1. Struttura del codice

ROS è strutturato in nodi. All'interno di un'applicazione vengono eseguiti più nodi, indipendenti l'uno dall'altro, che comunicano fra loro tramite messages, topics e services(…spiegare…).

Per quanto riguarda la nostra applicazione useremo il package bosch-ros-pkg, che ci offre una buona base per i nostri test e i nostri sviluppi.

L'esplorazione degli ambienti sfrutterà quindi nodi dei seguenti package:

- explore
- stage (stageros)
- gmapping
- move_base
- tf

Nel nostro caso particolare, la struttura del codice è la seguente.

Il nostro package (bosh-ros-pkg/trunk/stacks/exploration/explore_stage) contiene i seguenti file e directory:

- config/
- explore/
- explore.launch
- explore.vcg
- explore.xml
- explore_slam.launch
- explore_slam.xml
- manifest.xml
- move.xml

Il file manifest.xml descrive le dipendenze del nostro package da package esterni ad esso.

Per lanciare l'applicazione richiamiamo il file explore_slam.launch, ce si trova all'interno del path "/trunk/stacks/bosch_demos/explore_stage". Questo file permette di lanciare tutti insieme i diversi nodi che fanno parte della nostra applicazione e di collegarli tra loro, di specificare dei parametri e di includere file xml, che a loro volta possono includere file di configurazione yaml.

Ad esempio, il file explore_slam.launch include i file explore_slam.xml e move.xml.

Il file explore_slam.xml include e dichiara i seguenti file e parametri:

- file con scope limitato al node-package "explore":
  - (file) config/footprint.yaml
  - (file) config/costmap_common.yaml
  - (file) explore/explore_costmap.yaml

- altri parametri, anch'essi con scope limitato al node-package "explore":
  - (parametro) potential_scale

- (parametro) orientation_scale
- (parametro) gain_scale
- (parametro) close_loops

- <remap from="slam_entropy" to="gmapping/entropy"/>

Il file move.xml include i seguenti file con scope limitato al node-package “move_base”:

- config/footprint.yaml
- config/costmap_common.yaml (namespace = “global_costmap”)
- explore/global_costmap.yaml
- explore/navfn_params.yaml
- config/costmap_common.yaml(namespace = “local_costmap”)
- explore/local_costmap.yaml
- explore/trajectory_planner_params.yaml

Parametri

Parametri dichiarati nel file config/ footprint.yaml:

```
footprint: [[-0.2825, -0.395], [0.2825, -0.395], [0.2825, -0.2], [0.4025, -0.2],
[0.4025, 0.2], [0.2825, 0.2], [0.2825, 0.395], [-0.2825, 0.395]]
```

Parametri dichiarati nel file config/costmap_common.yaml:

```
map_type: costmap
transform_tolerance: 0.5
obstacle_range: 2.5
max_obstacle_height: 2.0
raytrace_range: 3.0
inscribed_radius: 0.385
circumscribed_radius: 0.685
inflation_radius: 0.6
cost_scaling_factor: 15.0
lethal_cost_threshold: 100

observation_sources: base_scan

base_scan: {
sensor_frame: base_laser_link,
data_type: LaserScan,
expected_update_rate: 0.2,
observation_persistance: 0.0,
marking: true,
clearing: true
}
```

Parametri dichiarati nel file explore/explore_costmap.yaml:

```
explore_costmap:
  global_frame: /map
  robot_base_frame: base_link
```

```
  update_frequency: 1.0
  publish_frequency: 0.0
  raytrace_range: 5.0
  obstacle_range: 5.0
  static_map: false
  rolling_window: false
  width: 60.0
  height: 60.0
  resolution: 0.2
  origin_x: -30.0
  origin_y: -30.0
  track_unknown_space: true
```

Parametri dichiarati nel file explore/global_costmap.yaml:

```
#Independent settings for the planner's costmap
global_costmap:
  global_frame: map
  robot_base_frame: base_link
  update_frequency: 1.0
  publish_frequency: 0.5
  raytrace_range: 30.0
  obstacle_range: 5.0
  static_map: false
  rolling_window: false
  width: 60.0
  height: 60.0
  resolution: 0.05
  origin_x: -30.0
  origin_y: -30.0
```

Parametri dichiarati nel file explore/local_costmap.yaml:

```
#Independent settings for the local planner's costmap
local_costmap:
  global_frame: odom
  robot_base_frame: base_link
  update_frequency: 1.0
  publish_frequency: 0.5
  static_map: false
  rolling_window: true
  width: 3.0
  height: 5.0
  resolution: 0.05
  origin_x: 0.0
  origin_y: 0.0
```

Parametri dichiarati nel file explore/navfn_params.yaml:

```
NavfnROS:
  transform_tolerance: 0.3
```

Parametri dichiarati nel file explore/trajectory_planner_params.yaml:

```
TrajectoryPlannerROS:
  transform_tolerance: 0.3
  world_model: costmap
  sim_time: 1.7
  sim_granularity: 0.025
  dwa: true
  vx_samples: 3
  vtheta_samples: 20
  max_vel_x: 0.65
  min_vel_x: 0.1
  max_vel_th: 1.0
  min_vel_th: -1.0
  min_in_place_vel_th: 0.4
  xy_goal_tolerance: 0.1
  yaw_goal_tolerance: 0.02
  goal_distance_bias: 0.8
  path_distance_bias: 0.6
  occdist_scale: 0.01
  heading_lookahead: 0.325
  oscillation_reset_dist: 0.05
  escape_reset_dist: 0.15
  escape_reset_theta: 0.30
  acc_lim_th: 1.0
  acc_lim_x: 0.5
  acc_lim_y: 0.0
  heading_scoring: false
  heading_scoring_timestep: 0.8
  holonomic_robot: false
  simple_attractor: false
```

Come utilizzare i parametri all'interno dell'applicazione

```
private_nh.param("planner_frequency", planner_frequency_, 1.0);
```

oppure

```
ros::NodeHandle nh;
std::string global_name, relative_name, default_param;
if (nh.getParam("/global_name", global_name))
{
  ...
}

if (nh.getParam("relative_name", relative_name))
{
...
}

// Default value version
nh.param<std::string>("default_param", default_param, "default_value");
```

Cuore dell'applicazione

Quella che è per noi la parte di maggior interesse dell'applicazione, cioè quella che coinvolge direttamente la strategia di esplorazione, è tutta racchiusa nei seguenti file cpp:

- bosh-ros-pkg/trunk/stacks/exploration/explore/src/explore.cpp
- bosh-ros-pkg/trunk/stacks/exploration/explore/src/explore_frontier.cpp

In particolare, il primo file è importante per discriminare tra percezione/decisione continua/discreta, mentre il secondo è importante per quanto riguarda la scelta dei goal sulla frontiera e la scelta del prossimo goal da raggiungere.

2. Modifiche da apportare al codice

2.1. Diverse strategie di esplorazione

Nuovo parametro per scegliere la strategia di esplorazione:

*exploration_strategy_*: [0, 1, 2, …].

Dove:

- 0 equivale RANDOM
- 1 equivale a WEIGHTED (pesata)
- 2 equivale a MCDM

Aggiungere questo parametro sia nei file di configurazione legati al package explore, sia nel codice sorgente dell'applicazione, più precisamente nel file explore.cpp

Nuove funzioni per le diverse strategie di esplorazione:

Bisognerà aggiungere una nuova funzione per ogni strategia di esplorazione che vogliamo prendere in considerazione e collegarla all'attuale chiamata di funzione per la scelta del prossimo goal (getExplorationGoals() ?), ma anche, ovviamente, collegarla al parametro relativo alla strategia di esplorazione.

Esempi di nomi di funzioni utilizzabili per le sopra citate strategie di esplorazione sono:

- getExplorationGoals_RANDOM()
- getExplorationGoals_WEIGHTED()
- getExplorationGoals_MCDM()

Queste nuove funzioni dovranno essere aggiunte nel file explore_frontier.cpp (valutare se aggiungere i riferimenti anche nel file explore_frontier.h e se aggiungere qualche altro riferimento nei file explore.h ed explore.cpp)

2.2. Diverse combinazioni delle modalità (continua e discreta) di percezione e decisione

Parametro per scegliere le diverse combinazioni delle modalità di esplorazione:

Valutare se aggiungere un nuovo parametro o se sfruttare i seguenti parametri (già esistenti):

- explore_stage/explore/explore_costmap.yaml: *update_frequency*
  frequenza con la quale il robot aggiorna la mappa dell'ambiente
- explore_stage/explore_slam.xml: *planner_frequency* (da aggiungere al file xml)
  frequenza con la quale il robot calcola il nuovo goal da raggiungere

Nel primo caso, è possibile utilizzare un unico nuovo parametro che ci permette di scegliere tra le diverse combinazioni (a, b, d):

decision_perception_discretization: [0, 1, 2].

Dove:

- 0 equivale a percezione continua, decisione continua (caso a.)
- 1 equivale a percezione continua, decisione discreta (caso b.)
- 2 equivale a percezione discreta, decisione discreta (caso d.)

In questo primo caso bisognerà definire il nuovo parametro nel file explore_slam.xml e poi riutilizzarlo nel file explore.cpp.

Bisognerà quindi modificare gli attuali metodi che pianificano l'esplorazione e che costruiscono la mappa, in modo tale che i comportino coerentemente con il parametro appena visto.

Lo stesso vale per il secondo caso, qui sotto, con l'unica differenza che i parametri da tenere in considerazione saranno due, separati fra loro, e che questi parametri sono già definiti.

Nel secondo caso invece sarà necessario assegnare il valore "-1" ai due parametri per ottenere le diverse combinazioni, nei seguenti modi:

- *planner_frequency* != -1 e *update_frequency*!= -1 (caso a.)
- *planner_frequency* != -1 e *update_frequency* = -1 (caso b.)
- *planner_frequency* =-1 e *update_frequency* = -1 (caso d.)

E successivamente fare in modo che l'applicazione si comporti nel modo desiderato, collegando alla scelta di tali parametri le scelte nel codice dell'applicazione.

Controllare che il goal sia stato raggiunto, prima di sceglierne uno nuovo:

- nuova variabile globale: *chosenGoal_pose*

mantiene salvato il goal scelto, per poter calcolare la distanza tra di esso e il robot.

- nuova funzione: *chosenGoal_reached ()*

calcola la distanza tra il robot e il goal scelto; se essa è inferiore a una determinata soglia, allora considera il nodo come raggiunto e consente di sceglierne uno nuovo.

Questi elementi vanno aggiunti nel file explore.cpp e nel file explore.h.

2.3. Nuovo nodo in ascolto

Valutare se creare un nuovo nodo per raccogliere e visualizzare i dati relativi alle performance di esplorazione, o se invece effettuare le modifiche direttamente all'interno del nodo explore.

3. Modifiche apportate al codice

Le modifiche sono segnalate in rosso nei blocchi di codice.

File bosch-ros-pkg/trunk/stacks/exploration/explore_stage/explore_slam.xml

```
<launch>
  <node pkg="explore" type="explore" respawn="false" name="explore" output="screen" >
    <rosparam file="$(find explore_stage)/config/footprint.yaml" command="load" />

    <rosparam file="$(find explore_stage)/config/costmap_common.yaml" command="load"
ns="explore_costmap" />
    <rosparam file="$(find explore_stage)/explore/explore_costmap.yaml" command="load" />

    <param name="planner_frequency" value="1.0"/>
    <param name=”decision_perception_discretization” value=”0”/>
    <param name="exploration_strategy" value="0"/>
    <param name="potential_scale" value="0.005"/>
    <param name="orientation_scale" value="0.0"/>
    <param name="gain_scale" value="1.0"/>
    <param name="close_loops" value="true"/>
    <remap from="slam_entropy" to="gmapping/entropy"/>
  </node>
</launch>
```

File: bosh-ros-pkg/trunk/stacks/exploration/explore/include/explore/explore.h

```
#ifndef NAV_EXPLORE_H_
#define NAV_EXPLORE_H_

#include <ros/ros.h>
#include <move_base_msgs/MoveBaseAction.h>
#include <actionlib/client/simple_action_client.h>
#include <geometry_msgs/PoseStamped.h>
#include <nav_msgs/GetMap.h>
#include <costmap_2d/costmap_2d_ros.h>
#include <costmap_2d/costmap_2d.h>
#include <navfn/navfn_ros.h>
#include <explore/explore_frontier.h>
#include <explore/loop_closure.h>
#include <visualization_msgs/MarkerArray.h>
#include <vector>
#include <string>
#include <boost/thread/mutex.hpp>


namespace explore {

/**
 * @class Explore
 * @brief A class adhering to the robot_actions::Action interface that moves the robot base to
explore its environment.
 */
class Explore {
public:
  /**
   * @brief  Constructor
   * @return
   */
  Explore();

  /**
   * @brief  Destructor - Cleans up
   */
  virtual ~Explore();

  /**
   * @brief  Runs whenever a new goal is sent to the move_base
   * @param goal The goal to pursue
```

```
   * @param feedback Feedback that the action gives to a higher-level monitor, in this case, the
position of the robot
   * @return The result of the execution, ie: Success, Preempted, Aborted, etc.
   */
//  virtual robot_actions::ResultStatus execute(const ExploreGoal& goal, ExploreFeedback& feedback);
  void execute();

  void spin();

private:
  /**
   * @brief  Make a global plan
   */
  void makePlan();

  /**
   * @brief  Publish a goal to the visualizer
   * @param  goal The goal to visualize
   */
  void publishGoal(const geometry_msgs::Pose& goal);

  /**
   * @brief publish map
   */
  void publishMap();

  void reachedGoal(const actionlib::SimpleClientGoalState& status, const
move_base_msgs::MoveBaseResultConstPtr& result, geometry_msgs::PoseStamped frontier_goal);

  /**
   * @brief  Resets the costmaps to the static map outside a given window
   */
//  void resetCostmaps(double size_x, double size_y);

  bool mapCallback(nav_msgs::GetMap::Request  &req, nav_msgs::GetMap::Response &res);

  bool goalOnBlacklist(const geometry_msgs::PoseStamped& goal);

  ros::NodeHandle node_;
  tf::TransformListener tf_;
  costmap_2d::Costmap2DROS* explore_costmap_ros_;

  actionlib::SimpleActionClient<move_base_msgs::MoveBaseAction> move_base_client_;

  navfn::NavfnROS* planner_;
  std::string robot_base_frame_;
  bool done_exploring_;

  ros::Publisher marker_publisher_;
  ros::Publisher marker_array_publisher_;
  ros::Publisher map_publisher_;
  ros::ServiceServer map_server_;

  ExploreFrontier* explorer_;

  tf::Stamped<tf::Pose> global_pose_;
  double planner_frequency_;
  int    visualize_;
  LoopClosure* loop_closure_;
  std::vector<geometry_msgs::PoseStamped> frontier_blacklist_;
  geometry_msgs::PoseStamped prev_goal_;
  unsigned int prev_plan_size_;
  double time_since_progress_, progress_timeout_;
  double potential_scale_, orientation_scale_, gain_scale_;
  boost::mutex client_mutex_;
  bool   close_loops_;

  //:::dt:::
  int decision_perception_discretization_;
  int exploration_strategy_;
  geometry_msgs::PoseStamped chosenGoal_pose_;

};

}

#endif
```

File: bosh-ros-pkg/trunk/stacks/exploration/explore/src/explore.cpp

```
.........
.........


#include <explore/explore.h>
#include <explore/explore_frontier.h>

#include <boost/thread.hpp>
#include <boost/bind.hpp>

using namespace costmap_2d;
using namespace navfn;
using namespace visualization_msgs;
using namespace geometry_msgs;

namespace explore {

double sign(double x){
  return x < 0.0 ? -1.0 : 1.0;
}

Explore::Explore() :
  node_(),
  tf_(ros::Duration(10.0)),
  explore_costmap_ros_(NULL),
  move_base_client_("move_base"),
  planner_(NULL),
  done_exploring_(false),
  explorer_(NULL),
  prev_plan_size_(0)
{
  ros::NodeHandle private_nh("~");

  marker_publisher_ = node_.advertise<Marker>("visualization_marker",10);
  marker_array_publisher_ = node_.advertise<MarkerArray>("visualization_marker_array",10);
  map_publisher_ = private_nh.advertise<nav_msgs::OccupancyGrid>("map", 1, true);
  map_server_ = private_nh.advertiseService("explore_map", &Explore::mapCallback, this);

  private_nh.param("navfn/robot_base_frame", robot_base_frame_, std::string("base_link"));
  private_nh.param("planner_frequency", planner_frequency_, 1.0);
  private_nh.param("progress_timeout", progress_timeout_, 30.0);
  private_nh.param("visualize", visualize_, 1);
  double loop_closure_addition_dist_min;
  double loop_closure_loop_dist_min;
  double loop_closure_loop_dist_max;
  double loop_closure_slam_entropy_max;
  private_nh.param("close_loops", close_loops_, false); // TODO: switch default to true once
gmapping 1.1 has been released
  private_nh.param("loop_closure_addition_dist_min", loop_closure_addition_dist_min, 2.5);
  private_nh.param("loop_closure_loop_dist_min", loop_closure_loop_dist_min, 6.0);
  private_nh.param("loop_closure_loop_dist_max", loop_closure_loop_dist_max, 20.0);
  private_nh.param("loop_closure_slam_entropy_max", loop_closure_slam_entropy_max, 3.0);
  private_nh.param("potential_scale", potential_scale_, 1e-3);
  private_nh.param("orientation_scale", orientation_scale_, 0.0); // TODO: set this back to 0.318
once getOrientationChange is fixed
  private_nh.param("gain_scale", gain_scale_, 1.0);

  //:::dt:::
  private_nh.param("decision_perception_discretization", decision_perception_discretization_, 0);
  private_nh.param("exploration_strategy", exploration_strategy_, 0);

  explore_costmap_ros_ = new Costmap2DROS(std::string("explore_costmap"), tf_);
  explore_costmap_ros_->clearRobotFootprint();

  planner_ = new navfn::NavfnROS(std::string("explore_planner"), explore_costmap_ros_);
  explorer_ = new ExploreFrontier();
  loop_closure_ = new LoopClosure(loop_closure_addition_dist_min,
        loop_closure_loop_dist_min,
        loop_closure_loop_dist_max,
        loop_closure_slam_entropy_max,
        planner_frequency_,
```

```
        move_base_client_,
        *explore_costmap_ros_,
        client_mutex_);
}

Explore::~Explore() {
  if(loop_closure_ != NULL)
    delete loop_closure_;

  if(planner_ != NULL)
    delete planner_;

  if(explorer_ != NULL)
    delete explorer_;

  if(explore_costmap_ros_ != NULL)
    delete explore_costmap_ros_;
}

bool Explore::mapCallback(nav_msgs::GetMap::Request  &req,
                          nav_msgs::GetMap::Response &res)
{
  ROS_DEBUG("mapCallback");
  Costmap2D explore_costmap;
  explore_costmap_ros_->getCostmapCopy(explore_costmap);

  res.map.info.width = explore_costmap.getSizeInCellsX();
  res.map.info.height = explore_costmap.getSizeInCellsY();
  res.map.info.resolution = explore_costmap.getResolution();
  res.map.info.origin.position.x = explore_costmap.getOriginX();
  res.map.info.origin.position.y = explore_costmap.getOriginY();
  res.map.info.origin.position.z = 0;
  res.map.info.origin.orientation.x = 0;
  res.map.info.origin.orientation.y = 0;
  res.map.info.origin.orientation.z = 0;
  res.map.info.origin.orientation.w = 1;

  int size = res.map.info.width * res.map.info.height;
  const unsigned char* map = explore_costmap.getCharMap();

  res.map.set_data_size(size);
  for (int i=0; i<size; i++) {
    if (map[i] == NO_INFORMATION)
      res.map.data[i] = -1;
    else if (map[i] == LETHAL_OBSTACLE)
      res.map.data[i] = 100;
    else
      res.map.data[i] = 0;
  }

  return true;
}

void Explore::publishMap() {
  nav_msgs::OccupancyGrid map;
  map.header.stamp = ros::Time::now();

  Costmap2D explore_costmap;
  explore_costmap_ros_->getCostmapCopy(explore_costmap);

  map.info.width = explore_costmap.getSizeInCellsX();
  map.info.height = explore_costmap.getSizeInCellsY();
  map.info.resolution = explore_costmap.getResolution();
  map.info.origin.position.x = explore_costmap.getOriginX();
  map.info.origin.position.y = explore_costmap.getOriginY();
  map.info.origin.position.z = 0;
  map.info.origin.orientation.x = 0;
  map.info.origin.orientation.y = 0;
  map.info.origin.orientation.z = 0;
  map.info.origin.orientation.w = 1;

  int size = map.info.width * map.info.height;
  const unsigned char* char_map = explore_costmap.getCharMap();

  map.set_data_size(size);
  for (int i=0; i<size; i++) {
    if (char_map[i] == NO_INFORMATION)
      map.data[i] = -1;
```

```
    else if (char_map[i] == LETHAL_OBSTACLE)
      map.data[i] = 100;
    else
      map.data[i] = 0;
  }

  map_publisher_.publish(map);
}

void Explore::publishGoal(const geometry_msgs::Pose& goal){
  visualization_msgs::Marker marker;
  marker.header.frame_id = "map";
  marker.header.stamp = ros::Time::now();
  marker.ns = "explore_goal";
  marker.id = 0;
  marker.type = visualization_msgs::Marker::ARROW;
  marker.pose = goal;
  marker.scale.x = 0.5;
  marker.scale.y = 1.0;
  marker.scale.z = 1.0;
  marker.color.a = 1.0;
  marker.color.r = 0.0;
  marker.color.g = 1.0;
  marker.color.b = 0.0;
  marker.lifetime = ros::Duration(5);
  marker_publisher_.publish(marker);
}

void Explore::makePlan() {
  ROS_INFO(":::tesi::: inizio makePlan()");
  //since this gets called on handle activate
  if(explore_costmap_ros_ == NULL)
    return;

  tf::Stamped<tf::Pose> robot_pose;
  explore_costmap_ros_->getRobotPose(robot_pose);

  std::vector<geometry_msgs::Pose> goals;
  explore_costmap_ros_->clearRobotFootprint();
  explorer_->getExplorationGoals(*explore_costmap_ros_, robot_pose, planner_, goals,
potential_scale_, orientation_scale_, gain_scale_);
  if (goals.size() == 0)
    done_exploring_ = true;

  bool valid_plan = false;
  std::vector<geometry_msgs::PoseStamped> plan;
  PoseStamped goal_pose, robot_pose_msg;
  tf::poseStampedTFToMsg(robot_pose, robot_pose_msg);

  goal_pose.header.frame_id = explore_costmap_ros_->getGlobalFrameID();
  goal_pose.header.stamp = ros::Time::now();
  planner_->computePotential(robot_pose_msg.pose.position); // just to be safe, though this should
already have been done in explorer_->getExplorationGoals
  int blacklist_count = 0;
  for (unsigned int i=0; i<goals.size(); i++) {
    goal_pose.pose = goals[i];

    //:::dt:::
    ROS_INFO(":::tesi::: goals[i] : goals.size()= %d", goals.size());
    ROS_INFO("goals[%d] : x=%f - y=%f -
z=%f",(int)i,(float)goal_pose.pose.position.x,(float)goal_pose.pose.position.y,(float)goal_pose.pose
.position.z);

    if (goalOnBlacklist(goal_pose)) {
      blacklist_count++;
      continue;
    }

    valid_plan = ((planner_->getPlanFromPotential(goal_pose, plan)) && (!plan.empty()));
    if (valid_plan) {
      //:::tesi::: modifiche per tesi
      chosenGoal_pose_.header.frame_id = goal_pose.header.frame_id;
      chosenGoal_pose_.header.stamp = goal_pose.header.stamp;
      chosenGoal_pose_.pose = goal_pose.pose;
      ROS_INFO("chosenGoal_pose_ : x=%f -
y=%f",(float)chosenGoal_pose_.pose.position.x,(float)chosenGoal_pose_.pose.position.y);

      break;
```

```
    }
  }

  // publish visualization markers
  if (visualize_) {
    std::vector<Marker> markers;
    explorer_->getVisualizationMarkers(markers);
    for (uint i=0; i < markers.size(); i++)
      marker_publisher_.publish(markers[i]);
  }

  if (valid_plan) {
    if (prev_plan_size_ != plan.size()) {
      time_since_progress_ = 0.0;
    } else {
      double dx = prev_goal_.pose.position.x - goal_pose.pose.position.x;
      double dy = prev_goal_.pose.position.y - goal_pose.pose.position.y;
      double dist = sqrt(dx*dx+dy*dy);
      if (dist < 0.01) {
        time_since_progress_ += 1.0f / planner_frequency_;
      }
    }

    ROS_INFO(":::tesi::: planner_frequency_ =  %f", planner_frequency_);

    // black list goals for which we've made no progress for a long time
    if (time_since_progress_ > progress_timeout_) {
      frontier_blacklist_.push_back(goal_pose);
      ROS_DEBUG("Adding current goal to black list");
    }

    prev_plan_size_ = plan.size();
    prev_goal_ = goal_pose;

    move_base_msgs::MoveBaseGoal goal;
    goal.target_pose = goal_pose;
    move_base_client_.sendGoal(goal, boost::bind(&Explore::reachedGoal, this, _1, _2, goal_pose));

    if (visualize_) {
      publishGoal(goal_pose.pose);
      publishMap();
    }
  } else {
    ROS_WARN("Done exploring with %d goals left that could not be reached. There are %d goals on our
blacklist, and %d of the frontier goals are too close to them to pursue. The rest had global
planning fail to them. \n", (int)goals.size(), (int)frontier_blacklist_.size(), blacklist_count);
    ROS_INFO("Exploration finished. Hooray.");
    done_exploring_ = true;
  }
  ROS_INFO(":::tesi::: fine makePlan()");

}

bool Explore::goalOnBlacklist(const geometry_msgs::PoseStamped& goal){
  //check if a goal is on the blacklist for goals that we're pursuing
  for(unsigned int i = 0; i < frontier_blacklist_.size(); ++i){
    double x_diff = fabs(goal.pose.position.x - frontier_blacklist_[i].pose.position.x);
    double y_diff = fabs(goal.pose.position.y - frontier_blacklist_[i].pose.position.y);

    if(x_diff < 2 * explore_costmap_ros_->getResolution() && y_diff < 2 * explore_costmap_ros_-
>getResolution())
      return true;
  }
  return false;
}

void Explore::reachedGoal(const actionlib::SimpleClientGoalState& status,
    const move_base_msgs::MoveBaseResultConstPtr& result, geometry_msgs::PoseStamped frontier_goal){

  ROS_DEBUG("Reached goal");
  if(status == actionlib::SimpleClientGoalState::ABORTED){
    frontier_blacklist_.push_back(frontier_goal);
    ROS_DEBUG("Adding current goal to black list");
  }

//  if(!done_exploring_){
//    //create a plan from the frontiers left and send a new goal to move_base
//    makePlan();
```

```
//  }
//  else{
//    ROS_INFO("Exploration finished. Hooray.");
//  }
}

//:::tesi::: modifiche per tesi
bool chosenGoal_reached(PoseStamped robot_pose_msg, PoseStamped selected_goal_pose, float
map_resolution){

  ROS_INFO("robot_pose_msg : x=%f -
y=%f",(float)robot_pose_msg.pose.position.x,(float)robot_pose_msg.pose.position.y);
  ROS_INFO("chosenGoal_pose : x=%f -
y=%f",(float)selected_goal_pose.pose.position.x,(float)selected_goal_pose.pose.position.y);
  ROS_INFO("map_resolution : %f",(float)map_resolution);

  float distance_x = (float)robot_pose_msg.pose.position.x -
(float)selected_goal_pose.pose.position.x;
  float distance_y = (float)robot_pose_msg.pose.position.y -
(float)selected_goal_pose.pose.position.y;
  float distance_from_goal = sqrt((distance_x * distance_x) + (distance_y * distance_y));
  ROS_INFO("distance_from_goal : %f",distance_from_goal);

  if(distance_from_goal<0.2){
    return true;
  }

  return false;
}

void Explore::execute() {
  while (! move_base_client_.waitForServer(ros::Duration(5,0)))
    ROS_WARN("Waiting to connect to move_base server");

  ROS_INFO("Connected to move_base server");

  // This call sends the first goal, and sets up for future callbacks.
  ROS_INFO(":::tesi::: prima di makePlan()");
  makePlan();
  ROS_INFO(":::tesi::: dopo makePlan()");

  ros::Rate r(planner_frequency_);
  while (node_.ok() && (!done_exploring_)) {

    //:::dt:::
    if(decision_perception_discretization_ == 0){

      //continuous perception – continuous decision

      if (close_loops_) {
        tf::Stamped<tf::Pose> robot_pose;
        explore_costmap_ros_->getRobotPose(robot_pose);
        loop_closure_->updateGraph(robot_pose);
      }

      makePlan();
    }
    else if(decision_perception_discretization_ == 1 || decision_perception_discretization_ == 2){

      //continuous perception – discrete decision

      tf::Stamped<tf::Pose> robot_pose_local;
    explore_costmap_ros_->getRobotPose(robot_pose_local);
      PoseStamped robot_pose_local_msg;
      tf::poseStampedTFToMsg(robot_pose_local, robot_pose_local_msg);
      float map_resolution = explore_costmap_ros_->getResolution();

      if(chosenGoal_reached(robot_pose_local_msg, chosenGoal_pose_, map_resolution)){

        if(decision_perception_discretization_ == 2){

          //discrete perception – discrete decision

          explore_costmap_ros_->start();
          //far girare il robot di 360 gradi????
        }

        if (close_loops_) {
```

```
          tf::Stamped<tf::Pose> robot_pose;
          explore_costmap_ros_->getRobotPose(robot_pose);
          loop_closure_->updateGraph(robot_pose);
        }

        makePlan();

      }
      else{
        if(decision_perception_discretization_ == 2){

          //discrete perception – discrete decision

          explore_costmap_ros_->stop();
        }
      }
    }


    r.sleep();
  }

  move_base_client_.cancelAllGoals();
}

void Explore::spin() {
  ROS_INFO(":::tesi::: prima di spinOnce()");
  ros::spinOnce();
  ROS_INFO(":::tesi::: dopo ros::spinOnce() : prima di thread");
  boost::thread t(boost::bind( &Explore::execute, this ));
  ROS_INFO(":::tesi::: dopo thread : prima di ros::spin()");
  ros::spin();
  ROS_INFO(":::tesi::: dopo ros::spin() : prima di thread.join()");
  t.join();
  ROS_INFO(":::tesi::: dopo thread.join()");
}

}

int main(int argc, char** argv){
  ros::init(argc, argv, "explore");

  explore::Explore explore;
  explore.spin();

  return(0);
}
```

File: -ros-pkg/trunk/stacks/exploration/explore/include/explore/explore_frontier.h

```
#ifndef EXPLORE_FRONTIER_H_
#define EXPLORE_FRONTIER_H_

#include <nav_msgs/GetMap.h>
#include <nav_msgs/OccupancyGrid.h>
#include <geometry_msgs/Pose.h>
#include <visualization_msgs/Marker.h>

#include <LinearMath/btVector3.h>

#include <costmap_2d/costmap_2d_ros.h>
#include <navfn/navfn_ros.h>
#include <tf/transform_listener.h>

namespace explore {

struct FrontierPoint{
  int idx;     //position
  btVector3 d; //direction

  FrontierPoint(int idx_, btVector3 d_) : idx(idx_), d(d_) {};
};

struct Frontier {
```

```
  geometry_msgs::Pose pose;
  int size;
  Frontier():pose(),size(0) {}
  Frontier(const Frontier& copy) { pose = copy.pose; size = copy.size; }
};

struct WeightedFrontier {
  Frontier frontier;
  float cost;
  bool operator<(const WeightedFrontier& o) const { return cost < o.cost; }
  WeightedFrontier():frontier(),cost(1e9) {}
  WeightedFrontier(const WeightedFrontier& copy) { frontier = copy.frontier; cost = copy.cost; }
};

/**
 * @class ExploreFrontier
 * @brief A class that will identify frontiers in a partially explored map
 */
class ExploreFrontier {
private:
  nav_msgs::OccupancyGrid map_;

  uint lastMarkerCount_;
  float costmapResolution_;

  navfn::NavfnROS* planner_;
protected:
  std::vector<Frontier> frontiers_;

  /**
   * @brief Finds frontiers and populates frontiers_
   * @param costmap The costmap to search for frontiers
   */
  virtual void findFrontiers(costmap_2d::Costmap2DROS& costmap_);

  /**
   * @brief Calculates cost to explore frontier
   * @param frontier to evaluate
   */
  virtual float getFrontierCost(const Frontier& frontier);

  /**
   * @brief Calculates how much the robot would have to turn to face this frontier
   * @param frontier to evaluate
   * @param robot_pose current pose
   */
  virtual double getOrientationChange(const Frontier& frontier, const tf::Stamped<tf::Pose>&
robot_pose);

  /**
   * @brief Calculates potential information gain of exploring frontier
   * @param frontier to evaluate
   */
  virtual float getFrontierGain(const Frontier& frontier, double map_resolution);

public:
  ExploreFrontier();
  virtual ~ExploreFrontier();

  /**
   * @brief Returns all frontiers
   * @param costmap The costmap to search for frontiers
   * @param frontiers Will be filled with current frontiers
   * @return True if at least one frontier was found
   */
  virtual bool getFrontiers(costmap_2d::Costmap2DROS& costmap, std::vector<geometry_msgs::Pose>&
frontiers);

  /**
   * @brief Returns a list of frontiers, sorted by the planners estimated cost to visit each
frontier
   * @param costmap The costmap to search for frontiers
   * @param start The current position of the robot
   * @param goals Will be filled with sorted list of current goals
   * @param planner A planner to evaluate the cost of going to any frontier
   * @param potential_scale A scaling for the potential to a frontier goal point for the frontier's
cost
```

```
   * @param orientation_scale A scaling for the change in orientation required to get to a goal
point for the frontier's cost
   * @param gain_scale A scaling for the expected information gain to get to a goal point for the
frontier's cost
   * @param exploration_strategy The chosen exploration strategy
   * @return True if at least one frontier was found
   *
   * The frontiers are weighted by a simple cost function, which prefers
   * frontiers which are large and close:
   *   frontier cost = travel cost / frontier size
   *
   * Several different positions are evaluated for each frontier. This
   * improves the robustness of goals which may lie near other obstacles
   * which would prevent planning.
   */
  virtual bool getExplorationGoals(costmap_2d::Costmap2DROS& costmap, tf::Stamped<tf::Pose>
robot_pose, navfn::NavfnROS* planner, std::vector<geometry_msgs::Pose>& goals, double cost_scale,
double orientation_scale, double gain_scale, int exploration_strategy);

  /**
   * @brief  Returns markers representing all frontiers
   * @param markers All markers will be added to this vector
   */
  virtual void getVisualizationMarkers(std::vector<visualization_msgs::Marker>& markers);
};

}

#endif /* EXPLORE_FRONTIER_H_ */
```

File: bosh-ros-pkg/trunk/stacks/exploration/explore/src/explore_frontier.cpp

```
#include <explore/explore_frontier.h>


using namespace visualization_msgs;
using namespace costmap_2d;

namespace explore {

ExploreFrontier::ExploreFrontier() :
  map_(),
  lastMarkerCount_(0),
  planner_(NULL),
  frontiers_()
{
}

ExploreFrontier::~ExploreFrontier()
{

}

bool ExploreFrontier::getFrontiers(Costmap2DROS& costmap, std::vector<geometry_msgs::Pose>&
frontiers)
{
  findFrontiers(costmap);
  if (frontiers_.size() == 0)
    return false;

  frontiers.clear();
  for (uint i=0; i < frontiers_.size(); i++) {
    geometry_msgs::Pose frontier;
    frontiers.push_back(frontiers_[i].pose);
  }

  return (frontiers.size() > 0);
}

float ExploreFrontier::getFrontierCost(const Frontier& frontier) {
  ROS_DEBUG("cost of frontier: %f, at position: (%.2f, %.2f, %.2f)", planner_-
>getPointPotential(frontier.pose.position),
      frontier.pose.position.x, frontier.pose.position.y, tf::getYaw(frontier.pose.orientation));
  if (planner_ != NULL)
```

```
    return planner_->getPointPotential(frontier.pose.position); // / 20000.0;
  else
    return 1.0;
}

// TODO: what is this doing exactly?
double ExploreFrontier::getOrientationChange(const Frontier& frontier, const tf::Stamped<tf::Pose>&
robot_pose){
  double robot_yaw = tf::getYaw(robot_pose.getRotation());
  double robot_atan2 = atan2(robot_pose.getOrigin().y() + sin(robot_yaw), robot_pose.getOrigin().x()
+ cos(robot_yaw));
  double frontier_atan2 = atan2(frontier.pose.position.x, frontier.pose.position.y);
  double orientation_change = robot_atan2 - frontier_atan2;
//  ROS_DEBUG("Orientation change: %.3f degrees, (%.3f radians)", orientation_change * (180.0 /
M_PI), orientation_change);
  return orientation_change;
}

float ExploreFrontier::getFrontierGain(const Frontier& frontier, double map_resolution) {
  return frontier.size * map_resolution;
}

bool ExploreFrontier::getExplorationGoals(Costmap2DROS& costmap, tf::Stamped<tf::Pose> robot_pose,
navfn::NavfnROS* planner, std::vector<geometry_msgs::Pose>& goals, double potential_scale, double
orientation_scale, double gain_scale, int exploration_strategy)
{
  findFrontiers(costmap);
  if (frontiers_.size() == 0)
    return false;

  geometry_msgs::Point start;
  start.x = robot_pose.getOrigin().x();
  start.y = robot_pose.getOrigin().y();
  start.z = robot_pose.getOrigin().z();

  planner->computePotential(start);

  planner_ = planner;
  costmapResolution_ = costmap.getResolution();

  //we'll make sure that we set goals for the frontier at least the circumscribed
  //radius away from unknown space
  float step = -1.0 * costmapResolution_;
  int c = ceil(costmap.getCircumscribedRadius() / costmapResolution_);
  WeightedFrontier goal;
  std::vector<WeightedFrontier> weightedFrontiers;
  weightedFrontiers.reserve(frontiers_.size() * c);
  for (uint i=0; i < frontiers_.size(); i++) {
    Frontier& frontier = frontiers_[i];
    WeightedFrontier weightedFrontier;
    weightedFrontier.frontier = frontier;

    tf::Point p(frontier.pose.position.x, frontier.pose.position.y, frontier.pose.position.z);
    tf::Quaternion bt;
    tf::quaternionMsgToTF(frontier.pose.orientation, bt);
    tf::Vector3 v(cos(bt.getAngle()), sin(bt.getAngle()), 0.0);

    for (int j=0; j <= c; j++) {
      tf::Vector3 check_point = p + (v * (step * j));
      weightedFrontier.frontier.pose.position.x = check_point.x();
      weightedFrontier.frontier.pose.position.y = check_point.y();
      weightedFrontier.frontier.pose.position.z = check_point.z();

      weightedFrontier.cost = potential_scale * getFrontierCost(weightedFrontier.frontier) +
orientation_scale * getOrientationChange(weightedFrontier.frontier, robot_pose) - gain_scale *
getFrontierGain(weightedFrontier.frontier, costmapResolution_);
//      weightedFrontier.cost = getFrontierCost(weightedFrontier.frontier) -
getFrontierGain(weightedFrontier.frontier, costmapResolution_);
//      ROS_DEBUG("cost: %f (%f * %f + %f * %f - %f * %f)",
//          weightedFrontier.cost,
//          potential_scale,
//          getFrontierCost(weightedFrontier.frontier),
//          orientation_scale,
//          getOrientationChange(weightedFrontier.frontier, robot_pose),
//          gain_scale,
//          getFrontierGain(weightedFrontier.frontier, costmapResolution_) );
      weightedFrontiers.push_back(weightedFrontier);
    }
```

```
  }

  goals.clear();
  goals.reserve(weightedFrontiers.size());
  std::sort(weightedFrontiers.begin(), weightedFrontiers.end());

  //:::tesi::: modifiche per tesi
  ROS_INFO("getExplorationGoals: weightedFrontiers.size()= %d", weightedFrontiers.size());

  for (uint i = 0; i < weightedFrontiers.size(); i++) {
    //:::tesi::: modifiche per tesi
    ROS_INFO("goals[%d] : x=%f - y=%f -
cost=%f",(int)i,(float)weightedFrontiers[i].frontier.pose.position.x,(float)weightedFrontiers[i].fro
ntier.pose.position.y,(float)weightedFrontiers[i].cost);

    goals.push_back(weightedFrontiers[i].frontier.pose);
  }
  return (goals.size() > 0);
}

void ExploreFrontier::findFrontiers(Costmap2DROS& costmap_) {
  frontiers_.clear();

  Costmap2D costmap;
  costmap_.getCostmapCopy(costmap);

  int idx;
  int w = costmap.getSizeInCellsX();
  int h = costmap.getSizeInCellsY();
  int size = (w * h);

  map_.info.width = w;
  map_.info.height = h;
  map_.set_data_size(size);
  map_.info.resolution = costmap.getResolution();
  map_.info.origin.position.x = costmap.getOriginX();
  map_.info.origin.position.y = costmap.getOriginY();

  // Find all frontiers (open cells next to unknown cells).
  const unsigned char* map = costmap.getCharMap();
  for (idx = 0; idx < size; idx++) {
//    //get the world point for the index
//    unsigned int mx, my;
//    costmap.indexToCells(idx, mx, my);
//    geometry_msgs::Point p;
//    costmap.mapToWorld(mx, my, p.x, p.y);
//
    //check if the point has valid potential and is next to unknown space
//    bool valid_point = planner_->validPointPotential(p);
    bool valid_point = (map[idx] < LETHAL_OBSTACLE);

    if ((valid_point && map) &&
        (((idx+1 < size) && (map[idx+1] == NO_INFORMATION)) ||
         ((idx-1 >= 0) && (map[idx-1] == NO_INFORMATION)) ||
         ((idx+w < size) && (map[idx+w] == NO_INFORMATION)) ||
         ((idx-w >= 0) && (map[idx-w] == NO_INFORMATION))))
    {
      map_.data[idx] = -128;
    } else {
      map_.data[idx] = -127;
    }
  }

  // Clean up frontiers detected on separate rows of the map
  idx = map_.info.height - 1;
  for (unsigned int y=0; y < map_.info.width; y++) {
    map_.data[idx] = -127;
    idx += map_.info.height;
  }

  // Group adjoining map_ pixels
  int segment_id = 127;
  std::vector< std::vector<FrontierPoint> > segments;
  for (int i = 0; i < size; i++) {
    if (map_.data[i] == -128) {
      std::vector<int> neighbors;
      std::vector<FrontierPoint> segment;
      neighbors.push_back(i);
```

```
    // claim all neighbors
    while (neighbors.size() > 0) {
      int idx = neighbors.back();
      neighbors.pop_back();
      map_.data[idx] = segment_id;

      btVector3 tot(0,0,0);
      int c = 0;
      if ((idx+1 < size) && (map[idx+1] == NO_INFORMATION)) {
        tot += btVector3(1,0,0);
        c++;
      }
      if ((idx-1 >= 0) && (map[idx-1] == NO_INFORMATION)) {
        tot += btVector3(-1,0,0);
        c++;
      }
      if ((idx+w < size) && (map[idx+w] == NO_INFORMATION)) {
        tot += btVector3(0,1,0);
        c++;
      }
      if ((idx-w >= 0) && (map[idx-w] == NO_INFORMATION)) {
        tot += btVector3(0,-1,0);
        c++;
      }
      assert(c > 0);
      segment.push_back(FrontierPoint(idx, tot / c));

      // consider 8 neighborhood
      if (((idx-1)>0) && (map_.data[idx-1] == -128))
        neighbors.push_back(idx-1);
      if (((idx+1)<size) && (map_.data[idx+1] == -128))
        neighbors.push_back(idx+1);
      if (((idx-map_.info.width)>0) && (map_.data[idx-map_.info.width] == -128))
        neighbors.push_back(idx-map_.info.width);
      if (((idx-map_.info.width+1)>0) && (map_.data[idx-map_.info.width+1] == -128))
        neighbors.push_back(idx-map_.info.width+1);
      if (((idx-map_.info.width-1)>0) && (map_.data[idx-map_.info.width-1] == -128))
        neighbors.push_back(idx-map_.info.width-1);
      if (((idx+(int)map_.info.width)<size) && (map_.data[idx+map_.info.width] == -128))
        neighbors.push_back(idx+map_.info.width);
      if (((idx+(int)map_.info.width+1)<size) && (map_.data[idx+map_.info.width+1] == -128))
        neighbors.push_back(idx+map_.info.width+1);
      if (((idx+(int)map_.info.width-1)<size) && (map_.data[idx+map_.info.width-1] == -128))
        neighbors.push_back(idx+map_.info.width-1);
    }

    segments.push_back(segment);
    segment_id--;
    if (segment_id < -127)
      break;
  }
}

int num_segments = 127 - segment_id;
if (num_segments <= 0)
  return;

for (unsigned int i=0; i < segments.size(); i++) {
  Frontier frontier;
  std::vector<FrontierPoint>& segment = segments[i];
  uint size = segment.size();

  //we want to make sure that the frontier is big enough for the robot to fit through
  if (size * costmap.getResolution() < costmap.getInscribedRadius())
    continue;

  float x = 0, y = 0;
  btVector3 d(0,0,0);

  for (uint j=0; j<size; j++) {
    d += segment[j].d;
    int idx = segment[j].idx;
    x += (idx % map_.info.width);
    y += (idx / map_.info.width);
  }
  d = d / size;
  frontier.pose.position.x = map_.info.origin.position.x + map_.info.resolution * (x / size);
```

```
    frontier.pose.position.y = map_.info.origin.position.y + map_.info.resolution * (y / size);
    frontier.pose.position.z = 0.0;

    frontier.pose.orientation = tf::createQuaternionMsgFromYaw(btAtan2(d.y(), d.x()));
    frontier.size = size;

    frontiers_.push_back(frontier);
  }

}

void ExploreFrontier::getVisualizationMarkers(std::vector<Marker>& markers)
{
  Marker m;
  m.header.frame_id = "map";
  m.header.stamp = ros::Time::now();
  m.id = 0;
  m.ns = "frontiers";
  m.type = Marker::ARROW;
  m.pose.position.x = 0.0;
  m.pose.position.y = 0.0;
  m.pose.position.z = 0.0;
  m.scale.x = 1.0;
  m.scale.y = 1.0;
  m.scale.z = 1.0;
  m.color.r = 0;
  m.color.g = 0;
  m.color.b = 255;
  m.color.a = 255;
  m.lifetime = ros::Duration(0);

  m.action = Marker::ADD;
  uint id = 0;
  for (uint i=0; i<frontiers_.size(); i++) {
    Frontier frontier = frontiers_[i];
    m.id = id;
    m.pose = frontier.pose;
    markers.push_back(Marker(m));
    id++;
  }

  m.action = Marker::DELETE;
  for (; id < lastMarkerCount_; id++) {
    m.id = id;
    markers.push_back(Marker(m));
  }

  lastMarkerCount_ = markers.size();
}

}
```

3.1. Aggiunto il parametro “planner_frequency”

Aggiunto il parametro “planner_frequency” nel file explore_slam.xml per permettere a chi lo volesse di modificare a piacimento la frequenza di pianificazione del nuovo goal verso il quale dirigersi.

Aggiunta la riga seguente nel file: explore_slam.xml

```
<param name="planner_frequency" value="1.0"/>
```

3.2. Aggiunto il parametro “decision_perception_discretization”

Aggiunto il parametro “decision_perception_discretization” nel file explore_slam.xml per evitare “effetti collaterali” in altre parti dell’applicazione in seguito alla eventuale modifica dei parametri “planner_frequency” e “update_frequency”, e per mantenere un unico parametro dedicato alla scelta della modalità di esplorazione e solo ad essa, sicuramente la soluzione più pulita.

Al parametro in questione è stato assegnato il valore “0”, che prevede decisione e percezione continue.

Aggiunta la riga seguente nel file: explore_slam.xml

```
<param name=”decision_perception_discretization” value=”0”/>
```

Per poter utilizzare questo parametro è necessario aggiungerlo anche nel codice sorgente, in particolare nei file explore.h ed explore.cpp.

Aggiunta la riga seguente nel file: explore.h

```
int decision_perception_discretization_;
```

Aggiunta la riga seguente nel file: explore.cpp

```
private_nh.param("decision_perception_discretization", decision_perception_discretization_, 0);
```

3.3. Aggiunta la variabile “chosenGoal_pose”

Manteniamo memorizzato in questa variabile il goal che stiamo cercando di raggiungere.

La variabile va aggiunta sia nel file explore.cpp che nel file explore.h

Aggiunta la riga seguente nel file: explore.h

```
geometry_msgs::PoseStamped chosenGoal_pose_;
```

Aggiunta la parte in rosso, all’interno del metodo “makePlan()”, nel file: explore.cpp

```
......
......

bool valid_plan = false;
std::vector<geometry_msgs::PoseStamped> plan;
PoseStamped goal_pose, robot_pose_msg;
tf::poseStampedTFToMsg(robot_pose, robot_pose_msg);

goal_pose.header.frame_id = explore_costmap_ros_->getGlobalFrameID();
goal_pose.header.stamp = ros::Time::now();
planner_->computePotential(robot_pose_msg.pose.position); // just to be safe, though this should
already have been done in explorer_->getExplorationGoals

//:::dt:::
ROS_INFO(":::dt::: goals.size()= %d", goals.size());

int blacklist_count = 0;
for (unsigned int i=0; i<goals.size(); i++) {
```

```
  goal_pose.pose = goals[i];

  if (goalOnBlacklist(goal_pose)) {
    blacklist_count++;
    continue;
  }

  valid_plan = ((planner_->getPlanFromPotential(goal_pose, plan)) && (!plan.empty()));
  if (valid_plan) {

    //:::dt:::
    chosenGoal_pose_.header.frame_id = goal_pose.header.frame_id;
    chosenGoal_pose_.header.stamp = goal_pose.header.stamp;
    chosenGoal_pose_.pose = goal_pose.pose;
    ROS_INFO("chosenGoal_pose_ : x=%f -
y=%f",(float)chosenGoal_pose_.pose.position.x,(float)chosenGoal_pose_.pose.position.y);

    break;
  }
}

//:::dt:::
ROS_INFO(":::dt::: blacklist_count= %d", blacklist_count);

......
......
```

3.4. *Aggiunto il metodo "chosenGoal_reached( )"*

In questo medodo si controlla se il goal scelto in precedenza sia stato raggiunto, o se comunque siamo ad una distanza minima da esso: si calcola la distanza tra il robot e il goal scelto, se essa è inferiore a una determinata soglia, allora si considera il goal come raggiunto e si consente di sceglierne uno nuovo.

L'importante è scegliere accuratamente la soglia della distanza entro la quale consideriamo raggiunto il goal.

Aggiunto il metodo seguente nel file explore.cpp

```
//:::dt:::
bool chosenGoal_reached(PoseStamped robot_pose_msg, PoseStamped selected_goal_pose, float
map_resolution){

  ROS_INFO("robot_pose_msg : x=%f -
y=%f",(float)robot_pose_msg.pose.position.x,(float)robot_pose_msg.pose.position.y);
  ROS_INFO("chosenGoal_pose : x=%f -
y=%f",(float)selected_goal_pose.pose.position.x,(float)selected_goal_pose.pose.position.y);
  ROS_INFO("map_resolution : %f",(float)map_resolution);

  float distance_x = (float)robot_pose_msg.pose.position.x -
(float)selected_goal_pose.pose.position.x;
  float distance_y = (float)robot_pose_msg.pose.position.y -
(float)selected_goal_pose.pose.position.y;
  float distance_from_goal = sqrt((distance_x * distance_x) + (distance_y * distance_y));

  ROS_INFO("distance_from_goal : %f",distance_from_goal);

  if(distance_from_goal<0.2){
    return true;
  }

  return false;
}
```

3.5. Modificato il metodo “execute( )”

Questo metodo è quello che gestisce la frequenza di pianificazione e ri-pianificazione del prossimo goal da raggiungere.

Dobbiamo quindi intervenire qui: se il parametro “decision_perception_discretization” è stato impostato ad un valore diverso da 0, dobbiamo gestire la discretizzazione della decisione e/o della percezione, bloccandole finché il goal non sia stato raggiunto.

Aggiunta la parte in rosso nel metodo “execute()”, nel file explore.cpp

```
void Explore::execute() {
  while (! move_base_client_.waitForServer(ros::Duration(5,0)))
    ROS_WARN("Waiting to connect to move_base server");

  ROS_INFO("Connected to move_base server");

  // This call sends the first goal, and sets up for future callbacks.
  makePlan();

  ros::Rate r(planner_frequency_);

  //:::dt:::
  if(exploration_strategy_==0){ //RANDOM
        ROS_INFO(":::dt::: exploration_strategy_ = %d (RANDOM)", exploration_strategy_);
  }
  else if(exploration_strategy_==1){ //WEIGHTED
        ROS_INFO(":::dt::: exploration_strategy_ = %d (WEIGHTED)", exploration_strategy_);
  }
  else if(exploration_strategy_==2){ //MCDM
        ROS_INFO(":::dt::: exploration_strategy_ = %d (MCDM)", exploration_strategy_);
  }

  if(decision_perception_discretization_==0){ //continuous perception - continuous decision
        ROS_INFO(":::dt::: decision_perception_discretization_ = %d (continuous perception -
continuous decision)", decision_perception_discretization_);
  }
  else if(decision_perception_discretization_==1){ //continuous perception - discrete decision
        ROS_INFO(":::dt::: decision_perception_discretization_ = %d (continuous perception -
discrete decision)", decision_perception_discretization_);
  }
  else if(decision_perception_discretization_==2){ //discrete perception - discrete decision
        ROS_INFO(":::dt::: decision_perception_discretization_ = %d (discrete perception - discrete
decision)", decision_perception_discretization_);
  }

  while (node_.ok() && (!done_exploring_)) {

    //:::dt:::
    if(decision_perception_discretization_ == 0){

      //continuous perception – continuous decision

      if (close_loops_) {
        tf::Stamped<tf::Pose> robot_pose;
        explore_costmap_ros_->getRobotPose(robot_pose);
        loop_closure_->updateGraph(robot_pose);
      }

      makePlan();
    }
    else if(decision_perception_discretization_ == 1 || decision_perception_discretization_ == 2){

      //continuous perception – discrete decision

      tf::Stamped<tf::Pose> robot_pose_local;
      explore_costmap_ros_->getRobotPose(robot_pose_local);
      PoseStamped robot_pose_local_msg;
      tf::poseStampedTFToMsg(robot_pose_local, robot_pose_local_msg);
      float map_resolution = explore_costmap_ros_->getResolution();

      if(chosenGoal_reached(robot_pose_local_msg, chosenGoal_pose_, map_resolution)){
```

```
        if(decision_perception_discretization_ == 2){

          //discrete perception - discrete decision

          explore_costmap_ros_->start();
          //far girare il robot di 360 gradi????
        }

        if (close_loops_) {
          tf::Stamped<tf::Pose> robot_pose;
          explore_costmap_ros_->getRobotPose(robot_pose);
          loop_closure_->updateGraph(robot_pose);
        }

        makePlan();

      }
      else{
        if(decision_perception_discretization_ == 2){

          //discrete perception - discrete decision

          explore_costmap_ros_->stop();
        }
      }
    }

    r.sleep();
  }

  move_base_client_.cancelAllGoals();
}
```

3.6. Aggiunto il parametro "exploration_strategy"

Aggiunto il parametro "exploration_strategy" nel file explore_slam.xml per poter scegliere una tra le strategie di esplorazione disponibili.

Al parametro in questione è stato assegnato il valore "1", che prevede la scelta della strategia WEIGHTED.

Aggiunta la riga seguente nel file: explore_slam.xml

```
<param name="exploration_strategy" value="1"/>
```

Per poter utilizzare questo parametro è necessario aggiungerlo anche nel codice sorgente, in particolare nei file explore.h ed explore.cpp.

Aggiunta la riga seguente nel file: explore.h

```
int exploration_strategy_;
```

Aggiunta la riga seguente nel file: explore.cpp

```
private_nh.param("exploration_strategy", exploration_strategy_, 0);
```

3.7. Aggiunto un nuovo metodo per ogni nuova strategia di esplorazione

Sono stati aggiunti i seguenti metodi all'interno del file explore_frontier.cpp:

- getExplorationGoals_RANDOM()
- getExplorationGoals_WEIGHTED()
- getExplorationGoals_MCDM()

Aggiunti i seguenti metodi all'interno del file explore_frontier.cpp

```
xxx
```

3.8. Modificato il metodo getExplorationGoals( )

Modificato, nel file explore_frontier.cpp, il metodo getExplorationGoals(), che restituiva i goal da esplorare in ordine di costo decrescente. Ora il metodo controlla quale strategia è stata scelta e richiama il corrispondente metodo per scegliere e ordinare i goal da raggiungere.

Modificata l'intestazione del metodo getExplorationGoals() nel file explore_frontier.h

```
  /**
   * @brief Returns a list of frontiers, sorted by the planners estimated cost to visit each
frontier
   * @param costmap The costmap to search for frontiers
   * @param start The current position of the robot
   * @param goals Will be filled with sorted list of current goals
   * @param planner A planner to evaluate the cost of going to any frontier
   * @param potential_scale A scaling for the potential to a frontier goal point for the frontier's
cost
   * @param orientation_scale A scaling for the change in orientation required to get to a goal
point for the frontier's cost
   * @param gain_scale A scaling for the expected information gain to get to a goal point for the
frontier's cost
   * @param exploration_strategy The chosen exploration strategy
   * @return True if at least one frontier was found
   *
   * The frontiers are weighted by a simple cost function, which prefers
   * frontiers which are large and close:
   *   frontier cost = travel cost / frontier size
   *
   * Several different positions are evaluated for each frontier. This
   * improves the robustness of goals which may lie near other obstacles
   * which would prevent planning.
   */
  virtual bool getExplorationGoals(costmap_2d::Costmap2DROS& costmap, tf::Stamped<tf::Pose>
robot_pose, navfn::NavfnROS* planner, std::vector<geometry_msgs::Pose>& goals, double cost_scale,
double orientation_scale, double gain_scale, int exploration_strategy);
```

Aggiunta la parte in rosso all'interno del file explore_frontier.cpp

```
bool ExploreFrontier::getExplorationGoals(Costmap2DROS& costmap, tf::Stamped<tf::Pose> robot_pose,
navfn::NavfnROS* planner, std::vector<geometry_msgs::Pose>& goals, double potential_scale, double
orientation_scale, double gain_scale, int exploration_strategy)
{
  findFrontiers(costmap);
  if (frontiers_.size() == 0)
    return false;

  geometry_msgs::Point start;
  start.x = robot_pose.getOrigin().x();
  start.y = robot_pose.getOrigin().y();
  start.z = robot_pose.getOrigin().z();

  planner->computePotential(start);

  planner_ = planner;
  costmapResolution_ = costmap.getResolution();

  //we'll make sure that we set goals for the frontier at least the circumscribed
```

```
  //radius away from unknown space
  float step = -1.0 * costmapResolution_;
  int c = ceil(costmap.getCircumscribedRadius() / costmapResolution_);
  WeightedFrontier goal;
  std::vector<WeightedFrontier> weightedFrontiers;
  weightedFrontiers.reserve(frontiers_.size() * c);

  //:::dt:::
  srand(static_cast<int>(time(NULL)));

  for (uint i=0; i < frontiers_.size(); i++) {
    Frontier& frontier = frontiers_[i];
    WeightedFrontier weightedFrontier;
    weightedFrontier.frontier = frontier;

    tf::Point p(frontier.pose.position.x, frontier.pose.position.y, frontier.pose.position.z);
    tf::Quaternion bt;
    tf::quaternionMsgToTF(frontier.pose.orientation, bt);
    tf::Vector3 v(cos(bt.getAngle()), sin(bt.getAngle()), 0.0);

    for (int j=0; j <= c; j++) {
      tf::Vector3 check_point = p + (v * (step * j));
      weightedFrontier.frontier.pose.position.x = check_point.x();
      weightedFrontier.frontier.pose.position.y = check_point.y();
      weightedFrontier.frontier.pose.position.z = check_point.z();

// intervengo sul calcolo del costo in base al parametro exploration_strategy:
// i goal verranno poi ordinati in base al costo, in ordine crescente

      //ROS_INFO(":::dt::: exploration_strategy = %d", exploration_strategy);

      if(exploration_strategy==0){ //RANDOM
        // srand(static_cast<int>(time(NULL)));
        // int magic = rand() % 100 + 1;
        int magic = rand();
        weightedFrontier.cost = magic;
      }
      else if(exploration_strategy==1){ //WEIGHTED
        //ROS_INFO(":::dt::: exploration_strategy = WEIGHTED");

        weightedFrontier.cost = potential_scale * getFrontierCost(weightedFrontier.frontier) +
orientation_scale * getOrientationChange(weightedFrontier.frontier, robot_pose) - gain_scale *
getFrontierGain(weightedFrontier.frontier, costmapResolution_);

//      weightedFrontier.cost = getFrontierCost(weightedFrontier.frontier) -
getFrontierGain(weightedFrontier.frontier, costmapResolution_);
//      ROS_DEBUG("cost: %f (%f * %f + %f * %f - %f * %f)",
//          weightedFrontier.cost,
//          potential_scale,
//          getFrontierCost(weightedFrontier.frontier),
//          orientation_scale,
//          getOrientationChange(weightedFrontier.frontier, robot_pose),
//          gain_scale,
//          getFrontierGain(weightedFrontier.frontier, costmapResolution_) );

      }
      else if(exploration_strategy==2){ //MCDM
        //ROS_INFO(":::dt::: exploration_strategy = MCDM");

      }

      weightedFrontiers.push_back(weightedFrontier);

    }
  }

  goals.clear();
  goals.reserve(weightedFrontiers.size());
  std::sort(weightedFrontiers.begin(), weightedFrontiers.end());

  for (uint i = 0; i < weightedFrontiers.size(); i++) {
    //:::dt:::
    //ROS_INFO("goals[%d] : x=%f - y=%f -
cost=%f",(int)i,(float)weightedFrontiers[i].frontier.pose.position.x,(float)weightedFrontiers[i].fro
ntier.pose.position.y,(float)weightedFrontiers[i].cost);

    goals.push_back(weightedFrontiers[i].frontier.pose);
  }
```

```
  return (goals.size() > 0);
}
```

Modificata la seguente chiamata all'interno del metodo "makePlan()" nel file explore.cpp

```
explorer_->getExplorationGoals(*explore_costmap_ros_, robot_pose, planner_, goals, potential_scale_,
orientation_scale_, gain_scale_, exploration_strategy_);
```

3.9. Nuovo file di configurazione per RVIZ

Creata nuova cartella "rviz" all'interno della cartella explore_stage. Copiato all'interno di tale cartella il file stage/rviz/stage.vcg e rinominato: rviz_config.vcg.

D'ora in avanti useremo il seguente comando per lanciare RVIZ:

```
rosrun rviz rviz -d $(rospack find explore_stage)/rviz/rviz_config.vcg
```

Modifico il file in modo tale da avere subito a disposizione tutti i topic necessari per la visualizzazione dell'esplorazione:

Modificato il file explore_stage/rviz/stage.vcg

```
Background\ ColorR=0
Background\ ColorG=0
Background\ ColorB=0
Fixed\ Frame=/map
Target\ Frame=<Fixed Frame>
Grid.Alpha=0,5
Grid.Cell\ Size=1
Grid.ColorR=0,5
Grid.ColorG=0,5
Grid.ColorB=0,5
Grid.Enabled=1
Grid.Line\ Style=0
Grid.Line\ Width=0,03
Grid.Normal\ Cell\ Count=0
Grid.OffsetX=0
Grid.OffsetY=0
Grid.OffsetZ=0
Grid.Plane=0
Grid.Plane\ Cell\ Count=50
Grid.Reference\ Frame=<Fixed Frame>
Laser\ Scan.Alpha=1
Laser\ Scan.Autocompute\ Intensity\ Bounds=0
Laser\ Scan.Billboard\ Size=0,01
Laser\ Scan.Channel=0
Laser\ Scan.Decay\ Time=60
Laser\ Scan.Enabled=1
Laser\ Scan.Max\ ColorR=1
Laser\ Scan.Max\ ColorG=1
Laser\ Scan.Max\ ColorB=1
Laser\ Scan.Max\ Intensity=0
Laser\ Scan.Min\ ColorR=0
Laser\ Scan.Min\ ColorG=0
Laser\ Scan.Min\ ColorB=0
Laser\ Scan.Min\ Intensity=0
Laser\ Scan.Selectable=1
Laser\ Scan.Style=1
Laser\ Scan.Topic=/base_scan
Map.Alpha=0,7
Map.Draw\ Behind=0
Map.Enabled=1
Map.Topic=/map
Markers.Enabled=1
Markers.Marker\ Topic=visualization_marker
Markers.explore_goal=1
```

```
Markers.frontiers=1
Markers.loop_closure=1
Path.Alpha=1
Path.ColorR=0,1
Path.ColorG=1
Path.ColorB=0
Path.Enabled=1
Path.Topic=/move_base/NavfnROS/plan
Transforms.All\ Enabled=1
Transforms.Enabled=1
Transforms.Show\ Arrows=0
Transforms.Show\ Axes=1
Transforms.Show\ Names=1
Transforms.Update\ Rate=1
Tool\ 2D\ Nav\ GoalTopic=goal
Tool\ 2D\ Pose\ EstimateTopic=initialpose
Camera\ Type=rviz::FixedOrientationOrthoViewController
Camera\ Config=-14.5018 0 30 0 -0.706391 -0.0318091 -0.0318091 0.706391
Property\ Grid\ State=selection=Laser Scan.Enabled;expanded=.Global
Options,Transforms./base_footprint/base_footprint,Transforms./base_laser_link/base_laser_link,Transf
orms./base_link/base_link,Transforms./map/map,Transforms./odom/odom,Transforms.Enabled.Transforms.Tr
ee,Transforms./mapTree/map,Transforms./odomTree/odom,Transforms./base_footprintTree/base_footprint,T
ransforms./base_linkTree/base_link,Transforms./base_laser_linkTree/base_laser_link,Markers.Enabled.M
arkers.StatusTopStatus,Markers.Enabled.Markers.Namespaces;scrollpos=0,0;splitterpos=150,301;ispagese
lected=1
[Display0]
Name=Grid
Package=rviz
ClassName=rviz::GridDisplay
[Display1]
Name=Transforms
Package=rviz
ClassName=rviz::TFDisplay
[Display2]
Name=Laser Scan
Package=rviz
ClassName=rviz::LaserScanDisplay
[Display3]
Name=Map
Package=rviz
ClassName=rviz::MapDisplay
[Display4]
Name=Markers
Package=rviz
ClassName=rviz::MarkerDisplay
[Display5]
Name=Path
Package=rviz
ClassName=rviz::PathDisplay
[Transforms.]
base_footprintEnabled=1
base_laser_linkEnabled=1
base_linkEnabled=1
mapEnabled=1
odomEnabled=1
```

==========[APPUNTI]==========

Capire dove va a incidere il parametro “update_frequency” per la frequenza di aggiornamento della mappa.

Update_frequency:

- explore/explore_costmap.yaml (explore_slam.xml) (scope-package: explore)
- explore/ local_costmap.yaml (move.xml) (scope-package: move-base)
- explore/global_costmap.yaml (move.xml) (scope-package: move_base)

Controllare nei sorgenti di explore e di move_base dove e per cosa viene usato il parametro update_frequency.

Sembra che il parametro update_frequency non venga mai utilizzato… controllare con RVIZ. (sbagliato, viene utilizzato)

Vedere se su rviz viene bloccata anche la percezione, insieme alla decisione..

Provare con la seguente chiamata all'interno del file explore.cpp:

explore_costmap_ros_->stop() per fermare l'aggiornamento della mappa

explore_costmap_ros_->start() per farlo ripartire

il file interessato dalle modifiche al parametro "update_frequency" è il seguente:

opt/ros/boxturtle/stacks/navigation/costmap2d/src/costmap_2d_ros.cpp

sistemare il metodo chosenGoal_reached(), in particolare decidere a quanto fissare la soglia della distanza per considerare il goal raggiunto

valutazione delle prestazioni:

- su rviz, vedere cosa sono i seguenti
    - wall time
    - wall elapsed
    - ROS time
    - ROS elapsed
- Devo considerare il tempo impiegato dal robot per capire che ha esplorato tutta la mappa, o il tempo esatto di copertura totale della mappa? (il secondo è inferiore al primo)

Vedi condizione per dichiarare esplorazione completata ("Hooray…")

Vedi fixed frame e target frame su rviz

Controllare se goal in blacklist quando strategia discreta (utile per conclusione esplorazione) oppure spostare il blocco di codice che controlla il tipo do strategia, mettendolo all'interno del makeplan().