Repairing of sequential plans in dynamic environments

Filippos Gouidis

Thesis submitted in partial fulfillment of the requirements for the

Master of Science degree in Computer Science

University of Crete
School of Sciences and Engineering
Computer Science Department
Voutes Campus, GR-70013, Heraklion, Crete, Greece

Thesis Supervisor: Professor *Dimitris Plexousakis*

This work was partially supported by **Institute of Computer Science, Foundation of Research and Technology Hellas** and was part-financed by the Action **"Scholarships Programmes by the State Scholarships Foundation"** with funds of the Operational Programme **"Education and Life Long Learning"** of the **European Social Fund** (ESF) within the **National Strategic Reference Framework**, 2007-2013

Contents

List of Figures

List of Tables

UNIVERSITY OF CRETE
COMPUTER SCIENCE DEPARTMENT

Repairing of sequential plans in dynamic environments

Thesis submitted by
Filippos Gouidis
in partial fulfillment of the requirements for the
Masters' of Science degree in Computer Science

THESIS APPROVAL

Author: _____
Filippos Gouidis

Committee approvals: _____
Dimitris Plexousakis
Professor, Thesis Supervisor

_____
Ioannis Tsamardinos
Assistant Professor, Committee Member

_____
Georgios Flouris
Assistant Researcher, Committee Member

Departmental approval: _____
Antonis A. Argyros
Professor, Director of Graduate Studies

Heraklion, Month Year

Dedicated to my Father

Abstract

Planning is one of the oldest and most fundamental research areas of Artificial Intelligence. Apart from its theoretical importance, it is utilized very frequently in a wide range of practical applications that spans from space missions to factory line production.

A common complication that occurs after the production of plans, is their rendering invalid or suboptimal during their execution, due to the dynamic nature of the environments where they are executed. A fast response mechanism can be proven crucial for domains where the assumption of a static environment is very optimistic, if not untenable. This thesis presents a new algorithm, Repairing Dynamic A^* (RDA^*), for plan repairing that utilizes previous information and computational effort, in order to accelerate the production of new plans that correspond to the altered conditions of their environment.

RDA^* is an expansion of the A^* algorithm, a standard planning algorithm of the relevant literature, upon which many of the state-of-the-art planners are based. This expansion is tailored to the repairing of the plans in non-static environments of certain characteristics. Namely, dynamic goal-sets and modifiable action costs can be addressed.

The experimental protocol that we used for the assessment of the RDA^* performance is the following. First, a plan is produced for the initial environment's conditions. Consequently, assuming that the plan has been executed up to a certain percentage, either the problem's goal-set or some of its actions' costs is changed. Finally, RDA^* and A^* are executed from the latter point. The type of domains and problems that we used for the evaluation are standard benchmarks, derived from the international planning competitions.

The experimental results indicate that the performance of RDA^* depends from the following factors: the ratio of the original graph search size to the final graph search size, the branching parameter of the problem, the density of the graph search, the percentage of the original plan already executed and the volume of the changes in the environment. For sparse search graphs and small to moderate environment changes, RDA^* outperforms A^* in terms of speed by a factor of 10% to 80% in the majority of the cases, if the percentage of the plan that has been already executed is less than 40% to 50%.

We consider that this thesis can provide useful insights and hints towards the development of more efficient plan repairing techniques, since the A^* constitutes the backbone of many actual planners. Moreover, we believe that our work can be further improved and expanded, by incorporating new features, such as a decentralized approach and a real-time response functionality.

Περίληψη

Ο σχεδιασμός ενεργειών ($planning$) αποτελεί μια από τις παλιότερες και βασικότερες περιοχές έρευνας της Τεχνητής Νοημοσύνης. Πέραν της θεωρητικής αξίας που έχει, χρησιμοποιείται σε έναν μεγάλο εύρος πρακτικών εφαρμογών που κυμαίνεται από διαστημικές αποστολές μέχρι την εργοστασιακή γραμμή παραγωγής.

Μια επιπλοκή που συμβαίνει συχνά μετά την παραγωγή σχεδίων ενεργειών, είναι ότι κατά τη διάρκεια ανάπτυξης των, αυτά δεν δύναται πλέον να εκτελεστούν ή παύουν να είναι βέλτιστα, εξαιτίας της δυναμικής φύσης του περιβάλλοντος όπου εκτελούνται. Ένας γρήγορος μηχανισμός ανταπόκρισης θα μπορούσε να αποδειχθεί καίριος για περιοχές για τις οποίες η παραδοχή ενός σταθερού και αμετάβλητου περιβάλλοντος είναι πολύ αισιόδοξη, εάν όχι ανεδαφική. Η παρούσα εργασία παρουσιάζει ένα αλγόριθμο επιδιόρθωσης σχεδίων ενεργειών, τον $RepairingDynamicA^*$ (RDA^*), ο οποίος χρησιμοποιεί την ήδη επεξεργασμένη πληροφορία , ούτως ώστε να επιταχυνθεί η παραγωγή νέων σχεδίων ενεργειών που να αντιστοιχούν στις μεταβεβλημένες συνθήκες του περιβάλλοντος.

Ο RDA^* αποτελεί επέκταση του αλγορίθμου A^* , ο οποίος είναι ένας από τους διασημότερους αλγορίθμους της σχετικής βιβλιογραφίας και πάνω στον οποίο βασίζονται πολλοί από τους σχεδιαστές ενεργειών ($planners$) τελευταίας γενιάς. Η συγκεκριμένη επέκταση είναι προσαρμοσμένη για μη-στατικά περιβάλλοντα συγκεκριμένων χαρακτηριστικών. Συγκεκριμένα, μπορούν να αντιμετωπιστούν δυναμικά σύνολα-στόχων ($goal - sets$) και μεταβαλλόμενα κόστη ενεργειών.

Η πειραματική μέθοδος που χρησιμοποιήσαμε για την εκτίμηση της απόδοσης του αλγορίθμου είναι η εξής. Πρώτα, παράγεται ένα σχέδιο ενεργειών για τις αρχικές συνθήκες του περιβάλλοντος. Κατόπιν, θεωρώντας ότι το σχέδιο έχει εκτελεσθεί μέχρι ενός συγκεκριμένου σημείου, προκαλούνται αλλαγές είτε στο σύνολο-στόχων του είτε στα κόστη κάποιων ενεργειών του. Τελικά, εκτελούνται ο RDA^* και ο A^* . Τα διάφορα περιβάλλοντα και τα αντίστοιχα προβλήματα που χρησιμοποιήθηκαν για τα πειράματα, προέρχονται από τα καθιερωμένα προβλήματα συγκριτικής αξιολόγησης.

Τα πειραματικά αποτελέσματα υποδεικνύουν ότι η απόδοση του αλγορίθμου εξαρτάται από τους επόμενους παράγοντες: τον λόγο του μεγέθους του αρχικού γράφου αναζήτησης προς το μέγεθος του αντίστοιχου τελικού γράφου, την παράμετρο διακλάδωσης ($branchingfactor$) του, την πυκνότητα του, το ποσοστό του ήδη εκτελεσθέντος σχεδίου και τον όγκο των αλλαγών στο περιβάλλον. Για αραιούς γράφους και μικρές έως μέτριες αλλαγές του περιβάλλοντος, ο RDA^* υπερέχει του A^* ωσάν αφορά την ταχύτητα, σε ποσοστό που κυμαίνεται από 10% έως 80%, εφόσον το ποσοστό του ήδη εκτελεσθέντος σχεδίου δεν ξεπερνά το 40% με 50%.

Εκτιμούμε ότι η παρούσα εργασία μπορεί να παρέχει χρήσιμες ιδέες και υποδείξεις που θα διευκολύνουν την ανάπτυξη αποδοτικότερων τεχνικών επιδιόρθωσης σχεδίων ενεργειών, δεδομένου του ότι ο αλγόριθμος A^* αποτελεί την ραχοκοκκαλιά πολλών σύγχρονων σχεδιαστών ενεργειών. Επιπροσθέτως, πιστεύουμε ότι η παρούσα δουλειά μπορεί να βελτιωθεί και να επεκταθεί περαιτέρω, ενσωματώνοντας νέα χαρακτηριστικά, όπως μια μη-κεντρική ($decentralized$) προσέγγιση και μια λειτουργικότητα ανταπόκρισης αμέσου χρόνου ($real - time$).

Ensuite, ils glorifièrent les avantages des sciences: que de choses à connaître! que de recherches – si on avait le temps!

— Gustave Flaubert, *Bouvard et Pécuchet*

A la desaforada esperanza, sucedió, como es natural, una depresión excesiva. La certidumbre de que algún anaquel en algún hexágono encerraba libros preciosos y de que esos libros preciosos eran inaccesibles, pareció casi intolerable.

— Jorge Luis Borges, *La biblioteca de Babel*

Every time a scientific paper presents a bit of data, it's accompanied by an error bar — a quiet but insistent reminder that no knowledge is complete or perfect.

— Carl Sagan, *The Demon-Haunted world*

Acknowledgements

First and foremost, I would like to thank my supervisor, Pr. Dimitris Plexousakis, for his guidance and support throughout the writing of this work. His believing in me and the patience he exhibited helped me pull through the difficult times that I encountered during this journey. I would also like to thank Pr. Ioannis Tsamardinos and Dr. Giorgios Flouris, members of the thesis committee, for their helpful comments and suggestions.

Moreover, I would like to express my gratitude to Dr. Theodoros Patkos and Dr. Giorgios Flouris(again), both post doctorate researchers at the Foundation of Research and Technology, who provided me with encouragement, useful advices and excellent comments when necessary. This work would not have been possible if it weren't for their constant support. In addition, I feel obliged to thank the students and staff members (too many to name!) of the Information Systems Laboratory at the Foundation of Research and Technology.

Finally, I would like to thank my family: my sister, Anthousa, my late brother, Lucas, and my late parents. Especially, I would like to dedicate this work to my father who passed away during the preparation of this work.

Chapter 1

Introduction

1.1 Planning

Planning is concerned with the finding of a set of actions that, when executed by one or more agents, will achieve a desired result.It is one of the oldest and most fundamental research areas of the Artificial Intelligence, with many algorithms, originally conceived for planning problems, permeating other fields of computer science.

In practical level, the importance of planning is demonstrated by the wide range of domains where its techniques and methods are applied successfully. For example, space missions[4][6], logistics scheduling[1], supply chain design[25], path finding[30], emergence response[8] and robotic assembly[15] are just a few of the areas where planning plays a decisive role.

Moreover, as the number of autonomous entities increases , whether these are robots or just virtual bots, planning gains even greater significance , since it focuses on the study of the interactions between environments and agents, which are autonomous entities per se.

A special case of planning arises when the execution of a plan fails. The term **re-planning** is used to refer to the production of a new plan in this case. Depending on the way in which it is carried out, re-planning falls in the next two categories: **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, the previous computational effort is utilized.

Consequently, applying the latter approach, enables, in certain cases, the re-utilization of a part of the already processed data that was used for the original plan, which, in turn, might accelerate the finding of the new plan in comparison to re-planning from scratch. However, in cases where the environment has changed in a significant degree, it is likely that much computational effort will be wasted for the processing of information that is no longer valid, and, consequently, plan repairing will result being less efficient than re-planning from scratch in terms of speed.

1.2 Motivation

Due to the simplified assumptions upon which many types of planning are based, plan invalidations occur frequently in many real-world problems. Namely, usually, the knowledge of the environment is assumed to be complete and accurate and its corresponding representation precise. Moreover, the actions of the agents are supposed to be deterministic and the cost of each action to be known in advance. Unfortunately, all the previous assumptions fail frequently for a number of reasons.

First, a complete knowledge of the environment in realistic scenarios is very often impossible. Besides, even when this is achievable, it results in huge quantities of information, the utilization of which is computationally infeasible. In addition, neither the agents behaviour, nor the outcome of the actions they execute can be predicted with total accuracy, since the planning agents may break down or perform a task wrongly or partially.

Furthermore, an extra factor that hinders frequently the unobtrusive execution of plans, is the dynamic nature of the environments where the latter are deployed. These real-world domains, in contrast to the theoretical models which simulate them, are rarely static. On the contrary, they are susceptible to changes that could render an executing plan sub-optimal or make its full realization impossible. Also, another parameter of contingency is that in many domains the original objective for which the plan was produced may change before its execution is completed.

From the previous reasons, that is, the widespread utilizations of planning in a diversity of domains and the high percentage of plan failures that occur in such cases, it becomes evident that the study of the re-planning problem has not just theoretical value, but is of significant practical use.

As a step towards this direction, in the context of this thesis, *we investigate the conditions under which, plan repairing is more efficient than re-planning from scratch*. For this, we focus our attention on A^* algorithm which is one of the most popular and studied algorithms in the field of Artificial Intelligence, and is considered the standard planning algorithm [37].

Our contribution lies in the development of a novel algorithm, Repairing Dynamic A^* (henceforth RDA^*), which extends A^* in such a way that it can be used for plan repairing. Namely, RDA^* 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.

1.3 Thesis structure

This thesis is structured as follows: In Chapter 2, we provide the necessary background knowledge for A^* , which is the algorithm upon which RDA^* is based. In Chapter 3, we present the related work on re-planning and plan repairing in particular. In Chapter 4, we describe RDA^* algorithm in full detail. In Chapter 5, we provide some simple examples which illustrate how the algorithm is executed. In Chapter 6, we present the findings of RDA^* experimental evaluation. In Chapter 7, we summarize our conclusions

and elaborate on the possible directions of future work.

Chapter 2

Background

2.1 Notation

We use the term **replanning** to refer to the planning process that takes place after the execution of a plan has stopped due to changes in the environment. With the terms **initial plan** and **updated plan** we refer to the plans produced originally and after the changes in the environments took place, respectively.

We use the term ris (abbreviation for replanning initial state) to denote the state from where the replanning begins. We distinguish between **plan repairing**, or just repairing, and **re-planning from scratch**. In the former case, information and computational effort deriving from the production of the initial plan is utilized in the search for the updated plan, whereas, in the latter case, any previous information or other computation relative to the original plan is discarded.

A state s_a is called **parent_state** of another state s_d, if s_a has generated s_d. With the term p-value$_{s_a \to s_d}$ we denote the resulting g-value of s_d after its generation of another state s_a. Likewise, ac$_{p_a \to s_d}$ is the corresponding action, that leads from s_a to s_d. Conversely, if s_a is the parent_state of state s_d, then s_d is the **successor_state** or **child_state** of s_a. We use the term **goal_state** to denote any state that satisfies the problem's goal-set.

A state s_p is called **lgv-parent_state** of another state s_d, if s_p has generated s_d and p-value$_{s_p \to s_d} \leq$ p-value$_{s' \to s_d}$ for every other state s' that is parent_state of s_d. From the previous, it is derived that the g-value of a state s_d is equal to p-value$_{s_p \to s_d}$, where s_p is the lgv-parent_state of s_d. The action with which a state s_d is generated by another state s_a, is called **generating_action**$_{s_a \to s_d}$. The generating action of the parent state is called **lgv-generating_action**.

If s_1 is **parent_state** of s_2,s_2 is **parent_state** of state s_3,... and s_{n-1} is **parent_state** of s_n, then the sequence of actions $ac_{s_1 \to s_2}, ac_{s_2 \to s_3}, ..., ac_{s_{n-1} \to s_n}$ is called **path**$_{s_1, s_2, ..., s_{n-1}, s_n}$. The cost of a path p is equal to the sum of its actions' costs. A path p$_{s_1, ..., s_n}$ between two states s_1 and s_n is called optimal, if any other path p'$_{s_1, ..., s_n}$ between s_1 and s_n has at least the same cost.

For every state s the priority queue where its **parent_state** are stored, is called **parent_queue**. The position that a state s' holds in the **parent_queue** is determined by

the p-value$_{s' \rightarrow s_2}$, with lower values corresponding to higher priorities. Likewise, the hash table where the corresponding actions are stored is called **action table**. A state s is called **valid** if there is at least one path between s and ris, otherwise it is called **invalid**. We use the term **search tree** to denote the open and closed list that are used during a RDA^* execution. **Initial search tree** refers to the final search tree after the production of the initial plan. For a given search tree, the **branching factor** is the average number of successor_states of a state and the **ancestry factor** is the average number of ancestor_states of a state.

2.2 A^* algorithm

A^* [14][33] is one of the most popular algorithms of Artificial Intelligence [18][16][31], with some of its most common uses including graph traversal and path-finding. Its key idea is the utilization of a heuristic value, that "guide" the search, a characteristic that is absent from other search algorithms. As a result, its performance depends on the quality of the function that generates the heuristic values. There are two variations of A^* : **tree search** and **graph search**.

At each step of the tree search, a state is selected, examined, and expanded. The selection of the state is determined by a value assigned to it, the **f-value**, which is, in turn, the sum of two other values: the **g-value** and **h-value**, respectively. The former refers to the sum of the actions' cost of the path from the initial state to it, whereas the latter is an estimation of the cost from it to the goal-state.

After the selection of the state, a check takes place, in order to determine if the selected state satisfies the goal-set. If this holds, then the search stops and the corresponding plan is extracted. Otherwise, the state is expanded by generating all its successor states. When a state is generated, its g-value and f-value are calculated. Moreover, a pointer to its parent state, .i.e. the state by which it was generated, is stored along with a reference to the corresponding action for this transition.

In case a state is generated again, its current g-value is compared to the g-value that ensues from the new generating state. If the new generation results in a smaller g-value, then the pointer to the parent state and the reference to the corresponding action changes appropriately and the state's g-value and f-value are updated respectively. The algorithm terminates either when a solution is found or when there is no state that can be expanded, in which case a plan does not exist.

Assuming that a state that satisfies the goal set has been found, by following the parent state pointers from the aforementioned state to the initial state, we have the reverse order of the successive states of the corresponding plan. Likewise, the sequence of action that corresponds to this traversal constitutes the plan.

For the whole procedure to be carried out, it is necessary that two auxiliary collections are utilized. The first is a priority queue called **open list**, and the second is a set, referred to as **closed list**. The priority for each element of the open list is determined by its f-value: the element with the lowest f-value is the one with the highest priority. The two lists are initially empty. Before, the algorithm begins, the initial state is added to the open list.

After a state is expanded, it is removed from the latter and added to the closed list. Conversely, when a state is generated it is added to the open list. The two structures can be represented in a compact way with a tree, which is, usually, referred to as **search-tree**, the nodes of which correspond to the states of the search. The tree's leaves correspond to the elements of the open list, i.e. the states that are candidate for expansion, whereas, the rest of the nodes correspond to the elements of the closed list respectively, i.e. the already expanded states. If the h-values that are used are consistent, then this variation of the algorithm is guaranteed to find an optimal solution, if a solution exists, and, moreover, the algorithm will not generate more states than any other algorithm that uses the same h-values [1]. A 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 the goal from S_M is not greater than the step cost of getting to S_N plus the estimated cost of reaching the goal from S_N :$hval_{S_N} \leq c + hval_{S_M}$.

Graph search differs from tree search in two points. First, an extra check takes place when a state is generated and, second, there is modification w.r.t. the termination of the algorithm in order for the optimality of the solution to be guaranteed. Specifically, 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. Otherwise, the algorithm continues as in the case of tree search.

Similarly, if the state is already in the open list, its current f-value is compared to its old f-value and if the new f-value is smaller, the state is removed from the open list and re-inserted in it with the new f-value. If the new f-value is equal or greater than the old f-value, the graph search continues as in the tree search. Usually, two hash tables are utilized for the purpose of checking if a state is inserted in the lists, one for each list respectively.

Regarding the termination criterion in the case of graph search, 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 with an f-value that is smaller than the cost of the plan already found[2]. In this case, it is not required that the h-values are consistent, but it suffices to be admissible. A h-value of a state is admissible, if it is not greater than the cost of the optimal path from the given state to a goal-state. Note that a consistent h-value is always admissible, but the opposite does not always hold.

2.3 Pseudo Code

[1] In case of algorithms that use identical h-values, the same tie-breaker criterion for states with same f-values is required.

[2] If more than one has been found during the search, the one with the lowest cost is kept.

```
1  OPEN⟵ new PriorityQueue();
2  CLOSED⟵ new Set();
3  OPEN.add(initial_state);
4  plan ⟵ NULL;
5  while OPEN is not empty do
6      curState ⟵ OPEN.poll();
7      if curState satisfies goal set then
8          plan ⟵ ExtractPlan(curState);
9          break;
10     foreach Applicable action ac of curState do
11         genState ⟵ curState.apply(ac);
12         gVal ⟵ curState.gValue + ac.cost;
13         if OPENdoes not contain genState then
14             generateNewState(genState,curState );
15             OPEN.add(genState);
16         else
17             if gVal < genState.gValue then
18                 OPEN.remove(genState);
19                 generateNewState(genState,curState );
20                 OPEN.add(genState);
21         CLOSED.add(curState );
22     end
23 end
24 return plan;
```

Algorithm 1: A^* algorithm

```
1  genState.gValue = aVal;
2  genState.hValue = ComputehValue(genState);
3  genState.fValue = genState.gValue + genState.hValue;
4  genState.parent ⟵ currentState;
5  genState.gValue ⟵ ac;
6  mark genState as valid and informed ;
```

Algorithm 2: Generation of a state

Chapter 3

Related work

In this section we present a review of the the existing and related works on plan repairing. We concentrate exclusively on approaches related closely to classical planning, since we consider that the review of other kinds of methods that are used for re-planning, such as contingent [12] and conformant planning [19] or hidden Markov Models[20], lies beyond the scope of this thesis. In order to facilitate the comparison between RDA^* and the other algorithms,we distinguish between algorithms that, as RDA^* , have been inspired from A^* and the rest of plan repairing algorithms, presenting each category in a different subsection. A succinct comparison between RDA^* and the other repairing algorithms is provided in the end of the next chapter.

3.1 Plan repairing algorithms based on A^*

Over the last years, a significant number of A^* - inspired plan repairing[1] algorithms has been developed, the majority of which is tailored to single-agent robotics problems. These algorithms falls typically into two main categories regarding their capacities for plan-repairing: a) algorithms that are specialized in addressing modifications of the original goal-set[34] [21][26] [13] and b) algorithms that are specialized in addressing changes of the actions costs[23] [38][22]. Finally, there are few other algorithms that can cope both with goal-set modifications and actions costs changes[35][36][37].

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* [34], and D*-lite[21]. Both of them utilized a backwards-directed search from the goal state to

[1]Typically, these algorithms are referred to as re-planning algorithms. However, we consider that this term might be misleading, since it can be confused with re-planning from scratch. Therefore, we use the term plan repairing instead.

the current state, saving, this way information, which allows fast plan production when changes in the environment occur. D*-lite has been further extended since, with some of its extensions have been used in the navigation algorithms for a Mars Rover [7], and in the DARPA Urban challenge competition [27].

In [35] is presented the Generalized Adaptive A* (GAA*) algorithm. 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 falls close to our work, is MP-GAA*[17], where some of the best paths for some nodes are stored. Most recently, there have been implemented Generalized Fringe-Retrieving A*[36] and Moving Target D* Lite[37] which, in the same way as GAA*, can address both goal-set modifications and actions costs changes.

Finally, for each of the algorithms a further distinction can be made depending on whether the optimality of the plan is the main concern. Namely, in some problems, the main objective is the production of a new plan within a certain time window after the invalidation of the original plan. In these cases, any plan with a cost which does not transcend a certain threshold w.r.t the cost of the optimal plan is considered a valid solution to the corresponding problem. From the algorithms described in this section, [34] [26] [22] [27] and [13] hold this real-time property.

3.2 Other plan repairing algorithms

[24] is a recent line of work that investigates the recovery after plan failures in multi-agent environments where the objective of plan repairing is the minimization of the exchange of messages between agents and not the plan optimality. Apart from the theoretical analysis that is conducted for this type of problem, an algorithm is presented that is tailored to domains where communication complexity matters more than time complexity.

A work resembling in certain aspects the previous, is [29] where a multi-agent decentralized approach to plan repairing is presented. In this case, each agent of the system is responsible for controlling the actions it executes, and for independently repairing its own plan when it detects an action failure, by following a local strategy. As in the previous case, optimality is not a central issue.

In [9] the concept of plan stability is introduced. The authors of this work, argue that plan stability is an important property for many planning problems and present a plan repairing algorithm having as first priority the minimal perturbation of the original plan instead of the optimality of the new plan. A similar approach to the previous, is [3], where the main objective of the plan repairing algorithm is the minimal perturbation of the original plan.

The notion of bookings and commitments are used in [39]. In this case, the plan repairing procedure is considered a sequence of refinement and unrefinement steps, which aim at producing a new plan that does not violate the agents original bookings and commitments. Again, in this case, the plan optimality is not of central importance. Likewise, [5] presents a theoretical analysis of the way in which the agents commitments are linked to the plan repairing procedure.

Finally, [10] is concerned with plan adaptation which can be regarded as an almost identical technique to plan repairing. The planning system, in this case, utilizes specialized heuristic search techniques in order to solve the plan adaptation tasks through the repairing of certain portions of the original plan. The procedure for the plan adaptation is incremental: first, a sub-optimal plan is found and, then, if possible, better solutions w.r.t. cost are sought.

Chapter 4

Repairing Dynamic A^*

4.1 General

RDA^* is an informed search re-planning algorithm, suited for the repairing of sequential plans. In order for the algorithm to be able to be used for re-planning, RDA^* must have been used for the production of the initial plan. RDA^* can address two types of environment's changes: a) goal-set modifications and b) actions' costs alterations. RDA^* is based principally 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 tree is not created from the scratch as in A^* . Instead, in the beginning of the algorithm the initial search tree is retrieved and used for the consequent 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.

RDA^* has two variations, each one tailored to a different replanning requirement. The first is fitted to address goal-set modifications, whereas, the second can address actions costs changes. Moreover, each of the variations can be executed in two different ways. Namely, the first variation has a general-case sub-variation that can address any type of goal-set modification and, thus, can be applied always. The second is applicable only in cases when the modified goal-set is a super-set of the original goal-set, and it is used in this case due to it being more efficient. Similarly, for the variation addressing costs' changes, there is a general-case sub-variation that can be applied always and a special one, that is applicable only in cases when none of the actions' costs have decreased. In total, there exits four sub-variations of RDA^* , which we will describe in the next section. The following theorem holds for RDA^* :

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

Proof. The proof is presented in the appendix. □

4.2 Differences with A^* and novel concepts

Although, RDA^* has many commons with A^* , the adjustments made and the novel concepts and techniques that are utilized have as a result that in certain points RDA^* diverges significantly from A^* .

4.2.1 re-Generation of a state

First, the way in which the re-generation of a state is handled is different. In the case of A^* , if the re-generation results in a smaller p-value than the state's g-value, then, the state's parent pointer and the action's reference changes and its g-value is updated respectively. RDA^* , instead, uses a special structure for each state, called **parent_queue**, where the generating states are inserted, regardless of the resulting p-value and a Hash Table, called **action table**, for the storage and retrieval of the generating actions.

4.2.2 Informed and valid states

Moreover, an extra check that takes place before the expansion of a state and a new routine for the informing of the states are introduced. Namely, a state has to be checked for being informed before it is tested for satisfying the goal set. If it is found to be uninformed, the special routine for its informing is executed in order to be determined if the state is valid or invalid. In the former case, RDA^* continues in the same way as A^* , whereas, in the latter, the current state is discarded and the search continues with the selection of a new state from the open list. By default, when the repairing starts, all the states of the search tree are considered uninformed except of the new initial state which is set as informed and valid when the algorithm begins.

4.2.3 Informing procedure

Since the search tree that is used contains states who have been generated during the production of the original plan, their pointers and p-values may no longer be valid. The informing procedure updates this information by searching for the optimal path, in case one exists, from the ris to the state being informed. It can be achieved in two different ways: **lazily** or **fully**. Lazy informing, exploiting the fact that the parent_states are stored in sorted order in the parent_queue, reduces, in principle, the number of paths that are traversed in comparison to full informing and is, therefore, in general, faster than the latter. However, if any of the actions costs has decreased only full informing can be used, because, otherwise, the solutions returned by the algorithm might not be optimal.

The procedure for the full informing of a state is the following(pseudo-code at page 26). First the state being examined is marked as pending (line 1). All its parent states are examined for being informed (lines 5 and 16). For any parent state found not to be informed and not to be pending, the procedure of full informing is followed (lines 6 and 17). If an parent state is pending, then the value of a variable that keeps track of the number of

partially informed ancestor states increases by 1[1] (lines 10 and 13), while the state being informed is added in a special list of the pending state, called successors list lines 11 and 14). This way, when the informing of the pending state finishes, it can inform, in turn, the states contained in this list.

If a valid parent state is found, its p-value is re-calculated and compared with the state's g-value and if found smaller, then it is set as the state's parent, while the parent action and the g-value are updated accordingly (lines 24-28). After the examination of the parent states finishes, if the state has any valid ancestor, it is marked as informed and valid (line 31). Moreover, if the number of its partially informed parent states is equal to zero, then the states contained in its successors list, are informed (lines 32-33). Namely, for each successor state, its g-value is compared to the p-value of the state, and if found greater, then the parent state and its parent action and g-value are updated accordingly. Next, the successor state's number of partially informed parent states is reduced by one. If it is equal to zero, then the successor state informs with the same procedure the states contained in its own successors list (the pseudo code of the updating procedure can be found at page 29).

In case no valid parent states have been found, if the number of the state's partially informed parent states is equal to zero, then the state is marked as informed and invalid (lines 35-36). Next, parent statess contained in its successors list, are updated: for each, the value of the variable counting the partially informed parent states is reduced by one. If the new value is equal to zero, then the successor states, updates in the same way the states contained in its successors list updated (pseudo code at page 29). Finally, the state is reset from pending (line 38).

The lazy informing is carried out in the same way with the full(pseudo-code at page 27), with the exception that the procedure stops if a valid parent state has been found and the next parent state that is to be examined does not have a smaller p-value (lines 16-17). The examination of the parent states follows their sorting order. That is, it begins with the parent state having the lowest p-value, i.e. the lgv-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.

4.2.4 Storing of initial closed and open list

When the search for the plan finishes, the closed and open list are stored, so that they can be used in case of re-planning. Before the open list is saved, the last removed state from it, is re-inserted in it. Subsequently, when the algorithm is executed, the previously save lists and retrieved and used. It should be noted that the algorithm can be utilized, without any additional adjustments, for the repairing of already repaired plans. We leave, however, the assessment of RDA^* performance for this type of repairing for a future work.

[1] A partially informed parent state is either pending or has one or more pending parent states.

4.2.5 Traversal of the closed list and Validation of the open list

Finally, in one case, the initial closed 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 which satisfy the new goal-set have been found, the one having the lowest g-value is returned as solution and the algorithm terminates. Similarly, in two sub-variations of the algorithm the open list is validated before the main search starts. Namely, every state is informed, fully in case of decreased actions costs and lazily otherwise, and, if it is valid, it is re-inserted in the open list. Moreover, in one sub-variation the h-value of the state is calculated again before the re-insertion.

In summary, RDA^* differentiates from the classic A^* in the following ways:

- The open and closed list from the previous plan are not created again. Instead the final search from the search for the original plan is utilized. This way the new search starts from a search-tree that is already constructed.
- In some sub-variations of the algorithm, the retrieved search tree is traversed and pre-processed before the main search begins.
- For each state, all pointers to its parent states are stored along with the corresponding generating actions and p-values.
- The states can be either informed or uninformed. The uninformed states are updated by the informing procedure which results in their validation or invalidation. Any invalid state is discarded.

4.3 Description

In this section we provide an elaborate description of the algorithm. In order to facilitate the understanding of the method, we will refer, when necessary, to the corresponding lines of the pseudo-code that is presented in the next section. Note, that the algorithm is carried out in the same way both for planning and plan re-repairing, with the only difference being that in the former case the closed that is retrieved is empty, while the open list contains only the initial state.

4.3.1 Repairing for goal-sets modifications: General Case

This variation of the algorithm can be applied in any type of goal-set modification (pseudo code at page 22). First, the new initial state is marked as informed and valid and, then the initial closed list is retrieved and traversed in the way described in the previous section (lines 2-6). In case the traversal has not resulted in the finding of a solution, the initial open list is retrieved and is validated (lines 7-8). Next, the main part of the algorithm begins (lines 9-43). At each step, the state having the lowest f-value is removed from the open list and, subsequently, the state is tested for satisfying the goal-set(lines

11-13). If it does, then the search stops, the corresponding plan is returned as solution and the algorithm terminates.

If the testing for the satisfaction of the goal-set fails, then the loop for the generation of the currently selected state's successor states is executed (lines 14-39). First, the corresponding successor state is generated and its p-value is calculated. If the successor state is not contained in the open or the closed list, then its g-value is set equal to the previously calculated p-value. Accordingly, the currently selected state is set as its parent_state the generating action is set as parent action. Finally, the successor state is marked as informed and valid and is inserted in the open list. Finally, the state is marked as informed and valid and is re-inserted in the open list.

If the successor state is contained in the closed list, then it is examined for being informed and, if found uninformed, the routine of lazy informing is executed (lines 21-22). Next, if it is found invalid the procedure continues in the same way as previously: the state's g-value is set to the calculated p-value and the parent-state point and action reference are updated accordingly, while the state is marked as informed and valid.

If the successor state is found to be valid, then its g-value is compared against the calculated p-value and if it is smaller, the state's h-value is re-calculated and the state's g-value, lgv-parent state pointer and action reference are updated (lines 27-28). Otherwise, a pointer for the currently selected pointer and a reference for the corresponding action is inserted in the successor state's parent queue and action queue respectively (lines 35-36). In contrast to the previous two cases, that is, the first generation of a state or the re-generation of an invalid state, the state is re-inserted in the open list, only if its g-value has been reduced.

After the generation of all of its successor states, the currently selected state is inserted in the closed list(line 37) and the algorithm continues as previously with the selection the state having the lowest f-value which, is, in turn tested and expanded as described previously. The algorithm terminates returning a plan, either when a state satisfying the new goal-set is found or when the open list becomes empty in which case the plan is null, since a solution does not exist for the problem. Before this, the closed and open list are saved (lines 40-42).

4.3.2 Repairing for goal-set modification : Special Case

This variation of the algorithm can be applied when the new goal-set is a superset of the initial goal-set (pseudo code at page 34). This means that any state that satisfies the new goal-set will satisfy the new goal-set as well. The algorithm in this case is executed in the same way described previously except for the next points. First, the closed list is not traversed , since, by definition, there is no state in the closed list that satisfies the initial goal-state and, therefore, no state can be found in the closed list that satisfies the new goal-set.

Moreover, the open list is not validated before the main loop of the algorithm begins. Consequently, any selected state from the open list might be invalid. Because of this possibility, a extra check is introduced (lines 6-13). If a selected state is not informed, the procedure for its lazy informing is followed. Next, if it is found invalid, it is discarded. In

case the state is valid, its f-value is updated and if it found greater than the f-value of the head of the open list, it is re-inserted in it. Otherwise, the rest of the main loop is executed in the same way as in the previous subsection, with the generations of the successor states.

The other difference is that the h-value of a valid state contained in the closed list, is not calculated again since the h-value remains admissible. This holds, because the new plan's cost will be at least the same as the initial plan's cost[2]. Note that the calculation is omitted for efficiency reasons since it is considered computationally expensive.

4.3.3 Repairing for actions costs alterations: General Case

The description will be brief, since the algorithm is executed in its most part in the same way as in the general case of the goal-set modification case. There are three differences, however. First, the traversal of the closed list does not take place, for the same reasons as in the case of the increased goal-set. Moreover, the states are informed fully and not lazily. Also, the h-values of the valid states are not re-calculated, since the goal-set has not changed. The corresponding code can be found at page 35.

First, the repairing initial state is marked as informed and valid, the initial open and closed lists are retrieved and the validation of the former takes place (lines 2-5). Next, the head of the open list is removed and examined for satisfying the goal set. If it does, the corresponding plan is extracted and returned and the algorithm terminates (lines 7-10). Otherwise, the successor states are generated (lines 12-13). If a successor state is generated for the first time, its parent state, parent action and g-value are set, the state is marked as informed and valid and it is inserted in the open list (lines 15-16).

If it is contained in the open or the closed list, it is examined for being informed. In case it isn't, the procedure of full informing is followed (lines 18-19). Next, if the state is not valid, the same steps, as with a state being generated for the first time, are followed. The state's lgv-parent state, parent action and g-value are updated accordingly, the state is marked as informed and valid and it is inserted in the open list (lines 21-22).

If the generated state is valid, then its g-value is compared to the p-value of the state that has re-generated it and, if greater, the latter becomes its lgv-parent, the parent action and g-value are updated and the state is re-inserted in the open list. Otherwise, the parent state and the corresponding generating action are inserted in the parent queue and action table respectively (lines 32-33).

Finally, the algorithm terminates returning a solution when one has been found, or returning an empty plan when the open list has been emptied. Before this, the open and closed lists are saved (lines 37-39).

4.3.4 Repairing for actions costs alterations: Special Case

This sub-variation is applicable only in cases when the changes of the actions costs are increases(pseudo code at 36). There are three differences with the sub-variation that is applied in the general case. First, the original plan is retrieved, and, then , it is lazily

[2]Otherwise, if a plan with lower cost is found, then this plan satisfying the initial goal state by definition and having lower cost than the original plan will render the original plan sub-optimal which is a contradiction.

informed in order for its cost to be updated (lines 5-6) . In addition, a new termination criterion is introduced. Namely, if the f-value of the state that has been removed from the open list, is not smaller than the updated cost of the original plan, then the algorithm stops and the informed original plan is returned as solution (lines 9-10).

Moreover, each state that is removed from the open list, is examined for being informed, and if not, it is lazily informed (lines 11-13). If invalid, it is discarded, and the head of the open list is removed. In case the state is valid, its f-value is updated and, if greater than the f-value of the head of the open list, it is re-inserted in it (lines 15-18). The rest of the algorithm is executed in the same way as in general case of actions costs alterations.

4.4 Comparison with other approaches

Concluding this section, we can make the following remarks w.r.t. the related work that was presented in the previous chapter. Although, there exists a number algorithms that are based on A^* and which re-utilize parts of the old search tree, these approaches differ from our work in significant ways. First, with the exception of [17],the re-use concerns exclusively the storing of the old g-values and h-values. Besides, even in the case of [17], where the best paths for certain tree nodes are kept, this approach concerns only a small fragment of the original search tree. Moreover, the algorithm is applicable only in cases of single-agent path planning.

Most importantly, these algorithms suffer from a significant limitation, since they are applicable only in certain single-agent settings where the terrain can be mapped to the search tree. Also, another limitation of these algorithms has to do with the type of goals that they can handle. Namely, in their case, the goal-set is always consisted of a single goal which refers to the visiting of a specific location in the problem terrain, which via an appropriate mapping corresponds to a node in the search tree. In contrast, RDA^* is more versatile and flexible, since it can be used in a wider variety of single-agent and multi-agent domains and can cope with goal-sets consisted of multiple sub-goals and with goals that refer to broader tasks. Finally, the majority of these algorithms are specialized in either goal-set or costs modifications, whereas, RDA^* can cope with both of these changes.

Regarding the rest of the presented plan-repairing algorithms, we can observe the following. In most of the cases, the weight is not put in the optimality of the solution but on other aspects of the plan such as the stability of the original plan or the respect of the agents commitments, rendering these approaches very different from ours, where only the optimal solutions are considered valid. Moreover, even in the cases when optimal solutions are sought, the performance in terms of speed is not the main priority of these systems and algorithms.

4.5 Pseudocode

```
1  plan ⟵ NULL;
2  mark newInitialState as valid and informed ;
3  CLOSED ⟵ previousCLOSED;
4  plan ⟵ searchCloseList(CLOSED);
5  if plan is not NULL then
6  |  break;
7  OPEN ⟵ previousOPEN;
8  validateOpen(OPEN);
9  while OPEN is not empty do
10 |  currentState ⟵ OPEN.poll();
11 |  if currentState satisfies goal set then
12 |  |  plan ⟵ ExtractPlan(currentState);
13 |  |  break;
14 |  foreach Applicable action ac of currentState do
15 |  |  genState ⟵ currentState.apply(ac);
16 |  |  pVal ⟵ currentState.gValue + ac.cost;
17 |  |  if OPENor CLOSED does not contain genState then
18 |  |  |  generateNewState(genState,currentState );
19 |  |  |  OPEN.add(genState);
20 |  |  else
21 |  |  |  if genState is not informed then
22 |  |  |  |  lazy_inform(genState);
23 |  |  |  if genState is not valid then
24 |  |  |  |  generateNewState(genState,currentState );
25 |  |  |  |  OPEN.add(genState);
26 |  |  |  else
27 |  |  |  |  if pVal < genState.gValue then
28 |  |  |  |  |  generateNewState(genState,currentState );
29 |  |  |  |  |  if OPEN contains genState then
30 |  |  |  |  |  |  OPEN.remove(genState);
31 |  |  |  |  |  if CLOSED contains genState then
32 |  |  |  |  |  |  CLOSED.remove(genState);
33 |  |  |  |  |  OPEN.add(genState);
34 |  |  |  |  else
35 |  |  |  |  |  genState.ParentQueue.add(currentState );
36 |  |  |  |  |  genState.ActionTable.add(ac);
37 |  |  CLOSED.add(currentState );
38 |  end
39 end
40 OPEN.add(plan.finalState);
41 previousOPEN ⟵ OPEN;
42 previousCLOSED ⟵ CLOSED;
43 return plan;
```

Algorithm 3: RDA^* algorithm variaton for goal-set modification: general case

```
1  plan ⟵ NULL;
2  OPEN ⟵ previousOPEN;
3  CLOSED ⟵ previousCLOSED;
4  while OPEN is not empty do
5      currentState ⟵ OPEN.poll();
6      if currentState is not informed then
7          lazy_inform(currentState );
8          if currentState is not valid then
9              continue;
10         currentState.updatefValue();
11         if currentState.fValue > OPEN.head.fValue then
12             OPEN.add(currentState );
13             continue;
14     if currentState satisfies goal set then
15         plan ⟵ ExtractPlan(currentState);
16         break;
17     foreach Applicable action ac of currentState do
18         genState ⟵ currentState.apply(ac);
19         pVal ⟵ currentState.gValue + ac.cost;
20         if OPENor CLOSED does not contain genState then
21             generateNewState(genState,currentState );
22             OPEN.add(genState);
23         else
24             if genState is not informed then
25                 lazy_inform(genState);
26             if genState is not valid then
27                 generateNewState(genState,currentState );
28                 OPEN.add(genState);
29             else
30                 if pVal < genState.gValue then
31                     generateNewState(genState,currentState );
32                     if OPEN contains genState then
33                         OPEN.remove(genState);
34                     if CLOSED contains genState then
35                         CLOSED.remove(genState);
36                     OPEN.add(genState);
37                 else
38                     genState.ParentQueue.add(currentState );
39                     genState.ActionTable.add(ac);
40         CLOSED.add(currentState );
41     end
42 end
43 OPEN.add(plan.finalState);
44 previousOPEN ⟵ OPEN;
45 previousCLOSED ⟵ CLOSED;
46 return plan;
```

Algorithm 4: RDA^* algorithm variaton for goal-set modification : special case

```
1  plan ⟵ NULL;
2  mark newInitialState as valid and informed ;
3  OPEN ⟵ previousOPEN;
4  CLOSED ⟵ previousCLOSED;
5  validateOpen(OPEN);
6  while OPEN is not empty do
7      currentState ⟵ OPEN.poll();
8      if currentState satisfies goal set then
9          plan ⟵ ExtractPlan(currentState);
10         break;
11     foreach Applicable action ac of currentState do
12         genState ⟵ currentState.apply(ac);
13         pVal ⟵ currentState.gValue + ac.cost;
14         if OPENor CLOSED does not contain genState then
15             generateNewState(genState,currentState );
16             OPEN.add(genState);
17         else
18             if genState is not informed then
19                 full_inform(genState);
20             if genState is not valid then
21                 generateNewState(genState,currentState );
22                 OPEN.add(genState);
23             else
24                 if pVal < genState.gValue then
25                     generateState(genState);
26                     if OPEN contains genState then
27                         OPEN.remove(genState);
28                     if CLOSED contains genState then
29                         CLOSED.remove(genState);
30                     OPEN.add(genState);
31                 else
32                     genState.ParentQueue.add(currentState );
33                     genState.ActionTable.add(ac);
34         CLOSED.add(currentState );
35     end
36 end
37 OPEN.add(plan.finalState);
38 previousOPEN ⟵ OPEN;
39 previousCLOSED ⟵ CLOSED;
40 return plan;
```

Algorithm 5: RDA^* algorithm variation for actions costs modification: general case

```
1  plan ⟵ originalPlan;
2  mark newInitialState as valid and informed ;
3  OPEN ⟵ previousOPEN;
4  CLOSED ⟵ previousCLOSED;
5  lazyInform(plan);
6  bestScore = plan.gValue;
7  while OPENis not empty do
8      currentState ⟵ OPEN.poll();
9      if currentState.gValue ≥ bestScore then
10         break;
11     if currentState is not informed then
12         lazy_inform(currentState );
13         if currentState is not valid then
14             continue;
15         currentState.updatefValue();
16         if currentState.fValue > OPEN.head.fValue then
17             OPEN.add(currentState );
18             continue;
19     if currentState satisfies goal set then
20         plan ⟵ ExtractPlan(currentState);
21         break;
22     foreach Applicable action ac of currentState do
23         genState ⟵ currentState.apply(ac);
24         pVal ⟵ currentState.gValue + ac.cost;
25         if OPENor CLOSED does not contain genState then
26             generateNewState(genState,currentState );
27             OPEN.add(genState);
28         else
29             if genState is not informed then
30                 lazy_inform(genState);
31             if genState is not valid then
32                 generateNewState(genState,currentState );
33                 OPEN.add(genState);
34             else
35                 if pVal < genState.gValue then
36                     generateNewState(genState,currentState );
37                     if OPEN contains genState then
38                         OPEN.remove(genState);
39                     if CLOSED contains genState then
40                         CLOSED.remove(genState);
41                     OPEN.add(genState);
42                 else
43                     genState.ParentQueue.add(currentState );
44                     genState.ActionTable.add(ac);
45         CLOSED.add(currentState );
46     end
47 end
48 OPEN.add(plan.finalState);
49 previousOPEN ⟵ OPEN;
50 previousCLOSED ⟵ CLOSED;
51 return plan;
```

Algorithm 6: RDA^* algorithm variation for actions costs modification: special case

```
1  set currentState as pending ;
2  parent ⟵ currentState.getParent ;
3  currentState.gValue =∝;
4  NumberOfPartiallyInformedAncestors = 0 ;
5  if parent is not informed & is not pending then
6  |  full_inform(parent);
7  if parent is Valid then
8  |  state.gValue = parent.gValue + action.cost;
9  if parentis pending then
10 |  NumberOfPartiallyInformedAncestors + + ;
11 |  parent.StatesList.add(currentState );
12 if parent.NumberOfPartiallyInformedAncestors > 0 then
13 |  NumberOfPartiallyInformedAncestors + + ;
14 |  parent.StatesList.add(currentState );
15 foreach ancestorState in currentState .parentQueue do
16 |  if ancestorState is not informed & is not pending then
17 |  |  full_inform(ancestorState );
18 |  if ancestorState is pending then
19 |  |  NumberOfPartiallyInformedAncestors + + ;
20 |  |  ancestorState .StatesList.add(currentState );
21 |  if ancestorState.NumberOfPartiallyInformedAncestors > 0 then
22 |  |  NumberOfPartiallyInformedAncestors + + ;
23 |  |  ancestorState .StatesList.add(currentState );
24 |  if ancestorState is Valid then
25 |  |  if ancestorState.pValue < currentState.gValue then
26 |  |  |  currentState.parent ⟵ ancestorState;
27 |  |  |  currentState.action ⟵
   |  |  |  currentState.ActionTable.getAction(ancestorState);
28 |  |  |  currentState.gValue ⟵ ancestorState.pValue;
29 end
30 if state.gValue ≠∝ then
31 |  mark state as valid and informed ;
32 |  if successorState.NumberOfPendingAncestors = 0 then
33 |  |  updateSuccessorStates();
34 else
35 |  if NumberOfPendingAncestors = 0 then
36 |  |  mark state as invalid and informed ;
37 |  |  updateSuccessorStates();
38 reset currentState from pending ;
```

Algorithm 7: Full informing

```
1  set currentState as pending ;
2  parent ⟵ currentState.getParent ;
3  currentState.gValue =∝;
4  NumberOfPartiallyInformedAncestors = 0 ;
5  if parent is not informed & is not pending then
6  |  lazy_inform(parent);
7  if parent is Valid then
8  |  state.gValue = parent.gValue + action.cost;
9  if parentis pending then
10 |  NumberOfPartiallyInformedAncestors + + ;
11 |  parent.StatesList.add(currentState );
12 if parent.NumberOfPartiallyInformedAncestors > 0 then
13 |  NumberOfPartiallyInformedAncestors + + ;
14 |  parent.StatesList.add(currentState );
15 foreach ancestorState in currentState .parentQueue do
16 |  if ancestorState.pValue ≥ state.gValue then
17 |  |  break;
18 |  if ancestorState is not informed & is not pending then
19 |  |  lazy_inform(ancestorState );
20 |  if ancestorState is pending then
21 |  |  NumberOfPartiallyInformedAncestors + + ;
22 |  |  ancestorState .StatesList.add(currentState );
23 |  if ancestorState.NumberOfPartiallyInformedAncestors > 0 then
24 |  |  NumberOfPartiallyInformedAncestors + + ;
25 |  |  ancestorState .StatesList.add(currentState );
26 |  if ancestorState is Valid then
27 |  |  if ancestorState.pValue < currentState.gValue then
28 |  |  |  currentState.parent ⟵ ancestorState;
29 |  |  |  currentState.action ⟵
   |  |  |  currentState.ActionTable.getAction(ancestorState);
30 |  |  |  currentState.gValue ⟵ ancestorState.pValue;
31 end
32 if state.gValue ≠∝ then
33 |  mark state as valid and informed ;
34 |  if successorState.NumberOfPendingAncestors = 0 then
35 |  |  updateSuccessorStates();
36 else
37 |  if NumberOfPendingAncestors = 0 then
38 |  |  mark state as invalid and informed ;
39 |  |  updateSuccessorStates();
40 reset currentState from pending ;
```

Algorithm 8: Lazy informing

```
1  plan ⟵ null ;
2  cost = MAX;
3  foreach State current_state in CLOSED do
4      if current_state satisfies goal set then
5          if current_state isnot Informed then
6              lazy_inform(current_state);
7          if current_state is Valid then
8              if current_state.gValue < cost then
9                  plan ⟵ ExtractPlan(current_state) ;
10                 cost = current_state.gValue;
11 end
12 return plan;
```

Algorithm 9: Traversal of the closed list

```
1  newOpenList ⟵ new Priority Queue();
2  foreach state in OPEN do
3      if state isnot Informed then
4          if goalrepairing then
5              lazy_inform(state);
6          else
7              full_infrom(state);
8          if state is Valid then
9              if goalrepairing then
10                 state.hValue = ComputehValue(state);
11                 state.updatefValue();
12         newOpenList.add(state);
13 end
14 OPEN ⟵ newOpenList;
```

Algorithm 10: Validation of the open list

```
1  if state is informed and valid then
2      foreach successor_state in state.StatesList do
3          ac ⟵ state.gValue + ac.cost;
4          pVal ⟵ state.gValue + ac.cost;
5          successor_state.NumberOfPartiallyInformedAncestors–;
6          if pVal < successor_state.gValue then
7              successor_state.gValue = pVal;
8              genState.parent ⟵ state;
9              genState.action ⟵ successor_state.ActionTable.getAction(state);
10             if successor_state.NumberOfPartiallyInformedAncestors=0 then
11                 marked successor_state as informed and valid;
                   successor_state.updateSuccessorStates();
12     end
13 else
14     foreach successor_state in state.StatesList do
15         successor_state.NumberOfPartiallyInformedAncestors–;
16         if successor_state.NumberOfPartiallyInformedAncestors=0 then
17             marked successor_state as informed and invalid;
18             successor_state.updateSuccessorStates();
19     end
```

Algorithm 11: Updating of a state's successor states

```
1  newOpenList ⟵ new Priority Queue();
2  foreach state in OPEN do
3      if state isnot Informed then
4          if goalrepairing then
5              lazy_inform(state);
6          else
7              full_infrom(state);
8          if state is Valid then
9              if goalrepairing then
10                 state.hValue = ComputehValue(state);
11                 state.updatefValue();
12         newOpenList.add(state);
13 end
14 OPEN ⟵ newOpenList;
```

Algorithm 12: Initialization of the data strucures

```
1  if genState is generated first time then
2  |   genState.hValue = ComputehValue(genState);
3  |   genState.gValue = pVal;
4  |   genState.fValue = genState.gValue + genState.hValue;
5  |   genState.parent ⟵ parentState;
6  |   return;
7  if goal set has changed then
8  |   genState.hValue = ComputehValue(genState);
9  if pVal < genState.gValue then
10 |   genState.gValue = pVal;
11 |   genState.parent ⟵ parentState;
12 genState.fValue = genState.gValue + genState.hValue;
13 mark genState as valid and informed ;
```

Algorithm 13: Generation of a state

Chapter 5

Algorithm's application examples

In this section we illustrate the way in which the algorithm is executed by providing some simple examples. Let D be the domain in which an agent Ag_1 is situated. The set of the different states of D is $S_{St} = \{A, B, C, D, E, F, G, H, I, J\}$ and the set of the propositions by which each state is defined, is $S_p = \{P_0, P_1, P_2, P_3, P_4\}$ respectively. Table 5.1 presents the successor states for each state, with the number between the parentheses being the cost of the corresponding generating action, and the propositions that hold for it respectively.

Table 5.1: States Table

State	Successor States	Propositions
A	B(3), C(2), D(2)	P_0
B	E(2), A(3), F(1), G(3)	P_1
C	G(2), H(2), D(1)	P_2
D	H(1), A(2)	P_0,P_2
E	$\emptyset$	P_1, P_3
F	J(2)	P_2, P_4
G	J(3)	P_2, P_3
H	I(2)	P_2, P_4
I	$\emptyset$	P_3, P_4
J	$\emptyset$	$P_2.P_3, P_4$

Production of initial plan

Let the initial state of the problem be $I = \{A\}$ and the goal state be $G = \{P_2, P_3\}$. The next steps are followed:

Step 1: State A is inserted in open list.

Step 2: State A is removed from open list, expanded and added in the closed list. States B,C and D are generated and inserted in the open list.

Step 3: State C is removed from open list, expanded at and added in the closed list. States G and H are generated and inserted in the open list. States A and D are re-generated.

Step 4: State B is removed from open list, expanded at and added in the closed list. States E and F are generated and inserted in the open list. State G is re-generated.

Step 5: State D is removed from open list[1], expanded at and added in the closed list. States A and H are re-generated.

Step 6: State H is removed from the open list, expanded at and added in the closed list. States I is generated.

Step 7: State G is removed from the open list and the algorithm terminates. The corresponding plan is $P = \{Ac_{A\to C}, Ac_{C\to G}\}$ and its cost is 5.

The final search tree is presented in figure 5.1. The three numbers at the w.r.t. of the each node are its g-values, its h-value and its f-value respectively. The states of the closed list are depicted in black color and the states of the open list in blue colour respectively. The parent pointers are depicted in continuous black line and the other ancestor states pointer in dashed line respectively. In total, 5 states are expanded and 9 are generated. For comparison reasons, the search tree of A^* is shown in figure 5.2. Note that the only differences between the two trees, are the extra pointers that are kept in the RDA^* tree.

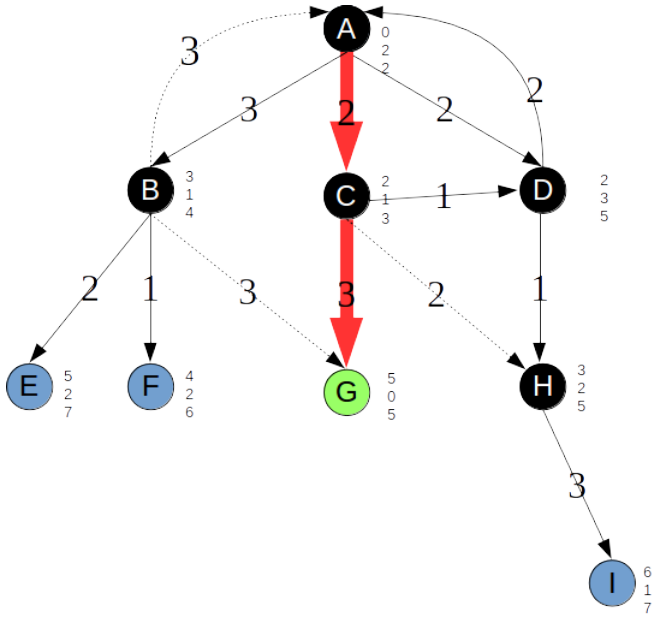


Figure 5.1: The final search tree for the initial problem after the execution of RDA^*

[1]If two states, have the same value, then the one with the lower g-value is expanded first.

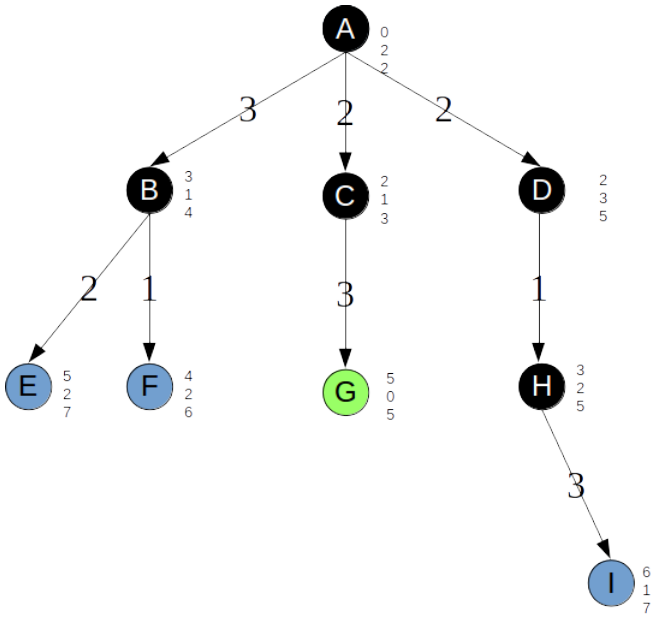


Figure 5.2: The final search tree for the initial problem after the execution of A^*

5.1 Example 1 - Goal-set increases

Let the new initial state be $I_{new} = \{C\}$ and the new goal state be $G_{new} = \{P_2, P_3, P_4\}$. Since the new goal set consists a superset of the initial goal set, the special case sub-variation of the algorithm is applied. The search tree with which the execution of RDA^* begins, is shown at figure 5.3. A question mark next to a node indicates that the corresponding state is not informed.

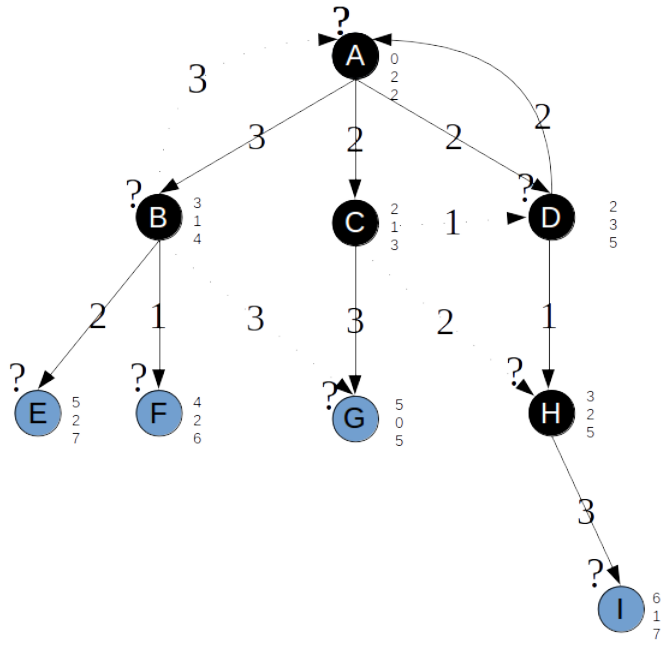


Figure 5.3: The search tree for example 1 when the execution of RDA^* begins

RDA^* is executed in the following way:

Step 1: State G is removed from open list. It is lazily informed (see section 5.5.1 for a step-by-step description) and, then , being valid, expanded and added in the closed list. State J is generated and inserted in the open list.

Step 2: State F is removed from open list. It is lazily informed (States B, A and D are informed during the procedure). Although state F is valid, its updated f-value is greater than the f-value of the head of the open list (state J), and, therefore, it is re-inserted in the open list.

Step 3: State J is removed from the open list and the algorithm terminates. The corresponding plan is $P = \{Ac_{C\rightarrow D}, Ac_{D\rightarrow H}, Ac_{G\rightarrow J}\}$ and its cost is 6.

The final search trees for RDA^* is presented in figure 5.4. Note that 2 states of the open list and one state of the closed list are not informed. In total, 5 states are informed, 1 is re-inserted in the open list, 1 is expanded and 1 is generated. The final search trees for the case of A^* is presented in figure 5.5. In this case, 4 states are expanded and 7 states are generated.

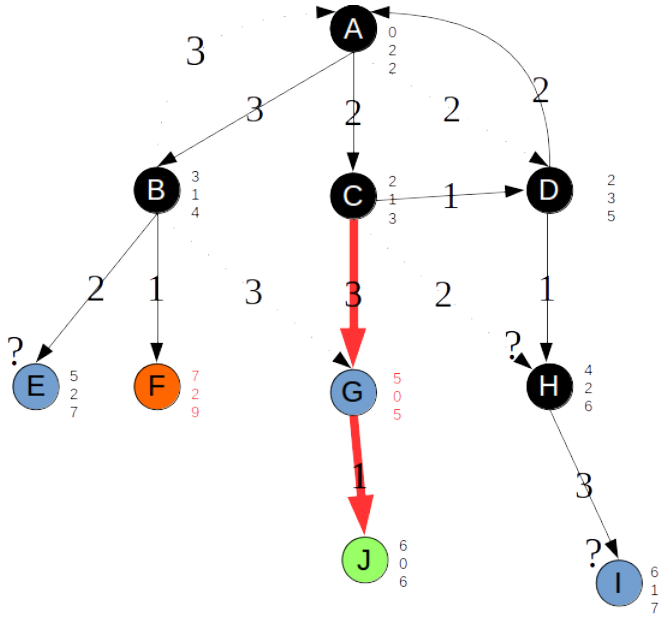


Figure 5.4: The final search tree of RDA^* for example 1

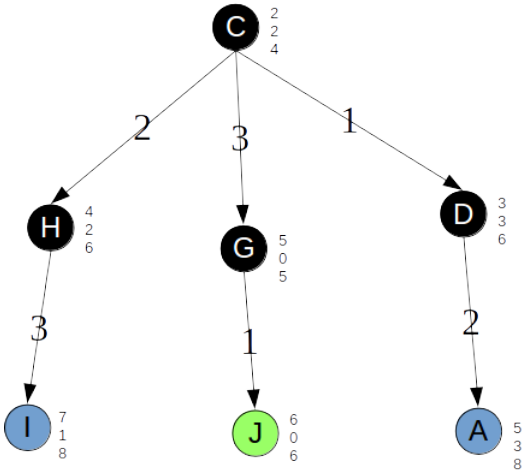


Figure 5.5: The final search tree of A^* for example 1

5.2 Example 2 - Goal-set changes

Let the new initial state be $I_{new} = \{C\}$ and the new goal state be $G_{new} = \{P_2, P_4\}$. The search tree with which the execution of RDA^* begins, is the same as in the case of the previous example. Since the new goal set is not a superset of the initial goal set, the general case sub-variation for the goal-set modification is executed. RDA^* is executed in the following way:

Step 1: The closed list is searched for states satisfying the new goal set. No state is found, and the execution of the algorithm continues.

Step 2: The open list is validated.

Step 3: State G is removed from open list, expanded and added in the closed list. State J is generated and inserted in the open list.

Step 4: State I is removed from the open list and the algorithm terminates. The corresponding plan is $P = \{Ac_{C\to D}, Ac_{D\to H}, Ac_{H\to I}\}$ and its cost is 7.

The search tree after the validation of the open list is shown in figure 5.6. The final search trees is presented in figure 5.7. In total, 8 states are informed, 1 state is expanded and 1 state is generated. The final search trees for the case of A^* is presented in figure 5.8. In this case, 4 states are expanded and 7 states are generated.

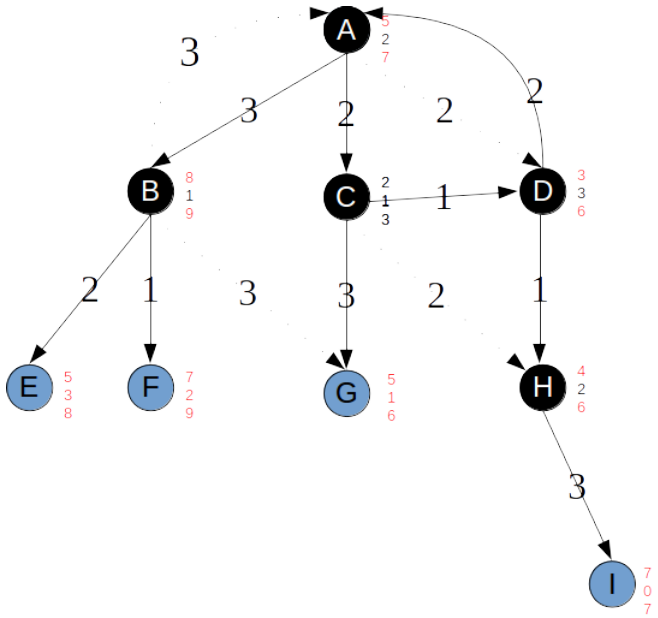


Figure 5.6: The search tree of example 2 after the validation of the open list

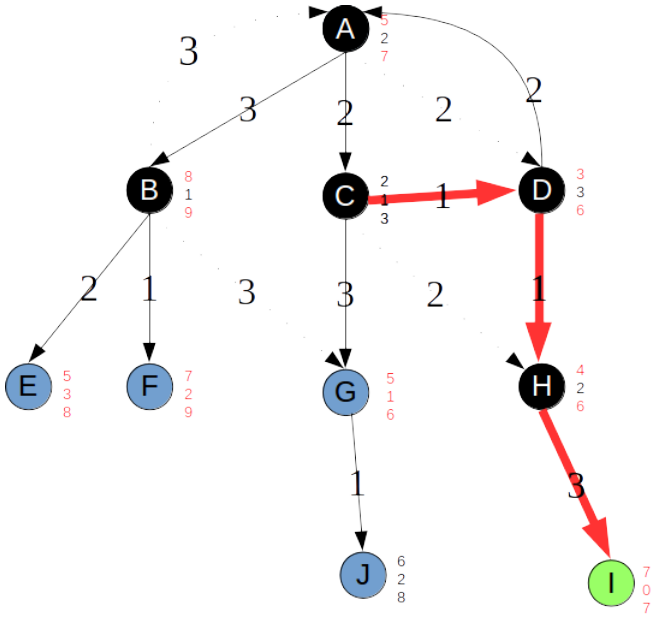


Figure 5.7: The final search tree of RDA^* for example 2.

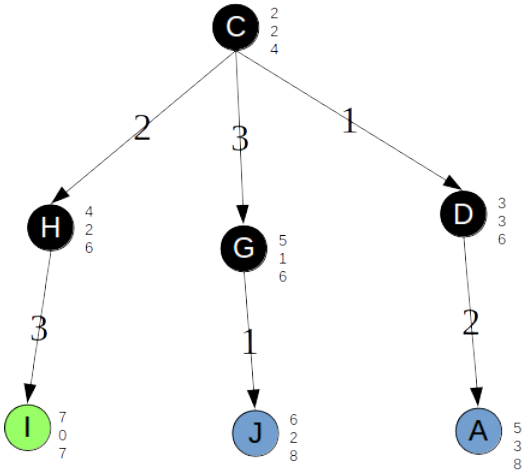


Figure 5.8: The final search tree of A^* for example 2

5.3 Example 3 - Increased actions costs

Let the new initial state be $I_{new} = \{C\}$. The goal state remains the same $G_{new} = \{P_2, P_3\}$. The following actions change their costs: $ac_{C \to G} = 4, ac_{D \to H} = 2, ac_{B \to E} = 4$. Since all the actions costs changes are increases, the sub-variation suited for cost increases is applied. The search tree with which the execution of RDA^* begins, is shown in figure 5.9.

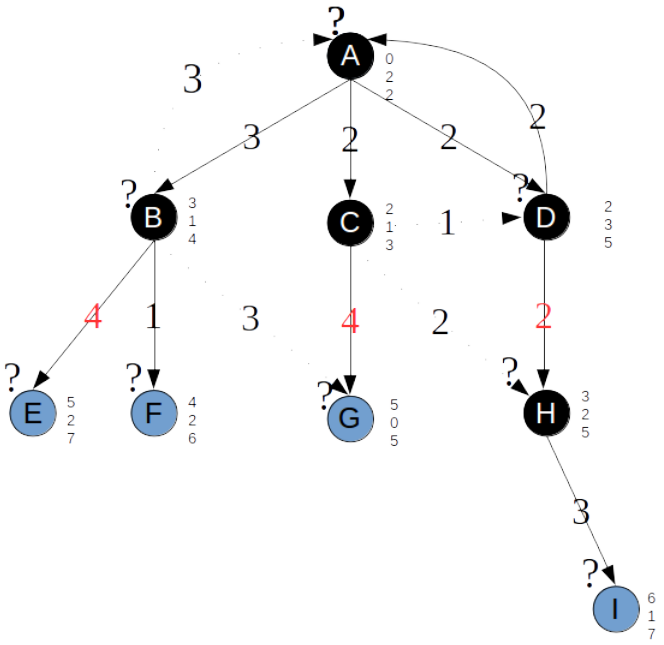


Figure 5.9: The search tree in the beginning of the execution of RDA^* for example 3

RDA^* is executed in the following way:

Step 1: The head of the open list from which the plan for the original problem was extracted, is removed from the open list. The plan of the original problem is lazily informed. Its updated cost is 6.

Step 2: State J is removed from open list. Since its f-value is not smaller than the updated cost of the validated plan, the algorithm terminates. The corresponding plan is $P = \{Ac_{C \to G}\}$ and its cost is 6.

The final search tree is presented in figure 5.10. In total, 1 state is informed. Note, that 4 states of the closed list and 3 states of the open list remain non-informed. In the case of A^* , 3 states are expanded and 6 states are generated. The corresponding final search tree is shown in figure 5.11.

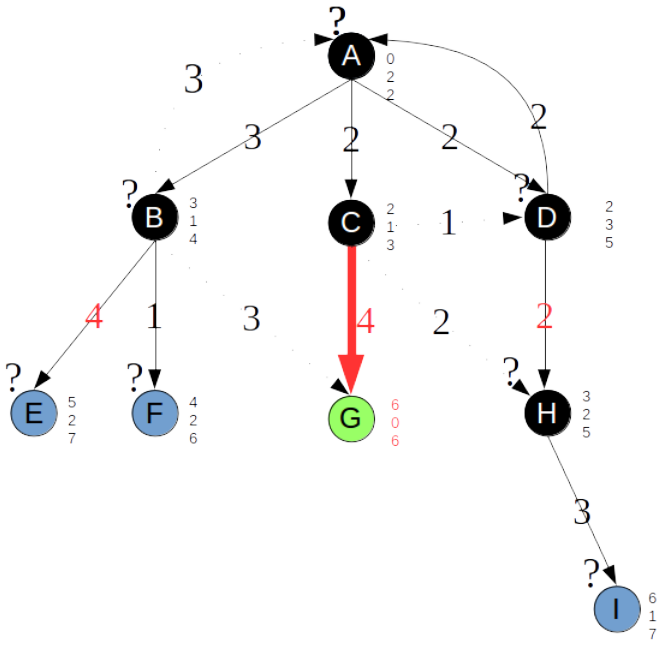


Figure 5.10: The final search tree of RDA^* for example 3.

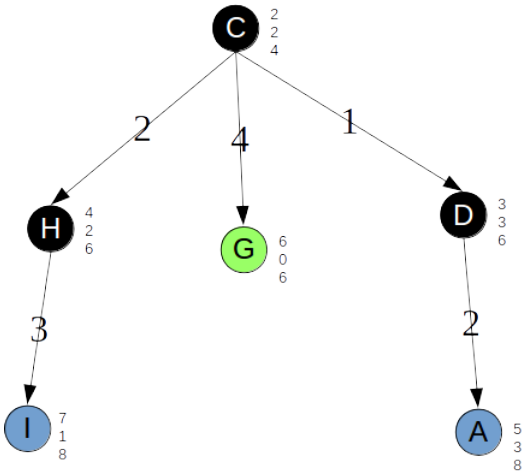


Figure 5.11: The final search tree of A^* for example 2

5.4 Example 4 - Decreased actions costs

Let the new initial state be $I_{new} = \{C\}$. The goal state remains the same $G_{new} = \{P_2, P_3\}$. The following actions change their costs: $ac_{C \to G} = 10, ac_{B \to E} = 1$. Since some of the actions costs changes are decreases, the general-case variation for cost changes

will be applied.

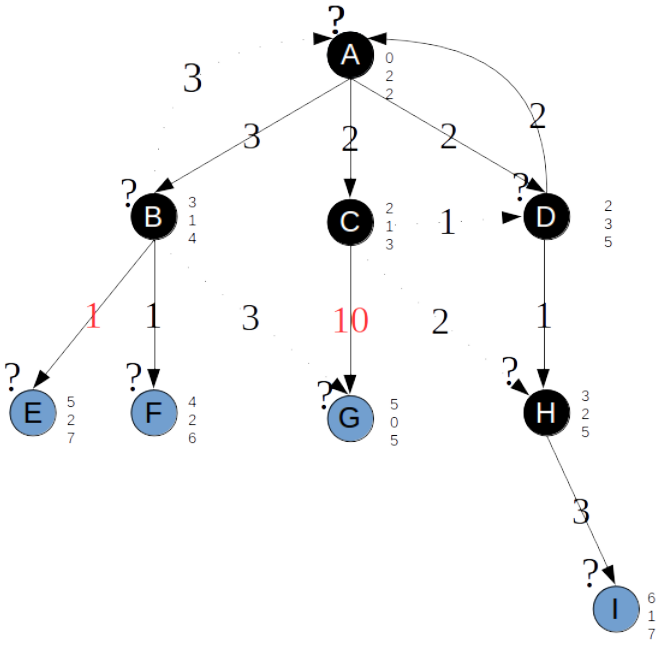


Figure 5.12: The search tree in the beginning of the execution of RDA^* for example 3

RDA^* is executed in the following way:

Step 1: The open list is validated (at section 5.5.2 a step-by-step description of the full informing of state G is provided).

Step 2: State E is removed from the open list, expanded and added in the closed list.

Step 3: State I is removed from the open list, expanded and added in the closed list.

Step 4: State F is removed from open list, expanded and added in the closed list. State J is generated and inserted in the open list.

Step 5: State G is removed from the open list and the algorithm terminates. The corresponding plan is $P = \{Ac_{C\to D}, Ac_{D\to A}, Ac_{A\to B}, Ac_{B\to G}\}$ and its cost is 11.

The search tree after the validation of the open list is shown in figure 5.13. The final search trees is presented in figure 5.14. In total, 8 states are informed, 3 states are expanded and 1 state is generated. The final search trees for the case of A^* is presented in figure 5.15. In this case, 8 states are expanded and 10 states are generated.

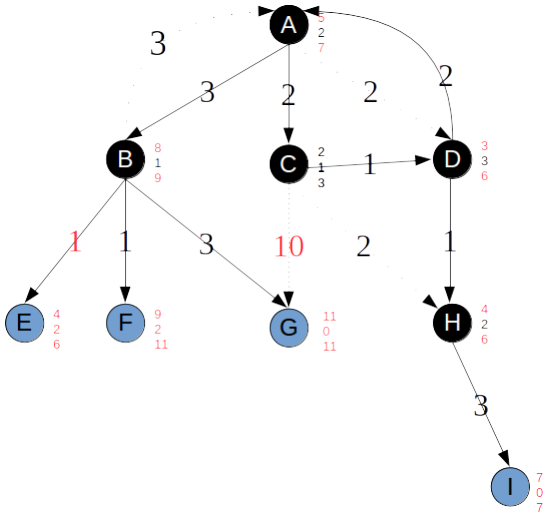


Figure 5.13: The search tree of example 4 after the validation of the open list

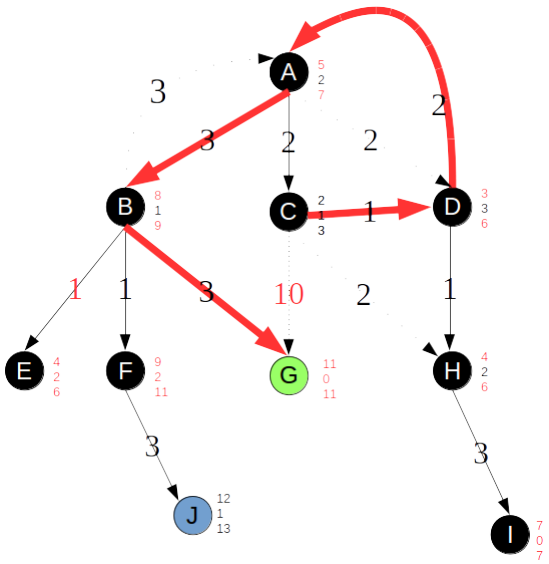


Figure 5.14: The final search tree of RDA^* for example 4.

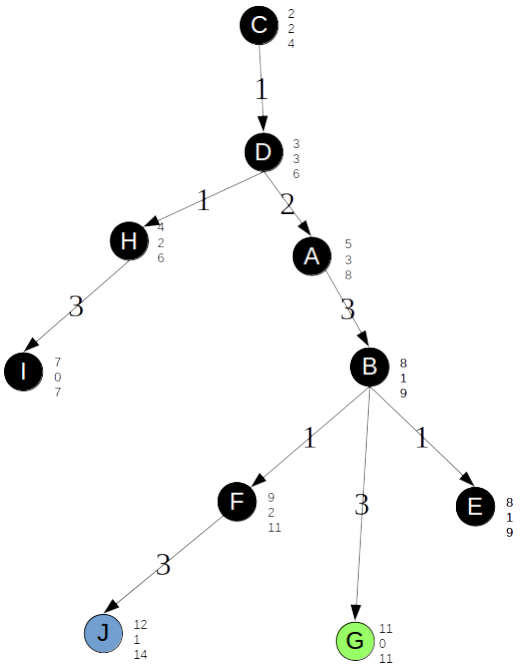


Figure 5.15: The final search tree of A^* for example 4

5.5 Informing

In order to illustrate the informing procedure, we present in this subsection a step-by-step analysis for two cases of informing deriving from the previous examples. Each description is accompanied by a graphical representation of certain steps. The representations follows the next notation.A red arrow from state s_1 to s_2 indicates that during the informing of s_1, s_2 was examined. A grey arrow indicate an ancestors state has not been yet examined. The values on the top of the arrows are the corresponding p-values. Red-coloured values are informed, whereas grey-coloured are not.

5.5.1 Lazy informing of state G in example 1

The steps that are followed are the next:

Step 1: State G is marked as pending. The lgv-parent state of state G, state C, is examined. State C is informed and valid. The corresponding p-value is updated. p-value$_{C \to G}$ remains 5, and, therefore, the g-value of state G does not change.

Step 2: The p-value of the next parent state of state G,state B is examined. p-value$_{C \to G}$ is 6 and since it is greater than the g-value of state G, the informing procedure stops.

Step 3: State G is marked as informed and valid and reset from pending.

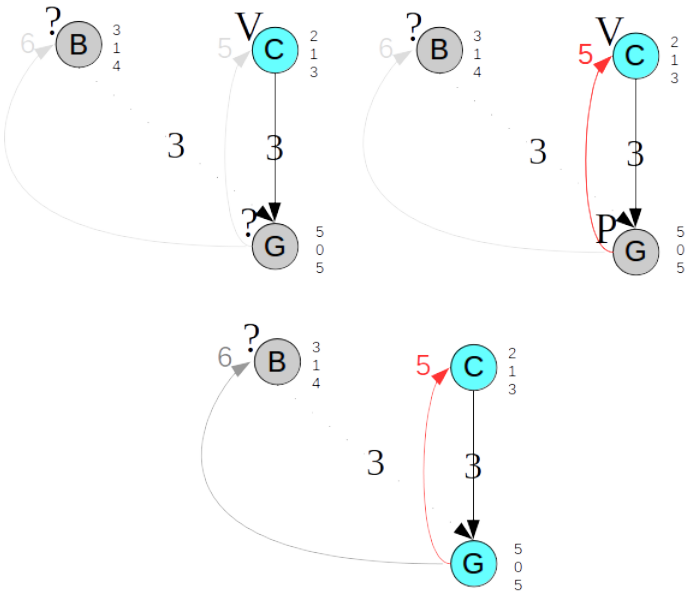


Figure 5.16: Lazy informing of state G. Steps are presented in a clockwise order beginning from top left.

5.5.2 Full informing of state G in example 4

The steps that are followed are the next:

Step 1: Stated G is marked as pending. The lgv-parent state of state G, state C, is examined. State C is informed and valid. p-value$_{C \rightarrow G}$ becomes 12, and the g-value of state G changes from 5 to 12.

Step 2: The next parent state of state G, state B, is examined. State B is not informed and the procedure for its full informing is followed.

Step 3: Stated B is marked as pending.The lgv-parent state of state B, state A, is examined. State A is not informed and the procedure for its full informing is followed.

Step 4: Stated A is marked as pending. The lgv-parent state of state A, state D, is examined. State D is not informed and the procedure for its full informing is followed.

Step 5: Stated D is marked as pending. The lgv-parent state of state D, state A, is examined. State A is informed-pending and is not examined. State D is inserted in the successors list of state A, and the corresponding p-value$_{A \rightarrow D}$ will be updated when the informing of state A is concluded. The number of partially informed parent states of state D is increased and becomes 1.

Step 6: The next ancestor state of state D, state C, is examined. State's C is informed and valid. p-value$_{C \rightarrow D}$ becomes 3, and, therefore, state C becomes the lgv-parent of state D and the g-value of state D changes from 2 to 3.

Step 7: State D has no other parent states. Its g-value becomes 3. State's D is marked as informed and valid and reset from pending. Since the number of partially informed parent states of state D is greater than zero, the states contained in its successor lists are not updated. The informing procedure of state A is resumed.

Step 8: State's A p-value$_{D \rightarrow A}$ changes from 4 to 5. State's A next parent state, state B, is examined. State B is informed-pending and is not examined. State A is inserted in the the successors list of state B, and the corresponding p-value$_{B \rightarrow A}$ will be updated when the informing of state A is concluded. The number of partially informed parent states of state A is increased and becomes 1.

Step 9: State A has no other parent states. Its g-value becomes 5. State A is marked as informed and valid and reset from pending. Since the number of partially informed parent states of state A is greater than zero, its informing is not finished. The informing procedure of state B is resumed.

Step 10: State's B p-value$_{A \rightarrow B}$ changes from 3 to 8. State B has no other ancestor states and its informing is finished. State B is marked as informed and valid and reset from pending. p-value$_{B \rightarrow A}$ is updated and becomes now 11. It is greater than the g-value of state A and, therefore, its parent pointer and g-value does not change. The number of partially informed ancestor states of state A is decreased by 1 and becomes 0. Since the number of partially informed parent states of state A is equal to zero, the p-values of the states contained in its successor lists are updated. State's D p-value$_{A \rightarrow D}$ becomes 7. It is greater than the g-value of state D and, therefore, its parent pointer and g-value does not change. The number of partially informed parent states of state D is decreased by 1 and becomes 0. Since no states are contained in the successor lists of state D, no updating takes place. The informing procedure of state G is resumed.

Step 11: State G p-value$_{A \rightarrow B}$ changes from 6 to 11.Because, p-value$_{A \rightarrow B}$ is smaller than the state's G g-value, state B becomes the parent of state G and its g-value changes from 12 to 11. State G has no other parent states and its informing is finished. It is marked as informed and valid and reset from pending.

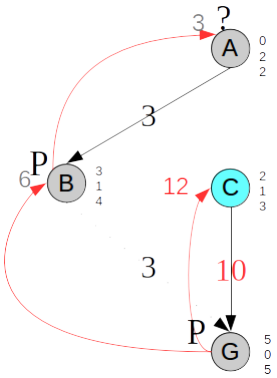


Figure 5.17: Step 3 of the full informing of state G

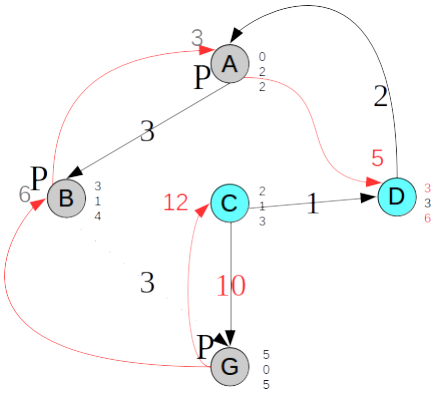


Figure 5.18: Step 6 of the full informing of state G

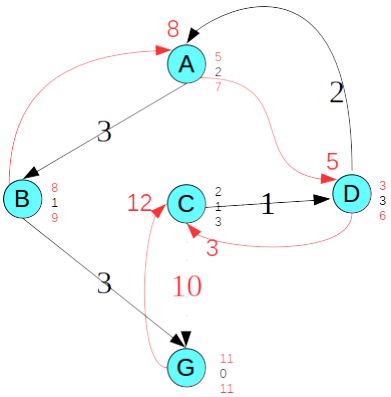


Figure 5.19: Step 11 of the full informing of state G

Chapter 6

Experimental Evaluation

6.1 Objective

The purpose of the experimental evaluation is to compare the performance of RDA^* against A^* in terms of speed for four different re-planning scenarios. Specifically, we compare the ratio of the runtime of the two algorithms with respect to:

- The percentage of the executed plan, which corresponds to the new initial state of the repairing problems w.r.t original plan (Experiments scenario 1,2);
- The percentage of the original goal-set modification (Experiments scenario 1 and 2);
- The percentage of the actions with decreased costs (Experiments scenario 3);
- The percentage of the actions with increased costs (Experiments scenario 4).

We do not measure their memory requirements because both of the algorithms show a linear complexity in the number of states in the state space.

6.2 Experiments setup

6.2.1 Benchmarks Domains

The benchmarks for the benchmarking evaluation derive from the third, fourth and eighth International Planning Competitions[11][28] [2]. In the following we provide a brief description for each.

- Logistics

  A number of packages is transported within cities via trucks, and between cities via air-planes. The locations that lie within a city are directly connected (trucks can move between any two such locations), and so are the cities respectively. To each city corresponds one truck, and each city has one location which serves as an airport. All the actions in this domain have uniform cost.

- Miconics (Simple)

  A number of passengers is transported with lifts from their origin-floors to their destination floors. All the actions in this domain have uniform cost.

- Blocksworld

  This domain consists of a set of blocks, a table and a robot hand. The blocks can be on top of other blocks or on the table; a block that has nothing on it is clear; and the robot hand can hold one block or be empty. The goal is to find a plan to move from one configuration of blocks to another. All the actions in this domain have uniform cost.

- Gripper

  In this domain there are a number of robots, with two gripper hands, and a set of rooms containing balls. The goal is to find a plan to transport balls from a given room to another. All the actions in this domain have uniform cost.

- Depots

  A number of trucks transport crates around which, then, must be stacked onto pallets at their destinations. The stacking is achieved using hoists. Trucks can behave like "tables", since the pallets on which crates are stacked are limited. All the actions in this domain have uniform cost.

- Logistics-cost

  It is the same domain as Logistics, with the only difference being that the actions do not have uniform costs.

- Depots-cost

  It is the same domain as Depots, with the only difference that the actions do not have uniform costs.

- Transport

  This domain is similar to the logistics domain. The difference is that the locations in a city can be more than one. The actions in this domain do not have uniform cost.

In table 6.1 the ancestry factors for each problem instance of the domains that was used for the experimental evaluation is shown.

Table 6.1: The ancestry factors of the problems

Problem	Ancestry Factor
Blocks 91	4.9
Blocks 92	4.6
Logistics 61	8.3
Logistics 62	8.3
Depot 1345	10.9
Depot 1935	8.6
Gripper x5	4.4
Gripper x6	4.6
Miconic 10	19.8
Miconic 11	21.7
Miconic 12	23.7
Depot-cost 1935	4.6
Logistics-cost 63	8.2
Transport 2533	8.9

6.2.2 Experiment Scenarios

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, respectively. Finally, a new plan is produced for the modified conditions. 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 $k \leq n$. In this case, the general-case goal sub-variation of RDA^* is used. The details of the conducted experiments for this scenario are shown in Table 6.2.

- Scenario 2
  The new goal set is produced by the addition of k goals in the original goal-set. The details of the conducted experiments for this scenario are shown in Table 6.3. In this case, second special-case goal sub-variation of RDA^* is used.

- 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 200% of its initial cost. The details of the conducted experiments for this scenario are shown in Table 6.4. In this case, the general-case cost sub-variation of RDA^* is used.

- 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 details of the conducted experiments for this scenario are shown in Table 6.5. In this case, the special-case cost sub-variation of RDA^* is used.

Table 6.2: Experiments of scenario 1

Experiment	Problem	Size of initial goal set	Number of removed goals	Number of added goals
1.1	Blocks 92	6	1	1
1.2	Depots 1935	5	1	1
1.3	Gripper x6	12	2	3
1.4	Logistics 62	5	1	1
1.5	Logistics 63	5	1	1

Table 6.3: Experiments of scenario 2

Experiment	Problem	Size of initial goal set	Number of added goals
2.1	Blocks 91	6	2
2.2	Blocks 91	7	1
2.3	Blocks 92	6	2
2.4	Blocks 92	7	1
2.5	Logistics 61	4	2
2.6	Logistics 61	5	1
2.7	Logistics 62	4	2
2.8	Logistics 62	5	1
2.9	Depot 1345	3	2
2.10	Depot 1345	4	1
2.11	Depot 1935	3	3
2.12	Depot 1935	4	2
2.13	Gripper x5	8	4
2.14	Gripper x6	9	5
2.15	Gripper x6	11	3
2.16	Miconic 10	7	3
2.17	Miconic 10	9	1
2.18	Miconic 11	7	4
2.19	Miconic 11	9	2
2.20	Miconic 12	9	3

Table 6.4: Experiments of scenario 3

Experiment	Problem	percentage of decreased actions costs	max percentage of plan's decreased actions costs
3.1	Transport 2533	[5-65]	10
3.2	Depots-cost 1935	[5-65]	10
3.3	TLogistic-cost 63	[5-65]	10

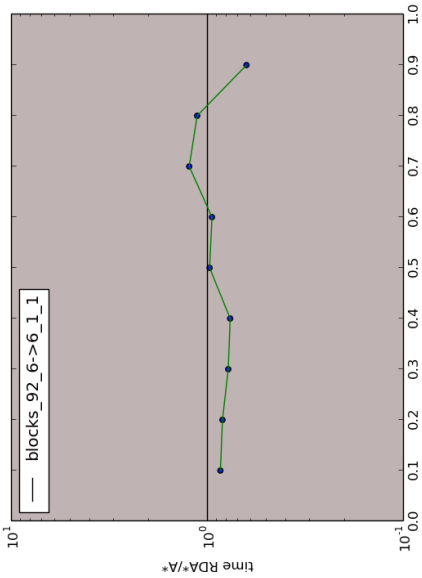
Table 6.5: Experiments of scenario 4

Experiment	Problem	percentage of increased actions costs	max percentage of plan's increased actions costs
4.1	Transport 2533	[5-65]	0
4.2	Depots-cost 1935	[5-65]	0
4.3	Logistic-cost 63	[5-65]	0

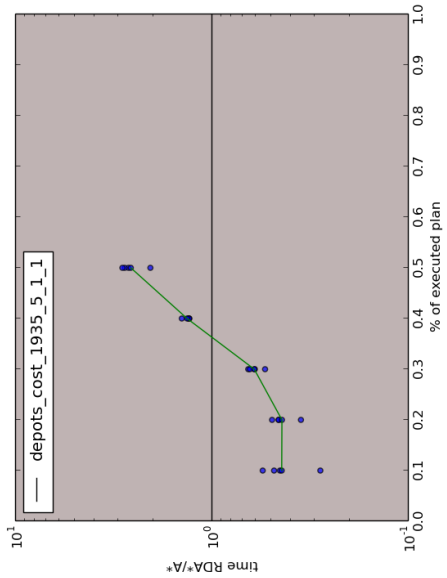
6.3 Results

This sections contains in form of diagrams the outcome of the experimental evaluation for the four scenarios, with every diagram reffering to a different problem. Namely, the ratio of RDA^* runtime to A^* runtime (y-axis) is plotted against the percentage of executed plan (x-axis). Each dot corresponds to a different experiment for the given problem and the continuous line corresponds to the mean value. The y-axis is in logarithmic scale. The data in each case has been normalized, by the exclusion of ratios with values below 10^{-1} and above 10^{1}.

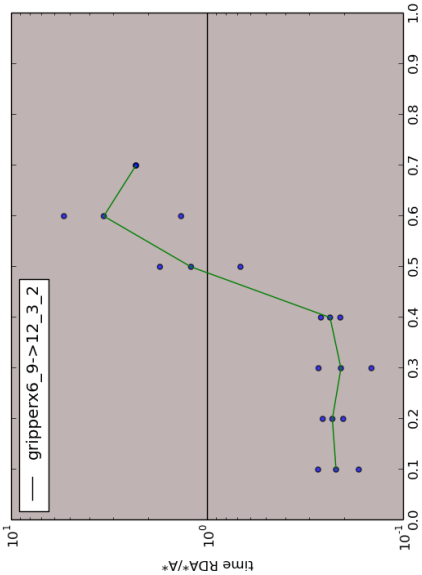
6.3.1 Scenario 1 diagrams



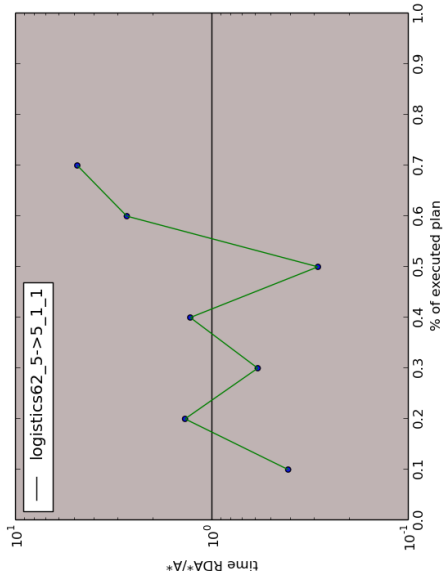
(a) Experiment 1.1



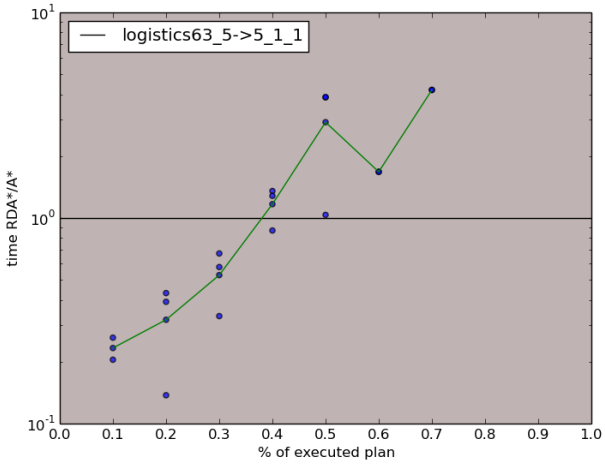
(b) Experiment 1.2



(c) Experiment 1.3



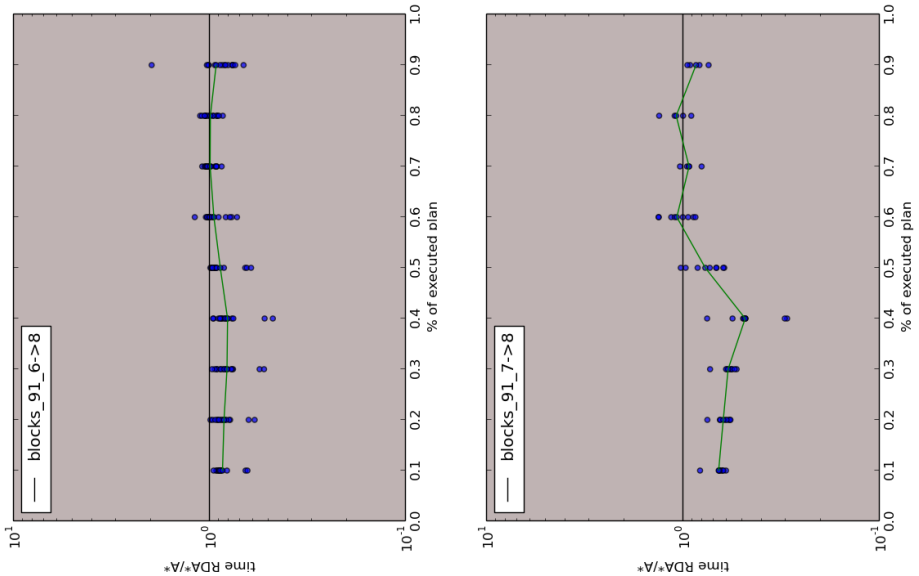
(d) Experiment 1.4



(e) Experiment 1.5

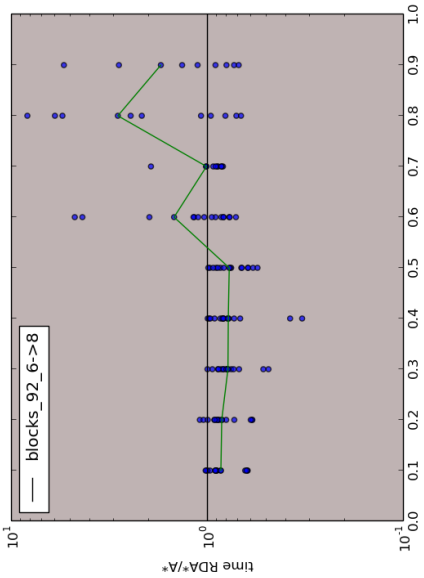
Figure 6.1: Diagrams of Experiments 1.1-1.5

6.3.2 Scenario 2 diagrams

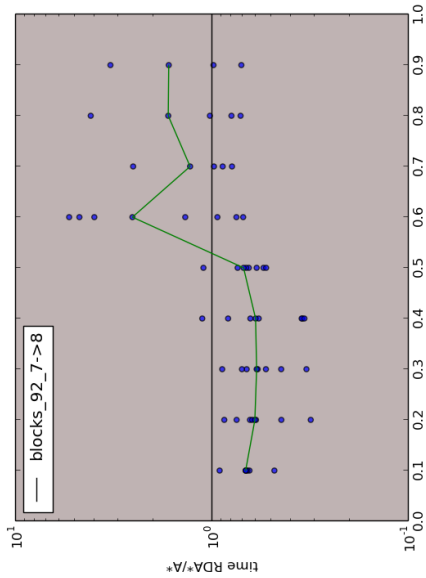


(a) Experiment 2.1

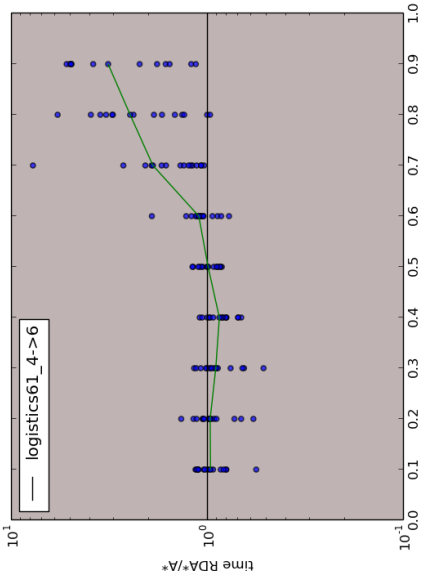
(b) Experiment 2.2



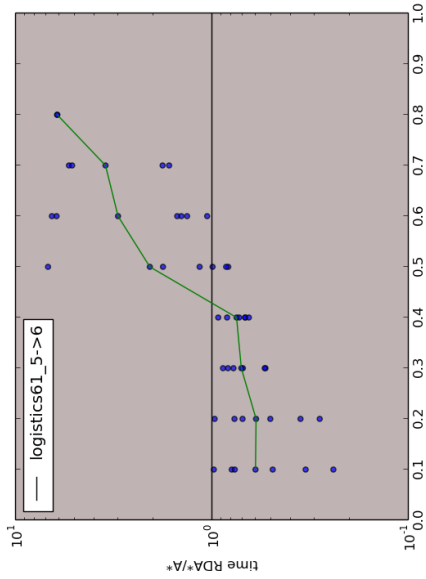
(c) Experiment 2.3



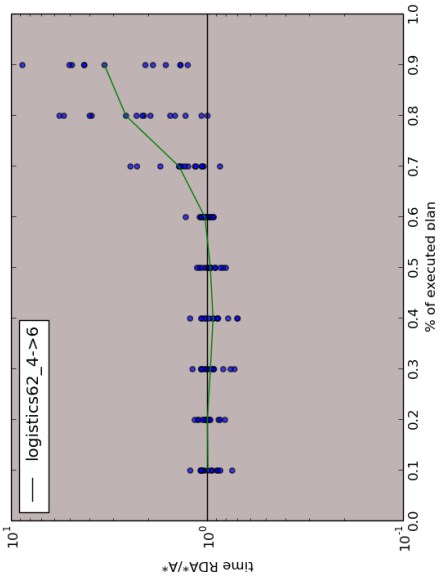
(d) Experiment 2.4



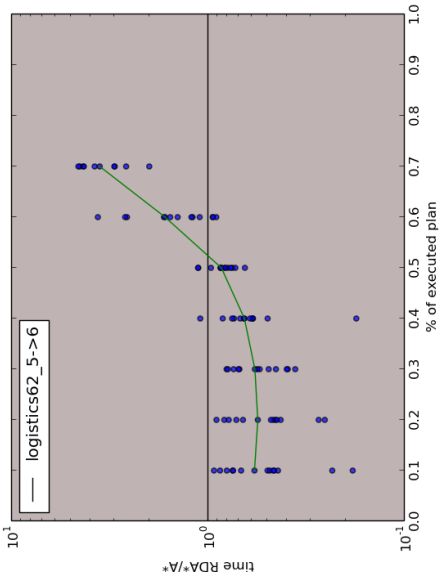
(e) Experiment 2.5



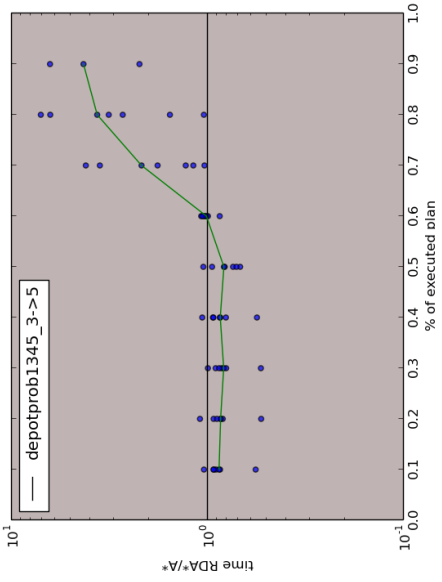
(f) Experiment 2.6



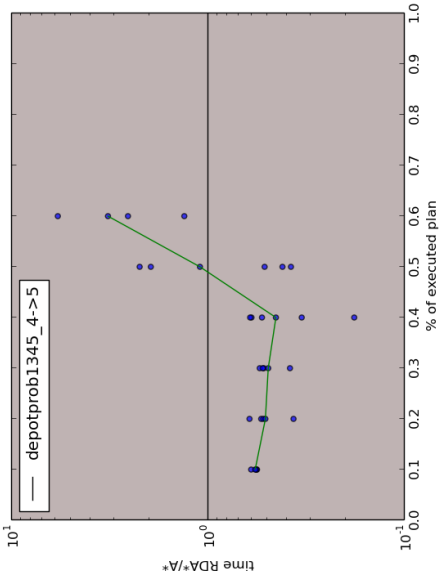
(g) Experiment 2.7



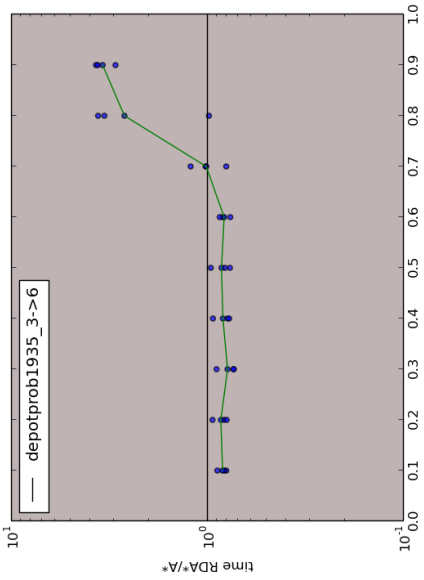
(h) Experiment 2.8



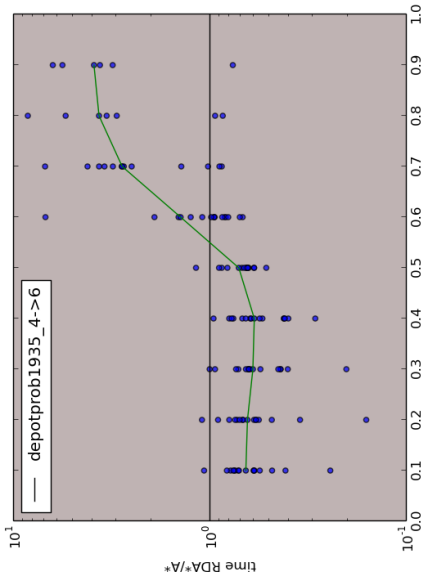
(i) Experiment 2.9



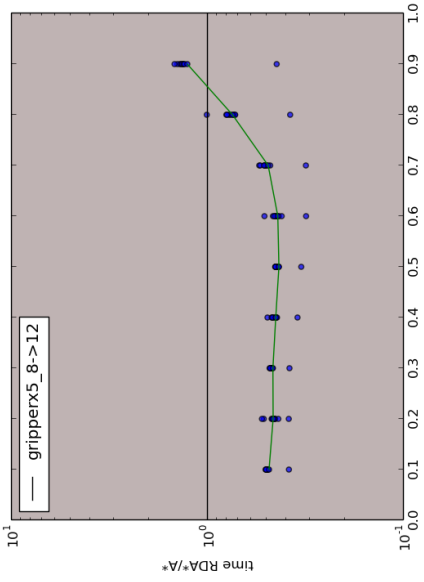
(j) Experiment 2.10



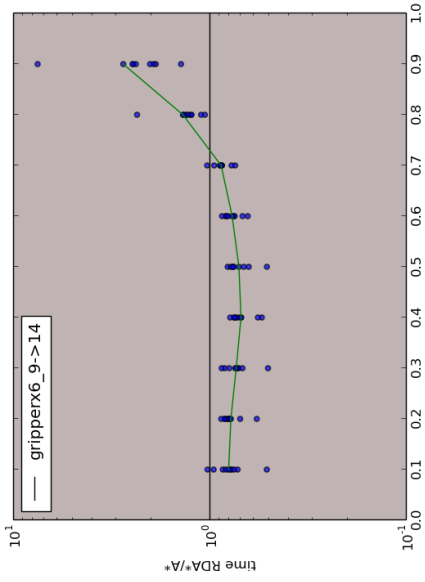
(k) Experiment 2.11



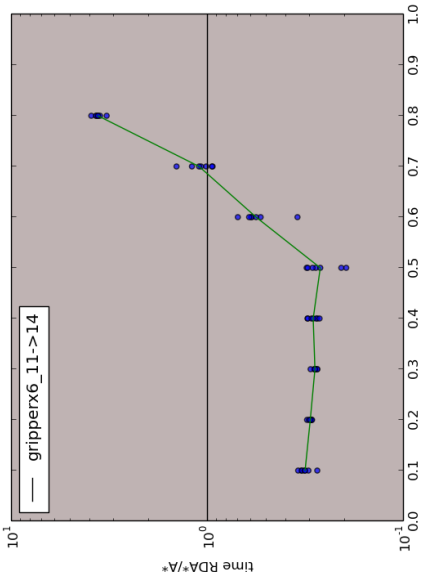
(l) Experiment 2.12



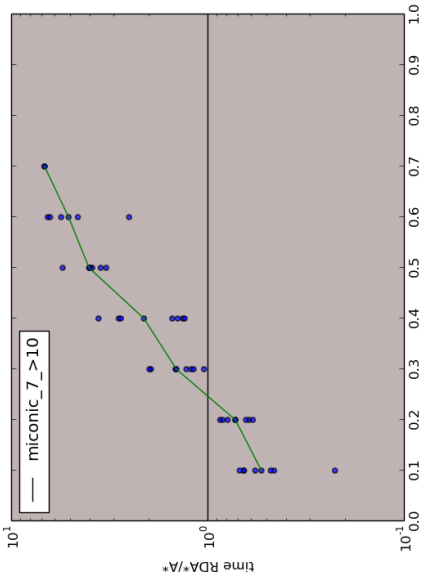
(m) Experiment 2.13



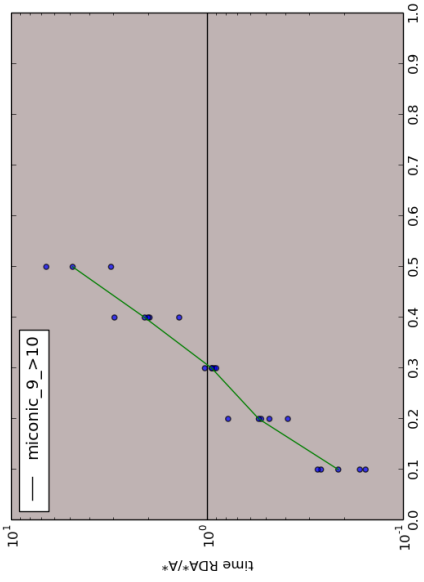
(n) Experiment 2.14



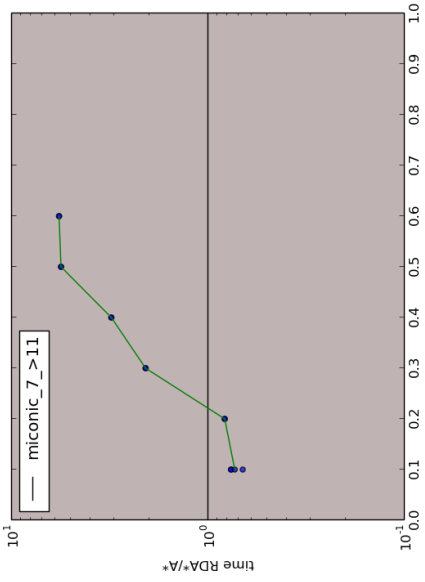
(o) Experiment 2.15



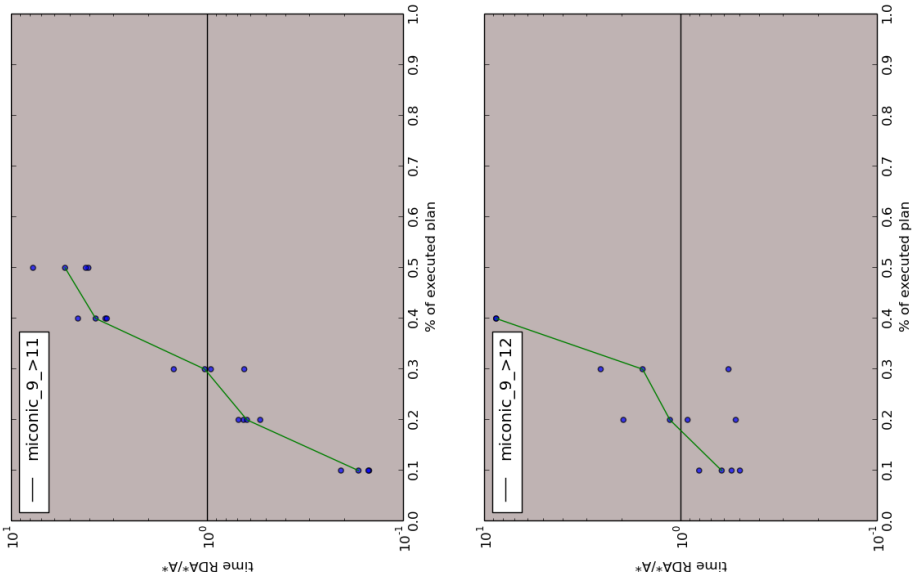
(p) Experiment 2.16



(q) Experiment 2.17



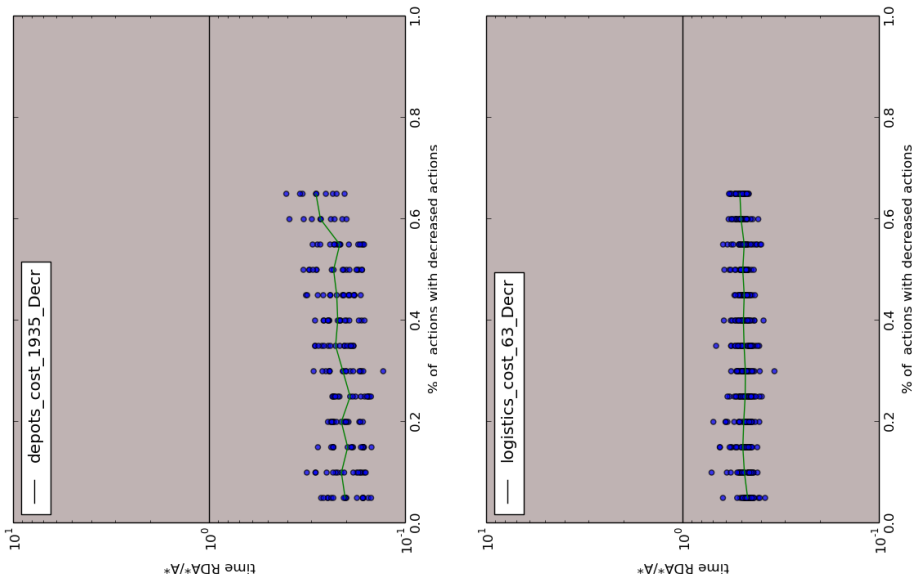
(r) Experiment 2.18



(s) Experiment 2.19 (t) Experiment 2.20

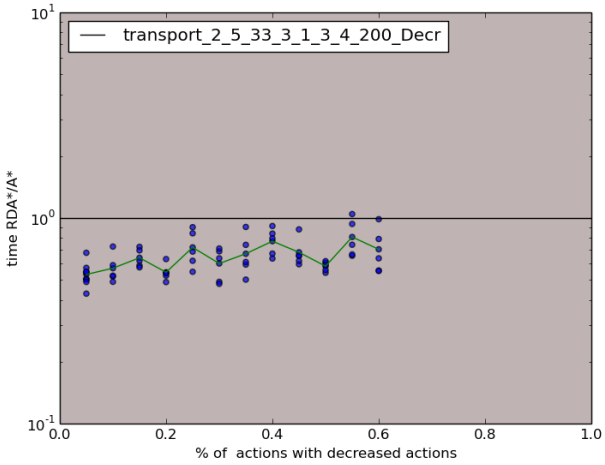
Figure 6.2: Diagrams of Experiments 2.1-2.20

6.3.3 Scenario 3 diagrams



(a) Experiment 3.1

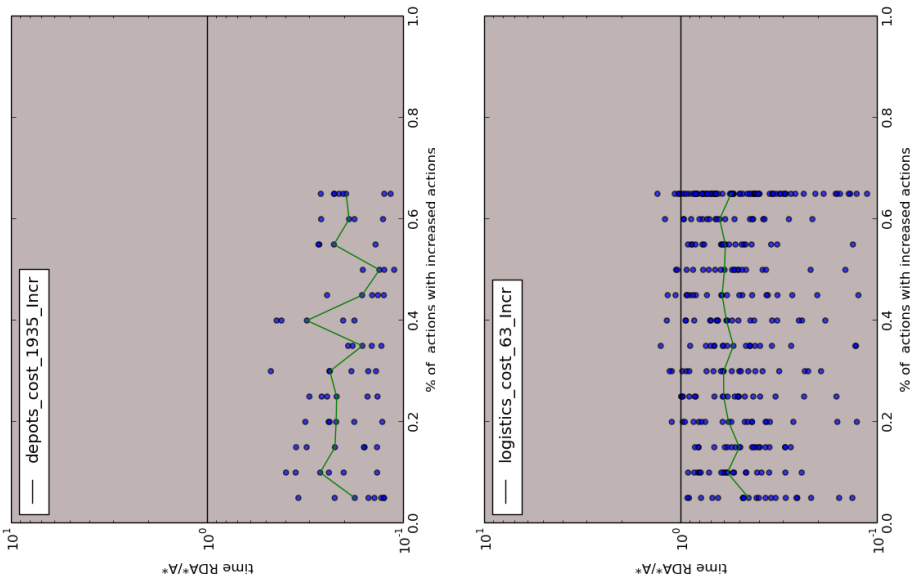
(b) Experiment 3.2



(c) Experiment 3.3

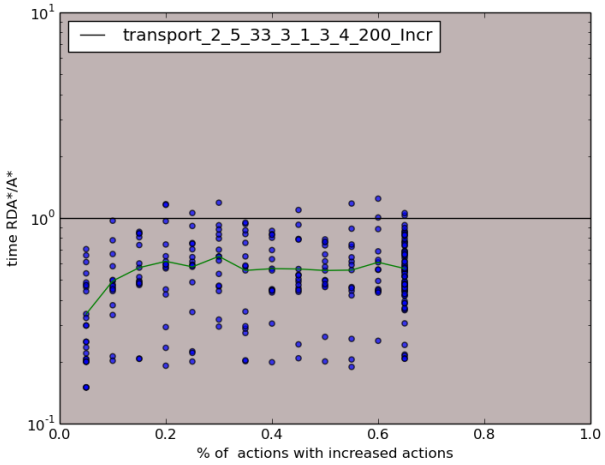
Figure 6.3: Diagrams of Experiments 3.1-3.3

6.3.4 Scenario 4 diagrams



(a) Experiment 4.1

(b) Experiment 4.2



(c) Experiment 4.3

Figure 6.4: Diagrams of Experiments 4.1-4.3

6.3.5 Discussion

From the diagrams of the previous subsections, the following conclusions regarding the performance of RDA^* against A^* can be drawn:

- As the percentage of the executed plan increases, the performance of RDA^* deteriorates with respect to the corresponding performance of A^* .
- As the percentage of the modified goal-set increases, the performance of RDA^* deteriorates with respect to the corresponding performance of A^* .
- As the ancestry factor increases, the performance of RDA^* deteriorates with respect to the corresponding performance of A^* .
- The performance of RDA^* does not vary significantly as the percentage of actions with decreased costs increases.
- The performance of RDA^* does not vary significantly as the percentage of actions with increased costs increases.
- For a given problem instance, RDA^* performs better in increases of the goal-set than in general modifications of the goal-set.
- For a given problem instance, RDA^* performs better in cases of increased actions costs than of decreased actions costs.

In the following we will attempt to explain the previous findings. RDA^* expands at most the same number of states as A^* , since a part of the search tree with which the the search begins, is already constructed. Moreover, during RDA^* 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 RDA^* performance against A^* .

As the percentage of the executed plan increases, the new root of the search tree, recedes further from the root of the original search tree, which, as a result, has one of the following two outcomes. A larger part of the search trees leaves, would either become invalid or would have its f-values increased. In either case, time is consumed for the informing of states which are useless for the search, since they will not be expanded.

Likewise, the fact 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 RDA^* in such cases in comparison to the general case handling of modified goal-sets. A similar line of reasoning can be applied in the case of modified actions costs. 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.

Finally, the deterioration of RDA^* performance with higher ancestry factors, can be attributed to the greater time that is necessary for the informing procedure. Namely, large

ancestry factors correspond to large average number of ancestor states, which results in a greater number of examined ancestor states during the informing procedure.

Chapter 7

Conclusions and future work

7.1 Conclusions

In this work we have presented a novel plan repairing algorithm, RDA^* , that extends one of the most popular and studied planning algorithm, A^* . RDA^* can address modifications in the goal-set and in the actions costs, rendering, thus, RDA^* suitable for plan repairing in dynamic environments, where changes of the aforementioned kinds occur.

The conducted experimental evaluation indicates that RDA^* outperforms A^* in most of the 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 40% to 50%. Moreover, in the case of goal-set modifications, the change in the goal-set should not be greater than 20% to 50%. The corresponding thresholds, for the previous two parameters, below which RDA^* should be preferred, depend on the ancestry factor of the re-planning problem, with higher ancestry factors corresponding to thresholds of lower values.

Finally, we would like to stress, that this work is not conclusive, but, a first attempt towards the development of an efficient plan-repairing algorithm, and can be, therefore, improved and extended in a number of ways. Nonetheless, we believe that the experimental findings provide a strong support for the utilization of RDA^* in re-planning scenarios. Besides, we consider that a more thorough experimental analysis could provide more useful hints and insights and help us to gain a more thorough understanding of the underlying mechanisms which determine the strengths and weaknesses of the algorithm.

7.2 Future work

We consider that there are some promising directions that are worth exploring and which may lead to the further enhancement of RDA^* and to a better understanding of its behaviour. In the following, we elaborate briefly on each of them.

7.2.1 Experimental evaluation for repeated repairing

As it has been already remarked, RDA^* can be used, without any extra adjustment, for repeated repairing scenarios. This comparison, apart from its academic value, could be, also, of practical use, since in some cases a need for repeated re-planning seems more close to certain dynamic environments. Similarly, the assessment of RDA^* performance for re-planning scenarios, where both the goal-set and the actions costs are modified, could provide useful insights.

7.2.2 Addressing of others types of dynamicity

Apart from goal-set and actions costs modifications, there exist other types of dynamicity that can be observed in real-world domains. For example, altered preconditions and effects for actions, additions and removals of planning agents and invalidations or insertions of new actions. Therefore, the extension of RDA^* in such a way that it can handle the aforementioned changes, could render RDA^* more flexible since it could be utilized in a wider variety of re-planning scenarios.

7.2.3 Distributed approach

The worst performance of RDA^* is observed in domains with large ancestry factors. The latter, is closely related to the branching factor which, in turn, depends mainly on the number of the agents activated for the re-planning procedure. Therefore, a distributed implementation of RDA^* , where each agent performs an independent search, could be proven more efficient. This intuition is further corroborated by the fact that a recent distributed implementationof A^* [32] managed to speed up significantly the runtime performance of the algorithm.

Chapter 8

Bibliography

[1] Mitchell Ai-Chang, John Bresina, Len Charest, Adam Chase, JC-J Hsu, Ari Jonsson, Bob Kanefsky, Paul Morris, Kanna Rajan, Jeffrey Yglesias, et al. Mapgen: mixed-initiative planning and scheduling for the mars exploration rover mission. *IEEE Intelligent Systems*, 19(1):8–12, 2004.

[2] Fahiem Bacchus. Aips 2000 planning competition: The fifth international conference on artificial intelligence planning and scheduling systems. *Ai magazine*, 22(3): 47, 2001.

[3] Julien Bidot, Bernd Schattenberg, and Susanne Biundo. Plan repair in hybrid planning. In *Annual Conference on Artificial Intelligence*, pages 169–176. Springer, 2008.

[4] Steve Chien, G Rabideau, R Knight, Robert Sherwood, Barbara Engelhardt, Darren Mutz, T Estlin, Benjamin Smith, F Fisher, T Barrett, et al. Aspen–automated planning and scheduling for space mission operations. In *Space Ops*, pages 1–10, 2000.

[5] William Cushing and Subbarao Kambhampati. Replanning: A new perspective. *Proceedings of the International Confer-ence on Automated Planning and Scheduling. Monterey, USA*, pages 13–16, 2005.

[6] Tara Estlin, Daniel Gaines, Caroline Chouinard, Rebecca Castano, Benjamin Bornstein, Michele Judd, Issa Nesnas, and Robert Anderson. Increased mars rover autonomy using ai planning, scheduling and execution. In *Proceedings 2007 IEEE International Conference on Robotics and Automation*, pages 4911–4918. IEEE, 2007.

[7] Dave Ferguson and Anthony Stentz. Field d*: An interpolation-based path planner and replanner. In *Robotics research*, pages 239–253. Springer, 2007.

[8] Frank Fiedrich and Paul Burghardt. Agent-based systems for disaster management. *Communications of the ACM*, 50(3):41–42, 2007.

[9] Maria Fox, Alfonso Gerevini, Derek Long, and Ivan Serina. Plan stability: Replanning versus plan repair. In *ICAPS*, volume 6, pages 212–221, 2006.

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

[11] Alfonso E Gerevini, Patrik Haslum, Derek Long, Alessandro Saetti, and Yannis Dimopoulos. Deterministic planning in the fifth international planning competition: Pddl3 and experimental evaluation of the planners. *Artificial Intelligence*, 173(5): 619–668, 2009.

[12] V Daniel R Guide, Vaidy Jayaraman, and Jonathan D Linton. Building contingency planning for closed-loop supply chains with product recovery. *Journal of operations Management*, 21(3):259–279, 2003.

[13] Eric A Hansen and Rong Zhou. Anytime heuristic search. *J. Artif. Intell. Res.(JAIR)*, 28:267–297, 2007.

[14] Peter E Hart, Nils J Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

[15] Frederik W Heger and Sanjiv Singh. Robust robotic assembly through contingencies, plan repair and re-planning. 2010.

[16] Malte Helmert. The fast downward planning system. *J. Artif. Intell. Res.(JAIR)*, 26: 191–246, 2006.

[17] Carlos Hernández, Roberto Asín, and Jorge A Baier. Reusing previously found a* paths for fast goal-directed navigation in dynamic terrain. In *AAAI*, pages 1158–1164, 2015.

[18] Jörg Hoffmann. Ff: The fast-forward planning system. *AI magazine*, 22(3):57, 2001.

[19] Jörg Hoffmann and Ronen I Brafman. Conformant planning via heuristic forward search: A new approach. *Artificial Intelligence*, 170(6):507–541, 2006.

[20] Leslie Pack Kaelbling, Michael L Littman, and Anthony R Cassandra. Planning and acting in partially observable stochastic domains. *Artificial intelligence*, 101(1): 99–134, 1998.

[21] Sven Koenig and Maxim Likhachev. D* lite. In *AAAI/IAAI*, pages 476–483, 2002.

[22] Sven Koenig and Maxim Likhachev. Real-time adaptive a. In *Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems*, pages 281–288. ACM, 2006.

[23] Sven Koenig, Maxim Likhachev, and David Furcy. Lifelong planning a*. *Artificial Intelligence*, 155(1):93–146, 2004.

[24] Antonín Komenda, Peter Novák, and Michal Pěchouček. Domain-independent multi-agent plan repair. *Journal of Network and Computer Applications*, 37:76–88, 2014.

[25] Nicholas Kushmerick, Steve Hanks, and Daniel S Weld. An algorithm for probabilistic planning. *Artificial Intelligence*, 76(1):239–286, 1995.

[26] Maxim Likhachev, Geoffrey J Gordon, and Sebastian Thrun. Ara*: Anytime a* with provable bounds on sub-optimality. In *Advances in Neural Information Processing Systems*, page None, 2003.

[27] Maxim Likhachev, David I Ferguson, Geoffrey J Gordon, Anthony Stentz, and Sebastian Thrun. Anytime dynamic a*: An anytime, replanning algorithm. In *ICAPS*, pages 262–271, 2005.

[28] Derek Long and Maria Fox. The 3rd international planning competition: Results and analysis. *J. Artif. Intell. Res.(JAIR)*, 20:1–59, 2003.

[29] Roberto Micalizio. A distributed control loop for autonomous recovery in a multi-agent plan. In *IJCAI*, pages 1760–1765, 2009.

[30] Alex Nash, Kenny Daniel, Sven Koenig, and Ariel Felner. Thetaˆ*: Any-angle path planning on grids. In *Proceedings of the national conference on artificial intelligence*, volume 22, page 1177. Menlo Park, CA; Cambridge, MA; London; AAAI Press; MIT Press; 1999, 2007.

[31] Raz Nissim and Ronen I Brafman. Multi-agent a* for parallel and distributed systems. In *Proceedings of the 11th International Conference on Autonomous Agents and Multiagent Systems-Volume 3*, pages 1265–1266. International Foundation for Autonomous Agents and Multiagent Systems, 2012.

[32] Raz Nissim, Ronen I Brafman, and Carmel Domshlak. A general, fully distributed multi-agent planning algorithm. In *Proceedings of the 9th International Conference on Autonomous Agents and Multiagent Systems: volume 1-Volume 1*, pages 1323–1330. International Foundation for Autonomous Agents and Multiagent Systems, 2010.

[33] Staurt J Russell and Peter Norvig. Artificial intelligence (a modern approach). pages 93–99, 2010.

[34] Anthony Stentz et al. The focussed dˆ* algorithm for real-time replanning. In *IJCAI*, volume 95, pages 1652–1659, 1995.

[35] Xiaoxun Sun, Sven Koenig, and William Yeoh. Generalized adaptive A*. In *Proceedings of the 7th international joint conference on Autonomous agents and multi-agent systems-Volume 1*, pages 469–476. International Foundation for Autonomous Agents and Multiagent Systems, 2008.

[36] Xiaoxun Sun, William Yeoh, and Sven Koenig. 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. International Foundation for Autonomous Agents and Multiagent Systems, 2010.

[37] Xiaoxun Sun, William Yeoh, and Sven Koenig. Moving target d* lite. In *Proceedings of the 9th International Conference on Autonomous Agents and Multiagent Systems: volume 1-Volume 1*, pages 67–74. International Foundation for Autonomous Agents and Multiagent Systems, 2010.

[38] Jur Van Den Berg, Dave Ferguson, and James Kuffner. 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, 2006.

[39] Roman Van Der Krogt and Mathijs De Weerdt. Plan repair as an extension of planning. In *ICAPS*, volume 5, pages 161–170, 2005.

Appendices

Appendix A

Proofs

A.1 Proof of soundness and optimality of RDA^*

In the theorems and lemmas that follow, it is assumed that a part of the original plan has been executed. Consequently, the g-value of the new initial state which is the final state of the executed plan is not set equal to zero, but equal to the cost of the plan. Accordingly, all the g-values refer to the cost of the path that starts from the original initial state. Note that this adjustment does not affect the validity of the theorems and lemmas that have to do with the optimality of the solution, which would still hold if the g-value of the new initial state was set equal to zero. That is because this convention has as a result that to every g-value, and, therefore, to every path the same constant is added, e.g. the cost of the already executed plan and, therefore, an optimal path will remain optimal. Finally, it is assumed that the number of actions is finite and all the cost actions are positive.

Theorem A.1.1. *Let S_{init} be the initial state of a RDA^* sr_1 search with gs_{init} as goal set, for which a plan has been found. If RDA^* is used for a new search sr_2 with $S_{newinit}$ as new initial state, then any state S_K will be marked as valid by the informing procedure, iff there is a path from $S_{newinit}$ to S_K.*

Proof. A state is marked as valid, only if at least one of its parent-states is valid. By default, only $S_{newinit}$ is marked as valid when sr_2 begins. Therefore, the first other state that will be marked as valid, will be a child-state of $S_{newinit}$, to which, by definition, there exists a path from $S_{newinit}$. Consequently, the next state that will be marked as valid, will be a child-state of one of the aforementioned states. From the previous, it follows by deduction that if there is a path from $S_{newinit}$ to a state, then this state will be marked as valid by the informing procedure. According to the previous, if there is no path from $S_{newinit}$ to S_K, then the informing procedure will not find any state marked as valid in the paths that are examined. Therefore S_K will not be marked as valid.

□

Lemma A.1.1. *Let S_{init} be the initial state of a RDA^* sr_1 search with gs_{init} as goal set, for which a plan has been found. If RDA^* is used for a new search sr_2 with $S_{newinit}$ as*

new initial state and a state S_K has been marked as invalid by the informing procedure, then there is no path from $S_{newinit}$ to S_K.

Proof. A state S_K is marked as invalid if the informing procedure has not found any valid parent-states of S_K. This according to theorem A.1.1 means that there is no path from $S_{newinit}$ to S_K's parent-states and, therefore, there is no path from $S_{newinit}$ to S_K. □

Theorem A.1.2. *Let S_{init} be the initial state of a RDA^* sr_1 search with gs_{init} as goal set, for which a plan has been found. If RDA^* is used for a new search sr_2 with $S_{newinit}$ as new initial state, then when a state S_K is informed the informing procedure computes the accurate cost of each valid path it examines.*

Proof. If during the informing of a state S_K, its parent-state state S_L is valid, then the corresponding p-value that refers to cost of the corresponding path is set equal to the g-value of S_L plus the cost of the generating action from S_L to S_K. Therefore, if the g-value of S_L is accurate, then p-value and, hence, the cost of the path will also be accurate. By default, the g-value of the $S_{newinit}$ is accurate since corresponds to the cost of the optimal path from S_{init} to $S_{newinit}$. The first p-value that will be computed, will be that of the first state that will be marked as valid, which according to theorem A.1.1 will be one of $S_{newinit}$ children-states. Since $S_{newinit}$ g-value and the generating action are also accurate, the corresponding p-value will be accurate. Consequently, the next p-value that will be computed will be that which corresponds to the cost of the path that leads to a child-state of one of the two aforementioned-states. From the previous, it follows by deduction that the computing of the cost of each valid path results always in the an accurate value. □

Theorem A.1.3. *Let S_{init} be the initial state of a RDA^* sr_1 search with gs_{init} as goal set, for which a plan has been found. If RDA^* is used for a new search sr_2 with $S_{newinit}$ as new initial state and S_K is a state to which there exists a path from $S_{newinit}$, then the full informing procedure will find the optimal path from $S_{newinit}$ to S_K.*

Proof. For a given state being informed, full informing procedure examines all the corresponding paths that lead to it. The examination of each path stops when one of the following three condition is met: a) the parent-pointer is marked as valid, b) the parent-pointer is marked as invalid or c) a circle has been detected. Case b) means that the path being examined is not valid. Case c) means that the path being examined is not optimal, since all actions have positive costs and therefore an optimal path cannot contain circles. Since according to theorem A.1.2, informing procedure computes the accurate cost of each path it examines, it follows that the valid path with the cost will be the optimal path from $S_{newinit}$ to S_K. □

Theorem A.1.4. *Let S_{init} be the initial state of a RDA^* sr_1 search with gs_{init} as goal set, for which a plan has been found. If RDA^* is used for a new search sr_2 with $S_{newinit}$ as new initial state, no actions costs have decreased and there is at least one path from $S_{newinit}$ to S_K, then the lazy informing procedure will find the optimal path from $S_{newinit}$ to S_K.*

Proof. For a given state being informed, lazy informing procedure examines one by one the corresponding paths that lead to in an increasing order with respect to to their original costs and stops if the cost of the path p_{min} which has the smaller updated cost of the already examined paths is not greater than the original cost of the path that is to be examined next. The examination of each path stops when one of the following three condition is met:a) the parent-pointer is marked as valid, b) the parent-pointer is marked as invalid or c) a circle has been detected. Case b) means that the path being examined is not valid. Case c) means that this path being examined is not optimal, since all actions have positive costs and therefore an optimal path cannot contain circles. Since no actions costs have decreased, the cost of any of these paths cannot have decreased. Therefore, since the updated cost of p_{min} is not greater than the original cost of another plan p_l, then it will be not greater than its updated cost. Also, since the paths were ordered in an increasing order according to their original costs, it follows that if the updated cost of p_{min} is not greater than the the original cost of the path that was to be examined after it, then it cannot be greater than the original cost of any other non-examined path. From the previous, it follows that there is no path with smaller cost than p_{min}, and therefore p_{min} is optimal, since, according to theorem A.1.2, informing procedure computes the accurate cost of each path it examines. □

Lemma A.1.2. *The informing procedure terminates.*

Proof. There are two ways that could lead to the non-termination of the informing procedure: a)an infinite number of examined paths or b) an infinite loop during the examination of a path. Since the number of the states in the open and closed list is finite, it follows that the number of paths that do not contain circles is finite too. Also, the examination of a path is aborted when a circle is detected. Therefore, neither the first nor the second condition that can lead to the non-termination of the informing procedure can hold and, therefore, the informing procedure always terminates.

□

Theorem A.1.5. *A^* is sound, if the heuristic function used produces h-values that are admissible.*

Proof. Assume that the cost of the optimal plan p_{op} for a given problem is equal to c_{op}. A^* expands at each step the state from the open list that has the minimum f-value. Supposing that a plan p' is found which is sub-optimal and the cost of which is equal to c'. The final state of the plan p_{op} was not expanded, otherwise p_{op} would have been found. If s_{Fop} and $s_{F'}$ are the final states of p_{op} and p respectively, then it must hold that $fval_{F_{op}} \geq fval_{F'}$ (1). Also, since the heuristic function produces admissible h-values, then it holds that

$$fval_{F_{op}} = gval_{F_{op}} + hval_{F_{op}} = gval_{F_{op}} + 0 = c_{op} \text{ (2)}$$
$$fval_{F'} = gval_{F'} + hval_{F'} = gval_{F'} + 0 = gval_{F'} = c' \text{(3)}$$

From (1),(2) and (3) it follows that $c_{op} \geq c'$ which is a contradiction, since we assumed that p' is sub-optimal and, thus, its cost must be grater than the cost of p_{op}. □

Theorem A.1.6. *A^* is complete, if the heuristic function used produces h-values that are admissible.*

Proof. Assuming that c_{op} is the cost of the optimal path, then in order for the path's final state to be expanded there must be only finitely many states with cost less than or equal to c_{op}. Due to the fact that every action's cost is positive and the number of actions is finite, it follows that the number of the paths from the initial state to to any other state which have cost less than c_{op} must be finite. Since the f-value of any state S_K is always greater than the cost of the corresponding cost of the path from the initial state to S_K, it follows that the number of states with a f-value less than c_{op} is finite, and, therefore, A^* is complete. □

Lemma A.1.3. *If RDA^* begins with an empty closed list and an open list which contains only its initial state, then it is equivalent to a A^* search that starts from the same initial state.*

Proof. Each step of RDA^* executed in the same way as each step of A^* : the state with the lowest f-value from the open list is selected and, if it is not a goal-state, it is generated and its children states are inserted in the open list, while the expanded state is inserted in the closed list. There are two differences between RDA^* and A^* . The first is that in the beginning of the execution its closed list might not empty and its open list might contain other states and not the initial state of the search as in the case of A^* , which in this case does not hold. The other difference has to do with the procedure of informing for states that lie in the closed or open list. In this case, since the only state inside the original open list is valid, all other states that are inserted in the closed or open list are also valid, by default, which means that the informing procedure does not take place for any state. Therefore, in this setting RDA^* is equivalent to A^* . □

Lemma A.1.4. *Let sr_1 be a A^* search. Suppose that during sr_1 a finite number of states is inserted in the open list and a finite number of states is inserted in the closed list. Also, suppose that some of the aforementioned states may have wrong f-values or may be invalid, i.e. not be accessible from the initial state. A^* is sound and complete if the following three conditions are met:i) the expanding of the states in the open list is carried out in a non-decreasing way with respect to to their f-values, ii) none of the inserted states in the closed list satisfies the goal-set and iii) the f-value of any state is never greater than the cost of the optimal path from this state to a goal-state.*

Proof. Since the invalid states are not utilized in any way, the execution of the algorithm is not affected by their insertion. Also, since the states are expanded in a non-decreasing way with respect to to their f-values and the f-values are never greater the cost of the optimal path that leads to a goal-state, if a plan is found, then it will be optimal in regard to all the states in the open list, since all other non-expanded states have at least the same f-value and, if any of them is a goal-state then the cost of the corresponding path will have at least the same cost. Moreover, since none of the inserted states in the closed list satisfies the goal-set, there can be no state in the closed list that is a goal-state. Consequently, the plan

found will be also optimal with respect to all the states in the closed list. Therefore A^* is sound in this scenario. Finally, since the number of the states inserted in the open list is finite, following the same line of reasoning as in theorem A.1.6, we reach the conclusion that A^* is also complete in this scenario.

□

Lemma A.1.5. *Let S_{init} be the initial state and gs_{init} the goal set for which RDA^* has found a plan p_1. Let $S_{newinit}$ be the new initial state that lies in p_1 and gs_{new} the new goal-set. Then, the new RDA^* search corresponds to a A^* search from $S_{newinit}$, in the open and closed list of which, a finite number of states has been inserted which may have wrong f-values or may be invalid, i.e. not be accessible from the initial state.*

Proof. According to lemma A.1.3 the initial search of RDA^* is equivalent to a A^* search that starts from S_{init}. The new RDA^* search utilizes the final closed and open list of the original search. Consequently, the new search from $S_{newinit}$ in this case begins using the open and closed lists from the previous search, which is equivalent to a A^* search from $S_{newinit}$ with a finite number of states inserted in its open and closed list which may have wrong f-values or may be invalid. Also, according to lemma A.1.2 the informing procedure of RDA^* always terminates, and, therefore, there can be no difference in the behaviour of the new RDA^* search and its corresponding A^* search because of this procedure.

□

Lemma A.1.6. *Let S_K and S_L be states respectively. If there is a path from S_K to S_L, then any state reachable from S_L is also reachable from S_K .*

Proof. Any state S_M reachable from S_L, can be reached from S_K by following the path from S_K to S_L and, then, the path from S_L to S_M. □

Lemma A.1.7. *Let S_K be a state that is located in the open list during the execution of a RDA^* search. Its g-value $gval_{S_K}$ corresponds to the cost of the optimal path from S_{init} to S_K for the given expanded states, i.e. there is no path from S_{init} to S_K with a smaller cost.*

Proof. $gval_{S_K}$ is equal to the cost of the path from S_{init} to S_K. If only one state has generated S_K, then there is only one path from S_{init} to S_K, which is, by definition, optimal. If there are more than one states, then each time S_K is re-generated, its current g-value is compared to the g-value that results from the new parent-state and the smaller value is kept and, therefore, $gval_{S_K}$ is equal to the cost of the optimal path from S_{init} to S_K. □

Lemma A.1.8. *Let S_K be a state that is located in the open list during the execution of a RDA^* search and S_M another state that lies in the optimal path from the initial state S_{init} to S_K. The cost of the optimal path from S_M S_K is equal to to $gval_{S_K} - gval_{S_M}$.*

Proof. If there is a path from S_M to S_K with a cost smaller than $gval_{S_K}$ - $gval_{S_M}$, then there will be a path from S_{init} to S_K with a cost smaller than $gval_{S_K}$, which cannot hold according to lemma A.1.7 . □

Lemma A.1.9. *Let S_M be a state which is located in the final open list of a RDA^* search that has been completed and let $S_{newinit}$ be the new initial state for a new RDA^* search. If $S_{newinit}$ was in the plan of the original search and if no actions costs have been decreased, then during the new RDA^* search, the g-value of any state S_L that was located in the final open list of the original search can never decrease, unless S_L is generated by a state which was not located in the final closed list of the original search.*

Proof. According to lemma A.1.8 , the difference $gval_{\mathrm{S_L}} - gval_{\mathrm{S_{newinit}}}$ is equal to the cost of the optimal plan from $\mathrm{S_{newinit}}$ to $\mathrm{S_L}$. $\mathrm{gval_{S_{newinit}}}$ is minimum since it lies in the optimal path from $\mathrm{S_{init}}$ to $\mathrm{S_{newinit}}$. Since no actions costs are decreases, then it follows than the cost of any path cannot decrease. Therefore, if there was a state in the closed list which generated $\mathrm{S_L}$ and resulted in a smaller $\mathrm{gval_{S_{newinit}}}$ it would correspond to a path with a decreased cost which is a contradiction according to the previous. □

Theorem A.1.7. *Let S_{init} be the initial state and gs_{init} the goal set of a RDA^* search that has been completed. If S_{new} is the new initial state and gs_{new} the new goal-set of a new RDA^* search, then RDA^* is sound and complete if the h-values that are used are admissible.*

Proof. According to lemma A.1.5, RDA^* corresponds in this case to a A^* search from $\mathrm{S_{newinit}}$ which instead of using a new open and closed list, utilizes the final closed and open lists from the previous search and which utilizes a informing procedure in order to compute the f-values of the states with respect to to $\mathrm{S_{newinit}}$ and determine which states are valid. RDA^* repairs lazily each state of the open list by re-computing its g-value and its h-value that correspond to the new goal-set, which results in an updated f-value for all the valid states, while all the invalid states are removed from the open list. Consequently, the open list is sorted according to the new f-values of its states. According to theorem A.1.4, lazy informing finds the optimal path for any state to which there is a path from $\mathrm{S_{newinit}}$ and, therefore, since RDA^* utilizes admissible heuristic, the f-value of every state $\mathrm{S_M}$ cannot be greater than the cost of the optimal path. Also according to the same theorem, lazy informing marks as invalid and, hence, does not expand the states to which no past exist from $\mathrm{S_{newinit}}$. Let's assume that there is state $\mathrm{s_A}$ with $\mathrm{fval_A}$, which is expanded before a state $\mathrm{s_B}$, which has a smaller $\mathrm{fval_B}$. In order for this to be possible, there must be another state $\mathrm{s_K}$ in the open list which generates $\mathrm{s_B}$ and has not been expanded before $\mathrm{s_A}$, which will result to $fval_{\mathrm{B}} < fval_{\mathrm{A}}$(1). Since all actions costs are positive it holds that $gval_{\mathrm{K}} < gval_{\mathrm{B}}$(2). Also since h-values are admissible, it holds that $hval_{\mathrm{K}} \leq hval_{\mathrm{B}} + c_{\mathrm{action_{KB}}}$(3). From (2) and (3), it is derived that $hval_{\mathrm{K}} + gval_{\mathrm{K}} < hval_{\mathrm{B}} + gval_{\mathrm{B}} \rightarrow fval_{\mathrm{K}} < fval_{\mathrm{B}}$ (4). From (1) and (4), it follows that $fval_{\mathrm{K}} < fval_{\mathrm{A}}$ and, therefore, $\mathrm{s_K}$ will be expanded before $\mathrm{s_A}$. But this is a contradiction, since we assumed that $\mathrm{s_K}$ has not been expanded before $\mathrm{s_A}$. From the previous, it is derived that the expanding of the states in the open list is carried out in a non-decreasing way with respect to to the f-values. Moreover, RDA^* does not utilize any invalid states. If none of the states in the final closed list satisfy $\mathrm{gs_{new}}$, RDA^* is sound and complete according to theorem A.1.4. Besides, before the execution of RDA^* begins, the original closed list is traversed and searched for goal-satisfying states in which case the one with the lowest g-value is kept. The path leading

to this state is returned as a solution when the priority of the open list becomes greater that this g-value. Since the f-value of every state S_M cannot be greater than the cost of the optimal path, the solution returned from the algorithm in this case is optimal. Therefore, RDA^* is sound and complete in this case also.

□

Lemma A.1.10. *Let S_{init} be the initial state and gs_{init} the goal set of a RDA^* search that has been completed. If S_{new} is a new initial state and gs_{new} a new goal-set which is a super-set of gs_{init}, no state in the final closed list of the original search can satisfy gs_{new}.*

Proof. Since gs_{new} is a superset of gs_{init}, every state satisfying gs_{new} satisfies also gs_{init}. Therefore, if a state exists in the final closed list that satisfies gs_{new} it will also satisfy gs_{init}. But this is impossible, since, by definition, this state would correspond to a solution for the original search in which case, it is not expanded and, therefore, it is not inserted in the closed list.

□

Theorem A.1.8. *Let S_{init} be the initial state and gs_{init} the goal set of a RDA^* search that has been completed. Supposing a new initial state S_{new}, unchanged actions' costs and a new goal-set gs_{new}, which is a super-set of gs_{init}, RDA^* is sound and complete if the h-values that are used are admissible.*

Proof. According to lemma A.1.5, RDA^* corresponds in this case to a A^* search from $S_{newinit}$ which instead of using a new open and closed list, utilizes the final closed and open lists from the previous search and which utilizes a informing procedure in order to compute the f-values of the states with respect to to $S_{newinit}$ and determine which states are valid. In this case, the closed list is not traversed since the new goal-set is a superset of the original goal-set(a set S is always a superset of itself), and according to lemma A.1.10 it is impossible that a state in the closed list satisfies the goal set. RDA^* , repairs lazily each state of the open list that is selected for expansion by re-computing a new g-value, which results in an updated f-value for it. According to theorem A.1.4, lazy informing finds the optimal path for any state to which there is a path from $S_{newinit}$ and, therefore, since RDA^* utilizes admissible heuristic, the f-value of every state S_M cannot be greater than the cost of the optimal path. Also according to the same theorem, lazy informing marks as invalid and, hence, does not expand the states to which no past exist from $S_{newinit}$.

According to lemma A.1.9 , since no actions costs changes are decreases, the g-value of a state can only decrease, if it is generated by another state that lies in the open list. Also, since gs_{new} is a superset of gs_{init}, then the initial h-value of a state is still smaller than the cost of the state from the state to a goal-state. From the previous, it is derived the f-value of a state can only decrease, if it is generated by a state that lies in the open list. Let's assume that there is state s_A with $fval_A$, which is expanded before a state s_B, which has a smaller $fval_B$. In order for this to be possible, there must be another state s_K in the open list which generates s_B and has not been expanded before s_A, which will result to $fval_B < fval_A$(1). Since all actions costs are positive it holds $gval_K < gval_B$(2). Also since h-values are admissible, it holds that $hval_K \leq hval_B + c_{action_{KB}}$(3). From (2) and

(3), it derives that $hval_{\mathrm{K}} + gval_{\mathrm{K}} < hval_{\mathrm{B}} + gval_{\mathrm{B}} \rightarrow fval_{\mathrm{K}} < fval_{\mathrm{B}}$ (4). From (1) and (4), it follows that $fval_{\mathrm{K}} < fval_{\mathrm{A}}$ and therefore, s_{K} will be expanded before s_{A}. But this is a contradiction, since we assumed that s_{K} has not been expanded before s_{A}. From the previous, it is derived that the expanding of the states in the open list is carried out in a non-decreasing way with respect to to the f-values. Therefore, according to theorem A.1.4 , RDA^* is sound and complete.

□

Theorem A.1.9. *Let S_{init} be the initial state and gs_{init} the goal set of a RDA^* search that has been completed. Supposing a new initial state, the same goal-set and a number of actions costs' changes, RDA^* is sound and complete if the h-values that are used are admissible.*

Proof. According to lemma A.1.5, RDA^* corresponds in this case to a A^* search from $S_{newinit}$ which instead of using a new open and closed list, utilizes the final closed and open lists from the previous search and which utilizes a informing procedure in order to compute the f-values of the states with respect to to $S_{newinit}$ and determine which states are valid. In this case, the closed list is not traversed since the new goal-set is a superset of the original goal-set(a set S is always a superset of itself), and according to lemma A.1.10 it is impossible that a state in the closed list satisfies the goal set. RDA^* repairs fully each state of the open list by re-computing its g-value and its h-value that correspond to the new goal-set, which results in an updated f-value for all the valid states, while all the invalid states are removed from the open list. Consequently, the open list is sorted according to the new f-values of its states. According to theorem A.1.3, full informing finds the optimal path for any state to which there is a path from $S_{newinit}$ and, therefore, since RDA^* utilizes admissible heuristic, the f-value of every state S_{M} cannot be greater than the cost of the optimal path. Also according to the same theorem, informing marks as invalid and, hence, does not expand the states to which no past exist from $S_{newinit}$.

Let's assume that there is state s_{A} with $fval_{\mathrm{A}}$, which is expanded before a state s_{B}, which has a smaller $fval_{\mathrm{B}}$. In order for this to be possible, there must be another state s_{K} in the open list which generates s_{B} and has not been expanded before s_{A}, which will result to $fval_{\mathrm{B}} < fval_{\mathrm{A}}$(1). Since all actions costs are positive it holds that $gval_{\mathrm{K}} < gval_{\mathrm{B}}$(2). Also since h-values are admissible, it holds that $hval_{\mathrm{K}} \leq hval_{\mathrm{B}} + c_{\mathrm{action_{KB}}}$(3). From (2) and (3), it derives that $hval_{\mathrm{K}} + gval_{\mathrm{K}} < hval_{\mathrm{B}} + gval_{\mathrm{B}} \Rightarrow fval_{\mathrm{K}} < fval_{\mathrm{B}}$ (4). From (1) and (4), it follows that $fval_{\mathrm{K}} < fval_{\mathrm{A}}$ and therefore, s_{K} will be expanded before s_{A}. But this is a contradiction, since we assumed that s_{K} has not been expanded before s_{A}. From the previous, it is derived that the expanding of the states in the open list is carried out in a non-decreasing way with respect to to the f-values. Therefore, according to theorem A.1.4, RDA^* is sound and complete.

□

Theorem A.1.10. *Let S_{init} be the initial state and gs_{init} the goal set of a RDA^* search that has been completed. Supposing a new initial state, the same goal-set and a number of actions costs' increases, RDA^* is sound and complete if the h-values that are used are admissible.*

Proof. According to lemma A.1.5, RDA^* corresponds in this case to a A^* search from $S_{newinit}$ which instead of using a new open and closed list, utilizes the final closed and open lists from the previous search and which utilizes a informing procedure in order to compute the f-values of the states with respect to to $S_{newinit}$ and determine which states are valid. In this case, the closed list is not traversed since the new goal-set is a superset of the original goal-set(a set S is always a superset of itself), and according to lemma A.1.10 it is impossible that a state in the closed list satisfies the goal set. RDA^* , repairs lazily each state of the open list that is selected for expansion by re-computing a new g-value, which results in an updated f-value for it. According to theorem A.1.4, lazy informing finds the optimal path for any state to which there is a path from $S_{newinit}$ and, therefore, since RDA^* utilizes admissible heuristic, the f-value of every state S_M cannot be greater than the cost of the optimal path. Also according to the same theorem, lazy informing marks as invalid and, hence, does not expand the states to which no past exist from $S_{newinit}$.

According to lemma A.1.9 , since no actions costs changes are decreases, the g-value of a state S_M can only decrease, if it is generated by another state that lies in the open list. Also, since the goal-set does not change then the initial h-value which was admissible remains smaller than the cost of the path from S_M to a goal-state. From the previous, it is derived the f-value of a state can only decrease, if it is generated by a state that lies in the open list. Let's assume that there is state s_A with $fval_A$, which is expanded before a state s_B, which has a smaller $fval_B$. In order for this to be possible, there must be another state s_K in the open list which generates s_B and has not been expanded before s_A, which will result to $fval_B < fval_A$(1). Since all actions costs are positive it holds $gval_K < gval_B$(2). Also since h-values are admissible, it holds that $hval_K \leq hval_B + c_{action_{KB}}$(3). From (2) and (3), it derives that $hval_K + gval_K < hval_B + gval_B \rightarrow fval_K < fval_B$ (4). From (1) and (4), it follows that $fval_K < fval_A$ and therefore, s_K will be expanded before s_A. But this is a contradiction, since we assumed that s_K has not been expanded before s_A. From the previous, it is derived that the expanding of the states in the open list is carried out in a non-decreasing way with respect to to the f-values. Therefore, according to theorem A.1.4 , RDA^* is sound and complete.

□

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

Proof. The general case variation of RDA^* for goal-set modifications is sound and complete according to theorem A.1.7. The special case variation of RDA^* for goal-set modifications is sound and complete according to theorem A.1.8. The general case variation of RDA^* for actions costs modifications is sound and complete according to theorem A.1.9. The special case variation of RDA^* for actions costs modifications is sound and complete according to theorem A.1.10. Therefore, RDA^* is sound and complete for every re-planning scenario it addresses.

□