Deadline Compliance, Predictability, and On-line Optimization in Real-Time Problem Solving

Babak Hamidzadeh
(hamidzad@cs.ust.hk)
Department of Computer Science
University of Science & Technology
Clear Water Bay, Kowloon, Hong Kong

Shashi Shekhar
(shekhar@cs.umn.edu)
Department of Computer Science
University of Minnesota
Minneapolis, MN 55455

Abstract

Real-time algorithms need to address the time constraints (e.g. deadlines) imposed by applications like process control and robot navigation. Furthermore, dependable real-time algorithms need to be predictable about their ability to meet the time constraints of given tasks. A real-time algorithm is predictable, if it can decide the feasibility of meeting time constraints of a given task or an arbitrary task from a task set well ahead of the deadline. Lastly, a real-time algorithm should exhibit progressively optimizing behavior (i.e. the quality of the solution produced should improve as time constraints are relaxed). We propose a new algorithm, SARTS, that is based on a novel on-line technique to choose the proper values of parameters which control the time allocated to planning based on the time constraints. SARTS also provides criteria to predict its ability to meet the time constraints of a given task. The paper provides theoretical and experimental characterization of SARTS as a dependable real-time algorithm.

1. Introduction

Real-time systems are increasingly being used in important applications such as avionics, process control, robot and vision systems, and command and control. These applications are characterized by attributes, such as strict time constraints posed as deadlines on the time by which the system is expected to produce a solution to a given problem, or the time by which the system is expected to predict deadline violation, in order to avoid the catastrophic consequences of violating deadlines. Another attribute of most of the above applications is that they require dynamic problem solving due to the lack of complete a priori information at compile time. Many real-time computing problems such as task scheduling on uni- and multi-processor architectures [1] are hard optimization problems for which finding solutions are computationally prohibitive, particularly if such problem solving is to take place on-line. Thus, dynamic problem solving in real-time is faced with the tradeoff between sub-optimal solutions and the time allocated to find such solutions. Techniques addressing dynamic problem solving in real-time are often required to progressively optimize rather than to optimize, namely they are expected to find the best answer within the allotted time, and improve upon it if additional time is allocated.

Many combinatorial optimization problems, including those in real-time computing, can be formulated as the search for an answer in the space of all partial and complete solutions, such that the answer minimizes (or maximizes) an objective function [2,3]. Most search-theoretic approaches to combinatorial optimization problems have addressed only the static version of the problem[3]. There has been a recent rise in research on search techniques for dynamic (on-line) optimization [4,5]. The problem with most of the dynamic optimization techniques is that they take an ad-hoc approach to handling the time allocated to optimization. In dynamic real-time computing, controlling the time to find a solution is as important as the time it takes to execute that solution. The total response time of a system to a problem instance, in such domains, is the sum of the time to search for the solution, and the time to execute that solution. Another major drawback of the existing algorithms is that presently they do not provide guarantees on meeting deadlines, as shown by the experimental analysis in[6]. Such algorithms have not provided tests to predict the possibility of missing a given deadline. Furthermore, these algorithms do not provide a mechanism to monitor the progress of the algorithm in meeting a deadline and to signal an alarm as soon as it is discovered that the deadline will not be met. Such mechanism is very useful in many applications where failure to meet a deadline is catastrophic. In such applications deadline violation should be signaled as soon as possible, in order for some contingency actions to be undertaken to prevent disaster. One of the main reasons for these limitations is that the existing approaches [4,7] do not adequately address the allocation of time to plan in context of the remaining time to deadline.

This paper examines an on-line parameter tuning approach to address the above problems. This approach uses the information available for the current problem instance at various stages of the planning procedure to control the time allocated to planning. Based on this technique we introduce a new algorithm called Self-Adjusting Real-Time Search algorithm (SARTS) and characterize its effectiveness theoretically. A major contribution of SARTS is its ability to adapt the parameters of the search (planning) process based on the remaining time to deadline (i.e. slack) and on the estimated execution costs, in order to improve deadline compliance. Feasibility tests are provided that enable the algorithm to predict deadline violation prior to the deadline. SARTS is able to continue monitoring the progress of the problem solving effort towards the goal during problem solving. This capability enables this algorithm to detect initially inconclusive tasks that turn out to be infeasible. SARTS is a progressively optimizing algorithm in the sense that it will continue to improve the solution quality as the time constraints are

relaxed. The algorithm finds the optimal solution if adequate planning time can be allocated in presence of larger slacks. The paper also provides methodologies for analyzing the predictive power of real-time search algorithms.

The remaining sections of this paper are organized as follows. Section 2 contains a formal statement of the problem addressed in this paper. Section 3 provides the specification and theoretical analysis of SARTS. Section 4 provides experimental evaluation of SARTS. Finally, section 5 provides a set of concluding remarks.

2. Problem Statement

A state space can be represented by a graph G(V,E) that consists of a set of nodes V, and a set of edges E connecting some of the nodes in the graph. A node *vieV* in the graph represents a certain configuration of the environment called a state. An edge (vi, *vj)eE* in the graph represents a transformation function in the state space that transforms a node vi, at one end of the edge to the node *Vj* at the other end of that edge. Based on the size of the state space, availability of information and the nature of the problem solving, the complete state space may not be generated and stored prior to the start of problem solving. Associated with each edge (vi, *vj)eE* is an execution cost ceV|V, which is regarded as the cost of transforming vi, to Vj, or the cost of traversing the edge (vi, vj;).

Problem solving in many important applications can be modeled as a problem of searching for and executing a path P in G before deadline D, that connects (transforms), via edges in G, the start node "s" to the goal node "g". Associated with the search phase (also referred to as planning) is a cost *CPp* that signifies the time that was taken by the search algorithm to find path P. Associated with the execution phase is a cost *CEp* that signifies the time that was taken by the system to execute path P. The total response time *Tresp* of performing a task is defined as the sum of the costs of planning and execution phases (i.e. *Tresp -CP + CE)*. The planning phase of a task consists of generating and evaluating a set of nodes in the state space, in order to find a solution path for a given problem instance. Expansion of a node in G by the search algorithm refers to the process of generating the successors of that node in the graph. The successors of a node *vt* are the set of all the K nodes (v|,...,vk) which are connected to *vi* via direct edges (vi, vj),…,(v, *vk).* A heuristic function h(n,g) is a function that provides an estimate of the distance between node "n" and the goal node "g". Evaluation of a node "n" in G by the search algorithm refers to the process of examining certain characteristics of the node such as its actual distance from "s" or its estimated distance from "g" via a heuristic function h(n,g), and possibly ordering a set of nodes based on those characteristics. The execution phase of a task consists of applying the transformation functions associated with each edge on a solution path P, in order to reach the goal node "g" from the start node "s". Using the above definitions, the underlying problem of performing a task in this model can be specified formally as follows.

Given a graph G(V,E), a start node "s", a goal node "g" and a deadline D, a real-time problem solver needs to search for and execute a path P = *[(s* v,),(v, *v2)).....(vm g))* that connects "s" and "g". to maximize the following objectives in decreasing order of priority: 1) Predict possible deadline violation as soon as possible (i.e. well before the deadline D arrives), 2) Maximize the probability of searching and executing the path P before the deadline D (i.e. Maximize *?T.(Tresp* < D)), and 3) If time permits, continue to improve the solution.

In this paper, we concentrate on analyzing the effectiveness of SARTS in accomplishing the first two objectives. The discussion of the behavior of SARTS in terms of the third objective has been covered elsewhere [8] and is, thus, omitted from this paper. We show that the predictability of SARTS is related to the accuracy of its heuristic functions and that the expected value of its predictability improves as the accuracy of the heuristic estimates increases. In our analysis of SARTS's ability to comply with deadlines, we separate different features of SARTS in different algorithms, in order to observe the effect of each feature in isolation.

3. Self-Adjusting Real-Time Search (SARTS)

To perform real-time tasks by their deadlines, SARTS plans partial solutions for a task and executes those partial solutions in interleaved planning and execution phases. The plan-execute phases are repeated until a goal is reached, or until deadline violation is predicted. SARTS uses a novel on-line parameter tuning and prediction technique to determine the time of a planning phase, and to predict deadline violation ahead of time. The allocated time to planning in this search algorithm is self-adjusted based on what is known on-line about the nature of the problem and the problem instance. The algorithm continually self-adjusts its parameters based on the remaining time to deadline. To determine the maximum time allowable for planning, SARTS estimates the slack (i.e. the difference of remaining time to deadline and the estimated execution time to reach the goal). The larger the slack, the greater is the allocated time to planning. Slack can also be used to detect infeasible problem instances based on the predicted maximum time allowed for planning (e.g. negative values). Slack-based determination of planning time is useful for improving the dependability via prediction of deadline violation. It also makes the algorithm progressively optimizing for many application domains where solution quality can improve monotonically with additional planning effort. In these domains, larger slacks lead to larger planning times which, in turn, result in improved solution quality.

During the plan phase of cycle i, a partial path from the start node s(i) towards the goal is planned. The plan phase of a cycle is terminated when allocated time to plan runs out, i.e. *CP(i) > aCE(i)* (1). This stopping criterion controls the planning cost incurred in the current cycle (CP(i)) as a fraction *(a)* of the execution cost (CE(i) to be incurred in the current cycle. Parameter *a* can change the behavior of SARTS in interesting ways. Large value of *a* may allow adequate planning time in the very first cycle to plan the optimal path obviating the need of additional cycles. Small values of *a* may restrict the planning time to a minimal in each cycle, reducing

SARTS to a greedy algorithm. Besides controlling planning cost, criterion (1) is also effective in achieving near optimal response time in sequential plan and execute paradigms[9].

CP(i) may represent elapsed time and may depend on the number of nodes expanded during the ith cycle. The expanded nodes are those, whose descendants were generated and were added, in the correct order, to the open list during cycle i. *CE(i)* represents the execution cost of a path(s(i), s(i+l)), where s(i+l) is the most promising frontier node found by the planning in cycle i. The most promising node refers to a node that has the best potential to be on the shortest path to goal. The most promising node found in a cycle becomes the start node of the next cycle. *CE(i)* in SARTS is calculated by adding the length of the edges on the current partial path(s(i),s(i+l)) = {(s(i) 1), ..., (m s(i+1))}.

At the end of planning in cycle i, SARTS leaves a special number at the s(i). This number represents the second best choice at each node on the traversed path. As the node with the best heuristic estimate is traversed, the heuristic value of the second best node at that decision point is left as the new heuristic estimate of the traversed node. A later cycle may utilize the result of a prior cycle via this special number to avoid looping in cycles indefinitely[4].

During the execution phase, the partial path planned in the plan phase of the current cycle is executed. The partial path may consist of one or more edges in the graph. The execution phase of a cycle in SARTS may consist of traversing the entire partial path. Planning carried out in each cycle consists of several iterations. An iteration of the SARTS algorithm is similar to an iteration of A* [10]. Each iteration removes the most promising node from a list of unexplored nodes (i.e. the open list), generates the immediate children of that node (expanding a node), adds those children to the open list, and sorts the new list with the most promising node first. The iterations of SARTS are treated as atomic, i.e. each iteration will carry out ail of the above mentioned tasks.

We note that *CE(i)* is undefined before the first iteration in cycle i. After an iteration in cycle i, CE(i) represents the execution cost of path(s(i),n), where n is the most promising node at the end of the iteration. The stopping criteria for cycle i is examined after each iteration. The minimal planning in a cycle is an iteration, which makes SARTS behave like a greedy local gradient descent algorithm. Such a cycle is called *greedy cycle*. During the plan phase, greedy cycle only expands the current node s(i). The set of s(i)'s descendants make up the open list. The most promising descendant is thus chosen at the end of planning. The execute phase consists of traversal of a single edge (s(i),s(i+1)).

3.1. Self-Adjustment

Planning time is controlled as a fraction of total execution time via the paramet α in SARTS. The value of α may be determined at compile time to remain constant from cycle to cycle. However, in many applications, it is desirable to adjust *a* from cycle to cycle, as additional information becomes available. Self-adaptation of *a* is useful in order to respond to a wide variety of deadlines in an progressively optimizing fashion or to respond to changes in the world at run-time. We define a set of basic concepts and explain how SARTS self-adjusts the parameter α to control planning lime based on slack.

Let $T^r(i)$ be the remaining time to deadline at the beginning of the ith cycle. $T^r(i)$ can be expressed as $D - T^s(i)$, where D is the absolute deadline by which time SARTS is required to plan and execute path(s,g), and $T^s(i)$ is the starting time of cycle i. Let $T_e^r(i)$ be the true execution time of executing a path from s(i) to the goal node. $T_e^r(i)$ is estimated at the beginning of cycle i with help from a heuristic function $h(s(i), g)$. Note that $T_e^r(i)$ can be expressed as the sum of CE(i) over forthcoming cycles i, i+1, ..., k where goal is reached in cycle k. Let $T_p^r(i)$l denote the time that SARTS will allocate to plan the complete path(s(i)$T_p^r(i)$ can be expressed as a sum of CP(i) over forthcoming cycles i, i+1, ..., k. In general, estimating planning time is a non-trivial problem, and can be as difficult as finding path(s(i),g). SARTS does not attempt to estimate the value of $T_p^r(i)$. Instead SARTS controls the maximum values for planriing times CP(i) and $T_p^r(i)$ on the basis of available slack.

Lemma 1: If all cycles of SARTS are non-trivial (i.e. CP(i) and CE(i) are large with respect to the cost of a single SARTS iteration), the stopping criterion of SARTS allocates approximately $\alpha T_e^r(i)$ time for planning, i.e. $T_p^r(i) \approx \alpha T_e^r(i)$. For a proof see [11].

Let slack, $T_{slack}(i)$, represent the remaining time to deadline after a path from s(i) to g is executed. Slack provides an upper bound on the time that can be allocated to planning. We note that: $T_{slack}(i) = T^r(i) - T_e^r(i) \geq T_p^r(i)$, or $T_e^r(i)\alpha(i) \leq T_{slack}(i)$ or $\alpha(i) \leq \frac{T_{slack}(i)}{T_e^r(i)}$, or $\alpha(i) \leq \frac{T^r(i)}{T_e^r(i)} - 1$ (2a), or $\alpha(i) \leq \frac{T^r(i)}{h^a(s(i), g)} - 1$ (2b), where h^a is an admissible heuristic. The above calculations are interesting since they provide upper bounds on the possible time to plan. Equations 2a and 2b show that the upper bound on the value of *alpha{i)* is directly proportional to slack and that shorter slack causes a reduction in the value of *a(i)*. Intuitively, small slack means that there is little time for planning and that the planning process of SARTS approaches a greedy local gradient descent search. We also note that the atomicity of SARTS iterations imposes lower bounds on the values of a(i)» since the smallest amount of planning carried out by SARTS in any cycle is equal to that in an iteration of A*. Self-adjustment of parameter *a(i)* must observe these bounds. It has been shown [11] that SARTS reduces to A* for large deadlines, and that SARTS reduces to a greedy local gradient descent search, if deadlines are tight leading to small slacks.

3.2. Prediction of Deadline Violation

A unique aspect of SARTS relates to the tests that it provides to classify tasks into feasible, and infeasible tasks, based on the knowledge about the nature of the heuristic function. SARTS may not accept a task, if the

infcasibility tests show that the deadline cannot be met. Since infeasibility tests cannot discriminate between feasible tasks and inconclusive tasks, SARTS continues to monitor a task from cycle to cycle to guard against the danger of an inconclusive tasks becoming an infeasible task as time goes by. Using the tests, SARTS is able to predict the possibility of deadline violation ahead of time. Given an application domain, feasibility tests may be designed based on the domain knowledge.

Infeasibility Test: Given an admissible heuristic function $h^0(s,g)$, SARTS considers a task to reach the goal node g from a start node s(i) at the beginning of cycle i to be infeasible if $\alpha(i) = \frac{T'(i)}{h^a(s(i),g)} - 1 < 0$. A negative value for $\alpha(i)$ shows that SARTS may not be able to meet the deadline, since the slack is negative. Negative slack times are indicative of the fact that the estimated total execution time is larger than the remaining time to deadline, making the task infeasible.

Feasibility Test: Given a perfect heuristic function $h(s,g)$, SARTS considers a task to reach goal node g from a start node s(i) at the beginning of cycle i to be feasible if $\alpha(i) = \frac{T'(i)}{h^*(s(i),g)} - 1 > \frac{n\sigma}{h^*(s(i),g)}$ e a is the cost of one iteration of A* and n is an estimate of the number of edges (decision points) to goal. The number of decision points to goal, n, depends on the distribution of execution cost per edge. For example, in a graph with the same execution costs for all edges, n can be estimated by the heuristic function, h. In general, an interval or an upper bound estimate for n can be derived from the distribution of edge execution costs for a given confidence level. We note that estimation of n does not need any assumptions about the planning cost.

Lemma 2: Given h*, SARTS will find and execute a solution to a problem instance within the problem's given deadline, if the problem is classified as feasible. For a proof see [11].

Note /that the discussions on a perfect heuristic are included here to argue that the correctness and predictability of the feasibility tests are correlated with the power of heuristics available for the particular domain, and that the expected utility of the tests increases with the accuracy of the heuristics.

4. Experimental Evaluation

In our experiments we have compared different versions of SARTS with each other and with RTA*(n)[12,13]. The problem instances of the experiments consist of graphs that represent the state space of all partial and complete solutions. Each version of SARTS focuses on a single characteristic of SARTS, in order to test the effect of that particular characteristic on the performance of SARTS, in isolation.

We will review the basic steps of RTA*(n) to facilitate interpretation of our observations. At each cycle, RTA*(n) first creates the successor nodes of the current state. The current state is the actual position of the system. As each successor node is created, its estimated distance from the goal (i.e. h), the cost from the current node (i.e. g), and the sum of h and g (i.e. f are calculated. The eudidean distance formula is used to calculate the heuristic values. This heuristic formula is monotonic [12] and is guaranteed to produce optimal solutions in A*. In the case of RTA*. this heuristic formula allows substantial pruning of frontier nodes without loss of valuable information in reaching a partial solution. Notice that, unlike A* in which g is the value of the total cost so far, namely from the start node to the current successor node, in RTA*, g is the value of the cost from the current node to each of its successor nodes. The h values are calculated via look-ahead search. The general rule of thumb is that the larger the number of look-aheads (i.e. the larger the n), the better the estimated f value (i.e. g+h) will be. However, we encountered cases in which the greater look-aheads led the algorithm to more costly solutions. Also, one must note that while greater look-aheads are generally helpful in finding shorter paths to the goal (i.e. lower execution cost), they require more processing and planning (i.e. higher planning cost). Once all the successor nodes and their f values are determined, the algorithm sorts these nodes with respect to their f values. The successor node with the smallest f value is chosen as the next physical move for the RTA* algorithm. This process is repeated until a solution is reached. While this algorithm can not guarantee termination in the case of graphs with no solutions, it does guarantee that it will not get stuck in local minima and graph cycles. This is done by penalizing cyclic and dead-end paths, and by leaving the h value of the second best path at each decision point [12]. The greedy algorithm is a special version of RTA*(n) in which the search algorithm examines only the immediate neighbors of the current node, without any look-ahead search, to make a decision about its next move.

The different versions of SARTS that are tested in the experiments of this section are as follows. The FL(n) algorithm performs a fixed number of iterations during each scheduling cycle, starting from the current node of the cycle as the start state. During the execution cycle, this algorithm traverses the whole partial path that was planned during the scheduling cycle (i.e. FL(n) may traverse more than one edge at each execution cycle). Note that this algorithm is not capable of controlling the scheduling effort at run time. FL(n) is also not sensitive to the remaining time to deadline. FL(n) differs from RTA*(n) in that it performs the look-ahead search starting from the current node rather than having a separate look-ahead search for each neighbor of the current node. The FA(α) algorithm differs from FL(n) in that it uses a stopping criterion similar to that of SARTS (inequality 1 of section 3) to terminate a scheduling cycle. The parameter a in the stopping criterion of FA(α), however, is fixed from one cycle to the next. This algorithm does not adjust the scheduling effort based on the remaining time to deadline. During an execution cycle, (α) traverses all edges in the partially planned path of the previous scheduling cycle. The SS algorithm is that version of SARTS which traverses a single edge during each execution cycle. Its scheduling phase differs from F(α)'s in that the parameter α of SS's stopping criterion is adjusted at each cycle, based on the remaining time to deadline. Table 1 summarizes different algorithms' characteristics in terms of their

execution effort per cycle and their planning effort allocation per cycle.

Table 1: Algorithms and their distinguishing characteristics

Alg.	execute/cycle	plan/cycle
RTA*(n)	1 edge	bounded look-ahead depth n for each neighbor of current node
FL(n)	partial path	n A* iterations starting from current node
FA(α)	partial path	(constant α)*(execution cost of partial path)
SS	1 edge	(adjustable α)*(execution cost of partial path)
SARTS	partial path	(adjustable α)*(execution cost of partial path)

4.1. Real-Time Tasks & Deadline Compliance

The ability of an algorithm to meet deadlines of a high percentage of tasks measures its capability to comply with the time constraints of tasks, its capability to predict deadline violation, and its capability to guarantee compliance with the time constraints of those tasks whose deadlines were predicted to be met. In this section, we compare different versions of SARTS with RTA*(n), via experiments that evaluate the deadline compliance ability of each of the algorithms. The parameters of the experiment are the graph size in terms of number of nodes k, its degree of connectivity β, different values of the look-ahead parameter n, different values of the parameter α of the stopping criterion of SARTS, and the deadlines. We chose the value of 4/k for the degree of connectivity of the graphs generated for our experiments. The problem instances consist of 870 distinct (start, goal) pairs on a randomly generated graph of 30 nodes. Each problem instance is run under a set of deadlines (i.e. 10, 20, ..., 1000). For this experiment, values n = 0, 1, 2, 3, and 4 were chosen for the look-ahead parameter in RTA*(n), and values a = 0.1,1,3, and 10 were chosen for the parameter α in different versions of SARTS.

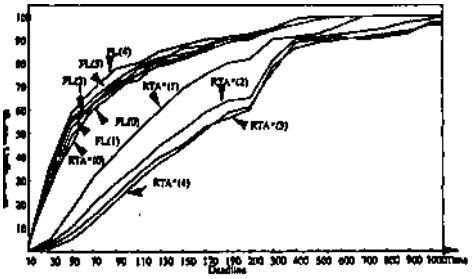

Figure 1.: Comparison of FL(n) and RTA*(n) in Complying with Deadlines

Figures 1, 2, and 3 demonstrate the results of the experiments on SARTS, SS, FA(a), FL(n), and RTA*(n), where n is the fixed look-ahead depth. Figure 1 compares the FL(n) with RTA*(n). This comparison was performed to evaluate the effect of a uniform-breadth look-ahead search for every neighbor of a current node versus that of an overall partial A* search which explores, more in depth, the more promising frontier nodes. RTA*(n) may lead to shorter paths in terms of execution costs. This algorithm, however, incurs larger planning costs than FL(n). As is shown in the figure, FL(n) (n=0,1,2,3,4) performs as well as or better than RTA*(n) (n=0,1,2,3,4) for most n. The performance of FL(n) is improved over RTA*(n), due to the fact that FL(n) expands fewer nodes by performing a top-level search starting from the current node at each cycle. Figure 2 compares FL(n) and FA(α). Comparison of FL(n) and F (α) was performed to evaluate the effect of a constant look-ahead bound versus that of a look-ahead bound that is a function of the execution costs. Using a look-ahead bound as a function of the execution costs aims at controlling the planning effort based on its tradeoff with execution cost to address minimized total response times. As is shown in the figure, FA(α) performs the same as FL(n) for α =0.1 and 1. FA(α) shows much improved performance, however, for α = 3 and 10. The improved performance is due to the fact that f(α) uses a stopping criterion that accounts for scheduling effort as well as the execution cost.

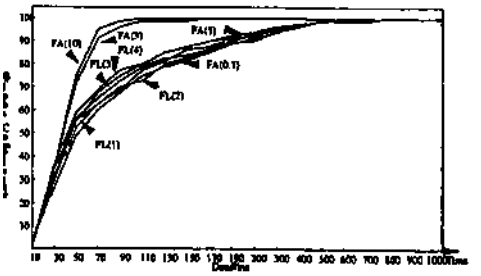

Figure 2.: Comparison of FL(n) and FA(a) in Complying with Deadlines

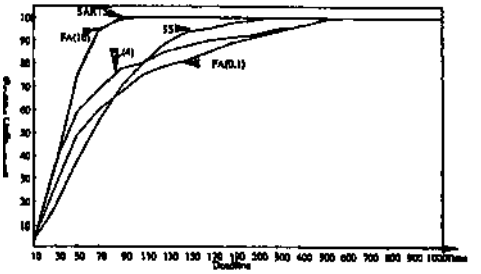

Figure 3.: Comparison of SARTS, SS, FL(4), FA(0.1), and FA(10) in Complying with Deadlines

Comparison of SARTS and ,(α) would reveal the effect of self-adjusting parameter *a* based on the remaining time to deadline versus a fixed *a* that is insensitive to the progress of the algorithm towards meeting its deadline. In comparing the SARTS and the SS algorithm we count the ratio of tasks whose deadlines were met over all 870 possible problem instances. This formula concentrates on finding the fraction of complied deadlines over all tasks. This comparison was mainly done to evaluate the effect of partial path traversal versus single-edge traversal during each execution phase. One of the consequences of single-edge traversal is that there will be more plan-execute cycles which allow the algorithm to monitor progress towards the goal. The drawback of single-edge traversal, however, is that it does not take full advantage of the planning effort in a cycle. This approach thus leads to increased overall planning for each problem instance. Figure 3 demonstrates the results of the experiment com-

paring the SARTS algorithm with SS, FL(4) and FA(O.I). As is shown in the figure, SARTS performs as well as the best F(α) by self-adjusting α based on slack. The SS algorithm was found to predict many more deadline violations and incurred a larger number of false alarms than SARTS. The large number of false alarm predictions in SARTS caused a poorer overall deadline compliance over all problem instances.

4.2. Deadline Violation Prediction

We examine the use of negative *a* in the infeasibility test of SARTS as an indicator of possible deadline violation in this sub-section. We also examine the effect of different heuristics on predictability of SARTS. We use true-negative (TN), false-negative (FN), false-positive(FP) and true-positive (TP) categories, in order to evaluate the test. The prediction accuracy of the test is measured as $\frac{TP+FN}{\#\ of\ instances} \times 100$. We note that the two measures of accuracy and true positivity together signify the dependability of an algorithm. The accuracy measures the predictive power of an algorithm. True-positivity, on the other hand, measures the deadline-compliance ability of an algorithm, since it represents the number of accepted problem instances which met their deadlines.

To explore the effect of different heuristics in predicting deadline violation, we ran a set of experiments on a grid world. In a grid, the nodes are arranged to form a rectangular grid in which each inner node with coordinates (i,j) is connected to all its neighboring nodes (ij+1), (i,j-1), (i+w,j), and (i-w,j) via an edge, where w is the width of the grid. An inner node is defined to be a node that is not on the periphery of the grid. For these experiments, a 10 x 20 grid was generated. 572 (start, goal) pairs were examined, where each start and goal node was selected to be an inner node.

Table 2: Deadline Compliance of SARTS with Euddean-Distance Function for 572 problem instances.

D	10	30	50	70	90	200
TN	0	30	18	17	6	0
FN	544	414	327	262	215	79
FP	0	0	0	0	0	0
TP	28	128	227	293	351	493
Acc.	100	95	97	97	99	100

Four different heuristics were examined in these experiments. Manhattan distance was used as an exact estimator of the remaining distance to goal in the grid world. As an over-estimator of the distance, we used twice the manhattan distance as one of our inadmissible heuristics. Euclidean distance was used as the underestimator of the distance. The fourth heuristic is also an inadmissible estimator based on the manhattan distance with introduced error that is randomly added to or subtracted from the exact estimate each time. In the figures, the plots corresponding to different heuristics are labeled as follows. "Perfect" denotes Manhattan Distance Heuristic, "Admissible" denotes Euclidean Distance, "Inadmissible" denotes Double Manhattan Distance (Over-estimate), and "Noisy" denotes Error Distance.

We expect the underestimating heuristic to have no false-positive instances over all deadlines. This is due to the fact that the true planning and execution costs incurred by the algorithm are expected to exceed the estimated value provided by the underestimator. In using the exact estimator of the remaining distance, we expect to see high degrees of predictability. The overestimator is expected to provide an upper bound on the true planning and execution costs incurred by the algorithm. Thus, a problem instance that is predicted to meet a deadline, using this heuristic, is expected to do so. The expected behavior of the algorithm with a noisy heuristic estimator is lower prediction accuracy.

Table 3: Deadline Compliance of SARTS with Double-Manhatan-Distance Function for 572 problem instances.

D	10	30	50	70	90	200
TN	0	0	0	0	0	0
FN	561	467	308	180	56	0
FP	0	0	0	0	0	0
TP	11	105	264	392	516	572
Acc.	100	100	100	100	100	100

Tables 2 through 5 show the results of these experiments. The row heading D, in the tables, denotes deadline values, and the row heading Ace. denotes the accuracy. Table 2 provides the data for SARTS with a euclidean-distance heuristic. As expected, this heuristic provides a good infeasibility test (i.e. a problem instance that was predicted to be infeasible, did in fact miss its deadline). Table 3 provides the data for SARTS with double-manhattan-distance heuristic. This estimate due to its pessimistic nature, provides 100% accuracy for all problem instances over all deadlines. This kind of heuristic is useful when missing deadlines can have highly undesirable effects. Note that all problem instances that were predicted to miss their deadlines, in this experiment, did in fact do so (i.e. *0%* TN's). Table 4 provides the data for SARTS with a manhattan-distance heuristic. This estimate provides very high prediction accuracies for all problem instances, over all deadlines. We note that the few number of deviations in predicting deadline violations are due to the cost of a single iteration (i.e. σ) that can be incurred in a SARTS cycle when the stopping criterion is met. Examining the problem instances that were not correctly predicted by SARTS, using a manhattan distance heuristic, revealed that the deadlines were missed by one time unit (i.e. σ in this experiment) in all cases. As is shown in the tables, the manhattan-distance heuristic produced the highest percentage of true-positive cases, which signifies the higher degree of deadline compliance for this heuristic. Finally, table 5 provides the data for SARTS with a noisy-manhattan-distance heuristic. As expected, this heuristic produced predictions with lower accuracy and lower deadline compliance even for larger deadlines. Figures 4 and 5 demonstrate the effect of different heuristics on the dependability of the SARTS algorithm. As is shown in figure 4, the perfect heuristic outperforms the other heuristics in its effect on deadline compliance of SARTS. Figure 5 demonstrates the effect of different heuristics on the predictability of SARTS. According to

the figure, the overestimating heuristic enables SARTS to provide 100% predictability. The perfect and the admissible heuristics perform similarly. The perfect heuristic, however, reaches 100% predictability sooner (at smaller deadlines) than the admissible heuristic.

Table 4: Deadline Compliance of SARTS with Manhattan-Distance Function for 572 problem instances.

D	10	30	50	70	90	200
TN	0	14	11	11	0	0
FN	544	404	244	84	4	0
FP	0	0	0	0	0	0
TP	28	154	317	477	568	572
Acc.	100	98	98	98	100	100

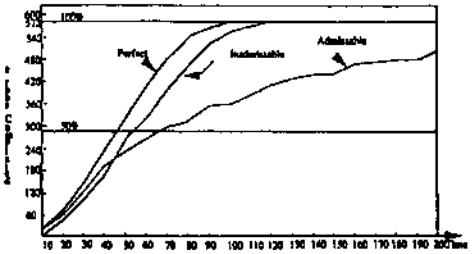

Figure 4.: Deadline Compliance Behavior of SARTS for Different Heuristics.

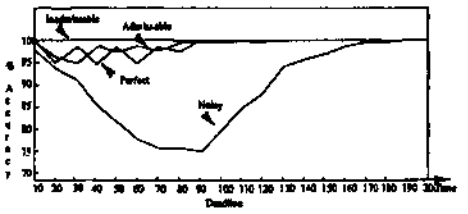

Figure 5.: Predictive Behavior of SARTS for Different Heuristics.

Table 5: Deadline Compliance of SARTS with Error-Distance Function for 572 problem instances.

D	10	30	50	70	90	200
TN	16	54	105	141	146	0
FN	540	430	282	158	69	4
FP	0	0	0	0	0	0
TP	16	88	185	273	357	568
Acc.	97	91	82	75	74	100

5. Conclusion

Dependable real-time algorithms should strive to meet the time-constraints of a given set of tasks. These algorithms should also provide tests to detect possible deadline violations ahead of time. Finally, a real-time algorithm should account for its own planning time as well as the execution cost of the solution it produces, in order to meet deadlines. SARTS is a new real-time search algorithm. It allocates time for planning based on the strictness of deadline and estimated slack. The self-adjustment and monitoring of planning time is a unique feature of SARTS. For very loose deadlines and large slack, it behaves like A* and finds high quality solutions. For tight deadlines and small slack, it behaves like a greedy algorithm in the hope of reducing the planning time. SARTS also provides an infeasibility predicate, which is monitored continuously to predict possible deadline violation ahead of time. Experiments show that SARTS provides higher deadline compliance than a well-known real-time search algorithm. The infeasibility test is found to be reasonably accurate in the experiments for various deadlines and its effectiveness is related to the accuracy of the heuristic functions used.

References

1. M. R. Garey and D. S. Johnson, "Complexity Results for Multiprocessor Scheduling Under Resource Constraints," *SIAM Journal of Computing,* pp. 397-411,1975.
2. W. Zhao, K. Ramamritham, and J. A. Stankovic, "Preemptive Scheduling Under Time and Resource Constraints," *IEEE Transactions on Computers,* August 1987.
3. J. Gu, *Constraint-Based Search,* Cambridge University Press, New York, 1993.
4. R. E. Korf, "Real-Time Heuristic Search," *Artificial Intelligence Journal,* vol. 42, no. 2-3, pp. 197-221,1990.
5. L. C. Chu and B. W. Wah, "Optimization in Real Time," *IEEE Real-Time Systems Symposium,* 1991.
6. B. Hamidzadeh and S. Shekhar, "Can Real-Time Search Algorithms Meet Deadlines?," *Proc. of the Tenth National Conference on Artificial Intelligence,* AAAI, 1992.
7. B. Hamidzadeh and S. Shekhar, "Specification and Analysis of Real-Time Problem Solvers," *IEEE Transactions on Software Engineering,* vol. 19, no. 8, August, 1993.
8. S. Shekhar and B. Hamidzadeh, "Self-Adjusting Real-Time Search: A Summary of Results," *Proc. of IEEE Conference on Tools for Artificial Intelligence,* 1993.
9. S. Shekhar and S. Dutta, "Minimizing Response Times In Real Time Planning And Search," *Proceedings of 11th International Joint Conference on Artificial Intelligence,* pp. 238-242, IJCAI, 1989.
10. D. Gelperin, "On the Optimality of A*," *Artificial Intelligence,* vol. 8, pp. 69-76, Elsevier, 1977.
11. S. Shekhar and B. Hamidzadeh, "SARTS: A Dependable Real-Time Search Algorithm," *UMN Tech. Report CSci TR 93-23,* University of Minnesota, 1993.
12. R.E. Korf, "Real-Time Heuristic Search: First Results," *Proc. AAAI Conference,* 1987.
13. R.E. Korf, "Real-Time Heuristic Search: New Results," *Proc. AAAI Conference,* 1988.