obstacle. The current implementation of the global planner in ROS assumes a circular-shape robot. This results in generating an optimistic path for the actual robot footprint, which may be infeasible path. Besides that, the global planner ignores kinematic and acceleration constraints of the robot, so the generated path could be dynamically infeasible.

2.1.2 Local Planner

The *local path planner* or the *controller* in ROS operates on the *local_costmap*, which only uses local sensor information to build an obstacle map and dynamically updated with sensor data. It takes the generated plan from the global planner, and it will try to follow it as close as possible considering the kinematics and dynamics of the robot as well as any moving obstacles information in the *local_costmap*. ROS provides implementation of two local path planning algorithms namely the Trajectory Rollout [13] and the Dynamic Window Approach (DWA) [14] in the package *base_local_planner*. Both algorithms have the same idea to first discretely sampled the control space then to perform forward simulation, and the selection among potential commands. The two algorithm differ in how they sample the robot's control space.

After the global plan passed to the *controller*, the *controller* will produce velocity commands to send to a mobile base. For each control cycle, *the controller* will try to process part from global path (determined by the size of the *local_costmap*).

First, the *controller* will sampled the control space of the robot discretely. The number of the samples will be specified by the controller parameters *vx_samples* and *vtheta_samples* (more details about the parameters can be found in the next section). Then, the controller will perform a simulation in advance for each one of those velocity samples from the current place of the robot to foresee the situation from applying each sample for amount of time (this time will be specified in the parameter *sim_time*). Then, the controller will evaluate each resultant path from the simulation and will exclude any path having collisions with obstacles. For the evaluation, the controller will incorporates the following metrics: distance from obstacles, distance to the goal position, distance from the global plan and robot speed. Finally, the controller will send the velocity command of the highest-scoring path to the mobile base to execute it.

The "Map Grid" is used to evaluate and score the velocities. For each control cycle, the controller will create a grid around the robot (the grid size determined by the size of the *local_costmap*), and the global path will be mapped onto this area. Then each grid cell will receive a distance value. The cells containing path points and the goal will be marked with 0. Then each other grid cell will be marked with its manhattan distance from nearest zero grid by a propagation algorithm. This "Map Grid" is then used in the evaluation and scoring of the velocities. As the "Map Grid" will cover small area from global path each time, the goal position often will lie outside that area. So in that case the first path point inside the area having a consecutive point outside the area will be considered as "local goal", and the distance from that local goal will be considered when scoring trajectories for distance to goal.

3 Relaxed A*

RA* is a time linear relaxed version of A*. It is proposed to solve the path planning problem for large scale grid maps. The objective of RA* consists of finding optimal or near optimal solutions with small gaps, but at much smaller execution times than traditional A*. The core idea consists of exploiting the grid-map structure to establish an accurate approximation of the optimal path, without visiting any cell more than once.

In fact, in A* the exact cost g(n) of a node n may be computed many times; namely, it is computed for each path reaching node n from the start position. However, in the RA* algorithm g(n) is approximated by the cost of the minimum-move path from the start cell to the cell associated to node n.

In order to obtain the relaxed version RA*, some instructions of A*, that are time consuming with relatively low gain in terms of solution quality, are removed. In fact, a node is processed only once in RA*, so there is no need to use the closed set of the A* algorithm. Moreover, in order to save time and memory, we do not keep track of the previous node at each expanded node. Instead, after reaching the goal, the path can be reconstructed, from goal to start by selecting, at each step, the neighbor having the minimum g(n) value. Also, it is useless to compare the g(n) of each neighbor to the g(n) of the current node n as the first calculated g(n) is considered definite.

Finally, it is not needed to check whether the neighbor of the current node is in the open list. In fact, if its g(n) value is infinite, it means that it has not been processed yet, and hence is not in the open list. The RA* algorithm is presented in Algorithm 1.

```
input : $Grid, Start, Goal$
$tBreak$ = 1+1/(length($Grid$)+width($Grid$));
// Initialisation:
$openSet$ = Start // Set of nodes to be evaluated;
for each vertex $v$ in $Grid$ do
    $g\_score(v)$= infinity;
end
$g\_score[Start] = 0$;
// Estimated total cost from Start to Goal:
$f\_score[Start]$ = heuristic_cost($Start, Goal$);
while $openSet$ is not empty and $g\_score[Goal] == infinity$ do
    $current$ = the node in $openSet$ having the lowest $f\_score$;
    remove $current$ from $openSet$;
    for each free neighbor $v$ of $current$ do
        if $g\_score(v) == infinity$ then
            $g\_score[v] = g\_score[current] + dist\_edge(current, v)$;
            $f\_score[v] = g\_score[v] + tBreak * heuristic\_cost(v, Goal)$;
            add neighbor to $openSet$;
        end
    end
end
if $g\_score(goal) \,!= infinity$ then
    return reconstruct_path($g\_score$) // path will be reconstructed based
    on $g\_score$ values;
else
    return failure;
end
```

Algorithm 1: Relaxed A*

Both terms g(n) and h(n) of the evaluation function of the RA* algorithm are not exact, then there is no guaranty to find an optimal solution.

4 Integration Steps

In this section, we present the steps of integrating a new path planner into ROS. The integration has two main steps: (1) writing the path planner class, and (2) deploying it as a plugin. Following, we describe them in details.

4.1 Writing the Path Planner Class

As mentioned before, to make a new global planner work with ROS, it must first adhere to the interfaces defined in `nav_core` package. A similar example can be found in the `carrot_planner.h` [15] as a reference. All the methods defined in `nav_core::BaseGlobalPlanner` class must be overridden by the new global path planner. For this, you need to create a header file, that we will call in our case, `RAstar_ros.h`

```
/** include the libraries you need in your planner here */
/** for global path planner interface */
#include <ros/ros.h>
#include <costmap_2d/costmap_2d_ros.h>
#include <costmap_2d/costmap_2d.h>
#include <nav_core/base_global_planner.h>
#include <geometry_msgs/PoseStamped.h>
#include <angles/angles.h>
#include <base_local_planner/world_model.h>
#include <base_local_planner/costmap_model.h>

using std::string;

#ifndef RASTAR_ROS_CPP
#define RASTAR_ROS_CPP

namespace RAstar_planner {

class RAstarPlannerROS : public nav_core::BaseGlobalPlanner {
public:

 RAstarPlannerROS();
 RAstarPlannerROS(std::string name, costmap_2d::Costmap2DROS* costmap_ros);

 /** overridden classes from interface nav_core::BaseGlobalPlanner **/
 void initialize(std::string name, costmap_2d::Costmap2DROS* costmap_ros);
 bool makePlan(const geometry_msgs::PoseStamped& start,
        const geometry_msgs::PoseStamped& goal,
        std::vector<geometry_msgs::PoseStamped>& plan
           );
 };
};
#endif
```

Now, we will explain the different parts of the header file.

```
#include <ros/ros.h>
#include <costmap_2d/costmap_2d_ros.h>
#include <costmap_2d/costmap_2d.h>
#include <nav_core/base_global_planner.h>
#include <geometry_msgs/PoseStamped.h>
#include <angles/angles.h>
#include <base_local_planner/world_model.h>
#include <base_local_planner/costmap_model.h>
```

It is necessary to include core ROS libraries needed for path planner. The headers:

```
#include <costmap_2d/costmap_2d_ros.h>
#include <costmap_2d/costmap_2d.h>
```

are needed to use the `costmap_2d::Costmap2D` class that will be used by the path planner as input map. This map will be accessed automatically by the path planner class when defined as a plugin. There is no need to subscribe to `costmap2d` to get the cost map from ROS.

```
#include <nav_core/base_global_planner.h>
```

is used to import the interface $nav\_core :: BaseGlobalPlanner$, which the plugin must adhere to.

```
namespace RAstar_planner {

 class RAstarPlannerROS : public nav_core::BaseGlobalPlanner {
```

It is a good practice, although not necessary, to define namespace for your class. Here, we define the namespace as `RAstar_planner` for the class `RAstarPlanner ROS`. The namespace is used to define a full reference to the class, as `RAstar_ planner::RAstarPlannerROS`. The class `RAstarPlannerROS` is then defined and inherits from the interface `nav_core::BaseGlobalPlanner`. All methods defined in `nav_core::BaseGlobalPlanner` must be overridden by the new class `RAstarPlannerROS`.

```
  RAstarPlannerROS();
```

Is the default constructor which initializes the planner attributes with default values.

```
  RAstarPlannerROS(std::string name, costmap_2d::Costmap2DROS* costmap_ros);
```

This constructor is used to initialize the costmap, that is the map that will be used for planning (`costmap_ros`), and the name of the planner (`name`).

```
  void initialize(std::string name, costmap_2d::Costmap2DROS* costmap_ros);
```

Is an initialization function for the `BaseGlobalPlanner`, which initializes the costmap, that is the map that will be used for planning (`costmap_ros`), and the name of the planner (`name`).

The initialize method for RA* is implemented as follows:

```
void RAstarPlannerROS::initialize(std::string name, costmap_2d::Costmap2DROS* ↩
     costmap_ros)
{
    if (!initialized_)
    {
        costmap_ros_ = costmap_ros;
        costmap_ = costmap_ros_->getCostmap();
        ros::NodeHandle private_nh("~/" + name);

        originX = costmap_->getOriginX();
        originY = costmap_->getOriginY();
        width = costmap_->getSizeInCellsX();
        height = costmap_->getSizeInCellsY();
        resolution = costmap_->getResolution();
        mapSize = width*height;
        tBreak = 1+1/(mapSize);
        OGM = new bool [mapSize];

        for (unsigned int iy = 0; iy < height; iy++)
        {
            for (unsigned int ix = 0; ix < width; ix++)
            {
                unsigned int cost = static_cast<int>(costmap_->getCost(ix, iy));
                if (cost == 0)
                    OGM[iy*width+ix]=true;
                else
                    OGM[iy*width+ix]=false;
            }
        }
        ROS_INFO("RAstar planner initialized successfully");
        initialized_ = true;
    }
    else
        ROS_WARN("This planner has already been initialized... doing nothing");
}
```

For the particular case of the `carrot_planner`, the initialize method is implemented as follows:

```
  void CarrotPlanner::initialize(std::string name, costmap_2d::Costmap2DROS* ↩
       costmap_ros){
    if(!initialized_){
      costmap_ros_ = costmap_ros; //initialize the costmap_ros_ attribute to the ↩
           parameter.
      costmap_ = costmap_ros_->getCostmap(); //get the costmap_ from ↩
           costmap_ros_

     /* initialize other planner parameters */
      ros::NodeHandle private_nh("~/" + name);
      private_nh.param("step_size", step_size_, costmap_->getResolution());
      private_nh.param("min_dist_from_robot", min_dist_from_robot_, 0.10);
      world_model_ = new base_local_planner::CostmapModel(*costmap_);

      initialized_ = true;
    }
    else
      ROS_WARN("This planner has already been initialized... doing nothing");
  }
```

```
 bool makePlan(const geometry_msgs::PoseStamped& start,
        const geometry_msgs::PoseStamped& goal,
        std::vector<geometry_msgs::PoseStamped>& plan
           );
```

Then, the method bool `makePlan` must be overridden. The final plan will be stored in the parameter `std::vector<geometry_msgs::PoseStamped>& plan` of the method. This plan will be automatically published through the plugin as a topic. An implementation of the `makePlan` method of the `carrot_planner` can be found in [16] as a reference.

Class Implementation In what follows, we present the main issues to be considered in the implementation of a global planner as plugin. The complete source code of the RA* planner can be found in [1]. Here is a minimum code implementation of the RA* global path planner (`RAstar_ros.cpp`).

```
#include <pluginlib/class_list_macros.h>
#include "RAstar_ros.h"

//register this planner as a BaseGlobalPlanner plugin
PLUGINLIB_EXPORT_CLASS(RAstar_planner::RAstarPlannerROS, nav_core::↩
     BaseGlobalPlanner)

using namespace std;

namespace RAstar_planner {
//Default Constructor
RAstarPlannerROS::RAstarPlannerROS(){

}
RAstarPlannerROS::RAstarPlannerROS(std::string name, costmap_2d::Costmap2DROS* ↩
     costmap_ros){
  initialize(name, costmap_ros);
}
void RAstarPlannerROS::initialize(std::string name, costmap_2d::Costmap2DROS* ↩
     costmap_ros){

}
bool RAstarPlannerROS::makePlan(const geometry_msgs::PoseStamped& start, const ↩
     geometry_msgs::PoseStamped& goal,
    std::vector<geometry_msgs::PoseStamped>& plan ){
 if (!initialized_) {
   ROS_ERROR("The planner has not been initialized, please call initialize () to use the ↩
          planner");
   return false;
 }
 ROS_DEBUG("Got a start: %.2f, %.2f, and a goal: %.2f, %.2f", start.pose.position.x, ↩
      start.pose.position.y,
           goal.pose.position.x, goal.pose.position.y);
 plan.clear();
 if (goal.header.frame_id != costmap_ros_->getGlobalFrameID()){
   ROS_ERROR("This planner as configured will only accept goals in the %s frame, but a ↩
          goal was sent in the %s frame.",
             costmap_ros_->getGlobalFrameID().c_str(), goal.header.frame_id.c_str↩
                  ());
   return false;
 }
 tf::Stamped < tf::Pose > goal_tf;
 tf::Stamped < tf::Pose > start_tf;

 poseStampedMsgToTF(goal, goal_tf);
 poseStampedMsgToTF(start, start_tf);
```

```

// convert the start and goal coordinates into cells indices to be used with RA* ↩
      planner
 float startX = start.pose.position.x;
 float startY = start.pose.position.y;
 float goalX = goal.pose.position.x;
 float goalY = goal.pose.position.y;

getCorrdinate(startX, startY);
getCorrdinate(goalX, goalY);

int startCell;
int goalCell;

if (isCellInsideMap(startX, startY) && isCellInsideMap(goalX, goalY)){
  startCell = convertToCellIndex(startX, startY);
  goalCell = convertToCellIndex(goalX, goalY);
}
else {
  cout << endl << "the start or goal is out of the map" << endl;
  return false;
}
/////////////////////////////////////////////////////////
// call RA* path planner
if (GPP->isStartAndGoalCellsValid(OGM, startCell, goalCell)){
  vector<int> bestPath;
  bestPath = RAstarPlanner(startCell, goalCell); // call RA*

  //if the global planner find a path
  if ( bestPath->getPath().size()>0)
  {
   // convert the path cells indices into coordinates to be sent to the move base
    for (int i = 0; i < bestPath->getPath().size(); i++){
      float x = 0.0;
      float y = 0.0;
      int index = bestPath->getPath()[i];

      convertToCoordinate(index, x, y);

      geometry_msgs::PoseStamped pose = goal;
      pose.pose.position.x = x;
      pose.pose.position.y = y;
      pose.pose.position.z = 0.0;
      pose.pose.orientation.x = 0.0;
      pose.pose.orientation.y = 0.0;
      pose.pose.orientation.z = 0.0;
      pose.pose.orientation.w = 1.0;

      plan.push_back(pose);
    }
    // calculate path length
    float path_length = 0.0;
    std::vector<geometry_msgs::PoseStamped>::iterator it = plan.begin();
    geometry_msgs::PoseStamped last_pose;
    last_pose = *it;
    it++;
    for (; it!=plan.end(); ++it) {
        path_length += hypot( (*it).pose.position.x - last_pose.pose.position.x, (*↩
              it).pose.position.y - last_pose.pose.position.y );
        last_pose = *it;
    }
    cout <<"The global path length: "<< path_length<< " meters"<<endl;
    return true;
  }
  else{
    cout << endl << "The planner failed to find a path " << endl
        << "Please choose other goal position, " << endl;
      return  false ;
    }
  }
  else{
    cout << "Not valid start or goal" << endl;
    return  false ;
  }
}
};
```

The constructors can be implemented with respect to the planner requirements and specification. There are few important things to consider:

- **Register the planner as BaseGlobalPlanner plugin**: this is done through the instruction:

```
PLUGINLIB_EXPORT_CLASS(RAstar_planner::RAstarPlannerROS, nav_core::↩
    BaseGlobalPlanner)
```

  For this it is necessary to include the library:

```
#include <pluginlib/class_list_macros.h>
```

- **The implementation of the makePlan() method**: The `start` and `goal` parameters are used to get initial location and target location, respectively. For RA* path planners, the start and goal first will be converted from x and y coordinates to cell indices. Then, those indices will be passed to the RA* planner. When the planner finish its execution, it will return the computed path. Finally, the path cells will be converted to x and y coordinates, then inserted into the plan vector (`plan.push_back(pose)`) in the for loop. This planned path will then be sent to the `move_base` global planner module which will publish it through the ROS topic `nav_msgs/Path`, which will then be received by the local planner module.

Now that your global planner class is done, you are ready for the second step, that is creating the plugin for the global planner to integrate it in the global planner module `nav_core::BaseGlobalPlanner` of the `move_base` package.

Compilation To compile the RA* global planner library created above, it must be added (with all of its dependencies if any) to the *CMakeLists.txt*. This is the code to be added:

```
add_library(relaxed_astar_lib src/RAstar_ros.cpp)
```

Then, in a terminal run `catkin_make` in your catkin workspace directory to generate the binary files. This will create the library file in the lib directory `~/catkin_ws/devel/lib/librelaxed_astar_lib`. Observe that "`lib`" is appended to the library name `relaxed_astar_lib` declared in the `CMakeLists.txt`

4.2 Writing Your Plugin

Basically, it is important to follow all the steps required to create a new plugin as explained in the plugin description page [17]. There are five steps:

Plugin Registration First, you need to register your global planner class as plugin by exporting it. In order to allow a class to be dynamically loaded, it must be marked as an exported class. This is done through the special macro PLUGINLIB_EXPORT_CLASS. This macro can be put into any source (.cpp) file that composes the plugin library, but is usually put at the end of the .cpp file for the exported class. This was already done above in RAstar_ros.cpp with the instruction

```
PLUGINLIB_EXPORT_CLASS(RAstar_planner::RAstarPlannerROS, nav_core::↩
     BaseGlobalPlanner)
```

This will make the class RAstar_planner::RAstarPlannerROS registered as plugin for nav_core::BaseGlobalPlanner of the move_base.

Plugin Description File The second step consists in describing the plugin in a description file. The plugin description file is an XML file that serves to store all the important information about a plugin in a machine readable format. It contains information about the library the plugin is in, the name of the plugin, the type of the plugin, etc. In our case of global planner, you need to create a new file and save it in certain location in your package and give it a name, for example relaxed_astar_planner_plugin.xml. The content of the plugin description file (relaxed_astar_planner_plugin.xml), would look like this:

```
<library path="lib/librelaxed_astar_lib">
 <class name="RAstar_planner/RAstarPlannerROS"
 type="RAstar_planner::RAstarPlannerROS"
 base_class_type="nav_core::BaseGlobalPlanner">
   <description>This is RA* global planner plugin by iroboapp project.</↩
        description>
 </class>
</library>
```

In the first line:

```
<library path="lib/librelaxed_astar_lib">
```

we specify the path to the plugin library. In this case, the path is lib/librelaxed_astar_lib, where lib is a folder in the directory ~/catkin_ws/devel/ (see Compilation section above).

```
 <class name="RAstar_planner/RAstarPlannerROS"
 type="RAstar_planner::RAstarPlannerROS"
 base_class_type="nav_core::BaseGlobalPlanner">
```

Here we first specify the name of the global_planner plugin that we will use later in move_base launch file as parameter that specifies the global planner to be used in `nav_core`. It is typically to use the namespace (`RAstar_planner`) followed by a slash then the name of the class (`RAstarPlannerROS`) to specify the name of plugin. If you do not specify the name, then the name will be equal to the type, which is in this case will be `RAstar_planner::RAstarPlannerROS`. It recommended to specify the name to avoid confusion.

The `type` specifies the name of the class that implements the plugin which is in our case `RAstar_planner::RAstarPlannerROS`, and the `base_class_type` specifies the name of the base class that implements the plugin which is in our case `nav_core::BaseGlobalPlanner`.

```
  <description>This is RA* global planner plugin by iroboapp project.</description>
```

The `<description>` tag provides a brief description about the plugin. For a detailed description of plugin description files and their associated tags/attributes please see the documentation in [18].

Why Do We Need This File? We need this file in addition to the code macro to allow the ROS system to automatically discover, load, and reason about plugins. The plugin description file also holds important information, like a description of the plugin, that doesn't fit well in the macro.

Registering Plugin with ROS Package System In order for pluginlib to query all available plugins on a system across all ROS packages, each package must explicitly specify the plugins it exports and which package libraries contain those plugins. A plugin provider must point to its plugin description file in its `package.xml` inside the export tag block. Note, if you have other exports they all must go in the same export field. In our RA* global planner example, the relevant lines would look as follows:

```
<export>
  <nav_core plugin="${prefix}/relaxed_astar_planner_plugin.xml" />
</export>
```

The ${prefix}/ will automatically determine the full path to the file `relaxed_astar_planner_plugin.xml`. For a detailed discussion of exporting a plugin, interested readers may refer to [19].

Important Note: In order for the above export command to work properly, the providing package must depend directly on the package containing the plugin interface, which is *nav_core* in the case of global planner. So, the *relaxed_astar* package must have the line below in its relaxed_astar/package.xml:

```
<build_depend>nav_core</build_depend>
<run_depend>nav_core</run_depend>
```

This will tell the compiler about the dependency on the `nav_core` package.

4.2.1 Querying ROS Package System for Available Plugins

One can query the ROS package system via `rospack` to see which plugins are available by any given package. For example:

```
$ rospack plugins --attrib=plugin nav_core
```

This will return all plugins exported from the `nav_core` package. Here is an example of execution:

```
turtlebot@turtlebot-Inspiron-N5110:~$ rospack plugins --attrib=plugin nav_core
rotate_recovery /opt/ros/hydro/share/rotate_recovery/rotate_plugin.xml
navfn /home/turtlebot/catkin_ws/src/navfn/bgp_plugin.xml
base_local_planner /home/turtlebot/catkin_ws/src/base_local_planner/blp_plugin.↩
     xml
move_slow_and_clear /opt/ros/hydro/share/move_slow_and_clear/recovery_plugin.xml
robot_controller /home/turtlebot/catkin_ws/src/robot_controller/↩
     global_planner_plugin.xml
relaxed_astar /home/turtlebot/catkin_ws/src/relaxed_astar/↩
     relaxed_astar_planner_plugin.xml
dwa_local_planner /opt/ros/hydro/share/dwa_local_planner/blp_plugin.xml
clear_costmap_recovery /opt/ros/hydro/share/clear_costmap_recovery/ccr_plugin.xml
carrot_planner /opt/ros/hydro/share/carrot_planner/bgp_plugin.xml
```

Observe that our plugin is now available under the package `relaxed_astar` and is specified in the file /home/turtlebot/catkin_ws/src/relaxed_astar/relaxed_astar_planner_plugin.xml. You can also observe the other plugins already existing in `nav_core` package, including `carrot_planner/CarrotPlanner` and `navfn`, which implements the Dijkstra algorithm.

Now, your plugin is ready to use.

4.3 Running the Plugin

There are a few steps to follow to run your planner in turtlebot. First, you need to copy the package that contains your global planner (in our case `relaxed_astar`) into the catkin workspace of your Turtlebot (e.g. `catkin_ws`). Then, you need to run `catkin_make` to export your plugin to your turtlebot ROS environment.

Second, you need to make some modification to `move_base` configuration to specify the new planner to be used. For this, follow these steps:

1. In Hydro, go to this folder /opt/ros/hydro/share/turtlebot_navigation/launch/includes

```
$ roscd turtlebot_navigation/
$ cd launch/includes/
```

2. Open the file `move_base.launch.xml` (you may need sudo to open and be able to save) and add the new planner as parameters of the global planner, as follows:

```
......
<node pkg="move_base" type="move_base" respawn="false" name="move_base" ↩
    output="screen">
<param name="base_global_planner" value="RAstar_planner/RAstarPlannerROS"↩
    />
....
```

Save and close the `move_base.launch.xml`. Note that the name of the planner is `RAstar_planner/RAstarPlannerROS` the same specified in `relaxed_astar_planner_plugin.xml`.
Now, you are ready to use your new planner.

3. You must now bringup your turtlebot. You need to launch `minimal.launch`, `3dsensor.launch`, `amcl.launch.xml` and `move_base.launch.xml`. Here is an example of launch file that can be used for this purpose.

```
<launch>
<include file="$(find turtlebot_bringup)/launch/minimal.launch"></include>

  <include file="$(find turtlebot_bringup)/launch/3dsensor.launch">
    <arg name="rgb_processing" value="false" />
    <arg name="depth_registration" value="false" />
    <arg name="depth_processing" value="false" />
    <arg name="scan_topic" value="/scan" />
  </include>

  <arg name="map_file" default="map_folder/your_map_file.yaml"/>
  <node name="map_server" pkg="map_server" type="map_server" args="$(arg ↩
      map_file)" />

  <arg name="initial_pose_x" default="0.0"/>
  <arg name="initial_pose_y" default="0.0"/>
  <arg name="initial_pose_a" default="0.0"/>
  <include file="$(find turtlebot_navigation)/launch/includes/amcl.launch.xml">
    <arg name="initial_pose_x" value="$(arg initial_pose_x)"/>
    <arg name="initial_pose_y" value="$(arg initial_pose_y)"/>
    <arg name="initial_pose_a" value="$(arg initial_pose_a)"/>
  </include>

  <include file="$(find turtlebot_navigation)/launch/includes/move_base.launch.xml↩
      "/>

</launch>
```

Note that changes made in the file `move_base.launch.xml` will now be considered when you bring-up your turtlebot with this launch file.

4.4 *Testing the Planner with RVIZ*

After you bringup your turtlebot, you can launch the rviz using this command (in new terminal).

```
$ roslaunch turtlebot_rviz_launchers view_navigation.launch --screen
```

5 ROS Environment Configuration

One important step before using the planners is tuning the controller parameters as they have a big impact on the performance. The controller parameters can be categorized into several groups based on what they control such as: robot configuration, goal tolerance, forward simulation, trajectory scoring, oscillation prevention, and global plan.

The robot configuration parameters are used to specify the robot acceleration information in addition to the minimum and maximum velocities allowed to the robot. We are working with the Turtlebot robot, and we used the default parameters from $turtlebot\_navigation$ package. The configuration parameters are set as follow: $acc\_lim\_x = 0.5$, $acc\_lim\_theta = 1$, $max\_vel\_x = 0.3$, $min\_vel\_x = 0.1$, $max\_vel\_theta = 1$ $min\_vel\_theta = -1$ $min\_in\_place\_vel\_theta = 0.6$.

The goal tolerance parameters define how close to the goal we can get. $xy\_goal\_tolerance$ represents the tolerance in meters in the x and y distance and should not be less than the map resolution or it will make the robot spin in place indefinitely without reaching the goal, so we set it to 0.1. $yaw\_goal\_tolerance$ represents the tolerance in radians in yaw/rotation. Setting this tolerance very small may cause the robot to oscillate near the goal. We set this parameter very high to 6.26 as we do not care about the robot orientation.

In the forward simulation category, the main parameters are: sim_time, $vx\_samples$, $vtheta\_samples$, and $controller\_frequency$. The sim_time represents the amount of time (in seconds) to forward-simulate trajectories, and we set it to 4.0. The $vx\_samples$ and $vtheta\_samples$ represent the number of samples to use when exploring the x velocity space and the theta velocity space respectively. They should be set depending on the processing power available, and we use the value recommended in ROS web-site for them. So, we set 8 to the $vx\_samples$ and 20 to $vtheta\_samples$. The $controller\_frequency$ represents the frequency at which this controller will be called. Setting this parameter a value too high can overload the CPU. Setting it to 2 work fine with our planners.

The trajectory scoring parameters are used to evaluate the possible velocities to the local planner. The three main parameters on this category are: $pdist\_scale$, $gdist\_scale$, and $occdist\_scale$. The $pdist\_scale$ represents the weight for how much the controller should stay close to the global planner path. The $gdist\_scale$ represents the weight for how much the controller should attempt to reach its goal by whatever path necessary. Increasing this parameter will give the local planner more freedom in choosing its path away from the global path. The $occdist\_scale$ represents the weight for how much the controller should attempt to avoid obstacles. Because the planners may generate paths very close to the obstacles or dynamically not feasible, we set the $pdist\_scale = 0.1$, $gdist\_scale = 0.8$ and $occdist\_scale = 0.3$ when testing our planners. Another parameter named dwa is used to specify whether to use the DWA when setting it to $true$, or use the Trajectory Rollout when setting it to $false$. We set it to $true$ because the DWA is computationally less expensive than the Trajectory Rollout.

6 Experimental Validation

For the experimental study using ROS, we have chosen the realistic Willow Garage map (Fig. 3), with dimensions 584 * 526 cells and a resolution 0.1 m/cel. In the map, the white color represents the free area, the black color represents the obstacles, and the grey color represents unknown area.

Three performance metrics are considered to evaluate the global planners: (1) *the path length*, it represents the length of the shortest global path found by the planner, (2) *the steps*, it is the number of the steps in the generated paths. (3) *the execution time*, which represents the amount of time that the planner spend to find its best path.

To evaluate the planners, we consider two tours each with 10 random points. Figure 3 shows the points for one of the tours, where the red circle represents the start location, the blue circle is the goal location, and the green circles are the intermediate waypoints. We run each planner 30 times for each tour. Figures 4, 5, 6 and 7 show the global plans for that tour generated by the RA*, ROS-A*, and ROS-Dijkstra for grid path and gradient descent method respectively.

Tables 1 and 2 shows the path length and the execution time of the planners for the complete tour.

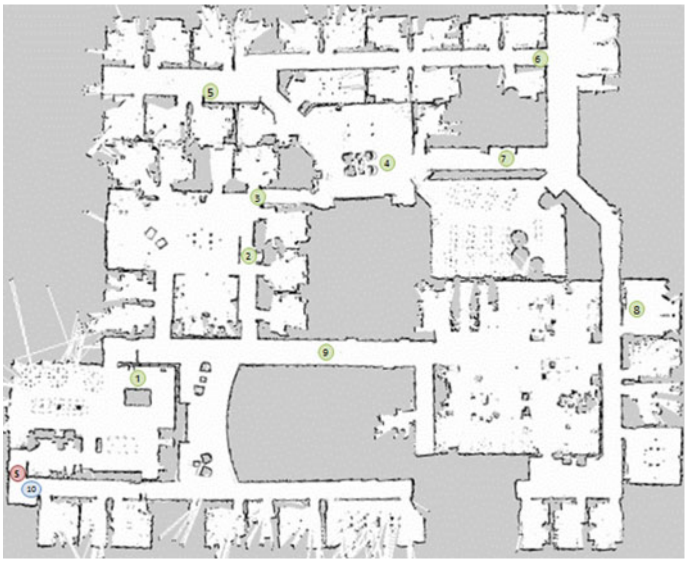

Fig. 3 Willow Garage map

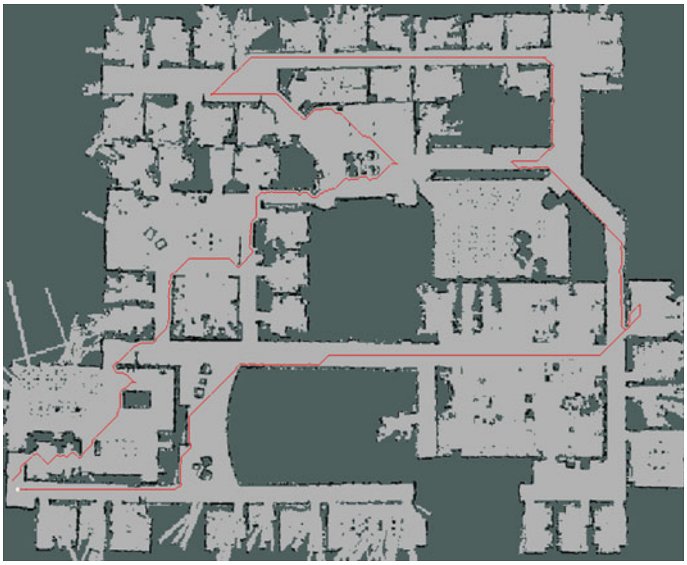

Fig. 4 RA* planner

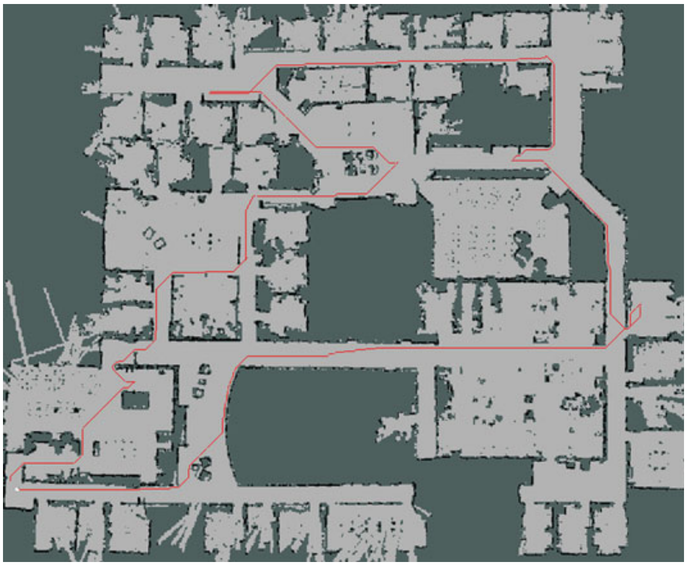

Fig. 5 ROS A* (grid path)

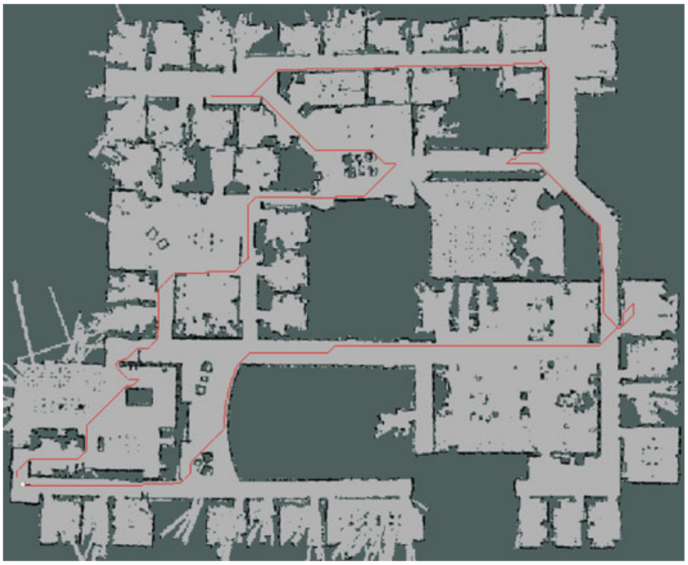

Fig. 6 ROS Dijkstra (grid path)

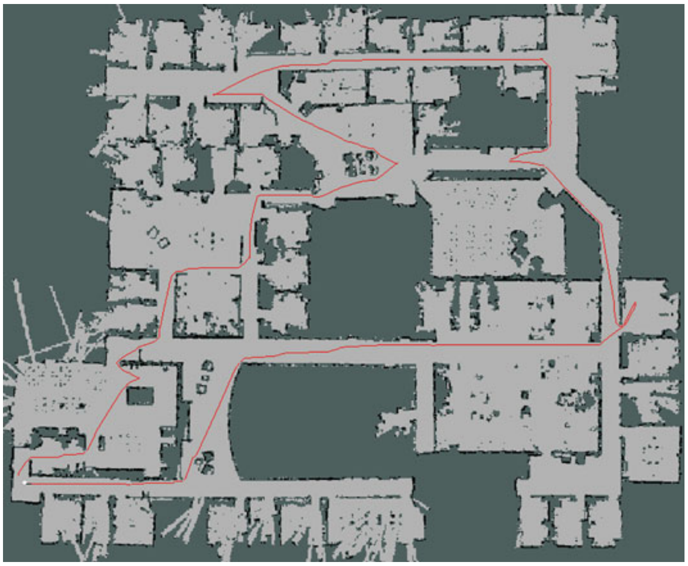

Fig. 7 ROS Dijkstra (gradient descent)

Table 1 Execution time in (microseconds) and path length in (meters) for tour 1

Planner	Execution time		Path length	Number of steps
	Total	Average		
RA* (grid path)	34.7532	3.4214 ± 0.4571	199.6486	1765
A* (grid path)	163.0028	14.4820 ± 1.1104	192.7875	1771
Dijkstra (grid path)	189.9566	19.0443 ± 1.2455	193.5490	1783
Dijkstra (gradient descent)	211.7877	19.1116 ± 1.1714	187.7220	3485

Table 2 Execution time in (microseconds) and path length in (meters) for tour 2

Planner	Execution time		Path length	Number of steps
	Total	Average		
RA* (grid path)	137.2426	13.7243 ± 0.7017	291.9332	2465
A* (grid path)	198.8332	19.8833 ± 1.2563	280.0695	2551
Dijkstra (grid path)	216.1717	21.6172 ± 1.2391	281.4142	2568
Dijkstra (gradient descent)	218.4983	21.8498 ± 1.2261	275.3235	4805

The ROS-Dijkstra with gradient descent method generates paths with more steps but little shorter in length than A* paths, this is because it has smaller granularity (more fine-grain exploration), so it has more freedom in the movements and more able to take smaller steps.

Other planners work at cell level (grid path), so they consider each cell as a single point, and they have to pass the whole cell in each step. As the resolution of the map

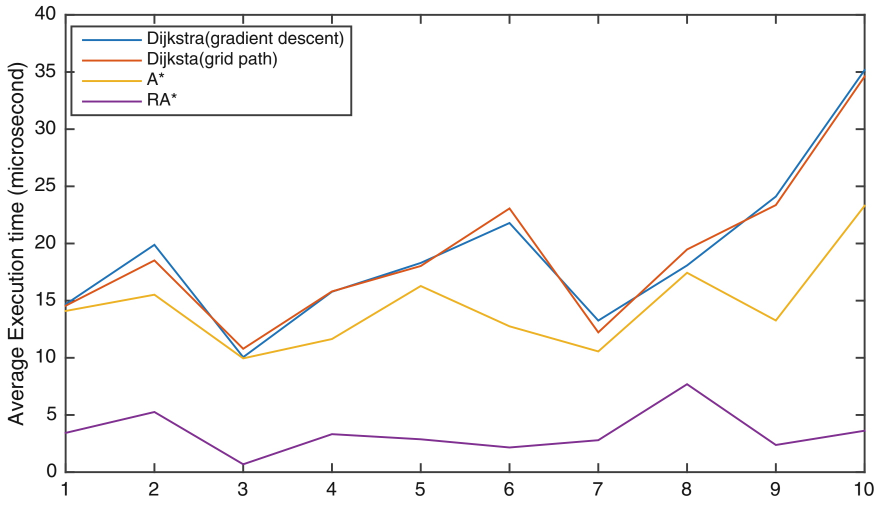

Fig. 8 Detailed execution time for tour 1

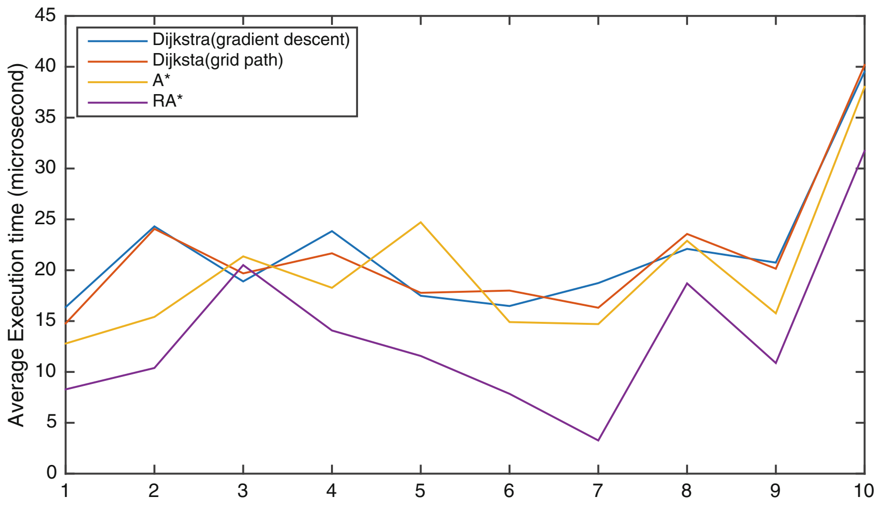

Fig. 9 Detailed execution time for tour 2

is 10 cm, so the distance in each step is at least 10 cm. Comparing the execution time, the RA*, extremely superior the other planners in all simulated cases. Using RA*, the execution time was reduced by more than 78 % from the ROS-A*. Figures 8 and 9 shows the average execution time for 30 run between each two points from the tour.

Acknowledgments This work is supported by the iroboapp project "Design and Analysis of Intelligent Algorithms for Robotic Problems and Applications" under the grant of the National Plan for Sciences, Technology and Innovation (NPSTI), managed by the Science and Technology Unit of Al-Imam Mohamed bin Saud University and by King AbdulAziz Center for Science and Technology (KACST). This work is also supported by the myBot project entitled "MyBot: A Personal Assistant Robot Case Study for Elderly People Care" under the grant from King AbdulAziz City for Science and Technology (KACST).

References

1. Relaxed A*. https://github.com/coins-lab/relaxed_astar (2014)
2. Robot Operating System (ROS). http://www.ros.org
3. K. Wyrobek, E. Berger, H. Van der Loos, J. Salisbury, Towards a personal robotics development platform: rationale and design of an intrinsically safe personal robot, in *IEEE International Conference on Robotics and Automation, ICRA 2008* (IEEE, 2008), pp. 2165–2170
4. M. Quigley, E. Berger, A.Y. Ng, STAIR: hardware and software architecture, in *AAAI, Robotics Workshop* (Vancouver, BC 2007), pp. 31–37
5. E. Marder-Eppstein, E. Berger, T. Foote, B. Gerkey, K. Konolige, The office marathon: robust navigation in an indoor office environment, in *2010 IEEE International Conference on Robotics and Automation (ICRA)*, May 2010, pp. 300–307
6. P. Goebel, *ROS By Example*, edited by. Lulu. http://www.lulu.com/shop/r-patrick-goebel/ros-by-example-hydro-volume-1/ebook/product-21393108.html (2013)

7. OpenSLAM. https://openslam.org/
8. G. Grisetti, C. Stachniss, W. Burgard, Improved techniques for grid mapping with rao-blackwellized particle filters. IEEE Trans. Robot. **23**(1), 34–46 (2007)
9. S. Thrun, D. Fox, W. Burgard, F. Dellaert, Robust Monte Carlo localization for mobile robots. Artif. Intell. **128**(1–2), 99–141 (2000)
10. Adding a global path planner as plugin in ROS. http://www.iroboapp.org/index.php?title=Adding_A_Global_Path_Planner_As_Plugin_in_ROS
11. Writing a global path planner as plugin in ROS. http://wiki.ros.org/navigation/Tutorials/Writing%20A%20Global%20Path%20Planner%20As%20Plugin%20in%20ROS
12. E.W. Dijkstra, A note on two problems in connexion with graphs. Numerische Mathematik **1**(1), 269–271 (1959)
13. B.P. Gerkey, K. Konolige, Planning and control in unstructured terrain, in *Workshop on Path Planning on Costmaps, Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)* (2008)
14. D. Fox, W. Burgard, S. Thrun, The dynamic window approach to collision avoidance. IEEE Robot. Autom. Mag. **4**(1), 23–33 (1997)
15. Carrot planner header. http://docs.ros.org/hydro/api/carrot_planner/html/carrot_planner_8h_source.html
16. Carrot planner source. http://docs.ros.org/hydro/api/carrot_planner/html/carrot_planner_8cpp_source.html
17. Pluginlib package. http://wiki.ros.org/pluginlib
18. Plugin description file. http://wiki.ros.org/pluginlib/PluginDescriptionFile
19. Plugin export. http://wiki.ros.org/pluginlib/PluginExport