International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems
© World Scientific Publishing Company

Probabilistic Task Planning with Action Grouping for Multiple ROS Robots [*]

Dongning Rao
School of Computer, Guangdong University of Technology
Guangzhou 510006, P. R. China
raodn@gdut.edu.cn

Guodong Hu
School of Computer, Guangdong University of Technology
Guangzhou 510006, P. R. China
raodn@gdut.edu.cn

Zhihua Jiang[†]
Department of Computer Science, Jinan University
Guangzhou 510632, P. R. China
tjiangzhh@jnu.edu.cn

Received (22 February 2019)
Revised (22 August 2019)
Accepted (22 November 2019)

The Robot Operating System (ROS) has become a thriving ecosystem. Recently, researchers form the automated planning community devoted to integrate classical planners on the ROS. However, path planning systems and motion planning systems had already been implemented on the ROS. Most of previous automated planning researches on the ROS were trying to reinvent the wheel. On the other hand, for multiple robots, existing studies only focused on task allocation instead of integrating task allocation and action planning for every single robot. Unfortunately, this real-world scene is not suitable for classical planners. For one thing, there are multiple robots and the actions should be taken concurrently. For another, the actions could fail. Therefore, we proposed a probabilistic task planning (ProTaP) approach for multiple ROS robots. The ProTaP is based on concurrent probabilistic planning and provided as a ROS node. By knowledge engineering, the ProTap generates problem files and feeds it along with a domain file to any state-of-the-art concurrent probabilistic planners. Actions in the result plan can be executed by calling ROS services.

Keywords: ROS; concurrent probabilistic planning; knowledge engineering.

[*] This work is supported by the Guangdong Natural Science Foundation (Nos. S2013020012865 and S2011040001427).

[†] Corresponding author.

1. Introduction

The Robot Operating System (ROS) is a flexible framework for writing robot software[a]. As creating truly robust, general-purpose robot software from nothing is hard, the ROS has become the empirical standard. It provided many efficient implemented components and encourages collaborative robotics software development. For example, the Moveit! motion planning framework[b] and the navigation path planning stack[c] are two of the most common used planning components in the ROS. They were often introduced at the very beginning in reference books for the ROS.[1] The ROS ecosystem is growing prosperity. The 47 sponsors of 2018 ROS conference[d] included the top 5 IT companies in the world and the top 5 robot companies in the world along with the ARM and the Amazon.

Automated planning is the sub-area of artificial intelligent which focused on organize actions to achieve goals.[2] The thriving of the ROS ecosystem inspired researchers to facilitate classical planning on the ROS. Action and path planning can be done with classical planners.[3] There are other studies focused on task and motion planning.[4] Moreover, researchers had noticed that classical planning should pay more attention on high level planning.[5] However, many existing researchs tried to reinvent the wheel. They did path planning and/or motion planning along with task planning.

However, real-world problems are often complex. For example, actions have probabilities of failure. On the other hand, real-world robots are also complex. For example, robots have different hardwares, which required device drivers. Further, there could be multiple robots. Unfortunatlely, for multiple robot scenario where the problem is called multiple robots task allocation (MRTA), classical planning is not so handy.[6,7] Moreover, multiple robots lead to concurrent planning because they are autonomous individuals.

To conquer the above issues, we proposed the probabilistic task planning (ProTaP) approach. The ProTaP is designed for multiple robots environment. Actions in the ProTaP have a failure probability and they can be executed concurrently. After identified the initial state and goals, the ProTaP generates problem files by knowledge engineering methods.[8] This file along with a general purpose domain file, will be feed into any state-of-the-art concurrent probabilistic planners. Motion planning for an action and a trip of navigation are all modeled as a single action. Then, the result plan can be executed by calling ROS services. For example, a navigation action is a goto action which will be executed by calling ROS services based on the navigation stack as previous studies.[7]

This paper is organized as follows. After this introduction, backgrounds of this paper will be briefly presented. Then, architecture of the ProTaP will be set up in

[a] http://www.ros.org
[b] https://moveit.ros.org/
[c] http://wiki.ros.org/navigation/
[d] https://roscon.ros.org/2018/

the next Section. The domain file will be explained in a separate section. Detailed algorithms will come right after that, including the problem file generation process. Experiments with analysis were put on the Section before last. Finally, we discussed the ProTap along with a looking to the future.

2. Backgrounds

2.1. *The ROS*

ROS is a standard infrastructure for robots. Developers can use packages that abstract individual sensors or actuators, and messaging infrastructure on the ROS. In addition, ROS integrates a comprehensive set of libraries of open-source software components such as gmapping for Simultaneous Localization and Mapping (SLAM), Moveit! for motion planning and navigation stack for path planning. ROS relies on the concept of computational graph, and a process in ROS is called a node. The nodes are responsible for performing computations and processing data collected from sensors. For example, *movebase* is a node that controls the robot navigation, *amcl* is another node responsible for the localization of the robot, and *mapserver* is a node that provides the map of the environment to other processes of the system. The central node of the communication is referred to as ROS Master, which acts as a name server. Parameters of nodes are saved in a parameter server. In fact, each ROS node is designed for specific task as a service, and the combination of interconnected nodes makes a complex robotic system. Physical data as a labeled data stream is exposed as *topic*. The communications are implemented as *topic* with subscriptions and the messages are packed into *bags*.

MoveIt![10] is a library used for arm manipulation on the ROS. It provides us the plan that the joints have to follow to move an arm from one position to another. It can also take charge of executing the plan. The Moveit! can avoid obstacles by introduce known obstacles in a prior step to planning. Currently, it provides four basic but efficiently implemented planners [e]. Customized planner can be developed and provided to MoveIt! as a plugin. There are many motion planning algorithms even many years ago.[11] However, motion planning algorithms are often most suitable for specific cases. Fortunately, taking advantage of the ROS, these plugins omit the development of drivers. Kernel of the Moveit! package is called *move_group*. It get environment descriptions as $URDF$, SRDF and configuration files. Then, it provide action services to UI and call controllers and sensors to realize these actions.

Navigation stack[12] is another important components of the ROS. It is on a conceptual level, and uses odometry and sensor streams as input. Then, outputs including velocity commands will be sent to a mobile base node. It incorporates both global and local path planners to support ROS-enabled robot navigation. Basic algorithms are defined for the global path planner including $Dijkstra$, $A*$, and

[e] http://moveit.ros.org/documentation/planners/

carrot planners[f]. Like motion planning, path planning is a hot topic with long history.[13] There are many contributors continuously working on improving motion and path planning on the ROS.[14] The core components of the navigation stack is called the *move_base*. The *move_base* node links together a global and local planner to accomplish its global navigation task. Either global or local planner has a costmap as private weighted map. Beside these, the *move_base* node also subscribed to map, odometry and sensor topic and finally implement the navigation actions. Navigation needs a map. Using fusion data to build map let the robot efficiently record the outline of hanging objects and sloping things with wide angle into map.For example, a laser-based SLAM (Simultaneous Localization and Mapping) called *slam_gmapping* is a common used ROS node provided by the ROS. Further, the ROS provided a *multirobot_map_merge* package to merge maps from different robot and its mrpt package supports multi-robot SLAM. Of course, the multi-robot SLAM is also an active research filed.[15] Although the *multimaster_fkie* library is a metapackage to establish and manage multi-master networks, path planning and task planning funtions are still unavailable.

2.2. *Automated planning*

Automated planning studies how to design information processing tools that give access to affordable and efficient planning resources.[2] At the beginning, the community focused only on deterministic planning, where the effects of an action are deterministic. Soon, non-deterministic[16] and probabilistic planning[17] got attentions from researchers. Research of automated planning inluding knowledge representation and problem solving.

Definition language is one of the keypoints for planning knowledge representation. The Planning Domain Definition Language (PDDL) is an attempt to standardize Artificial Intelligence planning languages.[18] It make the 1998/2000 International Planning Competition (IPC) possible, and then evolved with each competition [g]. The PDDL was proposed for deterministic planning, its probabilistic version is called PPDDL (Probabilistic PDDL).[19] However, concurrent actions make the effects globalized. Therefore, the Relational Dynamic Influence Diagram Language (RDDL) is proposed to represent concurrent probabilistic planning domains and problems.[20] The RDDL is rule-based, and its transitions are functions of states and actions. In RDDL, constraints can be put on states or actions. Goals are represented as reward function and come with the domain. The parameters including objects and probabilities along with initial state are defined in the RDDL problem files. The RDDL is the standard representation in the recent International Probabilistic Planning Competitions (IPPC).

Knowledge engineering is another keypoints for planning knowledge represen-

[f] http://wiki.ros.org/global_planner

[g] http://www.icaps-conference.org/index.php/Main/Competitions

tation. Writing domain descriptions is but it is necessary for planning.[8] Therefore, researchers tried to automatically acquire domain descriptions. Not only deterministic planning representations but also action models with probabilistic effects was learned[21] two decades ago. After the RDDL was proposed, we can learned domain descriptions in the RDDL.[22] Considering the cost of getting observations, we can facilitated active learning[23] in this knowledge engineering process.

Searching is the most continent method for problem solving. For deterministic planning, search techniques based on relaxed heuristic,[24] search after SAT encoding[25,26] and bidirectional A* algorithm[27,28] won the IPCs in this century. Nondeterministic planners, especially the probabilistic planners, are mostly based on random local searches. Exsiting RDDL planners either based on iterative deepening algorithms as the Glutton[29] or based on stochastic simulations as the PROST.[30] Both Glutton and PROST were proposed in the IPPC 2011. However, in the IPPC 2018, almost all competitors are successor of the PROST[h].

2.3. *Task planning of Robots*

Robot task planning research is activated for many years. Early studies often combined task planning with motion planning. A straightforward intuition is the reactivation, especially model-based. For example, T-REX[31] is a timeline based plan execution architecture. It based on reactors that modifies state variables on the different timelines. These reactors were build-in in physical components of robots.The system is synchronous and it enforces interdependencies between timelines. Complex hybrid systems use a deliberative high level, a reactive low level, and a synchronisation mechanism to mediate between the other two levels. For example, ROSco[32] and Smach,[33] which use Hierarchical Finite State Machines.

The combination of automated planning and the task planning with motion planning on the ROS comes natural. ROSPlan[34] generates a sequence of actions to achieve the goal when the initial model of the environment is known. It integrates heuristic planners to control behavior of the robots . Replanning mechanisms is used to deal with dynamic real-world problems. It has been successfully deployed on autonomous underwater vehicles.[3] However, probability is an unavoidable issue. For example, the AMPLE framework[35] generates and refines midterm probabilities, then it uses a classical Markov decision process (MDP) planner to plan. Complex hybrid systems can also take advantages from planning. For example, SkiROS[36] is a hybrid framework using classical planner. Hierarchical task network planning was used in hybrid systems[5] too.

Knowledge representation also plays a key role, especially when defining world models. Traditional solutions use their own representations rather than planning definition. For example, translation approach specific to an application.[37] Combining automated planning, domain and problem files are the intermediaries. The

[h] https://ipc2018-probabilistic.bitbucket.io/

ROSPlan requires manual definition of planning domains, but the problem file is automated generated. In SkiROS, the user can define and modify domain and problems from GUI.

In multi-robot enviorment, this problem is multiple robot task allocation (MRTA). There are centralized methods and decentralized solutions. Centralized methods rely on a central controller that allocates tasks to robots and optimal solutions are typically computed using Branch-and-Bound. Decentralized Solutions Decentralized approaches include distributed constraint optimization and market-based algorithms. However, research in stochastic MRTA problems is still sparse. Moreover, unlike believed by previous studies,[7] the ROS has provide supports for multi-robot systems.

3. The Architecture of the ProTaP

The ProTaP is a ROS node. It can be invoked by updated SLAM results. By calling external RDDL planner, the returned plan will be executed as instructions. For the navigation actions, the ProTaP calls navigation nodes in the ROS. For the robot arm actions, the ProTaP calls motion planning nodes in the ROS. For other actions, the ProTaP calls corresponding services if exists. See the figure of the overall architecture.

The overall architecture can be explained in three levels.

First, the robot level. In a multiple robot environment, the robots can communicated by available communication packages. For example, the *multimaster_fkie*[i]. After the SLAM was done, a map will be acquired. Under most circumstances, the communication is available, the multi-robot SLAM node will provide a global map including all robots with environment information as much as possible. The $slam_g mapping$[j] and *multirobot_map_merge*[k] will basically do this job, but more advanced tools might bring better maps.[15] However, the planning is done within and for only one robot. That is, although the plan is for multiple robot, the robot will only execute self relative actions. In this way, the multi-robot system can be much more robust. For example, when communication is down, single robot SLAM can invoke planning for one robot. In this case, assuming single robot environment and continue seems to be the most reliable solution.

Second, the ROS node level. Services in the ROS are provided as nodes. Information about the environment can be get from SLAM nodes. From these nodes, the robot can get the map with an initial state and then transfer them to the task planning node. Voice orders can be decoded by speech recognize nodes. Goals and constraints are inputed in this way, both for the robot and for the task planning node. Then, the task planning node will generate domain and problem files and feed them to RDDL planners. In this first step, we only generate problem files.

[i] http://wiki.ros.org/multimaster_fkie

[j] http://wiki.ros.org/slam_gmapping

[k] wiki.ros.org/multirobot_map_merge

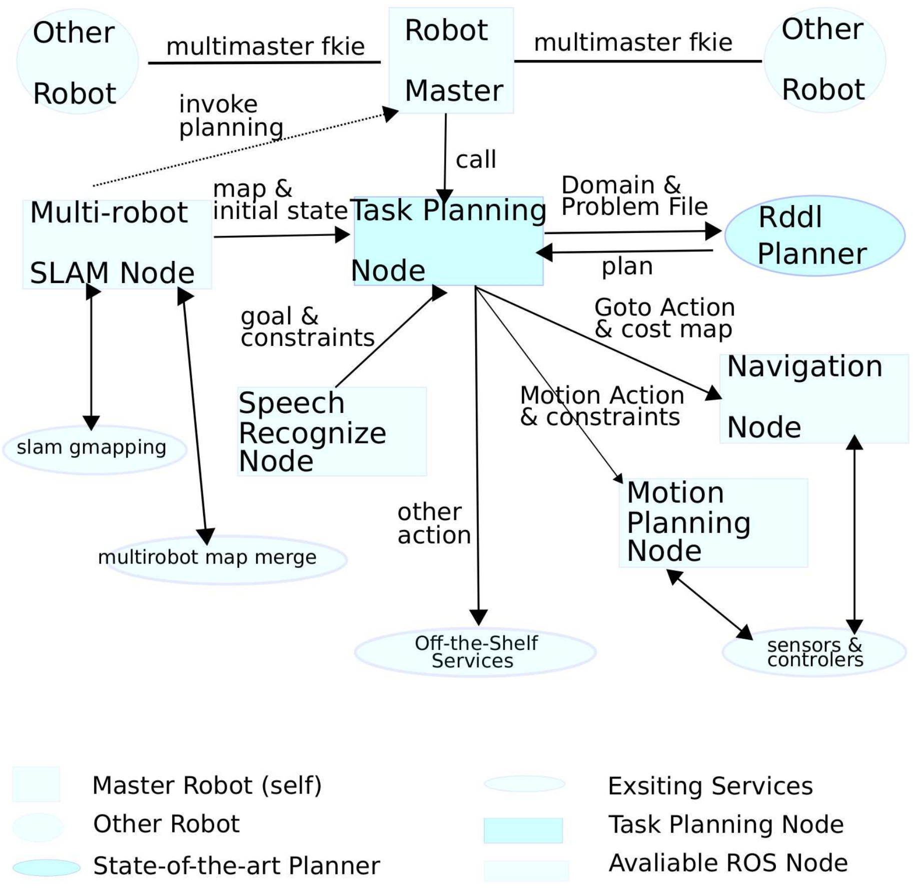

Fig. 1. A overall architecture of the ProTaP.

Normally, these RDDL planners are Linux based. Therefore the planning node uses a system call and wait for the result plan, as existing studies.[4,5,34] The plan steps are action sets. The planning node call services concurrently according to a action sets. That is, once a step. Navigation actions, which are called goto actions in the ProTaP, will be sent to ROS nodes based on the navigation stack. Motion actions will be executed by notifying ROS nodes based on the Moveit! framework. If there are other actions, which can be performed by registered services, can be handled as well.

Third, the services level. The ProTaP does not call services directly, except special services for actions which is not available for existing ROS nodes. SLAM related sensor are called by SLAM nodes, motion controllers and sensors are used by the Moveit! and path navigation related controllers are controled through the navigation stack. If there are sound, it will be processed by speech recognize nodes and provided to the planning node as text messages. In this way, millions of types of sensors[l] and hardware[m] can be facilitated without coding in the ProTaP.

As modern softwares, the ProTap follows three principles as below:

(i) Do not reinvent the wheel. Components of the ROS, as nodes, are contributed by programmers all over the world continuously. Re-implement functions already in these available nodes is unnecessary. In fact, due to too many reasons including commercial interests, calling these nodes is the most efficient way.

(ii) Mind own business but keep up with the times. The ProTaP focused on task planning. Path planning, motion planning and concurrent probabilistic planning are all active research field. Sooner or later, there will be new methods outperform the best existing solutions. Therefore, decoupled these functions will be the best strategy in long term.

(iii) Facing the complex world. There is failure in the real world. Therefore, probability is unavoidable. On the other hand, although separated, there are often many robots in the same frame. The robots are operated concurrently, thus the concurrent becomes a nature of the world.

4. The Target Domain of the ProTaP

The multiple robot probability task planning problems are in a specical domain. In this domain, there are multiple robots. These robots can execute actions concurrently. Moreover, every robots can do several unconflicted actions at the same time. One the other hand, every operation has a probability to fail. The goals are doing specific actions at specific points under specific constraints. These conditions are complexed interweaved. However, when we abstract the navigation task as a macro action called *goto* and simplify the robot arm controlling routine as a macro actions called *do*, the problems can be modeled in a single domain. That is the target domain of the ProTaP.

A formal definition of this domain is defined after basic terms defined as follows.

Definition 1. Fluent f is a relation and non-fluent $\bar{f}$ is a constant. The truth values of f vary from state to state, while that of non-fluent will never change.

Definition 2. The operation set $oper = goto, do, uda$.

(i) *goto* is a macro action for navigation task;

(ii) *do* is a macro action for sequence of arm manipulations for a robot;

(iii) *uda* is a macro action for *user defined actions*;

Definition 3. A constraint c is a universal function mapping any legal experssion to a fluent f, while f must be *true*.

[l] http://wiki.ros.org/Sensors

[m] http://wiki.ros.org/Industrial/supported_hardware

Definition 4. The multiple robot probability task planning domain is a tuple $\Sigma =< F, S, A, P, C, R >$

(i) F is a finite set of *fluents* f *and non-fluent* $\bar{f}$;
(ii) $S \subseteq Power(F)$ is the set of world states;
(iii) $A = oper^n$ is the set of concurrent actions, where *oper* is the operation set and n is the maximum number of concurrent;
(iv) P is a *tensor*, where $p_{ijk} \in P$ is the probability of the transition from $s_i \in S$ with action(s) $a_j \in A$ from $s_j \in S$, and for any i, j we have $\sum_{k=1}^{n} p_{ijk} = 1$;
(v) C is a finite set of constraints c;
(vi) R is a function calculating the reward value from any legal experssion.

The ProTaP uses a template domain file. There are six parts in this file.

- First, the requirements declaration. The target domain demands concurrent actions, and continuous for the compute of expressions. Further, the ProTaP uses state-actions constraints in the RDDL to express constraints. These three requirements are defined as:

```
requirements = {
    continuous,         // continuous is used for experssion computation
    constrained-state,  // this domain uses state constraints
    concurrent          // domain demands concurrent actions
}
```

- Second, object types statement. There are four kinds of objects. The *robot* object is the step-stone for multi-robot planning. The *action* object is the key for handling complex environments, e.g., differnet robots have different abilities. The position objects, *xPos* and *yPos*, are prepared for the navigation grid. There is a goal type objects, which let us define flexiable goals in the problem.

```
    types {
    robot : object; // necessary for multi-robot planning
    action: object; // differnet robots can have different abilities
    xPos  : object; // prepared for the navigation grid
    yPos  : object; // prepared for the navigation grid
    goal  : object; // goals can be defined flexiable
};
```

- Third, *fluent* definition. Actions have costs and goals have utilities, these are the *non-fluents*. The goal itself is a *non-fluent* but whether the goal has been achieved is a *state-fluent*. For our *operation set*, *goto* and *do* actions has to be defined too. Moreover, a *state-fluent* called *robotAt* is prepared for the *goto* action. However, in this first step, the user defined actions, *uda*, are omitted.

```
    pvariables {
```

```
// non-state fluent
  PROB_GOTO(robot) : { non-fluent, real, default = 0.9 };
GOTO_COST(xPos, yPos,xPos, yPos) : { non-fluent, real, default = 1 };
DO_COST(robot,action) : { non-fluent, real, default = 1 };
GOAL_UTILITY(goal) : {non-fluent, real, default = 100};

//goal, is the trajectory problem.
GOAL(goal, action, xPos, yPos): {non-fluent, bool, default = false};

// state-fluent
robotAt(robot, xPos, yPos) : {state-fluent, bool, default = false};
done(goal) : {state-fluent, bool, default = false};

// action fluent
goto(robot, xPos, yPos) : {action-fluent, bool, default = false};
do(robot,action): {action-fluent, bool, default = false};
};
```

- Fourth, the transition functions. For *goto* action, it will change the $robotAt fluents$ with the transition probabilities, P. For *do* action, whether a *goal* is achieved will be affected.

```
cpfs {
    robotAt'(?r,?x,?y) =
        if  (goto(?r, ?x, ?y))
        then Bernoulli ( goto(?r, ?x, ?y) * PROB_GOTO(?r) )
        else
            if (exists_{?xGoto : xPos, ?yGoto : yPos} [goto(?r,
                ?xGoto, ?yGoto)])
            then KronDelta( false )
            else robotAt(?r,?x,?y);

    done'(?g) =
        if (exists_{?r: robot, ?a: action, ?x : xPos, ?y : yPos}
            [do(?r, ?a) ^ GOAL(?g,?a,?x,?y)])
        then KronDelta( true )
        else done(?g) ;
};
```

- Fifth, the reward. Reward is the sum value of all utilities for achieved goal minus the cost of all actions.

```
// Reward is a sum of penalties for those actions, and goals achieved
reward = [sum_{?g : goal} [done(?g)*GOAL_UTILITY(?g)]]
-[sum_{?r : robot, ?a : action} [do(?r, ?a)]*DO_COST(?r, ?a)]
- [sum_{?r : robot, ?x : xPos, ?y : yPos, ?xGoto :xPos, ?yGoto :
    yPos}[[robotAt(?r,?x,?y)^goto(?r, ?xGoto, ?yGoto)]*GOTO_COST(?x,
    ?y, ?xGoto, ?yGoto)]];
```

- Sixth, the constraints. The robots are *mobile*[22] objects. That is, one robot can be at exactly on position at any time. Further, as the nature of our macro actions, the *goto* and *do* actions are all *singletonactions*.[22] That is, every robot can execute at most one *goto* and one *do* actions in the same time.

```
state-action-constraints {
// Robot at exactly one position
forall_{?r : robot} [sum_{?x : xPos, ?y : yPos} robotAt(?r, ?x,?y)]
    <= 1;
// one do action for one robot at a time
forall_{?r : robot} [(sum_{?a : action} [ do(?r,?a)] )<= 1];
// one goto for one robot at a time
forall_{?r : robot} [(sum_{?x : xPos, ?y : yPos} [ goto(?r, ?x, ?y)]
    )<= 1];
};
```

The complete domain file can be found at Appendix.A.

5. Algorithms in the ProTaP

5.1. *The main algorithm*

The main procedure is listed in the following algorithm. It is a wrapper for different services in the ROS environment. It first initialize and start the ROS node. Then, it will acquire initial stats, map, goals and constraints. If we are in real multi-robots environment, these parameters can be get for other ROS nodes. Otherwise, we can generated them randomly. Feeding these parameters into the $generate_{p}roblem$ function, the ProTaP get the problem definition file. Along with the existing domain file, the ProTap call concurrent probabilistic planners for plan. Then the plan will be executed concurrent. Actions will be dispatched to corresponding ROS nodes. For example, the *goto* actions will be sent to navigation nodes and the *do* actions are going to be processed by ROS nodes based on the Moveit! framework. A demonstration code can be found in the Appendix.B.

Algorithm 1: The ProTaP algorithm

```
Input: a tuple < the initial state $I$,
                 a map $cost\_map$,
                 goals $G$,
                 constriants $C$ >
       or $\emptyset$ for demonstration.
Output: A plan $P$ with execution
ros::init();
ros::start();
if in real multi-robots enviroment then
    Subscribe ROS messages ;
else
    Initialize seed for randomization ;
end
if in real multi-robots enviroment then
    Use map $cost\_mat$ and initial state $I$ from Multi-robot SLAM,
      e.g.,$mrpt\_slam$, to set $< x, yrobot >$ tuples;
else
    Set $< x, y, robot >$ tuples randomly ;
end
if in real multi-robots enviroment then
    Use goals $G$ and constraints $C$ to set $< g, x, y >$ tuples;
else
    Set $< g, x, y >$ tuples randomly ;
end
Call $generate\_problem()$ to generate problem ;
Call concurrent probabilistic planners to generate plan ;
Get and output the result plan $P$ ;
while still has plan steps in $P$ do
    Resolve operations from current plan step ;
    foreach operation oper in resolved operations do
        if oper is a goto action then
            Call ROS node based on the navigation stack asynchronous;
        else if oper is a do action then
            Call ROS node based on the Moveit! framework asynchronous ;
        else
            Call user defined services asynchronous ;
    end
end
ros::shutdown();
```

5.2. *The problem file generation algorithm*

The problem file generation algorithm is listed below. In this demonstration, we predefined parameters. However, the parameters for this function can be the addition parameters for this ROS node. In this way, these parameters can be defined at run time. The example problem file is located in Appedix.C.

Algorithm 2: The problem file generation algorithm

```
Input: the initial state $I$, goals $G$
Output: A RDDL problem file
Define the non-fluent section; For predefined number of robots, generate robot objects;
For predefined number of actions, generate action objects;
For predefined range of positions, generate x positions and y positions;
For goals in $G$, generate goal objects;
for every action of a robot, define the action cost;
for every pair of positions, define the navigation cost;
For goals in $G$, define the goal utility;
Define the instance section;
For the initial state $I$, generate true $fluent$ indicating the positions of robots;
Define maxim concurrent number according to the number of robots;
Define the horizon number according to the number of robots and the number of the goals;
Define the discount;
Output the RDDL problem file;
```

6. Experiments

TODO.

Table 1. Comparison of acoustic for frequencies for piston-cylinder problem.

Piston mass	Analytical frequency (Rad/s)	TRIA6-S_1 model (Rad/s)	% Error
1.0	281.0	280.81	0.07
0.1	876.0	875.74	0.03
0.01	2441.0	2441.0	0.0
0.001	4130.0	4129.3	0.16

7. Conclusion

We proposed a probabilistic task planning (ProTaP) approach for multiple ROS robots in this paper. The ProTaP is based on concurrent probabilistic planning and provided as a ROS node. It can generates problem files and feeds it along with a domain file to any state-of-the-art concurrent probabilistic planners. It decoupled task planning with hardware drivers because actions in the result plan are executed by calling ROS services.

Acknowledgements

The authors wish to thank the anonymous referees for their valuable advices. This work is supported by Natural Science Foundation of Guangdong Province of China 2016A030313700 and 2016A030313084

1. A. Koubâa, *Robot operating system (ros): The complete reference*, volume 1 (Springer, 2016).
2. M. Ghallab, D. Nau and P. Traverso, *Automated planning and acting* (Cambridge University Press, 2016).
3. M. Cashmore, M. Fox, D. Long, D. Magazzeni and B. Ridder, "Opportunistic planning in autonomous underwater missions", *IEEE Transactions on Automation Science and Engineering* **15** (2018) 519–530.
4. M. Crosby, R. P. Petrick, C. Toscano, R. C. Dias, F. Rovida and V. Krüger, "Integrating mission, logistics, and task planning for skills-based robot control in industrial kitting applications", in *Proceedings of the 34th Workshop of the UK Planning and Scheduling Special Interest Group (PlanSIG)* (2017), pp. 135–174.
5. J. Á. Segura-Muros and J. Fernández-Olivares, "Integration of an automated hierarchical task planner in ros using behaviour trees", in *Space Mission Challenges for Information Technology (SMC-IT), 2017 6th International Conference on* (IEEE, 2017), pp. 20–25.
6. M. L. Gini, "Multi-robot allocation of tasks with temporal and ordering constraints.", in *AAAI* (2017), pp. 4863–4869.
7. I. Gavran, R. Majumdar and I. Saha, "Antlab: a multi-robot task server", *ACM Transactions on Embedded Computing Systems (TECS)* **16** (2017) 190.
8. K. Wu, Q. Yang and Y. Jiang, "Arms: An automatic knowledge engineering tool for learning action models for ai planning", *The Knowledge Engineering Review* **22** (2007) 135–152.
9. L. Joseph, *Mastering ROS for robotics programming* (Packt Publishing Ltd, 2015).
10. S. Chitta, I. Sucan and S. Cousins, "Moveit![ros topics]", *IEEE Robotics & Automation Magazine* **19** (2012) 18–19.
11. J.-C. Latombe, *Robot motion planning*, volume 124 (Springer Science & Business Media, 2012).
12. J. Fabro, R. Guimares, A. Schneider de Oliveira, T. Becker and V. Amilgar Brenner, *ROS Navigation: Concepts and Tutorial* (2016), volume 625, pp. 121–160.
13. E. Galceran and M. Carreras, "A survey on coverage path planning for robotics", *Robotics and Autonomous systems* **61** (2013) 1258–1276.
14. H. Darweesh, E. Takeuchi, K. Takeda, Y. Ninomiya, A. Sujiwo, L. Y. Morales, N. Akai, T. Tomizawa and S. Kato, "Open source integrated planner for autonomous navigation in highly dynamic environments", *Journal of Robotics and Mechatronics* **29** (2017) 668–684.
15. K.-C. Ma, Z. Ma, L. Liu and G. S. Sukhatme, "Multi-robot informative and adaptive planning for persistent environmental monitoring", in *Distributed Autonomous Robotic Systems* (Springer, 2018), pp. 285–298.

16. A. Cimatti, M. Roveri and P. Traverso, "Automatic obdd-based generation of universal plans in non-deterministic domains", in *AAAI/IAAI* (1998), pp. 875–881.
17. N. Kushmerick, S. Hanks and D. S. Weld, "An algorithm for probabilistic planning", *Artificial Intelligence* **76** (1995) 239–286.
18. D. McDermott, M. Ghallab, A. Howe, C. Knoblock, A. Ram, M. Veloso, D. Weld and D. Wilkins, "Pddl-the planning domain definition language", .
19. H. L. Younes and M. L. Littman, "Ppddl1. 0: An extension to pddl for expressing planning domains with probabilistic effects", *Techn. Rep. CMU-CS-04-162* .
20. S. Sanner, "Relational dynamic influence diagram language (rddl): Language description", *Unpublished ms. Australian National University* (2010) 32.
21. T. Oates and P. R. Cohen, "Searching for planning operators with context-dependent and probabilistic effects", in *AAAI/IAAI, Vol. 1* (1996), pp. 863–868.
22. D. Rao and Z. Jiang, "Learning planning domain descriptions in rddl", *International Journal on Artificial Intelligence Tools* **24** (2015) 1550002.
23. D. Rao and Z. Jiang, "Cost-sensitive action model learning", *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems* **24** (2016) 167–193.
24. J. Hoffmann and B. Nebel, "The ff planning system: Fast plan generation through heuristic search", *Journal of Artificial Intelligence Research* **14** (2001) 253–302.
25. H. Kautz and B. Selman, "Unifying sat-based and graph-based planning", in *IJCAI* (1999), volume 99, pp. 318–325.
26. Z. Xing, Y. Chen and W. Zhang, "Maxplan: Optimal planning by decomposed satisfiability and backward reduction", in *Proceedings of the Fifteenth International Planning Competition, International Conference on Automated Planning and Scheduling* (2006), pp. 53–56.
27. S. Edelkamp and P. Kissmann, "Gamer: Bridging planning and general game playing with symbolic search", *IPC6 booklet, ICAPS* .
28. S. Edelkamp, P. Kissmann and Á. Torralba, "Bdds strike back (in ai planning).", in *AAAI* (2015), pp. 4320–4321.
29. A. Kolobov, P. Dai, M. Mausam and D. S. Weld, "Reverse iterative deepening for finite-horizon mdps with large branching factors", in *Twenty-Second International Conference on Automated Planning and Scheduling* (2012).
30. T. Keller and P. Eyerich, "Prost: Probabilistic planning based on uct.", in *ICAPS* (2012).
31. C. McGann, F. Py, K. Rajan, H. Thomas, R. Henthorn and R. McEwen, "A deliberative architecture for auv control", in *Robotics and Automation, 2008. ICRA 2008. IEEE International Conference on* (IEEE, 2008), pp. 1049–1054.
32. H. Nguyen, M. Ciocarlie, K. Hsiao and C. C. Kemp, "Ros commander (rosco): Behavior creation for home robots", in *Robotics and Automation (ICRA), 2013 IEEE International Conference on* (IEEE, 2013), pp. 467–474.
33. J. Bohren and S. Cousins, "The smach high-level executive [ros news]", *IEEE Robotics & Automation Magazine* **17** (2010) 18–20.
34. M. Cashmore, M. Fox, D. Long, D. Magazzeni, B. Ridder, A. Carrera, N. Palomeras, N. Hurtos and M. Carreras, "Rosplan: Planning in the robot operating system.", in *ICAPS* (2015), pp. 333–341.
35. C. P. C. Chanel, C. Lesire and F. Teichteil-Königsbuch, "A robotic execution framework for online probabilistic (re) planning.", in *ICAPS* (2014).
36. F. Rovida, M. Crosby, D. Holz, A. S. Polydoros, B. Großmann, R. P. A. Petrick and V. Krüger, "SkiROS — A skill-based robot control platform on top of ROS", in *Robot Operating System (ROS): The Complete Reference (Volume 2)*, ed. A. Koubaa (Springer, 2017), volume 707 of *Studies in Computational Intelligence*, pp. 121–160.

37. T. S. Vaquero, S. C. Mohamed, G. Nejat and J. C. Beck, "The implementation of a planning and scheduling architecture for multiple robots assisting multiple users in a retirement home setting.", in *AAAI Workshop: Artificial Intelligence Applied to Assistive Technologies and Smart Environments* (2015).

Appendix A. Domain File

```
////////////////////////////////////////////////////////////////////
//
// Multi ROS Robot Domain
//
// Author: Dongning Rao (raodn [at] gdut.edu.cn)
//
// In a grid, robots must get to goal positions to perform actions and
    avoid obstacles, along with tasks like communication.
// Different robots have random abilities.
// goto action: can call navigation
// moveit action: can call Moveit! for motion plannings
// other action: e.g., communication
////////////////////////////////////////////////////////////////////

domain multi_ros_robot_mdp {

   requirements = {
       continuous,         // continuous is used for experssion computation
       constrained-state,  // this domain uses state constraints
       concurrent          // domain demands concurrent actions
    }
    types {
       robot : object; // necessary for multi-robot planning
       action: object; // differnet robots can have different abilities
       xPos  : object; // prepared for the navigation grid
       yPos  : object; // prepared for the navigation grid
       goal  : object; // goals can be defined flexiable
    };

   pvariables {

      // non-state fluent
      PROB_GOTO(robot) : { non-fluent, real, default = 0.9 };
      GOTO_COST(xPos, yPos,xPos, yPos) : { non-fluent, real, default = 1 };
      DO_COST(robot,action) : { non-fluent, real, default = 1 };
      GOAL_UTILITY(goal) : {non-fluent, real, default = 100};

      //goal, is the trajectory problem.
      GOAL(goal, action, xPos, yPos): {non-fluent, bool, default = false};

      // state-fluent
      robotAt(robot, xPos, yPos) : {state-fluent, bool, default = false};
      done(goal) : {state-fluent, bool, default = false};
    // action fluent
    goto(robot, xPos, yPos) : {action-fluent, bool, default = false};
    do(robot,action): {action-fluent, bool, default = false};

  };

  cpfs {
    robotAt'(?r,?x,?y) =
    if  (goto(?r, ?x, ?y))
    then Bernoulli ( goto(?r, ?x, ?y) * PROB_GOTO(?r) )
    else    if (exists_{?xGoto : xPos, ?yGoto : yPos} [goto(?r, ?xGoto,
       ?yGoto)])
    then KronDelta( false )
    else  robotAt(?r,?x,?y);

    done'(?g) =
    if (exists_{?r: robot, ?a: action, ?x : xPos, ?y : yPos} [do(?r, ?a) ^
       GOAL(?g,?a,?x,?y)])
    then KronDelta( true )
    else
    done(?g) ;
  };

  // Reward is a sum of penalties for those actions, and goals achieved
  // For simplicity, we use the square of the distance as factor, rddl
     does not support absolute or sqrt functions.
  reward = [sum_{?g : goal} [done(?g)*GOAL_UTILITY(?g)]]
  -[sum_{?r : robot, ?a : action} [do(?r, ?a)]*DO_COST(?r, ?a)]
  - [sum_{?r : robot, ?x : xPos, ?y : yPos, ?xGoto :xPos, ?yGoto :
     yPos}[[robotAt(?r,?x,?y)^goto(?r, ?xGoto, ?yGoto)]*GOTO_COST(?x, ?y,
     ?xGoto, ?yGoto)]];

  state-action-constraints {
    // Robot at exactly one position
    forall_{?r : robot} [sum_{?x : xPos, ?y : yPos} robotAt(?r, ?x,?y)] <=
       1;
    // one do action for one robot at a time
    forall_{?r : robot} [(sum_{?a : action} [ do(?r,?a)] )<= 1];
    // one goto for one robot at a time
    forall_{?r : robot} [(sum_{?x : xPos, ?y : yPos} [ goto(?r, ?x, ?y)]
       )<= 1];
  };
}
```

Appendix B. Demo C++ File

```
#include <ros/ros.h>
```

```
#include <stdio.h>
#include <map>
#include <unistd.h>
#include <iterator>
#include <vector>
#include <algorithm>
#include <fstream>
typedef std::vector< std::tuple<int,int,int> > init_tuple_list;
typedef std::vector< std::tuple<int,int,int> > goal_tuple_list;
//typedef std::vector< std::tuple<std::string,std::string>>
    ability_tuple_list;
#ifdef RobotEnv
// include head files
#else
// define constants
#define xRange 5
#define yRange 5
#define numberOfRobot 2
#define robotNamePrefix "robot"
#define numberOfMoveitActions 2
#define moveitActionNamePrefix "MoveitAction"
#define numberOfGoals 3
#define OtherActionName "OtherAction"
#define numberOfOtherAction 1
#define numberOfAbility 3 // This should be <= (numberOfRobot*
    numberOfMoveitActions)

#define probGotoDefault 0.9
#define gotoCostDefault 1
#define probDoDefault 1
#define goalUtilityDefault 100


#define rddlsimPath "/mnt/d/projects/ecl-workspace/rddlsim-master"
#define problemName "multi_ros_robot_inst_mdp__1"
#define tempFilePath "/mnt/d/projects/r2files"
#define problemFile
    "/mnt/d/projects/r2files/multi_ros_robot_inst_mdp__1.rddl"
#define planFile "/mnt/d/projects/r2files/plan.txt"
#define rddlPlannerCall "./run rddl.sim.Simulator /mnt/d/projects/r2files
    rddl.policy.SPerseusSPUDDPolicy multi_ros_robot_inst_mdp__1 | grep \"
    Action taken:\" | cut -c19- > /mnt/d/projects/r2files/plan.txt "

#endif
/*
*  1  Using Multi-robot SLAM, mrpt_slam, to get Map and initial state;
*  2  Geting Goal and Constraints from Speech Command, pocketsphinx;
*  3  Generating RDDL files;
*  4  Call rddl planners to get plan;
*  5  Executing plan, for 3 types of actions:
```

```
*     a)    goto, calling Navigation stack;
*     b)    robot_action, calling Moveit;
*     c)    sending_msg, calling Webservice, Rosbridge ;
*/
//prepared for none-robot Env, two robot start from random position.
std::tuple<int, int, int> get_init_tuple(int id)
{
return std::make_tuple(id, rand() % xRange, rand() % yRange);

}

//prepared for none-robot Env, actions tobe done at random position.
std::tuple<int, int, int> get_goal_tuple(int id)
{

int randomActionNumber = rand() % (numberOfMoveitActions +
    numberOfOtherAction);
return std::make_tuple(randomActionNumber, rand() % xRange, rand() %
    yRange);
}

int generate_problem(const init_tuple_list & itl, const goal_tuple_list &
    gtl){
std::ofstream out;
out.open(problemFile,std::ios::trunc);
int i,j,iGoto,jGoto;
out<<"non-fluents nf_multi_ros_robot_inst_mdp__1 {" <<"\n";
out<<" domain = multi_ros_robot_mdp;" <<"\n";
out<<" objects {" <<"\n";
out<<"   robot : { ";
for(i=1;i<=numberOfRobot;i++)
{
if(i>1) out <<"," ;
out << "r" << i;
};
out <<"};\n";
out<<"   action : { ";
for(i=1;i<=(numberOfMoveitActions+numberOfOtherAction);i++)
{
if(i>1) out <<"," ;
out << "a" << i;
};
out <<"};\n";
out<<"   xPos : { ";
for(i=1;i<= xRange;i++)
{
if(i>1) out <<"," ;
out << "x" << i;
};
out <<"};\n";
```

```
out<<"   yPos : { ";
for(i=1;i<= yRange;i++)
{
if(i>1) out <<"," ;
out << "y" << i;
};
out <<"};\n";
out<<"   goal : { ";
for(i=1;i<= numberOfGoals;i++)
{
if(i>1) out <<"," ;
out << "g" << i;
};
out <<"};\n";
out<<" };" <<"\n";
out<<"non-fluents {" <<"\n";
for(i=1;i<=numberOfRobot;i++)
out << " PROB_GOTO(r" << i <<") = " << probGotoDefault <<";\n";

for(i=1;i<=numberOfRobot;i++)
for(j=1;j<=(numberOfMoveitActions+numberOfOtherAction);j++)
out << " DO_COST(r" << i << ",a"<< j <<") = " << probDoDefault <<";\n";

for(i=1;i<=xRange;i++)
for(j=1;j<=yRange;j++)
for(iGoto=1;iGoto<=xRange;iGoto++)
for(jGoto=1;jGoto<=yRange;jGoto++)
out << " GOTO_COST(x" << i << ", y"<< j
<<", xGoto" << iGoto << ", yGoto" << jGoto
<<") = " << round(100*sqrt(pow(i-iGoto, 2)+pow(j-jGoto,2))) / 100.0 <<";\n";

for(i=1;i<=numberOfRobot;i++)
out << " GOAL_UTILITY(g" << i <<") = " << goalUtilityDefault <<";\n";
i=1;
for(goal_tuple_list::const_iterator it = gtl.begin(); it != gtl.end(); ++it)
out << " GOAL(g" << i++ << ",a"<< std::get<0>(*it) <<", x" <<
    std::get<1>(*it) <<", y" << std::get<2>(*it) <<");\n";
out <<" };\n";
out <<"}\n";


out <<"instance multi_ros_robot_inst_mdp__1 {\n";

out <<" domain = multi_ros_robot_mdp;\n";
out <<" non-fluents = nf_multi_ros_robot_inst_mdp__1;\n";
out <<" init-state {\n";

for(init_tuple_list::const_iterator it = itl.begin(); it != itl.end(); ++it)
out << "   robotAt(r" << std::get<0>(*it) <<", x" << std::get<1>(*it) <<",
    y" << std::get<2>(*it) <<");\n";
```

```
out <<" };\n";

out << " max-nondef-actions = " << floor(2*numberOfRobot) << ";\n";
out << " horizon = 10;\n";
out << " discount = 1.0;\n";
out <<"}\n";

out.close();
return 0;
}

int main(int argc, char **argv) {
ros::init(argc, argv, "prost_wrapper_node");
// Start the node resource managers (communication, time, etc)
ros::start();
// Broadcast a simple log message
ROS_INFO_STREAM("Prost Wrapper Started!");
#ifdef RobotEnv
// subscribe ros messages
#else
/* initialize random seed: */
srand (time(NULL));
#endif

#ifdef RobotEnv
// 1 Using Multi-robot SLAM, mrpt_slam, to get Map and initial state
#else
// seting <x,  y  robot> tuples
init_tuple_list itl;
for (int i=0; i < numberOfRobot; i++)
itl.push_back(get_init_tuple(i));

// TODO: prepare for navi, setting up Map
#endif

ROS_INFO_STREAM("Step Logger: initial state got!");

#ifdef RobotEnv
// 2 Geting Goal and Constraints from Speech Command, pocketsphinx
#else
// seting Goal and Constraints
goal_tuple_list gtl;
for (int i=0; i < numberOfGoals; i++)
gtl.push_back(get_goal_tuple(i));

// seting Constraints, the first one is the ability of robots
// currently, for testing speed, not in the problem file.

// prepare for rddl file
#endif
```

```
ROS_INFO_STREAM("Step Logger: goal got!");
// 3 Generating RDDL files;
// TODO: prepare for rddl planner


generate_problem(itl, gtl);
ROS_INFO_STREAM("Step Logger: problem file got!");

// 4 calling rddl planner


chdir(rddlsimPath);
system(rddlPlannerCall);
std::string actions;
std::vector<std::string> vecOfAct;
// Open the File
std::ifstream pf(planFile);
// Read the next line from File untill it reaches the end.
while (std::getline(pf, actions))
{
// Line contains string of length > 0 then save it in vector
if(actions.size() > 0)
vecOfAct.push_back(actions);
}

ROS_INFO_STREAM("Step Logger: Plan got!");

/*  5  Executing plan, for 3 types of actions:
*     a)    goto, calling Navigation stack;
*     b)    robot_action, calling Moveit;
*     c)    sending_msg, calling Webservice;
*    ONLY actions for one robot will be executed,
*    the actions for THIS robot, default = r1.
*/


#ifdef RobotEnv
// for all actions in the actions, call corresponding services
// Create a client object for moveit or navi service.
ros::ServiceClient moveitActionClient
=nh.serviceClient<MoveitWrapper::Act>("moveit");

ros::ServiceClient naviActionClient
=nh.serviceClient<naviWrapper::Act>("navi");
// the costmap for navi should be initialized, by mrpt_slam

//Create the request and response objects.
NaviWrapper::Act::Request nreq;
NaviWrapper::Act::Response nresp;
```

```
//Fill in the request data members. According to specific planner

//Actually call the service

bool successM=moveitActionClient.call(mreq,mresp);
bool successN=naviActionClient.call(nreq,nresp);

//Check for success and use the response.


#else
// Dummy output
ROS_INFO_STREAM("Doing Actions!");

for(std::vector<std::string>::const_iterator it = vecOfAct.begin(); it !=
    vecOfAct.end(); ++it) {
//  str.replace(str.find(str2),str2.length(),"preposition");
actions=*it;
while(actions.find("___")!=std::string::npos)
actions = actions.replace(actions.find("___"),3,") and ");
while(actions.find("__")!=std::string::npos)
actions = actions.replace(actions.find("__"),2,"(");
while(actions.find("_")!=std::string::npos)
actions = actions.replace(actions.find("_"),2,",");

actions = actions + ")";
ROS_INFO_STREAM(actions);
}
#endif
ROS_INFO_STREAM("Step Logger: Motion and Navi actions done!");
#ifdef RobotEnv
// c)   sending_msg, calling Webservice;
#else
// Dummy output
ROS_INFO_STREAM("Broadcasting Msg: !");
#endif
ROS_INFO_STREAM("Step Logger: other actions done!");

#ifdef RobotEnv
// Process ROS callbacks until receiving a SIGINT (ctrl-c)
// ros::spin();
#endif
// Stop the node's resources
ros::shutdown();
// Exit tranquilly
}
```

Appendix C. Sample Problem File

```
non-fluents nf_multi_ros_robot_inst_mdp__1 {
domain = multi_ros_robot_mdp;
objects {
robot : { r1,r2};
action : { a1,a2,a3};
xPos : { x1,x2,x3,x4,x5};
yPos : { y1,y2,y3,y4,y5};
goal : { g1,g2,g3};
};
non-fluents {
PROB_GOTO(r1) = 0.9;
PROB_GOTO(r2) = 0.9;
DO_COST(r1,a1) = 1;
DO_COST(r1,a2) = 1;
DO_COST(r1,a3) = 1;
DO_COST(r2,a1) = 1;
DO_COST(r2,a2) = 1;
DO_COST(r2,a3) = 1;
GOTO_COST(x1, y1, xGoto1, yGoto1) = 0;
GOTO_COST(x1, y1, xGoto1, yGoto2) = 1;
GOTO_COST(x1, y1, xGoto1, yGoto3) = 2;
GOTO_COST(x1, y1, xGoto1, yGoto4) = 3;
GOTO_COST(x1, y1, xGoto1, yGoto5) = 4;
GOTO_COST(x1, y1, xGoto2, yGoto1) = 1;
GOTO_COST(x1, y1, xGoto2, yGoto2) = 1.41;
GOTO_COST(x1, y1, xGoto2, yGoto3) = 2.24;
GOTO_COST(x1, y1, xGoto2, yGoto4) = 3.16;
GOTO_COST(x1, y1, xGoto2, yGoto5) = 4.12;
......
GOTO_COST(x5, y5, xGoto4, yGoto1) = 4.12;
GOTO_COST(x5, y5, xGoto4, yGoto2) = 3.16;
GOTO_COST(x5, y5, xGoto4, yGoto3) = 2.24;
GOTO_COST(x5, y5, xGoto4, yGoto4) = 1.41;
GOTO_COST(x5, y5, xGoto4, yGoto5) = 1;
GOTO_COST(x5, y5, xGoto5, yGoto1) = 4;
GOTO_COST(x5, y5, xGoto5, yGoto2) = 3;
GOTO_COST(x5, y5, xGoto5, yGoto3) = 2;
GOTO_COST(x5, y5, xGoto5, yGoto4) = 1;
GOTO_COST(x5, y5, xGoto5, yGoto5) = 0;
GOAL_UTILITY(g1) = 100;
GOAL_UTILITY(g2) = 100;
GOAL(g1,a2, x4, y2);
GOAL(g2,a1, x1, y1);
GOAL(g3,a2, x3, y4);
};
}
instance multi_ros_robot_inst_mdp__1 {
domain = multi_ros_robot_mdp;
non-fluents = nf_multi_ros_robot_inst_mdp__1;
init-state {
robotAt(r0, x4, y1);
robotAt(r1, x0, y2);
};
max-nondef-actions = 4;
horizon = 10;
discount = 1.0;
}
```