# Dynamic Repairing A$^*$: a Plan-Repairing Algorithm for Dynamic Domains

Filippos Gouidis[1,2], Theodoros Patkos[1], Giorgos Flouris[1] and Dimitris Plexousakis[1,2]
[1] *Institute of Computer Science, FO.R.T.H, Heraklion, Crete, Greece*
[2] *Department of Computer Science, University of Crete, Heraklion, Crete, Greece*
{*gouidis, patkos, fgeo, dp*}@*ics.forth.gr*

Keywords: Planning, Plan repairing, Multi-agent systems, Graph search

Abstract: Re-planning is a special case of planning which arises when already produced plans become invalidated before their completion. In this work we investigate the conditions under which plan repairing is more efficient than re-planning from scratch. We present a new plan-repairing algorithm, Dynamic Repairing A$^*$(DRA$^*$), and we compare its performance against $A^*$ in a number of different re-planning scenarios. The experimental results indicate that if the percentage of the plan that has been already executed is less than 40% to 50% and the changes in the environment are small or moderate, *DRA*$^*$ outperforms $A^*$ in terms of speed by a factor of 10% to 80% in the majority of the cases.

## 1 INTRODUCTION

*Re-planning* is a special case of planning which arises during the deployment of a plan when either a plan being deployed no longer satisfies certain criteria (usually of time or actions' costs optimality) or some of its pending actions cannot be executed. In this case, a new plan has to be produced, and depending on the way in which this procedure is carried out, the next two categories can be distinguished: *re-planning from scratch* and *plan repairing*. In the former case, all the processed information that was used for the production of the original plan is discarded, whereas, in the latter, a part of the previous computational effort is utilized.

This line of work *investigates the conditions under which plan repairing is more efficient than re-planning from scratch*. To this end, we focus our attention on $A^*$ algorithm, which is one of the most popular and studied algorithms in the field of Artificial Intelligence. Specifically, our contribution lies in the development of a novel algorithm, Dynamic Repairing $A^*$ (henceforth *DRA*$^*$), which extends $A^*$ in such a way that it can be used for plan repairing.

Namely, *DRA*$^*$ is suited for the repairing of plans in dynamic environments and can address modifications in goal-sets and actions' costs during the execution of a plan, which are two of the most common causes of plan invalidation. Since many state-of-the-art planning algorithms in a variety of domains are based on $A^*$ the study can provide valuable hints and insights towards the improvement of the existing re-planning methods as well as towards the development of more efficient ones.

Moreover, although it has been demonstrated in the classic and highly influential work of (Nebel and Koehler, 1995) that *in the worst case* modifying an existing plan is not guaranteed to be more efficient than re-planning from scratch, the goal of a thorough understanding regarding the trade-offs between these two approaches is far from achieved. The current study wishes to explore in more depth this interaction, revealing practically important instances, where repairing is guaranteed to be the optimal choice.

The rest of the paper is organized as follows. In the second section, we describe briefly the $A^*$ algorithm. Next, we discuss related work regarding the re-planning problem. We then present *DRA*$^*$. In the fifth section, we continue by presenting our experimental evaluation comparing *DRA*$^*$ and $A^*$ in standard planning benchmarks. In section 6 we conclude.

## 2 BACKGROUND

$A^*$ (Hart et al., 1968) is one of the most popular algorithms of Artificial Intelligence, with some of its most common uses including graph traversal and path-finding. Its key idea is the utilization of a heuristic value, that "guides" the search. As a result, its performance depends on the quality of the function that generates the heuristic values.

$A^*$ can be implemented in two different ways: a) using a *tree search* or b) using a *graph search*. Typically, in both cases, two auxiliary collections are utilized during the execution of the algorithm: a priority queue, called *open list*, containing the states candidate for expansion, and a set, referred to as *closed list*, containing the already expanded states. Moreover, three special values for each state are used: *g-value*, *h-value* and *f-value*. The g-value of a state is equal to the cost from the initial state to it; the h-value is an estimation of the minimum cost from it to the goal-state and the f-value is equal to the sum of the g-value and h-value.

At each step of the tree search variation, the state of the open list having the lowest f-value is removed from it. The state is examined for satisfying the goal-set in which case the search stops and the corresponding plan is extracted. Otherwise, the state is expanded by generating all its successor states which are added in the open list, while the expanded state is added in the closed list. If the h-values that are used are consistent[1], then this variation of the algorithm is guaranteed to find an optimal solution, if one exists, and, moreover, not to generate more states than any other algorithm that uses the same h-values.

Graph search differs from tree search in two points. First, each time a state is generated it is examined for being contained in the closed and open list respectively. If it is not contained in neither of the lists, the same steps as in the case of tree search are followed. If it is already in the closed list, then its current f-value is compared to its old f-value, e.g. the one with which it was inserted in the closed list. If the new f-value is smaller, the state is removed from the closed list and re-inserted in the open list with the new f-value.

Moreover, the algorithm in this variation does not stop when a plan has been found, but it continues until there is no state in the open list with an f-value that is smaller than the cost of the plan. In this case, it is not required that the h-values are consistent, but it suffices to be admissible, i.e. not to be greater than the cost of the optimal path from the corresponding state to a goal-state.

[1]The h-value of a state $s_N$ is consistent, if for every state $s_M$ that can be generated from $s_N$, the estimated cost of reaching a goal-state from $s_N$ is not greater than the cost of getting to $s_M$ from $s_N$ plus the estimated cost of reaching a goal-state from $s_M$.

# 3 RELATED WORK

Over the last years, a significant number of $A^*$-inspired plan repairingalgorithms has been developed, with the majority of them tailored to single-agent robotics problems. These algorithms fall, typically, into two main categories w.r.t. their capacities for plan-repairing: a) algorithms that are specialized in addressing modifications of the original goal-set (Stentz et al., 1995; Koenig and Likhachev, 2002; Likhachev et al., 2003) (Hansen and Zhou, 2007) and b) algorithms that are specialized in addressing changes of the actions costs (Koenig et al., 2004; Van Den Berg et al., 2006; Koenig and Likhachev, 2006). Finally, there are few other algorithms that can cope with both changes (Sun et al., 2008; Sun et al., 2010a; Sun et al., 2010b).

In general, the efficiency of these algorithms derives from the exploitation of the geometrical properties of the terrain where the agent is situated, since in some single-agent settings, such as navigation or moving-target search, the search tree can be mapped to the problem terrain. However, this mapping cannot be realized in many single-agent settings or in a multi-agent environment and, as a consequence, these algorithms are not applicable for problems of this type.

Two of the most influential algorithms of the first category are, Focused D$^*$ (Stentz et al., 1995), and D$^*$-Lite (Koenig and Likhachev, 2002). Both of them utilized a backwards-directed search from the goal state to the current state, saving, this way, information, which allows fast plan production when changes in the environment occur.

The Generalized Adaptive A$^*$ (GAA$^*$) is presented in (Sun et al., 2008). GAA$^*$ learns h-values in order to make them more informed and can be utilized for moving target search in terrains where the action costs of the agent can change between searches. An extension of GAA$^*$ that is close to our work, is MP-GAA$^*$ (Hernández et al., 2015), where some of the best paths for some nodes of the search graph are stored. More recently, there have been implemented Generalized Fringe-Retrieving $A^*$ (Sun et al., 2010a) and Moving Target D$^*$-Lite (Sun et al., 2010b) which, in the same way as GA$A^*$ can address both goal-set modifications and actions costs changes.

Finally, we mention briefly some other plan repairing approaches that are not based on $A^*$ and where ad hoc techniques such as plan refinement and adaptation are utilized. For example, in (Gerevini and Serina, 2010) specialized heuristic search techniques are used in order to solve the plan adaptation tasks through the repairing of certain portions of the original plan. Similarly, in (Au et al., 2002) a special algorithm which uses analogy by derivation for plan adaptation is presented. However, in contrast with $DRA^*$, in these cases plan optimality is not a central issue.

# 4 DYNAMIC REPAIRING A*

$DRA^*$ is an extension of $A^*$ that is suited for the repairing of sequential plans and can address two types of changes in the environment: a) goal-set modifications and b) actions' costs alterations. $DRA^*$ is based on the graph search variation of $A^*$, utilizing the same search strategy: the selection, testing and expansion of a state at each step and the utilization of a heuristic value to guide the whole procedure.

Its novelty is that a new search graph is not created from scratch as in $A^*$. Instead, the initial search graph is retrieved at the start of the algorithm, and then used for the subsequent search. As with the case of graph search $A^*$, the utilization of admissible h-values is required in order for the solutions returned to be optimal. The corresponding pseudo-code is presented in pages 3 and 4.

## 4.1 Comparing DRA* with A*

$DRA^*$ differs from $A^*$ in a number of ways. First, while in the case of $A^*$ only the parent state, i.e. the state that results in the lowest g-value, is kept, in the case of $DRA^*$ every predecessor state of a given state is stored. Moreover, a special procedure, the *informing procedure*, takes place, in order to be determined if a state that is derived from a previous planning procedure and is encountered for the first time, is reachable from the new initial state, and its g-value and h-value to be updated if necessary. As a consequence, the novel concepts of an *informed*, *uninformed*, *valid* and *invalid state* are introduced, which serve for the description of the corresponding states. By default, when the algorithm begins, all the states of the search tree are marked as uninformed except of the new initial state which is set as informed and valid.

The informing procedure can be achieved in two different ways: *fully* or *lazily*. The former is applied in cases when there exist actions with decreased costs, whereas the latter is applied in the other cases.

The procedure for the full informing is the following. First, the state being examined is marked as pending and, consequently, all its predecessor states are examined for being informed. For any predecessor state found not to be informed and not to be pending, the procedure of full informing is followed. If

**Algorithm 1:** Dynamic Repairing A*

```
input  : New Initial State, Previous Closed List, Previous Open
         List, Original Goal set, New Goal set
output: The optimal plan for the new goal set
plan ⟵ NULL
mark newInitialState as valid and informed
CLOSED ⟵ previousCLOSED
if originalGoalSet = newGoalSet then
    plan ⟵ searchCloseList(CLOSED, newGoalSet)
if plan ≠ NULL & ∄ action with decreased costs then
    return plan
OPEN ⟵ previousOPEN
if newGoalSet is not superset of originalGoalSet then
    validateOpenList(OPEN,originalGoalSet,newGoalSet)
while OPEN is not empty do
    currentState ⟵ OPEN.poll()
    if currentState satisfies newGoalSet then
        plan ⟵ ExtractPlan(currentState)
        break
    foreach applicable action ac of currentState do
        succState ⟵ currentState.apply(ac)
        pVal ⟵ currentState.gValue + ac.cost
        if succState ∉ OPEN and ∉ CLOSED then
            OPEN.add(succState)
        else
            if succState is not informed then
                lazy_inform(succState)
            if succState is not valid then
                OPEN.add(succState)
            else
                if pVal < succState.gValue then
                    if OPEN ∋ succState then
                        OPEN.remove(succState)
                    if CLOSED ∋ succStatet then
                        CLOSED.remove(succState)
                    OPEN.add(succState)
                else
                    succState.predQueue.add(currentState )
        CLOSED.add(currentState )
    end
end
OPEN.add(plan.currentState )
previousOPEN ⟵ OPEN
previousCLOSED ⟵ CLOSED
return plan
```

**Algorithm 2:** Validation of the Open List

```
input  : Open List, Original Goal Set and New Goal Set
newOpenList ⟵ new Priority Queue()
foreach state in OPEN do
    if state is not Informed then
        if ∄ action with decreased costs then
            lazy_inform(state)
        else
            full_infrom(state)
        if state is Valid then
            state.updatefValue()
            newOpenList.add(state)
end
OPEN ⟵ newOpenList
```

**Algorithm 3:** Traversal of the Closed List

```
input  : Closed List and New Goal Set
output: A plan
plan ⟵ null
cost = ∞
foreach State state in CLOSED do
    if state satisfies goalSet then
        if state is not Informed then
            lazy_inform(state);
        if state is Valid then
            if state.gValue < cost then
                plan ⟵ ExtractPlan(state)
                cost =state.gValue
end
return plan
```

**Algorithm 4:** Successor States Update

```
input  : A state stateUpd
if stateUpd is informed and valid then
    foreach successor_state in stateUpd.StatesList do
        ac ⟵ successor_state.generatingAction
        pVal ⟵ stateUpd.gValue + ac.cost
        successor_state.nonInformedPredecessors−−
        if pVal < successor_state.gValue then
            successor_state.gValue = pVal
            genState.parent ⟵ stateUpd
            if successor_state.nonInformedPredecessors=0
              then
                marked successor_state as informed and valid
                successor_state.updatesuccStates()
    end
else
    foreach successor_state in stateUpd.StatesList do
        successor_state.nonInformedPredecessors−−
        if successor_state.nonInformedPredecessors=0 then
            marked successor_state as informed and invalid
            successor_state.updatesuccStates()
    end
```

a valid predecessor states is found, its *p-value*[2] is re-calculated and compared with the state's g-value and if found smaller, then this predecessor state is set as the state's parent. After the examination of the predecessor states finishes, if the state has any valid predecessor, it is marked as informed and valid. Finally, the state is reset from pending.

The lazy informing (the corresponding pseudocode is omitted due to lack of space) is carried out in the same way as the full, with the exception that the procedure stops if a valid predecessor state has been found and the next predecessor state that is to be examined does not have a smaller p-value. The examination of the predecessor states follows their sorting order. That is, it begins with predecessor state having the lowest p-value, i.e. the parent-state, and continues with the one having the second lowest and so forth. Note that in this case, some of the p-values of a state might not be correct and some of its parent states might not have been informed, without this affecting the correctness of the algorithm.

[2]With the term p-value$_{s_a \rightarrow s_d}$ we denote the resulting g-value of $s_d$ after its generation of another state $s_a$.

**Algorithm 5:** Full informing

```
input  : A state stateInf
set stateInf as pending
parentState ⟵ stateInf.getParent()
stateInf.gValue =∝
nonInformedPredecessors = 0
if parentState not informed & not pending then
    full_inform(parentState)
if parentState is Valid then
    stateInf.gValue = parentState.gValue + action.cost
if parentState is pending then
    nonInformedPredecessors + +
    parentState.StatesList.add(stateInf )
if parentState.nonInformedPredecessors > 0 then
    nonInformedPredecessors + +
    parentState.StatesList.add(stateInf )
foreach predState in stateInf.predQueue do
    if predState not informed & not pending then
        full_inform(predState )
    if predState is pending then
        nonInformedPredecessors + +
        predState .StatesList.add(stateInf )
    if predState.nonInformedPredecessors > 0 then
        nonInformedPredecessors + +
        predState .StatesList.add(stateInf )
    if predState is Valid then
        if predState.pValue < stateInf.gValue then
            stateInf.lgvParent ⟵ predState
            stateInf.gValue ⟵ predState.pValue
end
if stateInf.gValue ≠∝ then
    mark stateInf as valid and informed
else
    mark stateInf as invalid and informed
updatesuccStates(stateInf )
reset stateInf from pending
```

Another difference between the two algorithms is that in the case of $DRA^*$, when the search for the plan finishes, the closed and open lists are stored, so that they can be used in case of re-planning. Before the open list is saved, the last removed state is re-inserted in it. Subsequently, when the algorithm is executed, the previously save lists are retrieved and used.

In addition, if the new goal set is the same as the original goal set, the initial closed list is searched for containing solutions before the main part of the algorithm begins. During this traversal, each state is examined. The ones satisfying the new goal-set are lazily informed, when uninformed. In case one or more valid states satisfying the new goal-set have been found, the one having the lowest g-value is returned as solution and the algorithm terminates.

Finally, in cases where the new goal set is not a superset of the original goal set the open list is validated before the main search starts. Namely, every state is informed, fully if there exists actions with decreased costs and lazily otherwise, and, if it is valid, its h-value is re-calculated and it is re-inserted in the open list with its newly updated f-value.

**Theorem 1.** *DRA* is sound and complete for repairing scenarios of goal-set modifications or actions costs changes if the h-values that are used are admissible*[3].

# 5 EXPERIMENTAL EVALUATION

For the assessment of the capacity of *DRA*$^*$ in addressing re-planning problems, we compared its performance in terms of speed against $A^*$. To this end, we devised four different re-planning scenarios simulating real-world situations, where we compared the ratio of the runtime of the two algorithms by varying the following characteristics:

- The percentage of the original plan that was already executed at the time when the need for re-planning occurred (Scenarios 1, 2, 3 and 4);
- The percentage of the modification of the original goal-set (Scenarios 1 and 2);
- The percentage of the actions whose costs decreased (Scenario 3);
- The percentage of the actions whose costs increased (Scenario 4).

We opted for comparing *DRA*$^*$ against $A^*$ instead of other replanning algorithms for two reasons. First, $A^*$ is the most typical planning algorithm, the properties and behavior of which have been thoroughly studied. Therefore, the experimental results concerning the relative performance of the two algorithms, could provide us with valuable hints and insights for a better understanding of *DRA*$^*$. Second, most of the re-planning algorithms, as we already mentioned in section 3, are utilizable only in specific settings, which usually concern single-agent problems, and, therefore, are not applicable in the scenarios we are examining.

The experiments focus on time performance; memory requirements are not measured, because both $A^*$ and *DRA*$^*$ exhibit a linear complexity in the number of states in the state space. We should note, also, that we did not include in the experimental results the generated and expanded states of each algorithm since *DRA*$^*$ always expands and generates less states than $A^*$ when a part of the search tree is already constructed which is the case in all the conducted experiments.

[3]The proof is presented in (Gouidis et al., 2017).

## 5.1 Experimental Setup

The structure of the experiments is the same in every case. First, a plan is produced for the initial conditions of the problem, i.e. initial state, goal-set and actions' costs. Next, a parameter of the environment, according to the type of the experiment, is modified: in scenarios 1 and 2 the goal-set, and in scenarios 3 and 4 the costs of some actions. Finally, a new plan is produced for the modified conditions using both $A^*$ (replanning from scratch) and *DRA*$^*$ (repairing). The new initial state of the re-planning problems is a randomly-selected state of the initial plan. The changes for each scenario are the following:

- Scenario 1. The new goal set is produced by the removal of $k$ goals from the initial goal set consisted of n goals, and the insertion of m goals in it respectively, where $k \leq n$.
- Scenario 2. The new goal set is produced by the addition of $k$ goals in the original goal-set.
- Scenario 3. A $p\%$ percentage of the actions costs are decreased, none of which belongs to the initial plan. The maximum decrease for an action cost is a 90% of its initial cost.
- Scenario 4. A p% percentage of the actions costs are increased. $q\%$ of the actions with increased costs belongs to the initial plan. The maximum increase for an action cost is a 200% of its initial cost.

The benchmarks that were used for the evaluation are: Blocks, Depots, Gripper, Logistics, Miconic and Transports which derive from the $3^{rd}$, $4^{th}$ and $8^{th}$ International Planning Competitions (Bacchus, 2001; Long and Fox, 2003; Gerevini et al., 2009). In addition, since in the majority of the planning domains the actions costs are uniform, we created two variations of the domains Logistics and Depots, Logistics-cost and Depots-cost respectively, with actions of varied costs. The specifications of the scenarios are shown in Table 1.

Both algorithms were implemented in Java, using the same data structures, functions and routines for all the shared procedures, in order to ensure that the disparities in the runtimes reflect performative differences between the algorithms and are not due to their different implementations. The experiments were conducted on a 64-bit Ubuntu Workstation with two 8-core® Xeon® CPU E5-2630 processors running at a 2.30GHz server with 384 GB RAM, from which 10 GB were allocated for each experiment.

## 5.2 Experimental Results

The experimental results, presented in Table 2, indi-

Table 1: Specifications of the scenarios' experiments

| Experiment | Problem | Number of Initial Goals | Number of Removed Goals | Number of Added Goals | Average Branching Factor | Number of Conducted Experiments |
|---|---|---|---|---|---|---|
| 1.1 | Blocks | 6 | 1 | 1 | 4.61 | 5 |
| 1.2 | Depots | 5 | 1 | 1 | 8.77 | 5 |
| 1.3 | Gripper | 12 | 2 | 3 | 4.62 | 5 |
| 1.4 | Logistics | 5 | 1 | 1 | 8.33 | 5 |
| 1.5 | Logistics | 5 | 1 | 1 | 8.44 | 5 |
| 2.1-2.4 | Blocks | 6/7/6/7 | 2/1/2/1 | - | 4.92/4.95/4.62/4.69 | 28/28/28/28 |
| 2.5-2.8 | Logistics | 4/5/5/4 | 2/1/2/1 | - | 8.38/8.46/8.51/8.49 | 15/6/15/6 |
| 2.9-2.12 | Depot | 3/4/3/4 | 2/1/3/4 | - | 10.88/10.84/4.62/4.65 | 10/5/20/15 |
| 2.13-2.15 | Gripper | 8/9/11 | 4/5/3 | - | 4.38/4.35/4.66 | 40/40/40 |
| 2.16-2.19 | Miconic | 7/10/11/12 | 3/1/2/4 | - | 19.83/19.97/21.71/23.67 | 10/10/10/10 |

| Experiment | Problem | Percentage of Decreased (Increased) Actions Costs | Max Percentage of Plan's Decreased (Increased) Actions Costs | Average Branching Factor | Number of Conducted Experiments |
|---|---|---|---|---|---|
| 3.1a/3.1b/3.1c | Transport | 5/25/50 | 90 | 8.93 | 10 |
| 3.2a/3.2b/3.2c | Depots-cost | 5/25/50 | 90 | 4.62 | 10 |
| 3.3a/3.3b/3.3c | Logistic-cost | 5/25/50 | 90 | 8.23 | 10 |
| 4.1a/4.1b/4.1c | Transport | 5/25/50 | 200 | 8.94 | 10 |
| 4.2a/4.2b/4.2c | Depots-cost | 5/25/50 | 200 | 4.57 | 10 |
| 4.3a/4.3b/4.3c | Logistic-cost | 5/25/50 | 200 | 8.16 | 10 |

cate that *DRA*$^*$ outperforms $A^*$ in most of the goal set modification cases, provided that the next conditions are met. First, the percentage of the original plan that has been already executed, should not be greater than 50%. Moreover, the change in the goal-set should not be greater than 20% to 50%. The corresponding thresholds, for the previous two parameters, below which *DRA*$^*$ performs better, depend on the average branching factor of the re-planning problem, with average higher branching factors corresponding to thresholds of lower values. Moreover, according to the results, in the cases of modified actions costs, *DRA*$^*$ outperforms $A^*$ always.

Furthermore, we can make the following observations regarding the performance of *DRA*$^*$ compared to $A^*$:

1. As the percentage of the executed plan decreases, the relative performance is improved.
2. As the percentage of the modified goal-set decreases, the relative performance is improved.
3. As the average branching factor, i.e. the average number of predecessor states that a state has, decreases, the relative performance is improved.
4. The relative performance does not vary significantly as the percentage of actions with decreased costs increases.
5. The relative performance does not vary significantly as the percentage of actions with increased costs increases.
6. For a given problem instance, *DRA*$^*$ performs better in increases of the goal-set than in general modifications of the goal-set.
7. For a given problem instance, *DRA*$^*$ performs better in cases of increased actions costs than of decreased actions costs.

We consider that the previous findings can be explained by the following reasons. First, *DRA*$^*$ expands at most the same number of states as $A^*$, since a part of the search graph with which the search begins, is already constructed. Moreover, during *DRA*$^*$ execution, the procedures of states informing, open list validation and closed list traversal, which are absent from $A^*$, might take place. Therefore, it can be concluded, that the trade-off between the previous two factors determines *DRA*$^*$ performance against $A^*$.

Regarding the first finding, it can be due to the fact that as the percentage of the executed plan increases, the new root of the search graph recedes further from the root of the original search graph, which, as a result, has one of the following two outcomes: a larger part of the search graph leaves would either become invalid or would have its f-values increased. In either case, time is consumed for the informing of states that do not affect the search.

Likewise, the fact that the traversal of the closed list and the validation of the open list, is not carried out in the case of an increased goal-set seems to explain the better performance of *DRA*$^*$ in such cases in comparison to the general case of handling modified goal-sets (observation 6). A similar line of reasoning can be applied in the case of modified actions costs (observation 7). Namely, in the case of decreased costs, the open list is validated. Furthermore, the informing of the states is full, whereas, in the case of increased costs, the lazy informing is utilized, which, at worst case, requires the same time. Findings 4 and 5 can be ascribed to the fact that greater percentages of modi-

Table 2: Mean value of the ratio of the runtime of $DRA^*$ to $A^*$ for scenarios 1-4.

| | Experiment | Percentage of Executed Plan | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | 0.1 | 0.2 | 0.3 | 0.4 | 0.5 | 0.6 | 0.7 | 0.8 | 0.9 |
| Scenario 1 | 1.1 | 0.85±0.10 | 0.83±0.21 | 0.78±0.14 | 0.76±0.19 | 0.97±0.21 | 0.94±0.23 | 1.23±0.22 | 1.12±0.31 | 0.63±0.28 |
| | 1.2 | 0.44±0.10 | 0.44±0.05 | 0.61±0.05 | 1.33±0.05 | 1.78±0.35 | 2.58±0.31 | 2.95±0.32 | 3.21±0.41 | 4.04±0.54 |
| | 1.3 | 0.22±0.05 | 0.23±0.03 | 0.21±0.06 | 0.24±0.03 | 1.21±0.53 | 1.86±1.01 | 2.30±1.62 | 2.92±1.57 | 3.36±2.01 |
| | 1.4 | 0.41±0.07 | 0.29±0.06 | 0.58±0.09 | 1.36±0.16 | 1.28±0.14 | 2.70±0.37 | 4.82±0.45 | 5.67±0.64 | 7.03±0.82 |
| | 1.5 | 0.23±0.03 | 0.32±0.13 | 0.53±0.14 | 1.17±0.21 | 1.52±0.94 | 1.68±0.00 | 2.92±1.34 | 4.20±1.76 | 6.04±2.54 |
| Scenario 2 | 2.1 | 0.85±0.09 | 0.84±0.11 | 0.81±0.12 | 0.80±0.13 | 0.88±0.13 | 0.94±0.12 | 0.99±0.06 | 0.98±0.08 | 0.92±0.30 |
| | 2.2 | 0.65±0.06 | 0.62±0.06 | 0.59±0.06 | 0.48±0.13 | 0.77±0.15 | 1.07±0.17 | 0.93±0.08 | 1.08±0.15 | 0.85±0.08 |
| | 2.3 | 0.85±0.14 | 0.84±0.15 | 0.78±0.13 | 0.78±0.19 | 0.77±0.14 | 1.47±1.23 | 1.01±0.33 | 2.85±2.58 | 1.72±1.53 |
| | 2.4 | 0.67±0.13 | 0.60±0.17 | 0.59±0.17 | 0.60±0.27 | 0.69±0.18 | 2.53±1.89 | 1.29±0.71 | 1.66±1.42 | 1.65±1.15 |
| | 2.5 | 0.96±0.16 | 0.96±0.19 | 0.90±0.18 | 0.86±0.13 | 0.99±0.12 | 1.10±0.25 | 1.89±1.62 | 2.47±1.32 | 3.19±1.66 |
| | 2.6 | 0.60±0.26 | 0.59±0.24 | 0.70±0.13 | 0.75±0.10 | 2.06±2.14 | 2.99±2.36 | 3.46±1.76 | 4.50±1.93 | 6.10±2.44 |
| | 2.7 | 0.99±0.12 | 1.00±0.10 | 0.96±0.13 | 0.93±0.15 | 0.96±0.10 | 1.03±0.09 | 1.39±0.46 | 2.59±1.53 | 3.33±2.23 |
| | 2.8 | 0.58±0.22 | 0.55±0.20 | 0.57±0.15 | 0.65±0.21 | 0.85±0.14 | 1.64±0.82 | 3.54±0.88 | 4.73±1.35n | 5.74±1.66 |
| | 2.9 | 0.87±0.15 | 0.85±0.17 | 0.82±0.14 | 0.86±0.16 | 0.82±0.13 | 1.01±0.07 | 2.16±1.22 | 3.63±2.27 | 4.27±2.06 |
| | 2.1 | 0.57±0.02 | 0.51±0.09 | 0.49±0.06 | 0.45±0.17 | 1.09±0.81 | 1.09±0.89 | 1.50±1.23 | 2.42±1.76 | 3.45±2.03 |
| | 2.11 | 0.83±0.04 | 0.85±0.06 | 0.79±0.08 | 0.83±0.07 | 0.84±0.08 | 0.82±0.04 | 1.01±0.17 | 2.64±1.18 | 3.41±0.34 |
| | 2.12 | 0.65±0.19 | 0.64±0.21 | 0.60±0.20 | 0.59±0.18 | 0.71±0.16 | 1.40±1.49 | 2.78±1.64 | 3.65±2.63 | 3.87±1.96 |
| | 2.13 | 0.48±0.04 | 0.46±0.04 | 0.46±0.03 | 0.45±0.04 | 0.43±0.04 | 0.44±0.05 | 0.49±0.06 | 0.74±0.15 | 1.26±0.29 |
| | 2.14 | 0.80±0.14 | 0.78±0.09 | 0.73±0.11 | 0.69±0.08 | 0.71±0.09 | 0.77±0.07 | 0.87±0.08 | 1.37±0.38 | 2.75±1.83 |
| | 2.15 | 0.32±0.02 | 0.30±0.01 | 0.28±0.01 | 0.29±0.02 | 0.26±0.05 | 0.56±0.11 | 1.10±0.17 | 3.60±0.19 | 4.56±0.25 |
| | 2.16 | 0.53±0.15 | 0.72±0.10 | 1.44±0.35 | 2.11±0.86 | 4.01±0.70 | 5.10±1.46 | 6.75±0.00 | 8.43±3.21 | 10.03±3.42 |
| | 2.17 | 0.21±0.05 | 0.54±0.15 | 0.95±0.05 | 2.08±0.57 | 4.85±1.76 | 5.72±1.87 | 6.44±2.22 | 8.32±2.42 | 9.84±3.27 |
| | 2.18 | 0.17±0.03 | 0.63±0.07 | 1.03±0.34 | 3.70±0.60 | 5.30±1.71 | 6.90±2.41 | 7.72±2.74 | 9.24±3.03 | 10.11±2.87 |
| | 2.19 | 0.62±0.13 | 1.13±0.60 | 1.56±0.99 | 2.68±1.04 | 4.43±1.44 | 6.10±2.54 | 8.68±3.04 | 9.52±3.04 | 11.30±3.32 |
| Scenario 3 | 3.1a | 0.43±0.13 | 0.68±0.17 | 0.73±0.14 | 0.59±0.16 | 0.64±0.17 | 0.83±0.21 | 0.72±0.21 | 0.85±0.23 | 0.81±0.27 |
| | 3.1b | 0.50±0.12 | 0.54±0.14 | 0.59±0.14 | 0.62±0.21 | 0.54±0.22 | 0.69±0.25 | 0.69±0.22 | 0.92±0.26 | 0.78±0.25 |
| | 3.1c | 0.49±0.15 | 0.55±0.16 | 0.53±0.19 | 0.70±0.22 | 0.64±0.24 | 0.62±0.22 | 0.71±0.25 | 0.90±0.25 | 0.84±0.26 |
| | 3.2a | 0.17±0.03 | 0.18±0.04 | 0.24±0.04 | 0.22±0.05 | 0.24±0.06 | 0.21±0.07 | 0.20±0.06 | 0.24±0.08 | 0.30±0.07 |
| | 3.2b | 0.25±0.05 | 0.29±0.05 | 0.23±0.04 | 0.29±0.06 | 0.34±0.07 | 0.31±0.09 | 0.30±0.09 | 0.35±0.10 | 0.32±0.11 |
| | 3.2c | 0.26±0.07 | 0.32±0.06 | 0.28±0.06 | 0.29±0.07 | 0.37±0.04 | 0.39±0.11 | 0.35±0.12 | 0.36±0.11 | 0.39±0.12 |
| | 3.3a | 0.41±0.08 | 0.44±0.07 | 0.56±0.11 | 0.54±0.13 | 0.46±0.14 | 0.48±0.18 | 0.52±0.19 | 0.61±0.21 | 0.59±0.20 |
| | 3.3b | 0.45±0.08 | 0.46±0.08 | 0.48±0.12 | 0.50±0.15 | 0.54±0.18 | 0.52±0.17 | 0.61±0.22 | 0.60±0.24 | 0.59±0.22 |
| | 3.3c | 0.49±0.10 | 0.56±0.12 | 0.52±0.14 | 0.62±0.16 | 0.55±0.17 | 0.55±0.19 | 0.69±0.24 | 0.65±0.23 | 0.68±0.25 |
| Scenario 4 | 4.1a | 0.21±0.03 | 0.23±0.07 | 0.19±0.34 | 0.25±0.07 | 0.30±0.10 | 0.65±0.15 | 0.77±0.14 | 0.64±0.13 | 0.80±0.17 |
| | 4.1b | 0.48±0.13 | 0.62±0.14 | 0.75±0.19 | 0.79±0.18 | 0.87±0.19 | 0.79±0.21 | 0.77±0.20 | 0.73±0.24 | 0.85±0.23 |
| | 4.1c | 0.54±0.13 | 0.50±0.16 | 0.65±0.19 | 0.68±0.24 | 0.73±0.24 | 0.70±0.25 | 1.00±0.28 | 0.83±0.26 | 0.84±0.22 |
| | 4.2a | 0.10±0.03 | 0.08±0.04 | 0.12±0.04 | 0.10±0.06 | 0.24±0.11 | 0.35±0.14 | 0.27±0.13 | 0.28±0.13 | 0.30±0.12 |
| | 4.2b | 0.09±0.04 | 0.13±0.05 | 0.13±0.06 | 0.18±0.07 | 0.26±0.10 | 0.25±0.12 | 0.27±0.11 | 0.37±0.14 | 0.33±0.13 |
| | 4.2c | 0.21±0.10 | 0.27±0.13 | 0.25±0.11 | 0.35±0.17 | 0.30±0.15 | 0.29±0.16 | 0.37±0.16 | 0.35±0.15 | 0.39±0.13 |
| | 4.3a | 0.14±0.05 | 0.24±0.09 | 0.39±0.14 | 0.45±0.16 | 0.37±0.17 | 0.34±0.14 | 0.28±0.14 | 0.43±0.13 | 0.36±0.18 |
| | 4.3b | 0.26±0.05 | 0.33±0.11 | 0.38±0.12 | 0.48±0.13 | 0.41±0.15 | 0.44±0.15 | 0.42±0.16 | 0.39±0.17 | 0.45±0.15 |
| | 4.3c | 0.41±0.08 | 0.54±0.15 | 0.43±0.16 | 0.67±0.15 | 0.54±0.18 | 0.70±0.19 | 0.75±0.21 | 0.80±0.22 | 0.74±0.21 |

fied actions costs do not affect the execution of $DRA^*$.

Finally, the deterioration of $DRA^*$ performance with higher average branching factors (observation 3) can be attributed to the greater time that is necessary for the informing procedure, since large average branching factors correspond to a large average number of predecessor states, which results in a greater number of examined ancestor states during the informing procedure.

## 6 Conclusions

In this work we presented a novel plan repairing algorithm, $DRA^*$, that extends one of the most popular and studied planning algorithms, $A^*$, by addressing modifications in the goal-set and in the actions costs. Therefore, $DRA^*$, is suitable for plan repairing in dynamic environments, where changes of the aforementioned kinds take place.

The conducted experimental evaluation showed that

*DRA*$^*$ outperformed $A^*$ in most of the cases with modified goal-sets, provided that the percentage of the original plan that has been already executed, is not greater than 40% to 50%. and the change in the goal-set is not be greater than 20% to 50%. The overall performance depends on the average branching factor of the problem, with average higher branching factors corresponding to thresholds of lower values. For re-planning scenarios of modified actions costs, the experimental outcome was that *DRA*$^*$ outperformed $A^*$ in all experiments.

We believe that the experimental results provide a strong support for the utilization of *DRA*$^*$ in re-planning scenarios. Nevertheless, a more thorough experimental analysis could provide more useful hints and insights and help us to gain a more elaborate understanding of the underlying mechanisms which determine the strengths and weaknesses of the algorithm.

In particular, we would like to assess *DRA*$^*$ performance in scenarios of repeated repairing and in scenarios where both the goal-set and the actions costs are modified, which seem to represent more faithfully certain dynamic environments. Another direction that we wish to investigate is the addressing of other types of dynamicity that can be observed in real-world domains, such as altered preconditions and effects for actions, additions and removals of planning agents and invalidations or insertions of new actions.

Moreover, since the worst performance of *DRA*$^*$ is observed in domains with large branching factors which are directly related to the number of the agents activated for the re-planning procedure, we consider that a distributed implementation of *DRA*$^*$, where each agent performs an independent search, could improve substantially the performance of the algorithm.

## REFERENCES


Au, T.-C., Muñoz-Avila, H., and Nau, D. S. (2002). *On the Complexity of Plan Adaptation by Derivational Analogy in a Universal Classical Planning Framework*, pages 13–27. Springer Berlin Heidelberg, Berlin, Heidelberg.

Bacchus, F. (2001). AIPS 2000 planning competition: The fifth international conference on artificial intelligence planning and scheduling systems. *Ai magazine*, 22(3):47.

Gerevini, A. E., Haslum, P., Long, D., Saetti, A., and Dimopoulos, Y. (2009). Deterministic planning in the fifth international planning competition: PDDL3 and experimental evaluation of the planners. *Artificial Intelligence*, 173(5):619–668.

Gerevini, A. E. and Serina, I. (2010). Efficient plan adaptation through replanning windows and heuristic goals. *Fundamenta Informaticae*, 102(3-4):287–323.

Gouidis, F., Patkos, T., Flouris, G., and Plexousakis, D. (2017). The DRA$^*$ algorithm. Technical report, Foundation of Research and Technology.

Hansen, E. A. and Zhou, R. (2007). Anytime heuristic search. *J. Artif. Intell. Res.(JAIR)*, 28:267–297.

Hart, P. E., Nilsson, N. J., and Raphael, B. (1968). A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics*, 4(2):100–107.

Hernández, C., Asín, R., and Baier, J. A. (2015). Reusing previously found A$^*$ paths for fast goal-directed navigation in dynamic terrain. In *AAAI*, pages 1158–1164.

Koenig, S. and Likhachev, M. (2002). D$^*$ Lite. In *AAAI/IAAI*, pages 476–483.

Koenig, S. and Likhachev, M. (2006). Real-time adaptive A$^*$. In *Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems*, pages 281–288. ACM.

Koenig, S., Likhachev, M., and Furcy, D. (2004). Lifelong planning A$^*$. *Artificial Intelligence*, 155(1):93–146.

Likhachev, M., Gordon, G. J., and Thrun, S. (2003). ARA$^*$: Anytime $A^*$ with provable bounds on sub-optimality. In *Advances in Neural Information Processing Systems*, page None.

Long, D. and Fox, M. (2003). The 3rd international planning competition: Results and analysis. *J. Artif. Intell. Res.(JAIR)*, 20:1–59.

Nebel, B. and Koehler, J. (1995). Plan reuse versus plan generation: A theoretical and empirical analysis. *Artificial Intelligence*, 76(1-2):427–454.

Stentz, A. et al. (1995). The focussed D$^*$ algorithm for real-time replanning. In *IJCAI*, volume 95, pages 1652–1659.

Sun, X., Koenig, S., and Yeoh, W. (2008). Generalized adaptive A$^*$. In *Proceedings of the 7th international joint conference on Autonomous agents and multiagent systems-Volume 1*, pages 469–476.

Sun, X., Yeoh, W., and Koenig, S. (2010a). Generalized fringe-retrieving $A^*$: faster moving target search on state lattices. In *Proceedings of the 9th International Conference on Autonomous Agents and Multiagent Systems: volume 1-Volume 1*, pages 1081–1088.

Sun, X., Yeoh, W., and Koenig, S. (2010b). Moving target D$^*$ Lite. In *Proceedings of the 9th International Conference on Autonomous Agents and Multiagent Systems: volume 1-Volume 1*, pages 67–74.

Van Den Berg, J., Ferguson, D., and Kuffner, J. (2006). Anytime path planning and replanning in dynamic environments. In *Proceedings 2006 IEEE International Conference on Robotics and Automation, 2006. ICRA 2006.*, pages 2366–2371. IEEE.