Differential A^*

Karen I. Trovato and Leo Dorst

Abstract—A^* graph search effectively computes the optimal solution path from start nodes to goal nodes in a graph, using a heuristic function. In some applications, the graph may change slightly in the course of its use and the solution path then needs to be updated. Very often, the new solution will differ only slightly from the old. Rather than perform the full A^* on the new graph, we compute the necessary OPEN nodes from which the revised solution can be obtained by A^*. In this "Differential A^*" algorithm, the graph topology, transition costs, or start/goals may change simultaneously. We develop the algorithm and discuss when it gives an improvement over simply reapplying A^*. We briefly discuss an application to robot path planning in configuration space, where such graph changes naturally arise.

Index Terms—Graph search, uniform cost search, a-star search, A^* search, optimal navigation, incremental graph changes, dynamic scheduling, dynamic routing, robot path planning.

1 Introduction

In graph search, A^* is the classical uniform cost search that produces optimal paths between sets of start and goal nodes in an efficient manner and its use is widespread. When used for robot path planning [1], the graph to be searched is a representation of the states of the robot, where transitions indicate permitted motions with associated costs. When the "world" is known in advance, optimal plans can be generated easily. The goal of autonomous robotics, however, requires adapted (ideally, even optimal) plans whenever objects, opportunities, or rules change. Sensors enable frequent observation of the environment; therefore, in such graphs, small changes may occur frequently. These changes often have very localized effects on the graph, only occasionally affecting the path, yet traditional A^* would recompute even the unchanged portions. Observing this, we designed *Differential* A^* (∂A^*) [10], an algorithm which does only the *necessary* work to process the changes in the graph while allowing motion to proceed if the change does not cause an increase cost of the path. Because of its wider applicability, we describe it in this paper in a more general setting than the motivating application.

Differential A^* (∂A^*) is a general purpose exact graph-search algorithm that builds on A^* by managing topological and cost changes efficiently. Graph topology changes occur by creating or deleting nodes or transitions, which enables the dynamic construction of a graph (e.g., quadtrees). ∂A^* adapts search results, including all equivalent paths, between multiple sources and multiple destinations. Multiple heterogeneous node and edge changes are handled simultaneously. Differential A^* will always terminate for finite graphs and even for infinite graphs if a path exists [5] because it is merely a differential implementation of the basic A^* algorithm. Furthermore, it can handle transition costs of zero as well as nonnegative cycles. The topology ensures the permissible transitions. It is designed for general graphs, where adjacency is known, and is therefore not limited to any particular dimensionality. Compared to reapplying A^* on a changed graph, it is most beneficial when the graph is not fully connected or where the disparity in transition costs reduces the effective connectivity.

Differential A^* is useful for A^*-based applications calculating: vehicle navigation alternatives, plans in dynamically sensed environments, recovery motions for moving robots or cars [12], revised multiactor coordination plans [13], and many others. Game strategies may also be implemented, but only where adjacency can be defined, such as in the eight puzzle.

Planning methods tend to focus on efficient planning time or efficient execution time. The ∂A^* algorithm provides both and, moreover, guarantees optimal (execution) plans. In this arena, other change-based approaches are also found.

An incremental planning algorithm was proposed by Ramalingam and Reps [8] to address the related, single-source shortest-path problem assuming positive edge lengths. The method handles multiple heterogeneous edge changes simultaneously and is motivated by the context-free grammar (CFG) problem of Knuth [2]. The CFG rules provide a deterministic way to represent all possible "sentences" in a language. The grammar productions each define a value that is a function of the number of terminals and nonterminals, but also define the structure. The goal is to find the minimum-cost derivation (parse tree) for a terminal string. The algorithm is bounded as a function of the number of affected vertices; however, the number of vertices is not predicted in advance. The nature of the grammar structure does not require or handle multiple equivalent paths and does not utilize the opportunistic strategy of A^* with heuristics. This also causes the algorithm to focus on managing changes to costs, but not on changes to the structure.

- *K.I. Trovato is with Philips Research, 345 Scarborough Rd., Briarcliff Manor, NY 10510. E-mail: karen.trovato@philips.com.*
- *L. Dorst is with the University of Amsterdam, Kruislaan 403, NL-1098 SJ Amsterdam, The Netherlands. E-mail: leo@science.uva.nl.*

Manuscript received 19 Oct. 1998; revised 4 Dec. 2001; accepted 11 Apr. 2002.
For information on obtaining reprints of this article, please send e-mail to: tkde@computer.org, and reference IEEECS Log Number 108092.

Another incremental algorithm, D*, was provided by Stentz [9] to manage transition cost changes in a static, homogeneous 2D topology for robotics, with path-following embedded in the core method. Starting with a full or partially computed graph, incremental increases or decreases are propagated throughout the graph when changes occur. As might be expected, fewer "change" calculations are required when more of the space is precalculated. Unfortunately, the implementation of [9] does not detect when all paths between the start and goal are obstructed. In robotics, this is essential information and was a key trait of our earlier A^* work [1]. Finally, the method does not manage topological (create or delete) changes for nodes or transitions, essential for dynamic graphs.

The present paper is organized as follows: Since our Differential A^* algorithm uses A^* such that the OPEN nodes become both input and output parameters, it requires some new terms beyond the classical usage, which we introduce in Section 2. The novelty of the Differential A^* algorithm consists of careful processing of the graph's changes to determine their effect on OPEN so that the next invocation of A^* indeed produces the correct result. The case-by-case analysis for this "*Difference Engine*" forms Section 3, resulting in pseudocode for the ∂A^* algorithm. We attempt to give a characterization of the conditions under which the use of ∂A^* is beneficial in Section 4 and, in Section 5, we describe the salient elements of the robot path planning problem that was the original motivation for this work.

We assume a working knowledge of A^* as a prerequisite to reading this paper, but will briefly review and highlight the essential aspects and terminology used in our algorithm.

2 A^* Background

2.1 The Strategy

Classical A^* (as described by Pearl [7] and Nilsson [5], [6]) computes an optimal path between a start and goal for a cost-weighted, directed graph G. For efficiency, A^* uses an admissible heuristic h which guides the search without compromising optimality. The result of an A^* search is an "*explicated graph*" G_e in which pointers indicate transitions between nodes from the start to goal node. This explicated graph is then typically the input to a path following procedure. It is often convenient to see the execution of path following as a separate process since system design, communication, and control strategies can be extensive. Clearly, the capabilities for path following may affect the metric used for proper transitional cost-weighting. The assignment of proper (i.e., useful) weights will be assumed; in this paper, we focus on the search method itself.

The OPEN set is central to the internal functioning of A^* and can be managed efficiently by a heap. In principle, A^* could be stopped and restarted at any time as long as OPEN is preserved along with the part of the explicated graph which has already been computed. This is precisely what is needed for *differential* use. We now develop the terminology to allow an explicated graph and OPEN to be used explicitly.

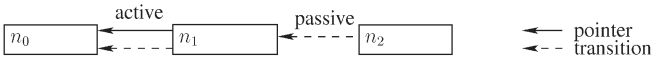


Fig. 1. Example for definitions (see text).

2.2 Graph G Terminology

Every directed edge in G from a node n_2 to a second node n_1 is called a *transition* $t(n_2, n_1)$. We use t_i to denote a particular transition. $c(n_2, n_1)$ represents the *cost* (e.g., time, distance, etc.) to make a transition *from* n_2 *to* n_1.

We make explicit a suggestion in Pearl [7] to manage multiple starts and goals using special nodes called "super-start σ" and "supergoal γ" to which the actual set of start and goal nodes are connected by transitions. They represent the nodes where the A* search begins and ends, respectively. A path is subsequently traced from an optimal goal node back to the associated starting node.

SCS(n) is the *successor* function of a given node n, returning all successors of n. Successor nodes have transitions *to* the node n in the directed graph G and can be considered adjacent to n. Lower-case scs(n) represents an individual, or typical, successor node of n. An scs(n) is a potential child of n (see below). By definition, SCS(super-start) is the set of start nodes.

PRED(n) is the *predecessor* function of a given node n, returning all predecessors of n. Predecessor nodes have transitions outgoing *from* node n and can be considered adjacent to n. Again, pred(n) represents an individual node and a pred(n) is potentially a parent of n (see below). We use pred(n) to represent an individual node of n, i.e., some typical element of PRED(n). We overload PRED() to operate on a *set* of nodes. By definition, PRED(supergoal) is the set of goal nodes. For a transition t, the expression pred(n) returns the node n to which the transition t points.

$h(n)$ is a consistent, admissible *heuristic* provided to guide A^* [5], [7]. It provides an estimate from the node n to the nearest goal in the goal set of γ. Therefore, γ is an implicitly understood argument of h. If $h = 0$, A^* becomes a uniform cost search.

2.3 Explicated Graph G_e Terminology

We observe that G_e is a subgraph of G, with additional attributes. To distinguish the two, we introduce a specific terminology for G_e in an extension of the standard terminology [7] to provide the additional precision required later in Section 3 to describe the Difference Engine of ∂A^*. This is illustrated in Fig. 1.

A *pointer* $p(n_1, n_0)$ from n_1 to n_0 in G_e is an *explicated transition* of G, i.e., an optimal transition from n_1 to n_0 realizing the smallest cost. We call the corresponding transition in G an *active transition* $t(n_1, n_0)$. A *passive transition* $t(n_2, n_1)$ of G is a transition that does*not* have a corresponding pointer in G_e from n_2 to n_1.

A *parent node* of n is a node of the explicated graph G_e that is pointed *to* by a given node n by a pointer out of n. PARENTS(n) is the set of all parents of n (which have all resulted in the same cost $g^*(n)$ and are therefore all on some optimal path through n). An individual parent node will be denoted as parent(n). Using the correspondence between nodes in G and G_e, we have PARENTS(n) $\subseteq$ PRED(n) since a parent is an explicated predecessor.

A *child node* of n is a node of the explicated graph G_e that points *into* a given node n. CHILDREN(n) is the set of all children of n. An individual child node will be denoted as child(n). Obviously, CHILDREN(n) $\subseteq$ SCS(n) since a child is an explicated successor.

The $g(n)$ of a node n is the calculated cost for transitioning between the node n and the superstart node. After a node has been fully treated (i.e., has been removed from OPEN), A^* will have produced the optimal value denoted by $g^*(n)$. If a node has not been treated, or has been "cleared," its value is denoted as ϕ. In relative cost, ϕ is always greater than any other value. Furthermore, a node with value ϕ has no parents. After having been fully treated by A^*, a node n has a value for g and PARENTS that achieve the values:

$$g(n) = \begin{cases} 0 & \text{if } n \text{ is a } \mathbf{start} \text{ node} \\ \min\{g(n')+c(n,n') \mid n' \in \mathbf{PRED}(n)\} & \text{for non}\mathbf{start} \text{ nodes.} \end{cases} \quad (1)$$

$$\mathbf{PARENTS}(n) = \{n' \mid g(n') + c(n,n') = g(n)\}. \quad (2)$$

The superstart node σ has no parents and, therefore, no pointers after A^*; it is assigned value 0 and $c(n,\sigma) = 0$.

The supergoal is assigned the minimum value of all goal nodes:

$$g(\gamma) \equiv \min\{g(\gamma_i) \mid \gamma_i \in \mathbf{PRED}(\gamma)\}.$$

This is taken to be ϕ when all goals have value ϕ. PRED(γ) represents the actual goal set, so the subset of PRED(γ) which realizes this minimum cost is denoted γ^*:

$$\gamma^* \equiv \{\gamma_i \in \mathbf{PRED}(\gamma) \mid g(\gamma_i) = g(\gamma)\}.$$

We use A^* to determine γ^* and then traverse the pointers encoding the optimal path from any node in γ^* to any of the corresponding start nodes realizing a path of length $g(\gamma)$, giving the equivalent shortest paths from the γ^* nodes to corresponding start nodes. If one wishes to identify paths connecting to all goals rather than the shortest path(s), then the A^* exit conditions using $g(\gamma)$ should be replaced with the equivalent derived from the maximum $g(\gamma_i)$ value. With either exit condition, A^* terminates for finite graphs. For infinite graphs, all required goals must be reachable from at least one start to terminate.

The order of node expansion in A^* requires the "*promise*" of a node, which is:

$$f(n) = g(n) + h(n). \quad (3)$$

2.4 A^* Pseudocode

There are only a few items to be clarified in the implementation of the A^* algorithm as we prepare for the ∂A^* algorithm.

The graphs G and G_e. G defines the topology, and is essential for all calculations. G_e is the explicated graph that is "erased" and subsequently recalculated by A^*. In ∂A^*, we *save* G_e and use it as input into ∂A^*. These graphs are so alike that they are sensibly stored within a single implementational structure of nodes and transitions by augmenting G with information concerning G_e. The parent nodes of G_e are defined through active transitions. The PARENTS may be implemented as a list or by ordinal bitwise indication stored within each node.

```
main( )
  for each σ_i ∈ SCS(σ) do g(σ_i) ← 0 end for
  add_to_OPEN(SCS(σ))
  // end of initialization
  if Astar(OPEN) ≥ g(γ)   // all minimum-cost goals reached
    γ* ← all nodes γ_i ∈ PRED(γ) such that g(γ_i) = g(γ)
    compute_path()
    exit('success')
  else
    exit('failure')

Astar(OPEN)
  // returns cost of last expanded node
  while OPEN is not empty
    n ← select_min_f(OPEN)
    if g(n) > g(γ)
      // finished, found all equivalent mimumum cost paths
      return(g(n))
    else
      expand(n)
  end while
  return(g(n))   // value of last expanded node

expand(n)
  for each n' ∈ SCS(n)
    g'(n') = g(n) + c(n', n)   // possible new value for n'
    if g'(n') < g(n')   // new value is lower
      g(n') ← g'(n'); f(n') ← g(n') + h(n')
      PARENTS(n') ← {n}; add_to_OPEN(n')
    else if g(n') = g'(n')
      PARENTS(n') ← PARENTS(n') ∪ {n}
    end if
  end for

compute_path()
  // note: technique may vary depending upon application
  follow pointers (PARENTS) iteratively
  from γ* to corresponding start node
```

Fig. 2. The A^* algorithm.

OPEN *at termination.* When A^* terminates, nodes in OPEN will remain there. They will be augmented by ∂A^*.

Initialized (or cleared) node. This node has no calculated f, g, or h value and, so, is defined: $g(n) \leftarrow \phi$ (implicitly, $f(n) \leftarrow \phi$) and PARENTS(n) $\leftarrow$ { }. The set of nodes with cost ϕ (i.e., no cost) is denoted by Φ. (Mnemonic: both symbols resemble the empty set symbol $\emptyset$.) In any comparison with a finite cost g, $\phi > g$ holds (so that ϕ acts like an infinite cost).

The *implementation* of the A^* algorithm we use is slightly modified from Pearl [7], see Fig. 2. Optional additional modifications can further improve efficiency, see [10, p. 18].

3 Differential A^* (∂A^*)

The ∂A^* algorithm is designed to produce a new G_e by running A^* on a properly modified graph G'_e and a revised set OPEN', as shown in Fig. 3. These two are produced by the Difference Engine using the explicated graph G_e and OPEN from the previous A^* and the changes to G.

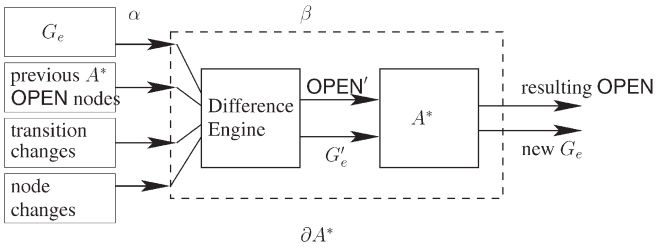


Fig. 3. Differential A^* consists of a Difference Engine followed by A^*.

TABLE 1
Cases of the Difference Engine

	transition		node		
	passive	active	typical	start	goal
increase	Case 3	Case 1			
decrease	Case 7	Case 5			
create	Case 8	Case 6	Case 9	Case 11	Case 13
delete	Case 4	Case 2	Case 10	Case 12	Case 14

3.1 Fundamental Operations of the Difference Engine

The main concern of ∂A^* is producing the revised OPEN set. These OPEN nodes are required as initialization to construct the correct new G_e from a properly prepared old G_e. It cannot always be decided immediately upon treatment whether a node is to be one of the OPEN nodes, so we collect *candidate* OPEN *nodes* in a set CAND. We process CAND in the Difference Engine (Section 3.3) to generate the actual set of OPEN nodes for the next A^*.

The fundamental operations in the Difference Engine are now the following:

- add_to_candidates(n): The set of candidates CAND is a repository for all candidate nodes that will be reviewed as possible OPEN nodes for the next A^*. The implementation is typically a list or array structure.
- The "*clear influence*" function, clear(n), is the key function in the Difference Engine. Given the changes, all nodes that will have an increased cost must be identified so that they may be recomputed with A^*. It is therefore the transitive closure of the CHILDREN function at n, but including n as well. The clear(n) function pinpoints these nodes by adding n and, recursively, all children of n into Φ, resetting them to $g(n) = \phi$. In addition, it finds the "forward perimeter" of the cleared nodes and adds this to the candidates. This *forward perimeter* PERIM(Z) of a set of nodes Z consists of the nodes that border on Z (not in Z) such that there is a transition from some Z-node to p. So, PERIM(Z) $\equiv$ PERIM(Z)/Φ. The resulting non-ϕ valued perimeter nodes are ultimately used as a source via OPEN for the subsequent A^* search. The function is specified in the pseudocode of Fig. 11.
- recompute_OPEN(): This function is used to update the value $h(n)$ of all nodes n in the OPEN set and, therefore, its promise function $f(n)$. The $g(n)$ value is not changed.

3.2 Case Analysis of the Difference Engine

Transition and node changes are the primary types of change to the graph G. Table 1 identifies the possibilities as cases for analysis. We can create or delete nodes or transitions to redefine the topology of the graph. We can change the cost of passive or active transitions. Finally, we can add or remove the identification of a node as start or goal. Each case is analyzed to identify the difference calculations required to recompute the graph G_e.

Each illustrated case will show the graphs G and G_e at time α before the changes and then at time β after the difference function; these will be denoted as G_α and G_β, respectively. G and G_e are sketched in the same figure, *passive transitions* are denoted by dashed arrows, and *active transitions (pointers)* by solid arrows. Costs for transitions will be preceded by a colon, i.e., $(t_1 : b > a)$. There is typically a node n or transition t_1 which is the focus of the case.

In all diagrams, nodes are labeled with their strongest relationship to the node (or transition) of interest. So, parent(n) is a typical instance of a parent of n (and, therefore, also a predecessor), but pred(n) is a typical predecessor of n, but *not* a parent. If a node is *not* related to n, it is denoted by n_i. Actions performed on a typical node are assumed to be taken for all elements in its class, e.g., treatment of parent(n) must be repeated for all elements of PARENTS(n). Finally, we only include the salient analysis here, a more exhaustive treatment is available in [14].

Case 1: Increase active transition cost. In this case (Fig. 4), the value of the active transition t_1 is increased. In G_α, p_1 indicates that n optimally points to parent(n). In G_β, the pointer of n may now be impacted because another node in PRED(n) may be the (optimal) new parent of n. Not only is the pointer affected, but the cost will now be increased (unless the pointer happens to result in an equivalent cost via the new parent). The recomputation to find the new parent of n can be achieved by examining all of the predecessors of n, determining the minimum $g(n)$, and assigning the corresponding pointer to the parent that produces the lowest $g(n)$. The predecessor computation and pointer revision must then be followed by the recomputation of all children of n, which must also be updated. Although we could perform two steps to recompute n and then clear(CHILDREN(n)), for algorithmic simplicity we choose to use clear(n), which is nearly as efficient. Subsequent recalculation of the cleared nodes occurs during the next A^*.

Required difference calculations: clear(n).

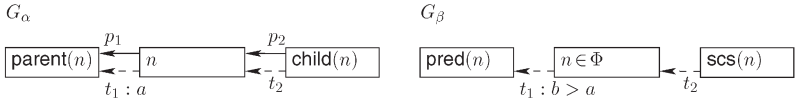


Fig. 4. Case 1.

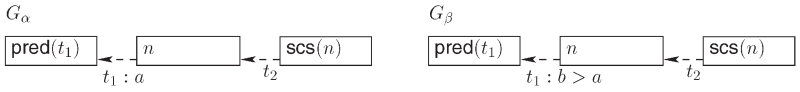


Fig. 5. Case 3.

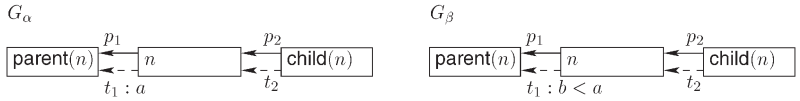


Fig. 6. Case 5.

Case 2: Delete active transition. This case is analogous to Case 1, except that t_1 is now deleted. As in Case 1, clear(n) will ensure that the children of n are set to ϕ, and all PRED(n) are added to CAND.

Required difference calculations: clear(n).

Case 3: Increase passive transition cost. In this case (Fig. 5), the value of t_1 is increased, but, in G_α, there is no pointer indicating that n optimally points to this predecessor of n, which means that pred(t_1) is not a parent of n. The transition t_1 is not in the explicated graph G_e and, therefore, increasing its cost will have no effect on the explicated graph (it would only make reaching n via this transition even more expensive). Therefore, there are no fundamental operations required.

Required difference calculations: None.

Case 4: Delete passive transition. This case is analogous to Case 3, except that t_1 is now deleted.

Required difference calculations: None.

Case 5: Decrease active transition cost. In this case (Fig. 6), the value of transition t_1 is decreased. In G_α, p_1 indicates that n optimally points to parent(n). In G_β, the optimality will not change the existence of pointer p_1 because t_1 is even more desirable than before. Because this transition can only affect the node n, followed by its children, it is sufficient to recompute n using the new lower cost of t_1 and add n to the set of candidates. The subsequent invocation of A^* will manage the propagation of this lower cost through to any children and, possibly, other nodes in the graph. So, we could recompute this individual node and then add it to the list of candidates; however (as in Case 1), we prefer to add_to_candidates (pred(t)) and allow the next A^* to provide the next recomputation of n.

Required difference calculations: add_to_candidates (pred(t)).

Case 6: Create an active transition. This case cannot occur since it attempts to change G_e directly. We mention it for completeness.

Required difference calculations: None.

Case 7: Decrease passive transition cost. This case is analogous to Case 5, except that the transition has no corresponding p_1 pointer. The cost change may trigger t_1 to become active, but this occurs automatically after invoking A^*.

Required difference calculations: add_to_candidates (pred(t)).

Case 8: Create a passive transition. This case is analogous to Cases 5 and 7 since it adds a new transition at a lower cost (lower than infinity), with no corresponding p_1.

Required difference calculations: add_to_candidates (pred(t)).

Case 9: Create a node n (Fig. 7). Creation of a node may be a frequent and fundamental topological change; however, it is actually a combination of cases. Although optional, we treat it as a separate case because it results in a difference calculation having greater overall efficiency.

Inserting a node has three steps. First, the node n must be created with a default (e.g., ϕ) value. Second, any predecessor and successor transitions such as t_1 and t_2 must be created. Third, n will be recalculated by A^* if we add PRED(n) to CAND. If each successor transition such as t_2 is added as a separate action, then n would be added to CAND many times.

We also observe that, during A^*, pred(n) calculates n, which, in turn, calculates scs(n). (If there are no predecessors of n and only successors, then n cannot be part of a path in G_e.) Treatment of PRED(n) will then automatically treat the successors. Therefore, treatment of the node and its associated transitions as a single unit rather than by separate cases is computationally more efficient because it removes redundant and overlapping actions.

Required difference calculations: add_to_candidates (pred(t)).

Case 10: Delete a node n (Fig. 8). Deleting a node may also be a frequent topological event. Deleting a node has two steps. The first step is deleting the active and passive transitions leading into and out of n. These four situations are shown in the diagram above. The second step is to delete the node itself. Because these elements may affect one another, each will be examined with an eye toward the consequences.

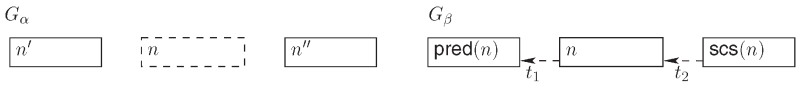


Fig. 7. Case 9.

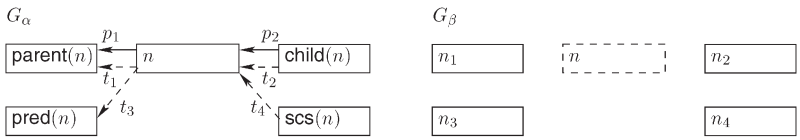


Fig. 8. Case 10.

Transition t_2 is an active transition with the associated pointer p_2 that leads from n to one of its children. The result of deleting the transition according to Case 2 means that PRED(child(n)), in the new situation G_β where $n \notin$ PRED(child(n)), will be added to the perimeter when clear(child(n)) is performed. This will ensure that another parent (if possible) will be found for all CHILDREN(n).

No action is required to delete the passive transitions t_3 and t_4 according to Case 4.

Deleting active transition t_1 according to Case 2 is that clear(n) will clear n and the recursive children of n. Note that, since the clear function uses both the successor and predecessor functions, the clear(n) function must be computed before n is deleted.

Even though these four situations could be managed separately by ensuring the proper order of computation, two improvements can be obtained by the same recommended change. Removal of t_2 and of t_1 contains duplicate activities because clear(n) also clears the children, including t_2. Taking these together, a more efficient overall solution is to first complete clear(n). Subsequently, because PRED(n) and CHILDREN(n) must remain valid during the clearing process, n and its related transitions may be deleted.

Required difference calculations: clear(n). Note: must delete node and transitions *after* clear as part of graph management.

Case 11: Create a start. In this case (Fig. 9), a node is converted in status to one of the start nodes. If the node does not exist, then it must first be created, as in Case 9, but this is an independent action. Topologically, we do not wish to disturb the relationships that already exist for node n, so we need only add a new transition t_0 between n and the superstart σ having cost 0. This, in turn, changes n to a starting node and successor of superstart σ. Similarly to Case 8 where we create a passive transition, the only action that is needed is to add the predecessor (i.e., superstart σ) to CAND. The transition t_0 must be created prior to invoking the pred(t_0) function.

Required difference calculations: add_to_candidates (pred(t)) = add_to_candidates (σ).

Case 12: Delete a start. When a start node is no longer in use, we remove the active transition to the superstart σ (t_0 of Case 11: G_β). As in Case 2, clear(n) is performed and *then* the transition is deleted.

Required difference calculations: clear(n) prior to removing the transition to superstart σ.

Case 13: Create a goal. The status of "goal" is managed by adding a transition from the supergoal(γ) to n, analogous to Case 11. This would normally mean that n would be added to candidates and then to OPEN. Because PRED(n) must be expanded in A^* prior to reaching n, this step becomes unnecessary. A change in the set of goal nodes (PRED(γ), will also change the value of the heuristic function $h(n)$ at every node, recalling that γ is an implicitly understood argument of h. If the (new) goal node is already treated, then the pointers in G_e will still be valid and A^* processing will be trivial because the terminating condition will be satisfied immediately. Unless the immediate termination is the final use of this graph, the values of $h(n)$ and dependent $f(n)$ must be recomputed for all nodes in OPEN.

Required difference calculations: recompute_OPEN().

Case 14: Delete a goal. The "goal" status of a node n is removed by deleting the transition between n and γ. If the transition is passive, no operations are required (Case 4). If it is active (Case 2), clear(n) will attempt to clear(γ), which is automatically defined. Therefore, clear(n) is unnecessary except to observe that γ will change. This may cause the heuristic to change; so, as in Case 13, OPEN must be adjusted in accordance with the new h value.

Required difference calculations: recompute_OPEN().

3.3 Differential A^* (∂A^*) Algorithm

In this section, we collect the cases into related groups, discuss ordering, list assumed constructs for the implementation and propose pseudocode to implement Differential $A^*(\partial A^*)$.

The cases are summarized in Table 2, grouped by related actions. The table shows the essence of the Difference Engine, but without an ordering for the Differential A^* algorithm. We now discuss the groups regarding efficiency and accuracy to arrive at a proposed ordering.

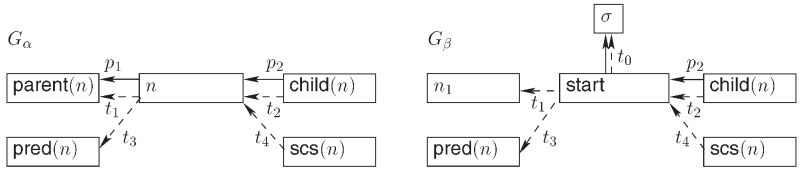


Fig. 9. Case 11.

TABLE 2
Grouped Differential A^* Cases

Case	Description	recompute	add_to_ candidates	clear
	Group A			n
1	Increase cost of an active transition			
2	Delete an active transition			
10	Delete a node n			
12	Delete a start			
	Group B	none		
3	Increase cost of a passive transition			
4	Delete a passive transition			
6	Create an active transition			
	Group C		pred(t)	
5	Decrease cost of an active transition			
7	Decrease cost of a passive transition			
8	Create a passive transition			
11	Create a start			
9	Create a node n		PRED(n)	
	Group D	OPEN		
13	Create a goal			
14	Delete a goal			

Within each group, difference calculations may be carried out in any order without any undesirable effect on *accuracy*. However, *efficiency* may vary. Observe that, in group A, the clear(n) function may clear a node only once, so it is bounded by $O(n)$. In group C, we see that the number of nodes resulting from the predecessor function will return the same number of nodes, regardless of the ordering. In group D, we also observe that a goal change affects the heuristic and, thus, the promise values (f) in OPEN.

Ideally, OPEN should be calculated when the size of OPEN is a minimum. This is entirely dependent upon the changes to the graph. Group A may increase or reduce the number of nodes in OPEN because clear(n) not only increases OPEN by the number of perimeter nodes, but also reduces the size of the previous OPEN by setting nodes to ϕ. Therefore, whether group D should precede group A is application and change specific. Group C will only serve to add nodes to CAND; therefore, group D should precede group C. Finally, the ordering of group A and C does not affect the total number of nodes in CAND.

As discussed in the cases, we must create a node or transition *before* any difference calculations occur. Similarly, deleting a node or transition must be done *after* the calculations.

We also observe that efficiencies can be improved by preprocessing a list of changes. Depending upon the assumptions about how changes are identified (e.g., chronologically), some obvious simplifications can be made. So, for example, if a transition or node is to be created and then deleted, these can be cancelled. Multiple cost changes of the same transition may be assumed to take the final cost. These preprocessing steps may be application specific, however, and are not assumed as part of the ∂A^* algorithm. Once the difference operations are complete, however, OPEN is generated by removing redundancies and ϕ values from OPEN and CAND.

For the basis of the algorithm, we assume the structure for the nodes and changes as described below. We also list efficient structures, where known. Each node contains:

- *Current values of* $g(n)$ *and* $h(n)$. $f(n)$ may be stored or calculated as needed. They may be referenced from OPEN.
- *List of* PARENTS(). Bitwise flags may manage low numbers (e.g., 32) of ordered (predictable) neighbors.
- *List of predecessors and successors.* Adjacency may be implied (calculable) based on known topology to save space, for instance, when G represents neighbors on a grid.
- *Transition cost* $c(n, n')$ represents the cost of the transition *from* n *to* n'. This value may be calculated on demand or precomputed.
- "*in*-OPEN *indicator*." A speed improvement can be achieved by simply adding a flag to each node, which is checked before adding the node to OPEN. This ensures only one copy of the node enters OPEN.

Moreover, *transition changes* (from node a to node b) are defined as a list of: (a, b, new_value), (a, b, delete), or (a, b, create); and node changes are defined as a list of (n, create, pred_list, scs_list), (n, delete), (n, create_start), (n, delete_start), (n, create_goal), (n, delete_goal). We propose pseudocode to implement the Differential A^* algorithm ∂A^* in Figs. 10 and 11.

4 Usefulness of Differential A^*

Functionally, both A^* and Differential A^* achieve the same results; they compute G_e, but with different levels of efficiency. Obviously, the issue is: "When is Differential A^* preferable to A^*?" The relative efficiencies are addressed in this section.

As we have seen, in Differential A^*, there are two general steps. The first is the use of the Difference Engine to

```
Differential_Astar (G_e, OPEN, list of transition changes T, list of node changes N)
  CAND ← OPEN
  for each transition (a,b) in T with associated cost new_value
    if action = create
      create_transition(a, b, new_value)
    else if is_active(a, b)   // note: from a to b (b is parent)
        if new_value > c(a, b)   // case 1
          clear(a)
        if action = delete   // case 2
          clear(a); delete_transition(a, b)
        if new_value < c(a, b)   // case 5
          add_to_candidates(b)   // predecessor of transition
    else   // passive transition
      if new_value < c(a, b) or action = create   // cases 7 and 8
        add_to_candidates(b)   // predecessor of transition
  for each node n in N
    switch action
    case create   // case 9
      create_node(n, pred_list, scs_list)
      add_to_candidates(PRED(n))   // all predecessors
    case delete   // case 10
      clear(n); delete_node(n)
    case create_start   // case 11
      create_transition(n, σ, 0)   // cost of n will become zero
      add_to_candidates(σ)   // σ = pred(n)
    case delete_start   // case 12
      clear(n); delete_transition(n, σ)
    case create_goal   // case 13
      create_transition(γ, n, 0)
      goal_change = true   // see below
    case delete_goal   // case 14
      delete_transition(γ, n)
      goal_change = true   // see below
  for each n in CAND
    if g(n) ≠ φ and n ∉ OPEN   // remove nodes cleared by parallel changes
      add_to_OPEN(n)
  if goal_change   // performed once, handles simultaneous goal changes
    recompute_OPEN( )
  Astar(OPEN)   // see Figure 2
```

Fig. 10. Pseudocode of the ∂A^* algorithm, part 1.

calculate the difference changes, generating a revised OPEN and G_e (depicted as OPEN′ and G'_e in Fig. 3) for the subsequent A^*. The second is to compute A^* beginning with those changed nodes rather than from a starting state. We analyze the fundamental difference calculations and, then, assess ∂A^* versus A^* for each of the groups of changes.

The time to perform A^*. It is well-known that, on a graph of N nodes, A^* takes $O(N \log N)$ time [5], [7]. This formula neglects details of the structure of the graph, such as its "branching factor" (the typical number of successors m of a node), which affects the growth of the OPEN set and, thus, determines the time to complete graph computation. Those details are taken into account in the order computation in [14], but they need not concern us here. For the general comparison between A^* and ∂A^*, the $O(N \log N)$ formula is sufficient.

The time to recompute_OPEN(). To perform the recompute_OPEN() function (required in Cases 13 and 14 of Group D), each node in OPEN must be recomputed. In the worst case, most of the nodes are in OPEN, such as if the graph is fully connected. In this case, the value of h needs to be revised for each node, requiring $O(n)$ operations. Furthermore, the heap needs to be revised based on the changed $f\ (= g + h)$ values, with a worst case of $O(N \log N)$. Because multiple goal changes may be requested simultaneously, there may be a strong effect on the heuristic function h. We also wish to perform such a large recalculation only once after all goal changes are known. So, although each of N nodes may be computed in $O(1)$, OPEN can be fully reconstructed in $O(N \log N)$.

The time to add_to_candidates. Adding to the candidates is required in the cases of Group C and is performed in constant time or $O(1)$. Adding a node to a list (which is how we prefer to implement the set CAND) can be performed in fixed time, but, for N candidates, we require $O(N)$ operations total.

The time to clear. To perform clear() in G_e, which occurs in the cases in Group A, all nodes Z recursively dependent on a given node are reset to ϕ and, then, the perimeter PERIM(Z) = PRED(Z)/Φ is added to the candidates. In

```
clear (n)
  g(n) ← φ
  for each n' in SCS(n)
    if is_active(n', n)   // is n' child of n?
      clear(n')
  PARENTS(n) ← {}   // clear pointers
  for each n' in PRED(n)   // n' adjacent to cleared area
    if g(n') ≠ φ
      add_to_candidates(n')   // perimeter added to CAND nodes

recompute_OPEN( )
  for each n in OPEN
    compute_h_and_f(n)

delete_transition(a, b)   // from a to b
  PRED(a) ← PRED(a) − {b}; SCS(b) ← SCS(b) − {a}

create_transition(a, b, new_value)   // from a to b
  PRED(a) ← PRED(a) ∪ {b}; SCS(b) ← SCS(b) ∪ {a}
  c(a, b) ← new_value

delete_node(n)
  for each n' in SCS(n)
    PRED(n') ← PRED(n') − {n}
    PARENTS(n') ← PARENTS(n') − {n}
  for each n' in PRED(n)
    SCS(n') ← SCS(n') − {n}
  free(n)

create_node(n, pred_list, scs_list)
  new(n)   // create the node
  g(n) ← φ;  h(n) ← φ;  f(n) ← φ; PARENTS(n) = { }
  for each n' in pred_list
    create_transition(n, n', new_value)
  for each n' in scs_list
    create_transition(n', n, new_value)

is_active(a, b)   // transition from a to b
  if b in PARENTS(a) return true else return false
```

Fig. 11. Pseudocode of the ∂A^* algorithm, part 2.

advance, it is difficult to estimate the number of nodes that will be affected by the clear function. The best case is that there are *no* successors to a node. The worst case, which may occur, but only in a rather pathological graph G, is when *all* nodes are (recursively) dependent on a given node, requiring $O(N)$ operations. Typically, however, only a small portion of the graph is affected. The clear function also finds the perimeter by using the predecessor function performed on each of the final set of cleared nodes.

With these complexities of main functions, we can determine for each group of Table 2 whether it is better to use Differential A^* or whether it is better to fully recompute the graph using A^*.

4.1 Group A

Group A consists of nodes and transitions that cause a ripple effect of increased costs to the node and its children. Because these are increased costs, the affected area must be cleared and recomputed since A^* can only serve to lower node costs. The clearing of the affected area is achieved with clear(). If this is the precise region that is required to complete the next A^* computation, then the primary overhead for Differential A^* is the time required for clear(), which involves setting up the perimeter. Often, the cleared set of nodes Φ represents only a portion of the given graph. Also, note that there may be nodes with cost ϕ that are not counted in Φ, such as those that have never had an assigned cost due to disconnects in the graph or, by chance, they occupy an unused area of the graph between any start and goal. Compared to complete recalculation with A^*, there will likely be some number of nodes that are saved from recalculation. The number of these nodes is $(N - |\Phi|)$ if the graph is connected and fully treated (e.g., when $h = 0$). When the overhead for clearing the nodes using clear() is less than recomputing the potentially saved nodes, then Differential A^* will be more effective. It is not guaranteed, however, that the cleared portion is fully recomputed nor that it is the only area affected by the change. For example, if we remove the last transition connecting one part of the graph to another, then the later graph will be cleared but never recalculated by A^*. The cleared region may be large, but the actual repropagation area is negligible. This is an extreme case. It is also possible that, in the same computation, some transitions will be added while others are removed. In this event, it is possible for the cleared nodes to be few, but there may be a major recalculation for A^*.

In this group of changes, the number of nodes cleared will vary according to the specific characteristics of the graph; therefore, the ability to estimate the number of cleared nodes for particular situations and some knowledge of the frequency that they occur can help predict whether Differential A^* or A^* is preferable. This can be difficult, but, for some applications, it may be estimated [14].

4.2 Group B

Group B consists of transitions that have no immediate impact on the graph G_e. The greatest gain in using Differential A^* occurs in the situations when the only changes are in Group B transitions. This is because the transitions have no effect on the graph G_e; therefore, no computations are required. If this was not identified, then these changes would ordinarily cause the complete and unnecessary recomputation of the graph.

In this group, Differential A^* is *always* more effective because it saves all A^* operations.

4.3 Group C

Group C consists of nodes and transitions that cause decreased costs. It consists of the addition of nodes or transitions, which change the topology and reduce costs. Nodes S between the start and additions are saved from recomputation. Nodes between the additions and the nearest goals (*distal* to the start) would have to be recomputed, even with A^*. Depending upon the relative positions, varying amounts can be saved. The group also includes the decrease in cost of an active or passive transition. Similarly to above, ∂A^* also saves the computation of nodes between the start and the transition, but distal nodes have to be recomputed. However, if the costs are not sufficiently low to force the propagation of new costs through to many children, then this change can save all

operations as it does in Group B. Even when cost propagation is required, then the *overhead* for using Differential A^* is essentially zero. Although conversion of a node to a start is similar, the number of nodes between "the nearest start" and the new start node, means $S = 0$ nodes in savings, with all distal nodes being recomputed. The savings occur in the number of nodes S that were saved from unnecessary recomputation. According to Section 4, this would have taken $O(S \log S)$ time.

In this group, Differential A^* is *at least as effective* as A^* because it saves all $O(S \log S)$ operations. Therefore, in this group, ∂A^* is *always preferable* to A^*.

4.4 Group D

Group D contains cases that change the goal set and, thus, affect the heuristic function. This must be recomputed after ∂A^* *once* and just before calling A^*. The time required depends on the size of the OPEN set, but may be $O(N \log N)$. The savings may be complete, such as if $h = 0$ (thus, independent of the goals), but more likely the savings is the difference between the size of the heap and the number of nodes in the full expansion between start and goal. In the worst case, for goal change only, all prior calculations may have led the search in the opposite direction from the new goal set. In this case, the recalculation of OPEN is not productive and the number of nodes managed in OPEN is unnecessarily large. Even still, the inefficiency is bounded by |OPEN|, which is, at most, $O(N)$. Characterizing the expected amount of change for Group D requires close analysis of the particular graph and its changes.

In this group, as in Group A, no general statement of Differential A^* versus A^* effectiveness can be made, although specific situations show clear benefit or bounded inefficiency.

4.5 Summary of the Groups

In summary, it is clear that Differential A^* is guaranteed to provide a dramatic improvement over A^* for Group B and C changes since there is virtually no overhead and fewer nodes must be recomputed than if the complete graph is recomputed. Group A, however, requires overhead for the clear function to identify and clear dependent (successor) nodes and find the perimeter from which to begin A^* so that small changes can be handled to advantage. Group D manages changes to the goals, having a worst-case overhead of recomputation and reorganizing the number of nodes on OPEN. The key Differential A^* computation time is then the sum of the time to reinitialize nodes with clear and to find the forward perimeter. These times depend rather strongly on the topology of the graphs G and G_e and it is therefore hard to draw general conclusions.

We illustrate the search with a robotics problem in the next section to convey the flavor of the considerations. The conclusion based on that application suggests that, in practice, ∂A^* is almost always preferable to full A^*.

5 Example: Differential A^* for a Robot Arm

Fig. 12 shows an example of the use of ∂A^* in the path planning for a simple anthropomorphic robot arm. Full details may be found in [1], [14], [11]. In brief, the left column shows an arm with two degrees of freedom, the shoulder angle θ_1, and the elbow angle θ_2, with equal length for the two limbs. This particular robot [4] is fully revolute in both joints as they are coplanar. The right column shows its configuration space [3] graph G, depicted as the grid of a 64 by 64 discretized space. Each node is characterized as (θ_1, θ_2) (with a resolution of 5.625 degrees, and has 16 neighbors (four straight, four diagonal, and eight at knight's-move positions). The cost of the transitions between these neighbors is computed to correspond to the distance the end effector of the robot travels:

$$c((\theta_1, \theta_2), (\theta_1 + \Delta\theta_1, \theta_2 + \Delta\theta_2)) = \sqrt{(\Delta\theta_1)^2 + (\Delta\theta_2)^2 + 2(\Delta\theta_1)(\Delta\theta_2)\cos(\theta_1 - \theta_2)}.$$

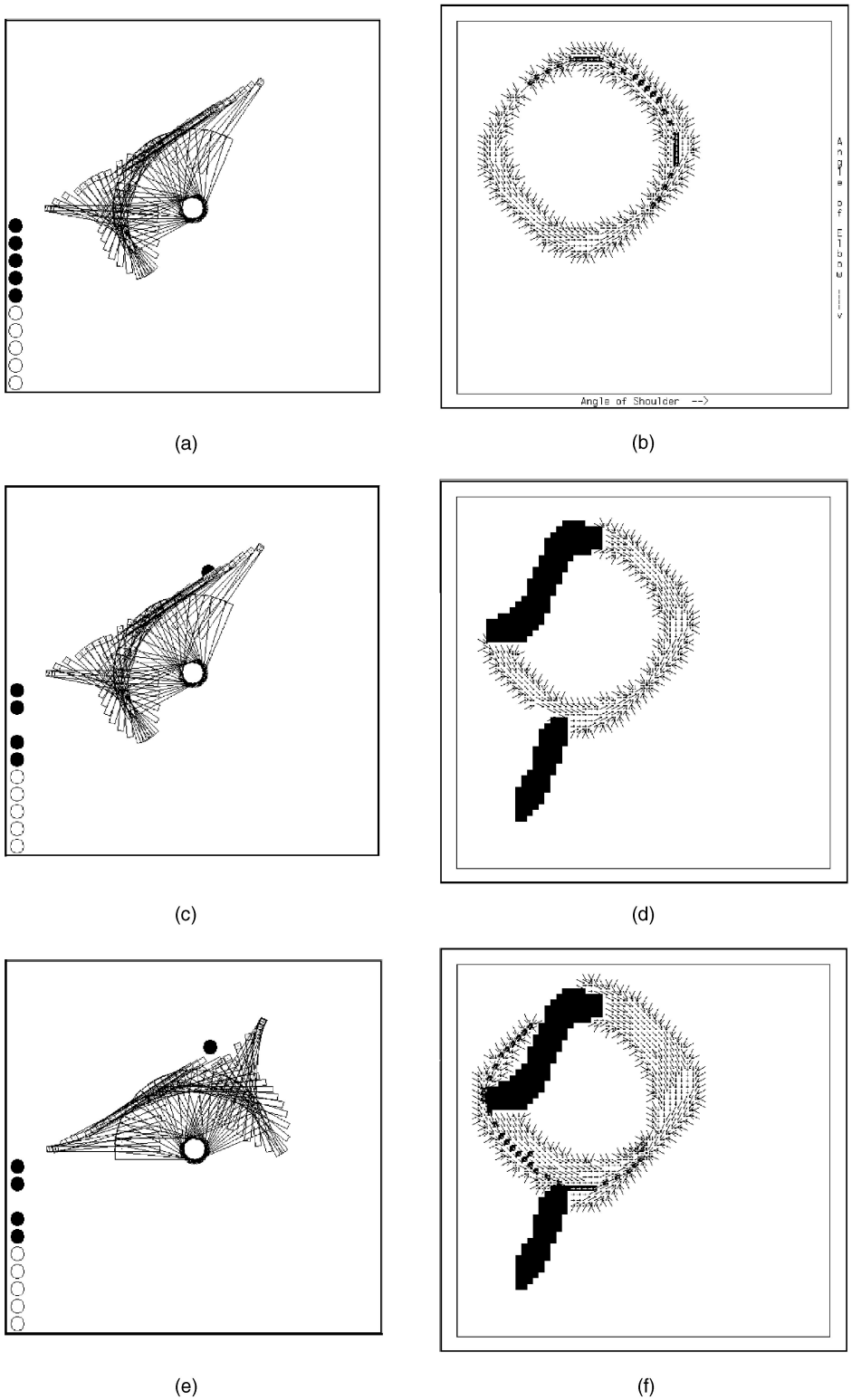
The heuristic h is from a node $n = (\theta_1, \theta_2)$ to a single goal node $\gamma = (\gamma_1, \gamma_2)$. We use the Euclidean distance of the end effector in both states:

$$h((\theta_1, \theta_2)) = \sqrt{(\cos\gamma_1 + \cos\gamma_2 - \cos\theta_1 - \cos\theta_2)^2 + (\sin\gamma_1 + \sin\gamma_2 - \sin\theta_1 - \sin\theta_2)^2}.$$

As in some search problems, this robotics problem benefits from searching from the *goal* toward the *start*. This is because the calculations resulting from the search produce a gradient field of arrows (pointers to parents) that always aim toward the goal. Since robots occasionally fall "off track" due to control problems or other interference, path following may resume from wherever a state has a valid pointer. Other than a terminology change and the benefit in path following, the ∂A^* and A^* algorithms are unchanged. With this new definition, the start $\sigma = (180^o, 180^o)$ leads to the goal $\gamma \approx (130^o, 110^o)$ by the pattern of pointers (G_e) depicted in Fig. 12b. The corresponding optimal path, shown in dark, is animated in Fig. 12a. The figure illustrates the status of the graph at the moment the search has completed, showing fully treated (what Pearl would call CLOSED) nodes as well as those still OPEN. The set Φ is indicated by white nodes. OPEN consists of the edge nodes along the inside and outside of the circular shape. Although all of the pointers in the path have their final values, the arrows along the "fringe" do not.

In Fig. 12c, a newly sensed obstacle is depicted in black. On the right in Fig. 12d, the corresponding forbidden states are indicated in black. Each forbidden state forces an infinite transition cost from any node into the state or may be considered as deleted transitions. Our representation uses the first form. The effect of the changes computed by the Difference Engine leads to a reduced set of pointers: Nodes "severed" from the original G_e have been cleared (reset to ϕ). The perimeter consists of the the nodes at the front "interface" of the obstacle, which is then added to OPEN. Once A^* is reapplied, G'_e results in Fig. 12f, including the new pointers and animated path in Fig. 12e. Note that the search area in Fig. 12f has "thickened" since the length of the path has increased due to the obstacle.

For this robotics example, the graph contains 4,096 (64 by 64) nodes. We analyzed the performance of A^* (on a Sun Sparc IPX) with $h = 0$ and the start and goal at 180 degree



(a) (b) (c) (d) (e) (f)

Fig. 12. Differential A^* for a robot arm.

opposite locations, finding the full computation of the graph to take 250ms. In Fig. 13, we show the time required for each component of a recalculation based on the number of nodes cleared (Group A).

This data suggests that, for these kinds of graphs and situations, as long as less than half of the number of nodes in G are cleared and recomputed, ∂A^* is preferable to A^*. Thus, our motivating application benefits from ∂A^*.

6 Conclusion

When a graph G changes, a portion of the explicated graph G_e may change and, hence, paths based on it. The objective

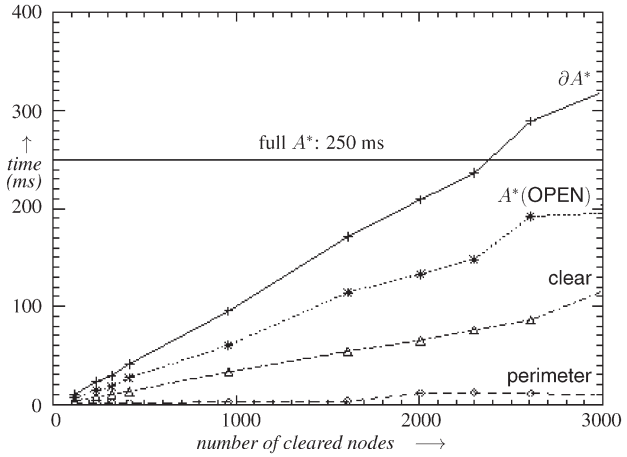


Fig. 13. The timing of differential A^* versus A^*.

of ∂A^* is to identify the portions that change and recompute only these areas.

We have provided a case-by-case derivation of ∂A^*, an algorithm that contains a Difference Engine which transforms the proposed changes to a graph G into a revised OPEN′ and G'_e (as in Fig. 3). They are used by A^* to complete the calculations. The resulting G_e is identical to what A^* would have produced in the original fashion.

The results are identical, but ∂A^* often achieves them much more efficiently. Although it was not possible to state, in general, that it is always better than A^*, we found groups of cases in which it is definitely more efficient than A^* and have found that, in most of our applications to robotics, it is a considerable improvement. Thus, ∂A^* can be used as a real-time, dynamic replanning tool in cases where A^* alone was previously considered too time-consuming.

REFERENCES

[1] L. Dorst and K. I. Trovato, "Optimal Path Planning by Cost Wave Propagation in Metric Configuration Space," *SPIE Advances in Intelligent Robotics Systems 1007,* pp. 186-197, Nov. 1988.
[2] D.E. Knuth, "A Generalization of Dijkstra's Algorithm," *Information Processing Letters,* vol. 6, no. 1, pp. 1-5, 1977.
[3] T. Lozano-Pérez and M.A. Wesley, "An Algorithm for Planning Collision-Free Paths Among Polyhedral Obstacles," *Comm. ACM,* no. 10, pp. 560-570, 1979.
[4] W. Newman, "High-Speed Robot Control in Complex Environments," PhD thesis, Massechusetts Inst. of Technology, Oct. 1987.
[5] N.J. Nilsson, *Problem Solving Methods in Artificial Intelligence.* McGraw-Hill, 1971.
[6] N. J. Nilsson, *Principles of Artificial Intelligence.* Tioga, 1980.
[7] J. Pearl, *Heuristics: Intelligent Search Strategies for Computer Problem Solving.* Reading Mass.: Addison-Wesley, 1984.
[8] G. Ramalingam and T. Reps, "An Incremental Algorithm for a Generalization of the Shortest-Path Problem," Technical Report 1087, Univ. of Wisconsin, May 1992.
[9] A. Stentz, "The Focussed D* Algorithm for Real-Time Replanning," *Proc. Int'l Joint Conf. Artificial Intelligence,* Aug. 1995.
[10] K.I. Trovato, "Differential A^*: An Adaptive Search Method Illustrated with Robot Path Planning for Moving Obstacles and Goals and an Uncertain Environment," *Int'l J. Pattern Recognition and Artificial Intelligence,* vol. 4, no. 2, 1990.
[11] K.I. Trovato and L. Dorst, *Differential Budding: Method and Apparatus for Path Planning with Moving Obstacles and Goals,* US Patent 4,949,277, 1990.
[12] K. I. Trovato, "Autonomous Vehicle Maneuvering," *Proc. SPIE Advances in Intelligent Robotic Systems Conf.,* vol. 1613, Nov. 1991.
[13] K.I. Trovato, "General Planning Method for Machine Coordination and Rendezvous," *J. Circuits, Systems, and Computers—Automotive Electronics,* vol. 4, no. 4, Dec. 1994
[14] K.I. Trovato, "A^* Planning in Discrete Configuration Spaces of Autonomous Systems," PhD thesis, Univ. of Amsterdam, 1996.

Karen I. Trovato received the MSc degree in computer science from New York University in 1982 and the PhD degree in computer science from the University of Amsterdam, The Netherlands, in 1996. She became a registered patent agent in 1992 in order to, with L. Dorst, understand software patent issues raised by early inventions. Since 1979, she has been working for Philips Research-USA, Briarcliff Manor, New York, where she is currently a principal member of the research staff. Her interests are in planning algorithms used for real-time systems, such as robots and other machines, but also for in-vehicle and web-based navigation.

Leo Dorst received the MSc (1982) and PhD (1986) degrees from the Applied Physics Department at Delft University of Technology, The Netherlands. His PhD thesis was on accurate geometrical measurements in discretized images. From 1986 to 1992, he worked as senior research scientist at Philips Laboratories, Briarcliff Manor, New York, focusing on robot path planning and task abstraction in goal-directed systems. Since 1992, he has been an assistant professor at the University of Amsterdam, The Netherlands, where his continued interest is on planning and representation in autonomous systems, with a recent total emphasis on the use of "geometric algebra" for geometric representation and computation.