

Robocup@Work Task-Planner

Francesco Chichi¹ Luca Monorchio²

Abstract

RoboCup@Work is a competition in RoboCup that targets the use of robots in work-related scenarios. It aims to foster research and development that enables use of innovative mobile robots equipped with advanced manipulators for current and future industrial applications, where robots cooperate with human workers for complex tasks ranging from manufacturing, automation, and parts handling up to general logistics. The value of classical task planning in soccer and rescue leagues is rather limited; if it used at all, these planners have to deal with constraints that are quite different from those in most industrial domains. Task planning may eventually be of great interest in RoboCup@Home, but so far most teams work with preprogrammed scripts or state machines executing routine activities. In RoboCup@Work, however, multiagent planning and scheduling with multi-criteria optimization is of immediate interest, and has a large potential for innovative applications. In this paper we will see a task planning approach based on a topological plan with focus on all the replanning in case of obstacles during the navigation.

¹Master in Artificial Intelligence and Robotics, mat. 1538229

²Master in Artificial Intelligence and Robotics, mat. 1650427

Contents

1	Problem approach	2
1.1	Build the topological plan	2
1.2	Distance matrix	3
1.3	Weighted graph based on current state	3
2	Planning structure	3
2.1	PDDL	3
2.2	Objects constraint	4
3	The Fast Downward Planning System	4
3.1	Lama 2011	5
4	Replanning Pddl	6
4.1	Obstacle's detection module	6
4.2	Topological plan updating	7
5	User interface	7
5.1	GUI explanation	7
5.2	GUI functionality	10
6	Readme	12



Introduction

The purpose of the Robocup@Work competition is to assess the ability of the robots for combined navigation and manipulation tasks. The robots have to deal with flexible task specifications, especially concerning information about object constellations in source and target locations, and task constraints such as limits on the number of objects allowed to be carried simultaneously. A single custom robot is used,



Figure 1. A youbot in action during an edition of Robocup.

which is initially positioned outside of the arena near a gate to the arena. The task is to get several objects from the source service areas (such as SH02, WS09, or CB02), and to deliver them to the destination service areas (e.g. WS11 and SH05). Robots may carry up to three objects simultaneously. The task specification consists of two lists: The first list contains for each service area a list of manipulation object descriptions. The second list contains for each destination service area a configuration of manipulation objects where the robot is supposed to achieve.

The following rules have to be obeyed:

- The robot has to start from outside the arena.
- The robot will get the task specification from the referee box.
- After the team's robot starts, it must move into the arena and attempt to complete the task.
- A manipulation object counts as successfully placed, if the robot has placed the object into the correct destination service area.
- It is not allowed to place manipulation objects anywhere except for the robot itself and any of the available service areas.
- A robot may carry up to three objects at the same time.
- The time is stopped when the robot has completed the task (delivered all objects to the right locations and left the arena through the exit gates). If a team cannot complete the task within the designated time, the run will be stopped.

In this project we propose a new task planning approach based on the construction of a topological plan in which the robot will perform the tasks assigned. We will see also how to replanning the entire task in real time in case of unexpected obstacles scenario. All code is based on the ROS Kinetic version and tested on Ubuntu 16.04 operating system. ROS (Robot Operating System) provides libraries and tools to help software developers create robot applications. It provides hardware abstraction, device drivers, libraries, visualizers, message-passing, package management, and more (ROS is licensed under an open source, BSD license).

1. Problem approach

Planning can seem simple enough at first. What makes the problem more complex is real-time rescheduling, exploiting the perception of the robot (thanks to the use of sensors) to find alternatives induce to negate the aborting of navigation, waste precious time and in the worst case fail the task. Topological plan allows the robot to conveniently manage all the zones of the map subject to risk of collision, in this regard the use of redundant paths facilitate navigation in order to achieve the preset goal.

Build the topological plan

The ros node **spqr_topological_plan** deals with the creation of the topological plan. The user at runtime can choose to add or delete nodes present on the map interface. This map is loaded directly from another node called **map server**, which offers map data as a ROS Service. Maps manipulated by the tools in this package are stored in a pair of files. The YAML file describes the map meta-data, and names the image file. The image file encodes the occupancy data. The map chosen for this project corresponds to the arena of the German-Open 2017 held in Magdeburg every year. The acquisition process has been done with the gmapping package that provides laser-based SLAM (Simultaneous Localization and Mapping), as a ROS node called **slam_gmapping**. Using **slam_gmapping**, we have created a 2D occupancy grid map (like a building floorplan) from laser and pose data collected by the youbot, as depicted in Figure 2. The topological plan also allows you to import the positions set by the organizers of the arena, in this way the person responsible for creating the topological graph will find it easier to link the various nodes of the topological graph. These particular locations come from the **location service** node and are represented by all the nodes having blue color (distinguished by those added later that have red color in Figure 4), as depicted in Figure 3. Once we have finished adding and connecting the various nodes properly, all that remains is to export the entire graph to file in order to proceed with the subsequent planning steps; for further informations, the list of commands available for this node is printed on the terminal every time the “h” key is pressed from the keyboard, for example “*press a to remove arc between nodes*”, “*press e to export the graph on file*”...

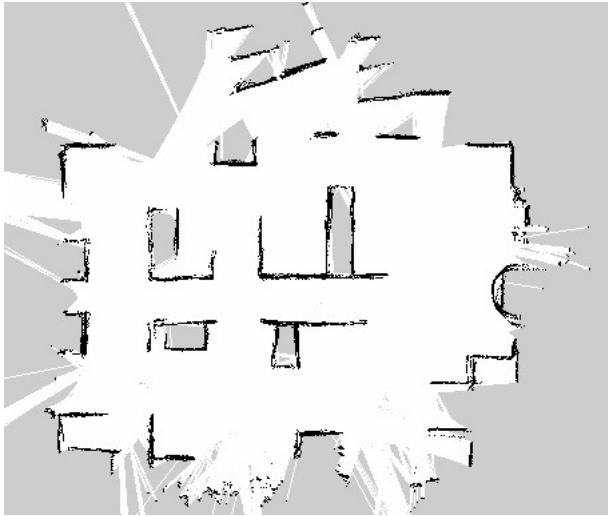


Figure 2. Gmapping's output map.

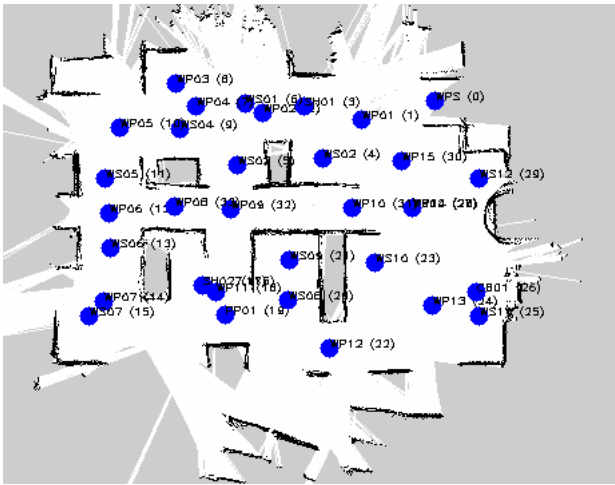


Figure 3. Imported Location Service on topological graph.

Distance matrix

Figure 4 shows how an arc that connects two nodes also takes into account the cost as a function of distance (red value). Knowing all these costs allows us to operate a minimization algorithm of the paths between the various nodes that make up the topological plan and consequently the possibility to obtain an optimal plan. The **master_planner_action** module includes everything necessary to minimize all the paths present in the graph by executing a Dijkstra algorithm (it was conceived by computer scientist Edsger W. Dijkstra in 1956):

For further details check the files `DistanceMatrix.cpp` and `DistanceMatrix.h` in the include directory.

Weighted graph based on current state

The main idea of the Robocup@Work is to have a central server-like hub called *Central Factory Hub* or *CFH* that serves all the services that are needed for executing and scoring tasks and successfully realize the competition. This hub is derived

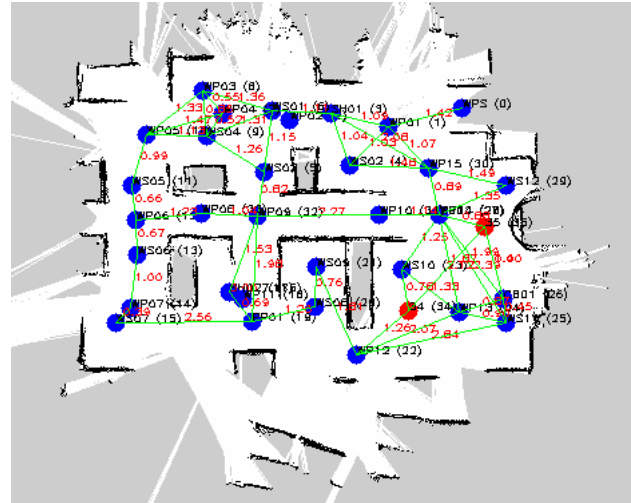


Figure 4. Final result of the topological graph plan.

from software systems well known in industrial business (e.g., SAP). It provides the robots with information regarding the specific tasks and tracks the production process as well as stock and logistics information of the RoCKIn'N'RoLLIn factory, in little terms, it is a plug-in driven software system where each plug-in is responsible for a specific task, functionality or other benchmarking module. The idea is to extract from the topological plan a sub-graph that takes into account all the information coming from the *CFH* (objects to be transported, available locations, etc.). Once Dijkstra has been applied based on the available information, we are ready to process this weighed graph in order to generate Pddl files.

2. Planning structure

We show the kernel of planning based on pddl creation and solving taking in consideration all the procedure seen in the latter sections.

PDDL

The Planning Domain Definition Language (PDDL) is an attempt to standardize Artificial Intelligence (AI) planning languages. It was first developed by Drew McDermott and his colleagues in 1998 (inspired by STRIPS and ADL among others) mainly to make the 1998/2000 International Planning Competition (IPC) possible, and then evolved with each competition. The PDDL planning task structure is organized in different components as:

- **Objects:** Things in the world that interest us.
- **Predicates:** Properties of objects that we are interested in.
- **Initial state:** The state of the world that we start in.
- **Goal specification:** Things that we want to be true.

Algorithm 1 Dijkstra(Graph, source):

```

for each vertex  $v \in \text{Graph}$  : do
     $\text{dist}[v] := \text{infinity}$ 
     $\text{previous}[v] := \text{undefined}$ 
 $\text{dist}[\text{source}] := 0$ 
 $Q := \text{the set of all nodes in Graph}$ 
end for

while  $Q$  is not empty : do
     $u := \text{node in } Q \text{ with smallest dist}[]$ 
    remove  $u$  from  $Q$ 
    for each neighbor  $v$  of  $u$  : do
         $\text{alt} := \text{dist}[u] + \text{dist\_between}(u, v)$ 
        if  $\text{alt} < \text{dist}[v]$ 
             $\text{dist}[v] := \text{alt}$ 
             $\text{previous}[v] := u$ 
        end if
    end for
end while
return  $\text{previous}[v]$ 

```

- **Actions/Operators:** Ways of changing the state of the world.

PDDL is intended to express the physics of a domain, that is, what predicates there are, what actions are possible, what the structure of compound actions is, and what the effects of actions are. Most planners require in addition some kind of advice, that is, annotations about which actions to use in attaining which goals, or in carrying out which compound actions and under which circumstances. For these reasons It is important to know that, in order to put the pieces together, planning tasks specified in PDDL are separated into two files: a *domain* file for predicates and actions, as depicted in Figure 5); a *problem* file for objects, initial state and goal specification, as depicted in Figure 6

Domain files look like this:

```

(define (domain <domain name>)
  <PDDL code for predicates>
  <PDDL code for first action>
  [...]
  <PDDL code for last action>
)

```

Figure 5. PDDL’s domain file structure.

Objects constraint

We know that a task assigned from CFH could have several objects of the same type to be transported to the various target locations. The fact that objects with same instance substantially will be considered the same objects (no data association

Problem files look like this:

```

(define (problem <problem name>)
  (:domain <domain name>)
  <PDDL code for objects>
  <PDDL code for initial state>
  <PDDL code for goal specification>
)

```

Figure 6. PDDL’s problem file structure.

for the goal) allow us to improve planning performance by adding a constraint of distance between the various objects and the destination locations. In this way these objects, that are located in the various start locations will be assigned by the plan in the closer target locations.

3. The Fast Downward Planning System

Fast Downward, i.e., link at: <http://www.fast-downward.org/>, is a classical planning system based on heuristic search. It is performed directly inside the ROS Kinetic environment and It can deal with general deterministic planning problems encoded in the propositional fragment of PDDL2.2, including advanced features like ADL conditions and effects and derived predicates (axioms). Like other well-known planners such as HSP and FF, Fast Downward is a progression planner, searching the space of world states of a planning task in the forward direction. However, unlike other PDDL planning systems, Fast Downward does not use the propositional PDDL representation of a planning task directly. Instead, the input is first translated into an alternative representation called multivalued planning tasks, which makes many of the implicit constraints of a propositional planning task explicit. Exploiting this alternative representation, Fast Downward uses hierarchical decompositions of planning tasks for computing its heuristic function, called the causal graph heuristic, which is very different from traditional HSP-like heuristics based on ignoring negative interactions of operators. As a planning system based on heuristic forward search, Fast Downward is clearly related to other heuristic planners such as HSP (Bonet & Geffner, 2001) or FF (Hoffmann & Nebel, 2001) on the architectural level. the resolution of the planning task are solved in three different phases, as depicted in Figure 7:

- **Translation:** this component is responsible for transforming the PDDL2.2 input into a nonbinary form which is more amenable to hierarchical planning approaches. It applies a number of normalizations to compile away syntactic constructs like disjunctions which are not directly supported by the causal graph heuristic and performs grounding of axioms and operators.
- **The knowledge compilation:** this component generates four kinds of data structures that play a central role during search: Domain transition graphs encode how,

and under what conditions, state variables can change their values. The causal graph represents the hierarchical dependencies between the different state variables. The successor generator is an efficient data structure for determining the set of applicable operators in a given state. Finally, the axiom evaluator is an efficient data structure for computing the values of derived variables.

- **The search:** this component implements three different search algorithms to do the actual planning. Two of these algorithms make use of heuristic evaluation functions: One is the well-known greedy best-first search algorithm, using the causal graph heuristic. The other is called multiheuristic best-first search, a variant of greedy best-first search that tries to combine several heuristic evaluators in an orthogonal way; in the case of Fast Downward, it uses the causal graph and FF heuristics. The third search algorithm is called focused iterative-broadening search; it is closely related to Ginsberg and Harvey's (1992) iterative broadening. It is not a heuristic search algorithm in the sense that it does not use an explicit heuristic evaluation function. Instead, it uses the information encoded in the causal graph to estimate the “usefulness” of operators towards satisfying the goals of the task.

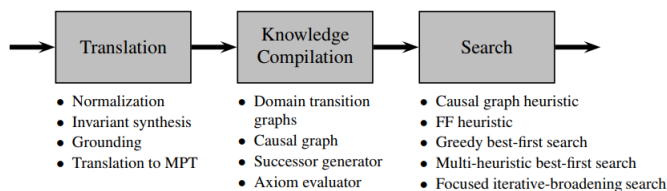


Figure 7. The three phases of Fast Downward's execution.

Let us focus more on the third block of the chain, called *Search*. Unlike the translation and knowledge compilation components, for which there is only a single mode of execution, the search component of Fast Downward can perform its work in various alternative ways. There are three basic search algorithms to choose from:

- **Greedy best-first search:** This is the standard textbook algorithm (Russell & Norvig, 2003), modified with a technique called deferred heuristic evaluation to mitigate the negative influence of wide branching.
- **Multi-heuristic best-first search:** This is a variation of greedy best-first search which evaluates search states using multiple heuristic estimators, maintaining separate open lists for each.
- **Focused iterative-broadening search:** This is a simple search algorithm that does not use heuristic estimators, and instead reduces the vast set of search possibili-

ties by focusing on a limited operator set derived from the causal graph.

Lama 2011

In this project we use a powerful heuristic function called *LAMA 2011*. It is a planning system based on heuristic state space search, in the spirit of *FF* (Hoffmann and Nebel 2001) and *Fast Downward* (Helmert 2006). It won the sequential satisficing track of the International Planning Competition in 2008 and 2011. As mentioned before, *LAMA* builds on the Fast Downward System, inheriting the general structure of Fast Downward, the translation of PDDL tasks with binary state variables to representations with finite-domain variables, and a search architecture that is able to exploit several heuristics simultaneously. One core feature of *LAMA* is the use of landmarks, i.e. variable assignments that must occur at some point in every solution plan, as a heuristic and for generating preferred operators. Using the data structures generated by the knowledge compilation module, the search engine attempts to find a plan using heuristic search with some enhancements, such as the use of preferred operators (similar to helpful actions in *FF*) and deferred heuristic evaluation, which mitigates the impact of large branching factors in planning tasks with fairly accurate heuristic estimates (Richter and Helmert 2009). Deferred heuristic evaluation means that states are not evaluated upon generation, but upon expansion. States are thus not selected for expansion according to their own heuristic value, but according to that of their parent. If many more states are generated than expanded, this leads to a substantial reduction in the number of heuristic estimates computed, if at a loss of heuristic accuracy. The rules of the 6th International Planning Competition (IPC 2008), for which *LAMA* was designed, suggest a type of search that takes plan quality into account. *LAMA* first runs a greedy best-first search, aimed at finding a solution as quickly as possible. Once a plan is found, it searches for progressively better solutions by running a series of weighted A searches with decreasing weight. The cost of the best known solution is used for pruning the search, while decreasing the weight makes the search progressively less greedy, trading speed for solution quality (Richter, Thayer, and Ruml 2010). The search engine is configured to use several heuristic estimators (namely, the *FF* heuristic and the landmark heuristic) within an approach called multi-heuristic search (Helmert 2006; Roger and Helmert 2010). This technique attempts to exploit strengths of the utilised heuristics in different parts of the search space in an orthogonal way. To this end, it uses separate open lists for each of the different heuristics as well as separate open lists for the preferred operators of each heuristic. Newly generated states are evaluated with respect to all heuristics, and added to all open lists (with the value estimate corresponding to the heuristic of that open list). When choosing which state to expand next, the search engine alternates between the different heuristics, and expands states from preferred-operator queues with higher priority than states from other queues.

4. Replanning Pddl

We have come to the most critical phase of this project which is the re-planning of the entire task in real time due to the unexpected presence of obstacles that prevent the continuation of navigation for the robot through the topological plan. We divide the problem into two phases based on the detection of obstacles on the map by means of the youbot's laser sensor and finally on updating the topological plan with PDDL regeneration taking into account the current state of the environment like, for example, the position of the robot in the map, number of slots empty.

Obstacle's detection module

This node exploits the OpenCV library, i.e., an open source computer vision library designed for computational efficiency and with a strong focus on real-time applications, to filter and classify all the information coming from lasers that could be interpreted as information about the presence of arena walls, locations or obstacles. Suppose we find ourselves in this scenario in which the robot is represented by the blue square and the obstacle by the red square on the map in the Figure 8. To manage all this information in real time, we show in a

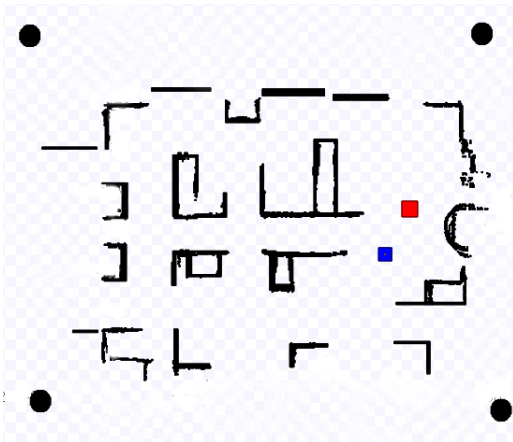


Figure 8. Robot and obstacle in the arena.

schematic way all the key points:

- **Synchronisation between topics:** without this constraint, the entire detection would be compromised because you would not have the right information about the position of the robot in relation to the obstacle at the time of detection and this is the reason for which the *AMCL* topic (probabilistic localization system for a robot moving in 2D) is synchronized with the laser *scan* topic.
- **Localization flag:** a check on the covariance matrix in the *ACML* topic allows us to determine whether or not the robot is delocalized and, consequently, when it is necessary to enable the detection of obstacles or

not. The procedure is done in this way: during navigation the covariance matrix is read from the topic *amclPose* → *pose.covariance*. After this we need to compute the eigen values of the matrix that come from the SVD (singular value decomposition). The result is a vector of six float numbers, three for position and the others for the orientation; finally, by applying appropriate thresholds it is possible to link the obstacle's detection to the localization of the robot.

- **Image processing:** the information obtained from the various topics and from the map (loaded from the map server) is converted into various images. The idea is to apply several filters that manually eliminate the various disturbances that could compromise the final detection quality. Walls, edges and all that concerns known information on the map are expanded like a *distance map* to avoid, for example, confusing a wall with an obstacle, as depicted in Figure 9.



Figure 9. The distance map of the arena.

- **Accumulator and sliding window:** the laser sensor is nothing more than a series of rays partitioned within a sector of 120 degrees, such rays when they affect a surface are reflected back from the point where they are ejected, precisely the laser sensor; it is therefore possible to establish the position in pixels in the map relative to the point of impact of each single ray, for each single interval of time. Using this information we have inserted in the map a technique called an accumulator that consists of adding a weight to a certain pixel every time the ray impacts on it, if that pixel reaches a weight higher than a threshold, then it will surely be an obstacle present in the map, as shown in the Figure 10. We must make an observation in all this: it is clear that in the presence of an obstacle more pixels close to each other will be detected, what we do to pull out the position of the obstacle is to apply a sliding window of opencv called *minmaxloc*. This function find the

minimum and maximum element values and their positions. The extremums are searched across the whole array or, if mask is not an empty array, in the specified array region. Finally a topic will publish both the robot position and obstacle position.

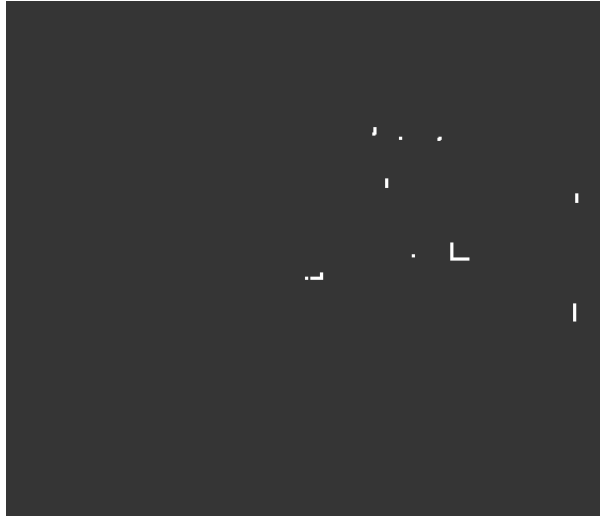


Figure 10. The accumulator's scan result.

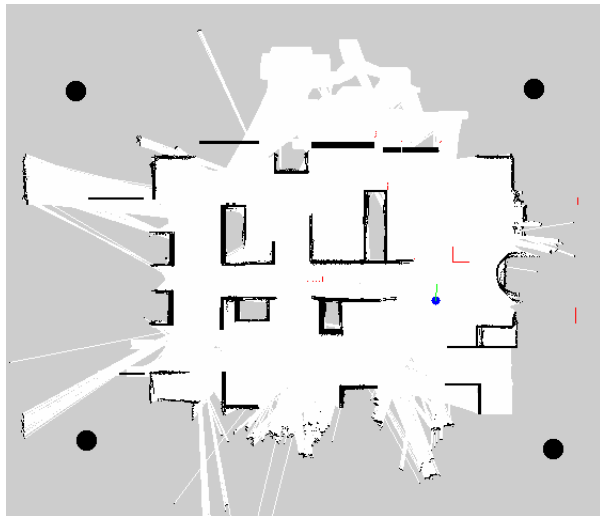


Figure 11. The module's final output.

Topological plan updating

Last step of the re-planning: update the topological plan with the information obtained from the laser obstacle detection node in order to re-elaborate a new pdl file updated to the current state. The idea is to take the current position of the obstacle and eliminate all the arches of the topological plane whose distance between the center of the arch and the obstacle is equal to half the length of the arc itself. In this example, as shown in the Figure 12, three arches near the obstacle will be removed from the topological plan (highlighted in red):

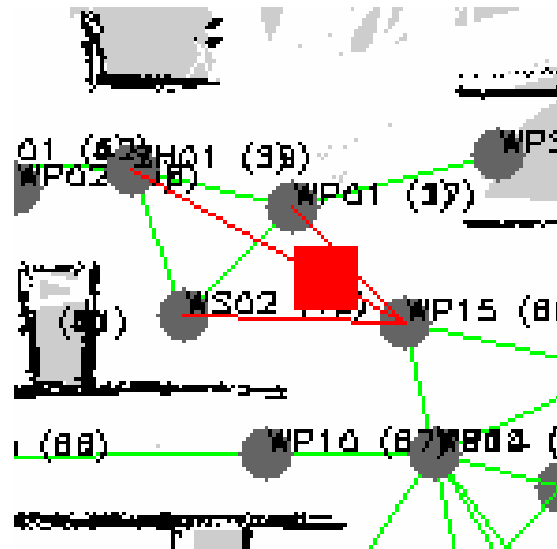


Figure 12. Example of replanning with arches founded.

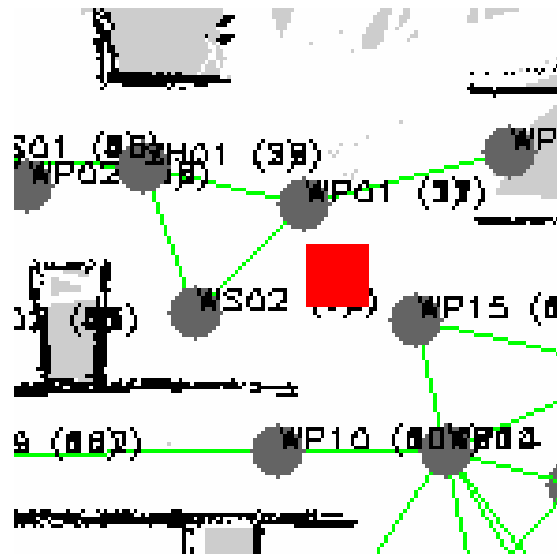


Figure 13. The update of the topological plan.

5. User interface

We now introduce the section dedicated to the graphical interface, a simple way to understand and compare the PDDL planning' solution output available with respect the others planning systems.

GUI explanation

To get into details this GUI is composed by:

- **Environment's map:** if the user insert an image path related to a map in the command line that image is used, as depicted in Figure 14; otherwise, due to the fact that there isn't a path to the map, the system directly takes

the map's data from the *map_server* node, shown on Figure 15.

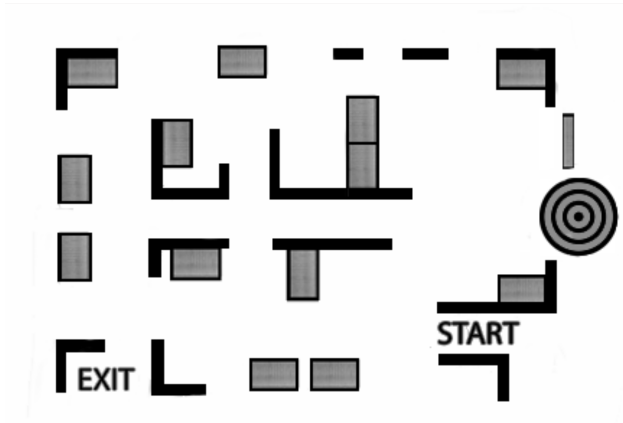



Figure 14. Cleaned map.





Figure 15. Original map.

- **Legend:** figure 16 shows the actual representation of the world, for instance we can see a colored dot on the robot pose with the relative number of planning step inside it. Four possible colors are available:
 - **Blue:** used to show the current robot position in the environment.
 - **Gray:** show all the precedent poses of the robot.
 - **Green:** means that the robot takes an object from the environment in order to put it in one of the free slot available.
 - **Red:** means that the robot takes an object from a specific slot and drops it in the position represented by the dot.
- **Slots:** A dynamic representation of the three youbot's slots, in particular different labels about slot numbers are shown on the top.

Legend:

Current robot pose: 

Robot poses: 

Take: 


Drop: 

Figure 16. Legend.

An empty slot is represented by a green rectangle with a label "empty" inside; otherwise, by a red one with the label used to show the name of the object dropped in this specific slot, as depicted in Figures 17, 18.

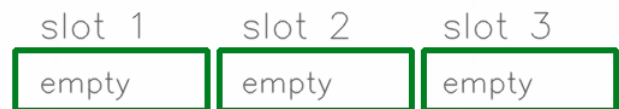


Figure 17. All the robot's slots are empty.

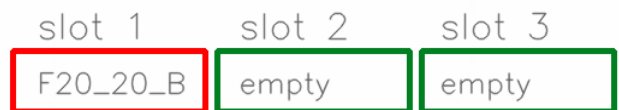


Figure 18. In the first robot's slot there is the object F20_20_B.

- **Planning steps:** the left click of the mouse allows the user to keep the planner moving forward or use the right click to come back. The actions up to the current moment are shown in the console (Figure 19).

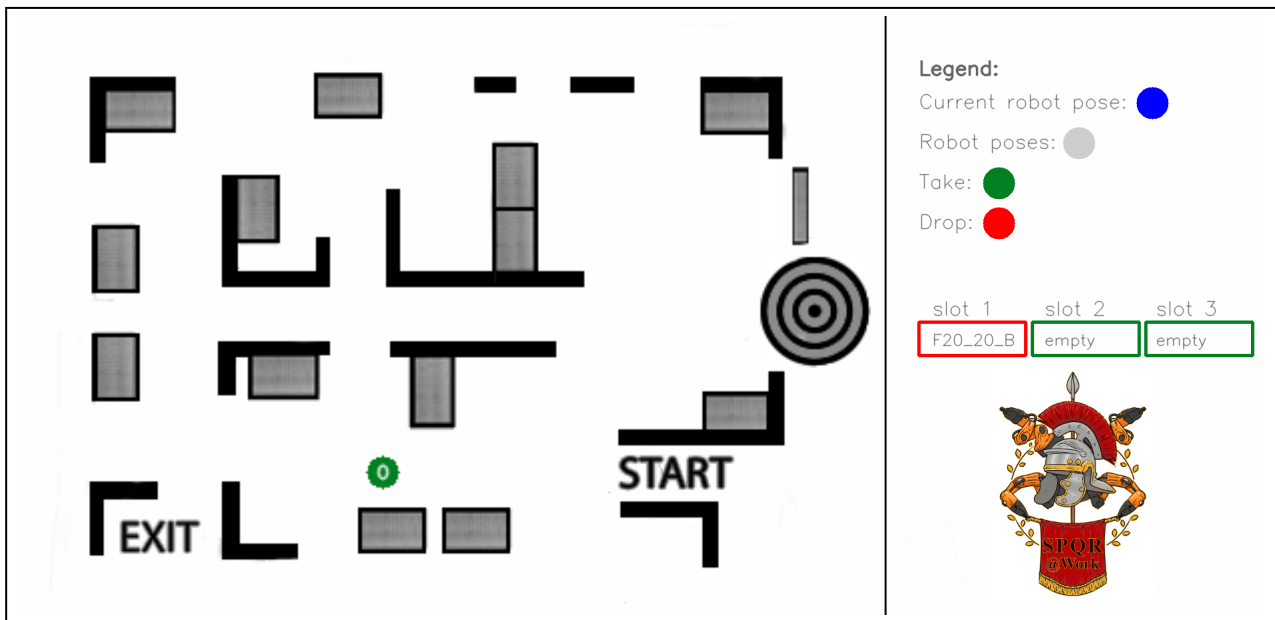

```
action: take object F20_20_B at WS01 from slot1
action: move from WS01 to SH02
action: move from SH02 to WS08
action: take object BEARING at WS08 from slot2
action: take object F20_20_B3 at WS08 from slot3
action: move from WS08 to WS07
action: drop object F20_20_B3 at WS07 from slot3
action: take object R20 at WS07 from slot3
action: move from WS07 to WS06
action: drop object R20 at WS06 from slot3
action: move from WS06 to WS07
action: take object MOTOR at WS07 from slot3
action: move from WS07 to SH02
action: move from SH02 to CB02
action: move from CB02 to WS10
action: drop object F20_20_B at WS10 from slot1
action: drop object BEARING at WS10 from slot2
action: move from WS10 to CB02
action: take object S40_40_B at CB02 from slot1
action: move from CB02 to WS12
action: drop object MOTOR at WS12 from slot3
action: move from WS12 to SH01
action: take object MOTOR1 at SH01 from slot2
action: take object MOTOR2 at SH01 from slot3
action: move from SH01 to SH02
action: drop object S40_40_B at SH02 from slot1
action: move from SH02 to WS03
action: take object BEARING_BOX at WS03 from slot1
action: move from WS03 to WS09
action: drop object BEARING_BOX at WS09 from slot1
action: drop object MOTOR1 at WS09 from slot2
action: drop object MOTOR2 at WS09 from slot3
action: move from WS09 to WS04
action: take object F20_20_G at WS04 from slot1
action: move from WS04 to PP01
action: drop object F20_20_G at PP01 from slot1
```

Figure 19. Planed action's sequence

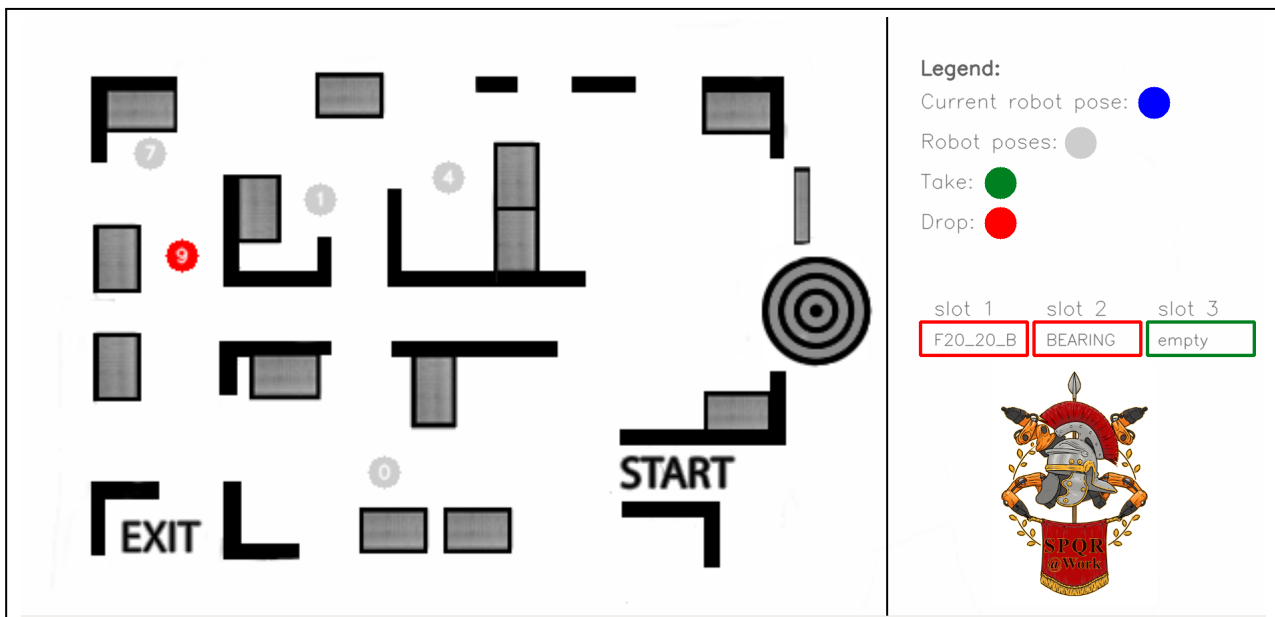
GUI functionality

Let us talk about how the planned actions are implemented:

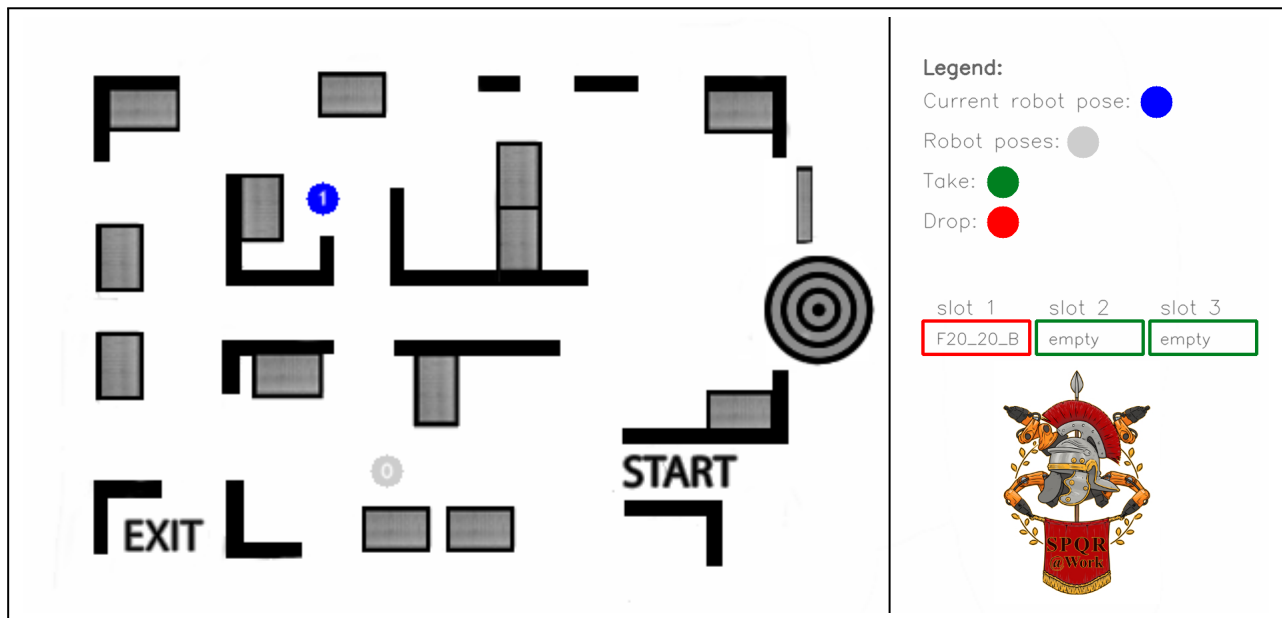
- **Take:** The robot take the F20_20_B



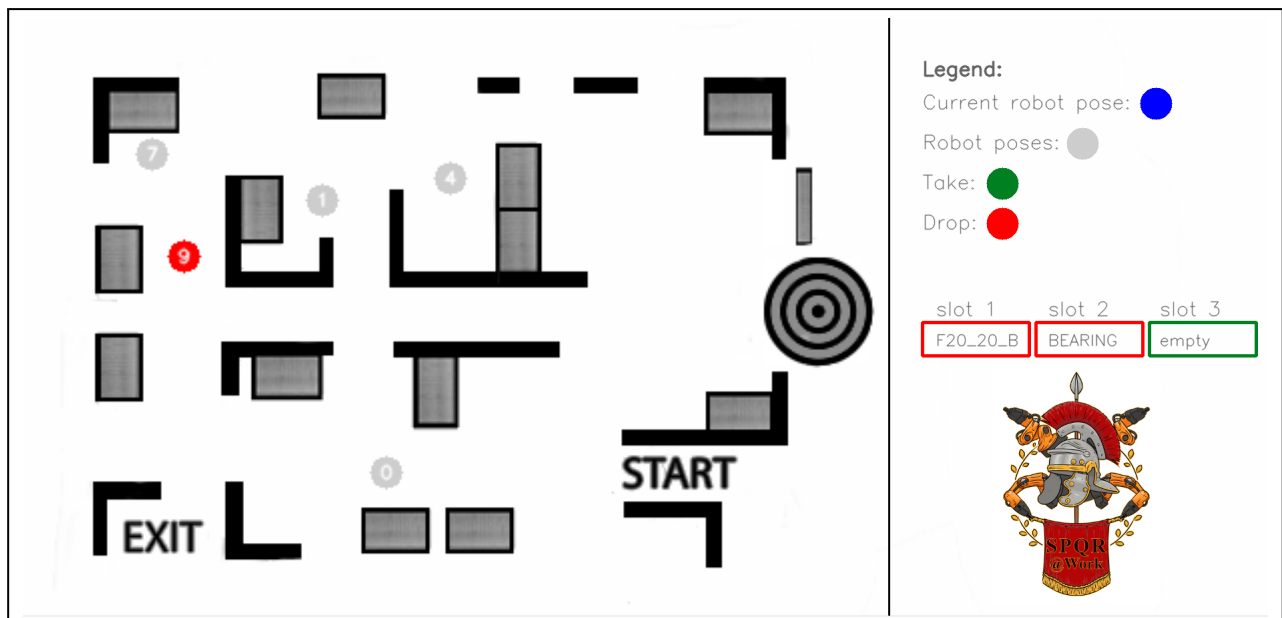
- **Drop:** The robot drops the object that was in the third slot in the current position.



- **Move:** The robot changes position



- **Task completed:** The task is completed and all the objects are in the final position. obviously, the robot will have all the three slots empty.



6. Readme

ROS packages:

Install the following ros packages:

```
sudo apt-get install ros-<your_version>-downward
sudo apt-get install ros-<your_version>-map-server
```

How to compile the repository: Ubuntu 14.04 - 16.04

```
cd <your_ws> catkin_make
```

Modify your .bashrc

In order to compile the ws, you have to add the ws into your .bashrc:

```
source <your_ws>/devel/setup.bash
```

How to run the task_planner

First of all, launch the roscore:

```
roscore
```

Launch the location service.

```
roslaunch location_service loc_service.py
```

Run the map_server with a map:

```
roslaunch map_server map_server <your_map_path> //e.g.:
```

```
roslaunch map_server map_server planning_ws/src/spqr_planning_and_reasoning/spqr_topological_plan/config/maps/map_magdeburg_real.yaml
```

Run the topological_plan:

```
roslaunch spqr_topological_plan spqr_build_topological_graph_node
```

Start the master_plan action_server, in order to generate the problem.pddl based on the current state:

```
roslaunch master_plan_action master_plan_action_server
```

In order to start the planning,

```
cp <your_ws>/src/spqr_planning_and_reasoning/master_planner_action/config/pddl/problem.pddl
<your_ws>/src/spqr_planning_and_reasoning/jsk_planning/pddl/pddl_planner/Robocup_task
```

Move to downward directory

```
cd <your_ws>/src/spqr_planning_and_reasoning/jsk_planning/pddl/pddl_planner/Robocup_task
```

Run downward

```
roslaunch downward plan_domain.pddl ./problem.pddl ipc seq-
sat-lama-2011 --plan-file sample.plan
```

In order to use the viewer, execute the following row:

```
roslaunch map_server map_server <path_to_yaml>
```

```
roslaunch location_service loc_service.py
```

```
roslaunch spqr_task_planner spqr_path_viewer_node <plan_path>
```

```
<optional_path_to_image>
```