# Multi-Domain Real-time Planning in Dynamic Environments Supplementary Document

Mubbasir Kapadia*[1], Alejandro Beacco†[2], Francisco Garcia‡[3], Vivek Reddy§[1], Nuria Pelechano¶[2], and Norman I. Badler‖[1]

[1]University of Pennsylvania
[2]Universitat Politècnica de Catalunya
[3]University of Massachusetts Amherst

## 1 Events

Tasks monitor the following events:

START_CHANGED: A task receives this event when the start state changes. This event may be triggered when the agent moves, changing its current position – or by the propagation of changes through the task dependency chain. This event triggers an update to account for the change in start state, requiring plan refinement, and potentially invalidating the current solution if the new start position does not lie along the path.

GOAL_CHANGED: A task receives this event when the desired goal state changes. This event triggers an update to account for change in goal and invalidates the current plan. When waypoints along $\Pi(\Sigma_2)$ change, it triggers start and goal updates for tasks in $\Sigma_3$ which are responsible for generating a path between the waypoints.

WORLD_CHANGED: Different domains account for world changes at different levels. The global navigation mesh domain $\Sigma_1$ which considers only static immovable geometry does not monitor this event. The dynamic navigation mesh domain $\Sigma_2$ keeps track of the number of dynamic objects in each polygon of its triangulation which contributes to the cost of the traversal. The grid domain $\Sigma_3$ accounts for the current position of obstacles and agents in its plan vicinity. The space-time domain $\Sigma_4$ monitors for deviation in the plans of neighboring agents which it accounts for while planning. Note that event registration for WORLD_CHANGED is based on spatial and temporal locality. Tasks monitor this event only for aspects of the environment that may change the current plan or are contained in the visibility frustum of the agent. This ensures that planners only consider changes in the environment of interest which require an update.

TUNNEL_CHANGED: Planners can exploit plans in one domain in order to accelerate searches in another domain. For example, the path computed in $\Sigma_3$ can be used to focus and accelerate the search in $\Sigma_4$. Tasks with this dependency must monitor other tasks and repair its own solution when the plan changes. Section **??** describes the use of tunnel search.

PLAN_STATUS_CHANGED: The status of the plan is monitored by the task itself (requiring a change in task priority) and by the task that it is dependent on. An invalid or sub-optimal solution gives the task a higher priority while an optimal solution does not require any further processing. If a task is unable to come up with a solution, it requires a change in task parameters (e.g. increasing the tunnel width to increase the search focus) or it means that current problem definition cannot be solved, requiring a new problem definition from the task higher up in the task dependency chain.

*mubbasir@seas.upenn.edu
†abeacco@lsi.upc.edu
‡fmaxgarcia@gmail.com
§vivreddy@seas.upenn.edu
¶npelechano@lsi.upc.edu
‖badler@seas.upenn.edu

## 2 Algorithmic Details for Planning Task

```
1  key($s$)
2      if $g(s) > rhs(s)$ then
3          return $[rhs(s) + \epsilon \cdot \mathrm{h}(s, s_{start}); rhs(s)]$
4      else
5          return $[g(s) + \cdot \mathrm{h}(s, s_{start}); g(s)]$
6  UpdateState($s$)
7      if ($s \neq s_{goal}$) then
8          $s' = \arg_{s' \in \mathrm{pred}(s)} \min(\mathrm{c}(s, s') + g(s'))$
9          $rhs(s) = \mathrm{c}(s, s') + g(s')$
10         $prev(s) = s'$
11     if ($s \in$ OPEN) remove s from OPEN
12     if $g(s) \neq rhs(s)$ then
13         if ($s \notin$ CLOSED) insert s in OPEN with key(s)
14         else insert s in INCONS
15     Insert $s$ in VISITED
16 ComputeOrImprovePath ($t_{max}$)
17     while ($\min_{s \in OPEN}(key(s) < key(s_{start}) \vee rhs(s_{start}) \neq$
       $g(s_{start}) \vee \Pi(s_{start}, s_{goal}) = NULL) \wedge t < t_{max}$ do
18         $s = \arg_{s \in \mathrm{OPEN}} \min(key(s))$
19         if ($g(s) > rhs(s)$) then
20             $g(s) = rhs(s)$
21             CLOSED = CLOSED $\cup\ s$
22         else
23             $g(s) = \infty$
24             UpdateState($s$)
25         foreach $s' \in succ(s)$ do
26             UpdateState($s'$)
27 ExecutePlanTask ($t_{max}$)
28     Move states from INCONS to OPEN
29     CLOSED = NULL
30     if START_CHANGED then StartChangeUpdate ($s_c$)
31     if GOAL_CHANGED then GoalChangeUpdate ($s_{new}$)
32     if WORLD_CHANGED then
33         foreach (obstacle change $s \rightarrow s'$ ) ObstacleChangeUpdate ($s$,$s'$)
34     if TUNNEL_CHANGED then
35     TunnelChangeUpdate ($\Pi'(\Sigma_{ld}, s_{start}, s_{goal})$)
36     ComputeOrImprovePath ($t_{max}$)
37     trigger PLAN_STATUS_CHANGED
```

**Algorithm 1**: AD* Planner used to compute and update paths for planning tasks $T(\Sigma)$ in each of the 4 domains.

**ExecutePlanTask** (Algorithm **??** [28–37]) is invoked each time the planning task is executed. This function monitors events and calls the appropriate event handlers, described in Algorithm **??**. Given a maximum amount to deliberate $t_{max}$, it refines the plan and publishes the $\epsilon$-suboptimal solution using the AD* planning algorithm [**?**]. We briefly describe our implementation of the AD* algorithm and how we handle changes in start, goal, obstacle movement, and tunnel updates, and refer the readers to [**?**] for more details.

AD* performs a backward search and maintains a least cost path

```
1  StartChangeUpdate ($s_c$)
2    if $s_c \notin \Pi(s_{start}, s_{goal}) \wedge \boldsymbol{d}(s_c, \Pi(s_{start}, s_{goal})) > t_{max}$ then
3        ClearPlanData()
4        $\epsilon = \epsilon_0$
5    else
6        $s_{start} = s_c$
7        foreach $s \in OPEN$ do
8            Update key($s$)
9  GoalChangeUpdate ($s_{new}$)
10   ClearPlanData()
11   $\epsilon = \epsilon_0$
12   $s_{goal} = s_{new}$
13 ObstacleChangeUpdate ($s$,$s'$)
14   if $s' \in \Pi(s_{start}, s_{goal})$ then
15       $\Pi(s_{start}, s_{goal}) = \Pi(s_{start}, s_{goal})$ - $s'$
16       $\epsilon = \epsilon_0$
17   if $pred(s) \bigcap VISITED \neq NULL$ then UpdateState($s$)
18   $g(s') = \infty$
19   if $s' \in CLOSED$ then
20       foreach $s'' \in succ(s')$ do
21           if $s'' \in VISITED$ then UpdateState($s''$)
22 TunnelChangeUpdate ($\Pi(s_{start}, s_{goal})$)
23   foreach $s \in VISITED$ do
24       if $|\boldsymbol{d}(s, \Pi(s_{start}, s_{goal}))| > t_w$ then
25           $g(s) = \infty$
26           if $s \in CLOSED$ then
27               foreach $s' \in succ(s)$ do
28                   if $s' \in VISITED$ then UpdateState($s'$)
29       else
30           $h_t(s, s_{start}) = h(s, s_{start}) + |\mathbf{d}(s, \Pi(s_{start}, s_{goal}))|$
```

**Algorithm 2**: Event handlers for change in start state, goal state, environment, and tunnel.

from the goal $s_{goal}$ to the start $s_{start}$ by storing the cost estimate $g(s)$ from $s$ to $s_{goal}$. However, in dynamic environments, edge costs in the search graph may constantly change and expanded nodes may become inconsistent. Hence, a one-step look ahead cost estimate $rhs(s)$ is introduced [**?**] to determine node consistency.

$$rhs(s) = \begin{cases} 0 \text{ if } s = s_{goal} \\ \arg\min(c(s, s') + g(s')) \text{ else} \end{cases} \tag{1}$$

The priority queue OPEN contains the states that need to be expanded for every plan iteration, with the priority defined using a lexicographic ordering of a two-tuple **key**($s$), defined for each state. OPEN contains only the inconsistent states ($g(s) \neq rhs(s)$) which need to be updated to become consistent. Nodes are expanded in increasing priority until there is no state with a key value less than the start state. A heuristic function $h(s, s')$ computes an estimate of the optimal cost between two states, and is used to focus the search towards $s_{start}$.

Instead of processing all inconsistent nodes, only those nodes whose costs may be inconsistent beyond a certain bound, defined by the inflation factor $\epsilon$ are expanded. It performs an initial search with an inflation factor $\epsilon_0$ and is guaranteed to expand each state only once. An INCONS list keeps track of already expanded nodes that become inconsistent due to cost changes in neighboring nodes. Assuming no world changes, $\epsilon$ is decreased iteratively and plan quality is improved until an optimal solution is reached ($\epsilon = 1$). Each time $\epsilon$ is decreased, all states made inconsistent due to change in $\epsilon$ are moved from INCONS to OPEN with **key**($s$) based on the reduced inflation factor, and CLOSED is made empty. This improves efficiency since it only expands a state at most once in a given search and reconsidering the states from the previous search that were inconsistent allows much of the previous search effort to be reused, requiring only a minor amount of computation to refine the solution. **ComputeOrImprovePath** (Algorithm **??** [16–26]) gives the routine for computing or refining a path from $s_{start}$ to $s_{goal}$.

When change in edge costs are detected, new inconsistent nodes are placed into OPEN and node expansion is repeated until a least cost solution is achieved within the current $\epsilon$ bounds. When the environment changes substantially, it may not be feasible to repair the current solution and it is better to increase $\epsilon$ so that a less optimal solution is reached more quickly.

An increase in edge cost may cause states to become under-consistent ($g(s) < rhs(s)$) where states need to be inserted into OPEN with a key value reflecting the minimum of their old cost and their new cost. In order to guarantee that under-consistent states propagate their new costs to their affected neigbhors, their key values must use uninflated heuristic values. This means that different key values must be computed for under- and over-consistent states, as shown in Algorithm **??** [1 – 5]. This key definition allows AD* to efficiently handle changes in edge costs and changes to inflation factor.

AD* uses a backward search to handle agent movement along the plan by recalculating key values to automatically focus the search repair near the updated agent state. It can handle changes in edge costs due to obstacle and start movement, and needs to plan from scratch each time the goal changes. The routines to handle change in start, goal, and world changes are described below.

**StartChangeUpdate**. When the start moves along the current plan, the key values of all states in OPEN are recomputed to re-prioritize the nodes to be expanded. This focuses processing towards the updated agent state allowing the agent to improve and update its solution path while it is being traversed. When the new start state deviates substantially from the path, it is better to plan from scratch. Alg **??** [1–8] provides the routine to handle start movement.

**GoalChangeUpdate**. Alg **??** [9–12] clears plan data and resets $\epsilon$ whenever the goal changes and plans from scratch at the next step.

**ObstacleChangeUpdate**. Alg **??** [13–21] handles change in obstacles. An obstacle movement from $s$ to $s'$ results in a free state at $s$ and an invalidation of the previously valid state $s'$. Nodes in the vicinity of the obstacle movement (i.e., successors of $s$ and $s'$) become inconsistent and may have invalid references to $s'$, which is no longer free, requiring them to be updated. If the obstacle movement invalidates the current plan, we reset $\epsilon$ to quickly produce a valid path at the next step, which can be refined in subsequent iterations.

**TunnelChangeUpdate**. This routine is used when the planning task monitors the computed path of another planning task $T(\Sigma_{ld})$ in a lower-dimensional domain to focus and accelerate its own searches, as described in Section **??**.

## 3 Performance of Tunnel Search

We evaluate the performance of tunnel-based search on 100 randomly sampled problem definitions (environment configuration, start and goal state) in the $\Sigma_4$ with a constraint enforcing the maximum Euclidean distance between $s_{start}$ and $s_{goal}$ to be 20 grid units. This corresponds to comparable problem definitions for these planners in our multi-domain framework. For a given problem instance, we first execute $T(\Sigma_3)$ to generate a spatial path $\Pi(\Sigma_3)$ which is used to focus the search in $T(\Sigma_4, \Pi(\Sigma_3))$. In addition, we solve the problem instance without a tunnel constraint to provide a basis for comparison.

Table **??** provides the number of nodes expanded and the total planning time for the three planning tasks. The aggregate performance of using $T(\Sigma_3)$ and $T(\Sigma_4, \Pi(\Sigma_3))$ is provided for reference. We notice that tunnel greatly expedites the search process by expanding $4X$ fewer nodes, and providing a $3X$ performance boost on average. Out of the $100$ scenarios, $8$ scenarios resulted in the tunnel search not being able to initially find a solution and had to increase its tunnel width and replan. Even in these cases, the tunnel search outperformed $T(\Sigma_4)$. For 3 scenarios, $T(\Sigma_4)$ could not generate a solution within the maximum time allotment of $100ms$. These were problem instances with a local minima where the search heuristic alone falsely focused the search down a wrong path but the use of the tunnel mitigated the need of the exploration of local minima in the space-time domain.

| **Planning Task** | **# of nodes** | **Time (ms)** |
|---:|:---:|:---:|
| $T(\Sigma_3)$ | 47 | 3.5 |
| $T(\Sigma_4, \Pi(\Sigma_3))$ | 246 | 7.3 |
| $T(\Sigma_3) + T(\Sigma_4, \Pi(\Sigma_3))$ | 293 | 10.8 |
| $T(\Sigma_4)$ | 1041 | 21.2 |

**Table 1:** *Performance evaluation of using tunnel based search.*