# Dynamic Temporal Planning for Multirobot Systems

**C. Ugur Usug and Sanem Sariel-Talay**

Computer Engineering Department

Istanbul Technical University, Istanbul, Turkey

{usugc,sariel}@itu.edu.tr

### Abstract

The use of automated action planning techniques is essential for efficient mission execution of mobile robots. However, a tremendous effort is needed to represent planning problem domains realistically to meet the real-world constraints. Therefore, there is another source of uncertainty for mobile robot systems due to the impossibility of perfectly representing action representations (e.g., preconditions and effects) in all circumstances. When domain representations are not complete, a planner may not be capable of constructing a *valid* plan for dynamic events even when it is possible. This research focuses on a generic domain update method to construct alternative plans against real-time execution failures which are detected either during runtime or earlier by a plan simulation process. Based on the updated domain representations, a new *executable* plan is constructed even when the outcomes of existing operators are not completely known in advance or valid plans are not possible with the existing representation of the domain. A failure resolution scenario is given in the realistic Webots simulator with mobile robots. Since TLPlan is used as the base temporal planner, makespan optimization is achieved with the available knowledge of the robots.

## Introduction

The achievement of the overall mission is the highest priority goal of a multirobot system, and this problem is treated previously as *task allocation* and *selection* problem. Although several optimization criteria could be considered for the efficiency of the mission execution (*i.e.*, the cost of task execution is taken into account), the use of automated onboard high-level action planning techniques is crucial for online generation of action sequences while satisfying precedence and resource constraints. This research focuses on continual high-level action planning and scheduling issues for efficient completion of multirobot missions.

Temporal planners, capable of generating efficient sequences of durative actions, are convenient to be used with real robot missions. There have been significant advances in devising efficient temporal planners for continually enhanced benchmark planning domains. However, these algorithms operate at a high level and are not completely capable of dealing with hardware or physical environment limitations. In particular, real world is partially observable and involves different sources of uncertainty due to noisy sensor information, unexpected outcomes of actions and failures.

Failure detection and diagnosis is investigated in earlier planning frameworks (de Jonge, Roos and Witteveen 2009) and plan repairing methods are proposed for recovering from failures (van der Krogt and de Weerdt 2005 and Micalizio 2009). However, in some real-world cases, a planner may not come up with a valid (re)plan with the available operators at hand (Brenner and Nebel 2009, Göbelbecker et al. 2010) but with unforeseen opportunistic features or outcomes of actions. This is due to lack of detailed and realistic representations (*e.g.*, preconditions and effects) of actions (*i.e.*, real-world instantiations of planning language operators) in a planning domain or the abstraction of the problem for reducing complexity. When action representations are incorrect or incomplete as for many real-world problems, learning methods are required. However, even for simple learning methods, a certain amount of background knowledge is needed.

This paper presents a dynamic temporal planning framework for mobile robots to handle action failures due to environmental issues. Different from earlier studies, the presented solution constructs alternative plans even when guidance by experts, relevant new information or repairing operators for replanning is not available. Real outcomes of existing operators may not be completely known in advance (*e.g.*, due to abstraction) or valid plans are not possible with the existing representation of the domain. The proposed framework includes a domain update procedure that provides flexible reasoning tools to replan accordingly for the resolution of a failure by using the intuitive principle of locality of failure. Background knowledge needed for updating the existing domain operators is almost negligible and a generic approach is applied by searching for effects in existing operators that may resolve the failure. After the required domain updates are performed, a replanning approach is employed as opposed to repairing since makespan (near-) optimal solutions are targeted (Cushing, Benton and Kambhampati 2008). TLPlan (Bacchus and Ady 2001) is used as the

forward chaining temporal planner in the system to construct makespan-optimized plans. Low and high-level planning procedures of the robot system are made compatible by using efficient domain (*e.g.*, map) representations.

The rest of the paper is structured as follows. The next section presents the formulation of the investigated problem. The following section describes the dynamic temporal planning framework as a proposed solution to the presented problem and the developed algorithms for domain updates and replanning. The experimental results are presented in the following section and then, the paper is concluded with suggestions for future work.

## Problem Statement

A planning domain is as a tuple $\Delta = (\mathcal{T}, \mathcal{C}, \mathcal{P}, \mathcal{O})$ where $\mathcal{C}$ is a set of constants, $\mathcal{T}$ is a set of types, $\mathcal{P}$ is a set of predicates and $\mathcal{O}$ is a set of planning operators representing real-world actions that can be executed by robots. A planning problem is modeled as $P_r = (\Delta, I, G)$ where $I$ and $G$ are initial and goal states, respectively. Each world state includes a set of facts including the representations of resources and robots in the system ($r_i \in R$). An operator $o \in \mathcal{O}$ is represented with a set of parameters $param(o)$, a set of preconditions $pre(o)$ and add/delete effects $add(o)/del(o)$. An operator $o$ is only applicable in the current state $S$ if $pre(o) \subseteq S$. Whenever $o$ is applied, the world state $S$ is transformed to a successor $S'$ which is represented as $add(o) \cup (S \setminus del(o))$.

A temporal plan $P$, a solution to a problem $P_r$, satisfying $makespan(P)$ optimization with a sequence of instantiated and scheduled operator instances, which correspond to real-world actions ($operator(a) = o$). Each action $a$ is represented with a set of arguments $arg(a)$, a set of allocated robots for execution and a start time $t_s(a)$ in the constructed temporal plan. Action $a$ has $dur(a)$ representing the total duration (*i.e.*, the amount of duration between $t_s(a)$ and the time step that $at - end$ effects are available) of the corresponding real-world action. $valid(P, S, G)$ is a simulation control function (performed prior to execution) of plan $P$ from current state $S$ to achieve goal state $G$. A *valid* plan is a plan for which $valid(P, S, G)$ function value is true. An *executable* plan is a *valid* plan of which real-world execution is possible with the existing resources. In some situations, even when a plan $P$ is *valid*, its runtime execution may fail. In this case, $P$ becomes *non-executable* during the execution of action $a_{failed}$ in state $S_{failed}$. The *cause* of failure is represented as $c \in S_{failed}$ and it has an attribute list of $attr(c)$ defined in $\Delta$ and each element $attr_i$ of this list has a value $value(attr_i, S)$ in $S$. It is assumed that $c$ is detected as just the subject of the action or encountered objects by the low level components (*e.g.*, perception and path planning modules) of the robot. Note that, a *non-executable* plan may still be treated as *valid* from the point of view of the planner even if $c$ is encoded in the planning domain. The corresponding domain update procedure transforms the planning problem from $P_r$ to $P_r'$. $P_r'$ includes the encoded *cause*. Even for $P_r'$, the planner may still fail to find an *executable* plan although an *executable* plan exists if $P_r'$ is transformed to $P_r''$. $P_r''$ should include necessary updates to generate an *executable* plan.

The problem that is investigated in this paper is updating the domain representation appropriately whenever an execution failure occurs (*i.e.*, telling the *cause* of the failure in the knowledge base). The domain updates should be made in such a way to autonomously find an *executable* plan involving alternative actions different than the failed ones. Note that, before the domain is updated, these alternative actions may not be applicable for achieving the goal state. Therefore, an *executable* (also *valid*) plan is to be found at $P_r''$ even when no *valid* plan is found at $P_r'$ or there are several *valid* but *non-executable* plans (*i.e.*, with non-executable actions $a$ for which $pre(a) \subseteq S_{failed}$) generated by replanning.

The presented problem involves two types of action failures, namely, *temporary* and *permanent* failures. *Temporary failures* occur at runtime and can be resolved by replanning. Temporarily failed actions become available whenever replanning is possible in some way. When an action *permanently* fails, there is no way to execute that action to achieve the goals.

As a motivating example to illustrate the above mentioned problem, a single-robot scenario can be given with an object, located on one end of a corridor, to be moved to the other end. The robot is capable of executing several actions such as pick/drop objects by its gripper, move from one location to another and push movable objects. However, since the environment is unstructured, the move action fails *temporarily* while transferring the object due to an unknown obstacle on the pathway to the destination. This failure is detected by the path planner module of the robot after updating its map and path plan accordingly. Although a *valid* plan is impossible for this case, there indeed exists a *valid* and *executable* plan according to which the robot pushes this obstacle until the target is reachable. However, *push* action is not designed to have such a related effect ("clear the pathway to reach at the target") to be included in the replan.

## Dynamic Temporal Planning Framework

One way to handle a real-time execution failure is changing the initial and goal state representations (Fox et al. 2006) to reconstruct a *valid* plan for the new

representation. However, it is not guaranteed that this new plan is *executable*. Additional precautions should be taken to prevent generating such plans (Cushing and Kambhampati 2005). There may be different intuitive solutions to the presented problem: (1) *disabling* the failed actions, (2) *suspending* the failed actions, (3) incremental or subgoal planning. Disabling the failed actions may prevent their use in the constructed plan although needed. The second method overcomes this problem by just suspending the selection of the failed action for a certain time period. In this case, since the duration of suspension is domain-dependent and unpredictable, a complicated reasoning process is needed to estimate the realistic duration for the suspension. Incremental or subgoal planning is another method, however, in this case, suboptimal solutions may be observed or the planner may fail to find a subplan to achieve subgoals.

The solution that is presented in this paper involves a dynamic temporal planning framework for a multirobot system to handle *temporary* action failures. The constructed solution to the presented problem involves a generic reasoning approach about the *cause* of the failure and updating the domain representation appropriately. Depending on the quality of the reasoning about the failure *cause*, the results of the update procedure can be more specific when semantic attachments can be made. For example, if an obstacle-related *cause* is given, only the attributes of *cause* which change the location of the object should be considered for replanning. However, if no such reasoning is available, irrelevant preconditions including attributes such as the color of the obstacle may also be considered. After appropriate domain updates are performed, a new *executable* plan can be constructed if any, even when there is not a *valid* plan with the given operator representations.

The overall framework employs four interleaved processes, namely, planning, execution, monitoring, and recovering from failures if any. Temporal planning is continually performed as in Algorithm 1 for robust continual mission execution. There are four execution states: *start*, *executing*, *suspended* and *failed*. In state *start*, a new plan is constructed and the robots start executing this renewed plan. State *executing* is active whenever robots are in execution of the plan. State *failed* is activated when an action failure occurs, and state *suspended* is activated until the execution is ended completely after state *failed*.

The algorithm starts with state *start* for planning (line 22) using the existing domain representation. TLPlan (Bacchus and Ady 2001) is used as a forward chaining temporal planner to provide makespan optimality at the action planner side. When a plan is constructed, it is sent to the robots.

In state *executing*, the algorithm is suspended on WAITMESSAGE subroutine waiting for any messages from the robots (at line 7). Each message contains the state of action execution for a specified action, perceptual information about the domain and the *cause* of failure if any. Receiving a message, Algorithm 1 resumes and calls the MONITOR subroutine (Algorithm 2). Algorithm 1 remains in state *executing* until a failure message is received from a robot in the domain or VALID$(P, S, G)$ subroutine returns *false* when the corresponding plan becomes *invalid*. In this case, a *stop-execution* message is sent to all robots. Receiving a stop message, each robot stops the execution of its action.

**Algorithm 1** DYNAMICTEMPORALPLANNING $(S, G)$

**Input:** current state $S$, goal state $G$
**Output**: returns success or failure: messages to robots are sent

```
1:  msg = NULL
2:  P = ∅
3:  status = start
4:  for S ⊉ G do
5:    if status = executing then
6:      if VALID(P, S, G)
7:        msg = WAITMESSAGE()
8:        status = MONITOR(msg, S)
9:      else
10:       SENDALLROBOTSSTOPEXECUTION()
11:       status = suspended
12:   else if status = failed then
13:     if VALID(P, S, G) then
14:       UPDATEDOMAINDESCRIPTION(msg, Δ, S, P)
15:     status = suspended
16:   else if status = suspended then
17:     for all robots stop execution do
18:       msg = WAITMESSAGE()
19:       MONITOR(msg, S)
20:     status = start
21:   else
22:     P = PLANNER(Δ, S, G)
23:     if P = ∅ then
24:       return NOPLAN
25:     status = executing
26:     SENDPLANTOROBOTS(P)
27: return SUCCESS
```

Whenever the plan is no longer executable by the robots, the cause of the failure is encoded in the domain representation and state *failed* is activated. Then, the validity of the failed plan is checked for the updated state information. If the failed plan is still treated as valid from the planner's perspective, replanning cannot handle this failure case. In this case, UPDATEDOMAINDESCRIPTION subroutine is called to make the required domain updates and handle the failure. The body of this subroutine is given in Algorithm 3. After the required changes are performed

in the domain representation, Algorithm 1 waits for *execution-stopped* messages from all robots. After then, the state is changed to *start* and a new plan is constructed.

**Algorithm 2** MONITOR $(msg, S)$

**Input:** message $msg$, currrent state $S$
**Output**: status of planning, current state $S$

1: $status = executing$
2: $a = msg.action$
3: $state_a = msg.state$
4: $percv = msg.perception$
5: **if** $state_a = started$ **then**
6: $S =$ APPLYSTARTEFFECTS$(a, S)$
7: **else if** $state_a = finished$ **then**
8: $S =$ APPLYENDEFFECTS$(a, S)$
9: $S =$ MERGE$(S, percv)$
10: **else if** $state_a = failed$ **then**
11: $S =$ MERGE$(S, percv)$
12: $status = failed$
13: **else if** $state_a = stopped$ **then**
14: $S =$ MERGE$(S, percv)$
15: **return** $status$

Algorithm 2 is used to monitor the status of the existing plan and to parse messages coming from robots. Monitoring of the existing plan is performed by simulating the plan from the current world state. The plan is simulated by applying the start and end effects of the actions to the fact base. The perceptions are applied on the current state, when *finished*, *failed* and *stop* messages are received and the status is set to *executing*. However, when a *failed* message is received, it is set to *failed*.

Algorithm 3 implements the appropriate domain updates. At lines (5) to (9), the failed action is *locked* by defining a new predicate (newPredicate) with the name of the operator (concatenated with string "_locked") and its parameters in the domain. This predicate is added to the preconditions of the failed operator. Then, a corresponding fact (newFact) is included in the domain description. At lines (15) to (19), a new pseudo operator ($o_{pseudo}$) is created for the failed action-cause pairs $(a_{failed} - c)$, if not created before. Line (21) assigns the pseudo operator to the corresponding failed action and cause pair. This pseudo operator diverges from the original domain operators by its specific preconditions, effects and parameters for representing the failure *cause*. It has additional preconditions for restricting its selection (for preventing cycling plans) only when the corresponding failed action is *locked* and the related *cause* is given as an argument. It has also additional effects for unlocking the failed action. The preconditions are generated by MAKEFAILURE PRECONDITIONS subroutine (Algorithm 4). This subroutine first searches for the domain operators which possess related effects to the attributes of *cause*.

**Algorithm 3** UPDATEDOMAINDESCRIPTION (msg,Δ,S,$P$)

**Input:** message $msg$, planning domain Δ,
current state $S$, current failed plan $P$
**Output**: updated planning domain Δ

1: $c = msg.cause$
2: $a_{failed} = msg.action$
3: $o_{failed} = operator(a_{failed})$
4: $preList = \emptyset$
5: newPredicate.name = CONCAT$(name(o_{failed}), "_locked")$
6: newPredicate.parameters = param$((o_{failed}))$
7: $pre(o_{failed}) = pre(o_{failed}) \cup \neg\, newPredicate$
8: newFact = GENERATEFACT (newPredicate, $arg(a_{failed})$)
9: $S = S \cup newFact$
10: $\Delta = \Delta \cup newPredicate$
11: $preList = (newParam = c)$
12: $preList = preList \cup newPredicate$
13: $o_{pseudo} =$ GETPSEUDOOPERATOR$(a_{failed}, c)$
14: **if** $o_{pseudo} = NIL$ **then**
15: $param(o_{pseudo}) = param(o_{failed}) \cup newParam$
16: $pre(o_{pseudo}) = preList$
17: $pre(o_{pseudo}) = pre(o_{pseudo})$
18: $\cup$ MAKEFAILUREPRECONDITIONS$(c, \Delta, S, \emptyset)$
19: $del(o_{pseudo}) = del(o_{pseudo}) \cup newFact$
20: $\Delta = \Delta \cup o_{pseudo}$
21: SETPSEUDOOPERATOR$(a_{failed}, c, o_{pseudo})$
22: **else**
23: $a_{pre} =$ GETPREDECESSORACTION$(P, o_{pseudo}, c)$
24: $o_{pre} = operator(a_{pre})$
25: $executedList =$ GETEXECUTEDOPERATORS$(a_{failed}, c)$
26: $executedList = executedList \cup o_{pre}$
27: SETEXECUTEDOPERATORS$(a_{failed}, c, executedList)$
28: $pre(o_{pseudo}) = preList$
29: $pre(o_{pseudo}) = pre(o_{pseudo}) \cup$
30: MAKEFAILUREPRECONDITIONS$(c, \Delta, S, executedList)$

**Algorithm 4** MAKEFAILUREPRECONDITIONS $(c, \Delta, S, l)$

**Input:** failure cause $c$, planning domain Δ,
current state $S$, executed operator list $l$
**Output**: disjunctive precondition list $prelist$

1: $prelist = \emptyset$
2: **for each** $o_i$ in Δ but not in $l$ **do**
3: **for each** *effect* $e_j$ in $add(o_i) \cup del(o_i)$ **do**
4: **for each** *attribute* $attr_k$ in $attr(c)$ **do**
5: **if** $e_j$ affects $attr_k$ **then**
6: $value = value(attr_k, S)$
7: $prelist = prelist \cup \neg(attr_k = value)$
8: **return** $prelist$

Then, it creates a precondition list as a disjunction of the inequality statements between the attributes of $cause$ and

their values in failed state $S_{failed}$. Therefore, these inequality statements ensure enabling the failed actions when the status of cause is changed in a desired way. If the pseudo operator is created earlier, and the plan is still non-executable, that means the alternative action does not produce the desired effect for resolving the failure. In this case, lines (23) to (30) in Algorithm 3 remove the related effects of this alternative action from the list of the preconditions of the pseudo operator. Therefore, it is not considered as an alternative action in future plans. An inappropriate alternative action is not considered in a future plan for the resolution of the related failure.

In the proposed framework, low-level path planning and high-level action planning procedures are successfully integrated by a common representation framework. Since there is a tight coupling between these procedures, topological mapping approach is convenient to be used. This approach provides the desired flexibility in representation and abstraction in both levels. The topological map is represented as a graph of the traversable nodes representing free spaces in the environment and the connecting edges between them. The map is generated incrementally during the mission execution. In this incremental approach, a new node is generated at a certain distance to the previous node (Simmons and Koenig 1995), whenever the robot has sharp turns or detects new obstacles (Ranganathan and Dellaert 2009). When new obstacles are detected on the path, invalid edges are removed from the map. The path planner provides such information to update the map, when pairs of nodes in interest become disconnected. The nodes of the map could also be represented as the facts for locations or objects in the action planning domain. If the failure cause is a discovered obstacle on the path, this information could be encoded in the knowledge base to be used in both levels.

## Experimental Results

A failure resolution scenario is analyzed for presenting the solution in the realistic Webots simulator with two mobile robots. This scenario includes a *temporary* action failure case to test and validate the success of the proposed method. There are 6 operators available in the planning domain: operator *(move-to-loc ?robot ?loc)* is to forward robots to a destination, operator *(move-to-obj ?robot ?obj)* is to forward robots to a position nearby an object to act on it, operators *(pick ?robot ?smallObj)* and *(drop ?robot ?smallObj)* are for picking/dropping a small object by the grippers of the robots, operator *(push ?robot ?largeObj)* is used to change the position of a large object by dragging, and operator *(paint ?robot ?obj ?color)* to paint an object.

The overall goal in the planning problem is transferring the small red objects to a target location behind the obstacle. However, the robots are not informed about the existence of the obstacle which blockades the corridor.

During the execution of the *move-to-loc* action, one of the robots detects the obstacle and the path planner returns a failure since it is impossible to generate a path from the current position to the destination. The failure *cause* of the action *move-to-loc* is reported as *obstacle* with its estimated location. This new information is encoded in the knowledge base ($P_r'$). If there is a valid solution at this step, replanning is performed. At this moment, it is assumed that there is no knowledge of how to resolve this failure (*i.e.*, operator *push* would not be selected by the planner since it does not have an effect "clear the pathway to reach at the target"). Even when replanning is not possible, the proposed generic domain update method is capable of creating an alternative plan by means of the UPDATEDOMAINDESCRIPTION subroutine. This subroutine adds a pseudo operator for the reported *cause* to the domain to transform to a new planning problem $P_r''$.

```
(def-adl-operator (move-to-loc ?loc)
    (pre
        ...
       (not (move-to-loc_locked ?loc))
    )
    ...
)
(def-adl-operator (pseudo1 ?obj ?loc)
    (pre
       ...
    (and
            (= ?obj obstacle)
            (move-to-loc_locked ?loc)
            (or
              (not (xcoord ?obj obs_x))
              (not (ycoord ?obj obs_y))
              (not (color ?obj red))
        )
    )
    )
    (del (move-to-loc_locked ?loc))
)
```

Figure 1: Domain updates for the resolution of the failure.

MAKEFAILUREPRECONDITIONS subroutine searches for the operators in the domain and finds *push* and *paint* as related operators to the *cause* (*obstacle)* since they have the effects to change the attributes of an object. A new pseudo operator *pseudo1* is created with preconditions related to both the failed action (*move-to-loc*) and the effects *x* and *y coordinates* and *color* of the operators *push* and *paint*, respectively. The domain is updated with the inclusion of a disabling fact (*move-to-loc_locked obstacle*) of the failed action and the new pseudo operator. All domain updates including the change made on the disabled action are illustrated in Figure 1. Overall plan execution is illustrated in Figure 2. Based on the updated domain information, a new *executable* plan is generated. The new plan includes *push* action for the first robot. Action *paint* is not included in the final plan since its cost is higher than that of *push*. If proper reasoning is possible, the selection

of action *push* could be enforced. Note that, the duration of action *push* is to be specified before the construction of the plan. In the implementation, this value is calculated based on the estimated length of the corridor. The incrementally constructed topological map for the first robot is presented in Figure 3. Just a single robot's map is illustrated for simplification purposes.

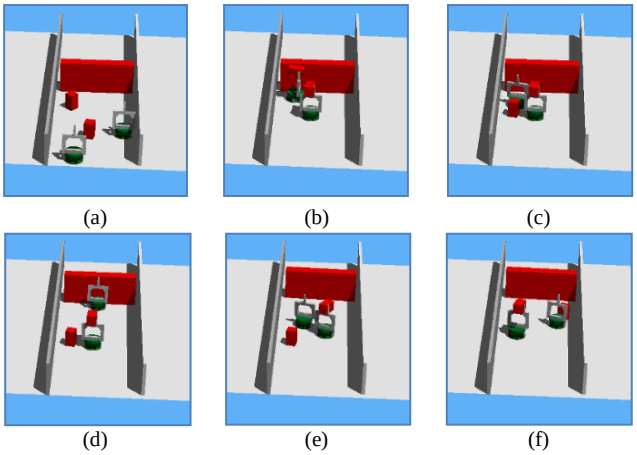


Figure 2: The overall plan execution of moving red objects to their destinations. (a-b) Initial plan is executed. In (b), the domain updates are performed due to the failure and a new plan is generated. (c-f) the new plan involves pushing the obstacle to an appropriate location before red objects are moved.

## Conclusion

In this paper, the realistic constraints of robotic mission execution are investigated and a dynamic replanning framework is proposed against environmental failures. The proposed method can efficiently handle *temporary* failures. Appropriate domain updates are performed to replan and generate alternative *executable* plans even when the domain representation is not completely specified. An example failure resolution scenario in Webots simulator is given to validate the proposed approach. As the simulation scenario illustrates, *executable* plans are generated by replanning with new pseudo operators in the updated domains. These initial experiments in the given scenario illustrate that the proposed domain update method and the dynamic replanning framework is promising for robust mission execution of robotic systems. The near future work includes extending the algorithm to handle *permanent* failures and porting the framework on real robots.

## Acknowledgments

Authors would like to thank Dogan Altan, Sami Dikici and Sertac Karapinar for their contribution in the simulation experiments.

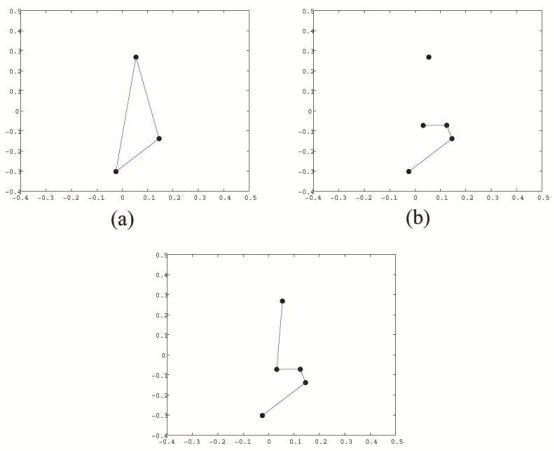


Figure 3: The topological map is created and updated incrementally. (a) The nodes of the graph represent the robot, object, and target locations. (b) The path planner recognizes the obstacle on the path to the target and updates the map. (c) Whenever the obstacle is removed from the path, the map is updated accordingly.

## References

Bacchus, F., and Ady, M. 2001. Planning with resources and concurrency: A forward chaining approach. In *Proceedings of IJCAI*, 417-424.

Brenner, M., and Nebel, B. 2009. Continual planning and acting in dynamic multiagent environments. In *AAMAS*, 19(3):297-331

Cushing W., and Kambhampati S. 2005. Replanning: A new perspective. In *Proceedings of ICAPS*.

Cushing, W.; Benton, J.; and Kambhampati, S. 2008. Replanning as a deliberative re-selection of objectives. *Arizona State University, Tech. Rep*.

de Jonge, F.; Roos, N.; Witteveen, C. 2009. Primary and secondary diagnosis of multi-agent plan. In *AAMAS*, 18(2): 267-294.

Fox, M.; Gerevini, A.; Long, D.; and Serina, I. 2006. Plan stability: Replanning versus plan repair. In *Proceedings of ICAPS*, 212-221.

Göbelbecker, M.; Keller, T.; Eyerich, P.; Brenner, M.; Nebel, B. 2010. Coming up with good excuses: What to do when no plan can be found. In *Proceedings of The International Conference on Automated Planning and Scheduling (ICAPS)*, 81-88.

Micalizio, R. 2009. A distributed control loop for autonomous recovery in a multiagent plan. In *Proceedings of IJCAI*, 1760-1765.

Ranganathan A., and Dellaert F. 2009. Bayesian surprise and landmark detection. In *Proceedings of ICRA*, 1240-1246.

Simmons R., and Koenig S. 1995. Probabilistic Robot Navigation in Partially Observable Environments. In *Proceedings of IJCAI*, 1080-1087.

van der Krogt, R., and de Weerdt, M. 2005. Plan repair as an extension of planning. In *Proceedings of The International Conference on Automated Planning and Scheduling (ICAPS)*, 161-170.