October 1993 *LIDS-P-2210*

EFFICIENT ALGORITHMS FOR GLOBALLY OPTIMAL TRAJECTORIES[†]

John N. Tsitsiklis[††]

Abstract

We present serial and parallel algorithms for solving a system of equations that arises from the discretization of the Hamilton–Jacobi equation associated to a trajectory optimization problem of the following type. A vehicle starts at a prespecified point x_0 and follows a unit speed trajectory $x(t)$ inside a region in $\Re^m$, until an unspecified time T that the region is exited. A trajectory minimizing a cost function of the form $\int_0^T r(x(t))\,dt + q(x(T))$ is sought. The discretized Hamilton–Jacobi equation corresponding to this problem is usually solved using iterative methods. Nevertheless, assuming that the function r is positive, we are able to exploit the problem structure and develop one–pass algorithms for the discretized problem. The first algorithm resembles Dijkstra's shortest path algorithm and runs in time $O(n \log n)$, where n is the number of grid points. The second algorithm uses a somewhat different discretization and borrows some ideas from a variation of Dial's shortest path algorithm that we develop here; it runs in time $O(n)$, which is the best possible, under some fairly mild assumptions. Finally, we show that the latter algorithm can be efficiently parallelized: for two–dimensional problems, and with p processors, its running time becomes $O(n/p)$, provided that $p = O(\sqrt{n}/\log n)$.

[†] Research supported by the the ARO under contract DAAL03–92–G–0115.

[††] Laboratory for Information and Decision Systems and the Operations Research Center, Room 35–214, Massachusetts Institute of Technology, Cambridge, MA 02139; e–mail: jnt@mit.edu.

1. INTRODUCTION

Consider a vehicle that is constrained to move in a subset G of $\Re^m$. The vehicle starts at an initial point x_0 and moves according to $dx/dt = u(t)$, subject to the constraint $\|u(t)\| \leq 1$, where $\|\cdot\|$ denotes the Euclidean norm. At some unspecified time T, the vehicle reaches the boundary of G and incurs a terminal cost $q(x(T))$. We also associate a traveling cost $\int_0^T r(x(t))\,dt$ to the trajectory followed by the vehicle. We are interested in a numerical method for finding a trajectory that minimizes the sum of the traveling and the terminal cost. We assume that $\inf_{x \in G} r(x) > 0$, which forces the vehicle to exit G in finite time.

This problem formulation allows us to enforce a desired destination x_f: for example, we may let $G = \Re^n - \{x_f\}$ and $q(x_f) = 0$. It can also incorporate "hard obstacles"; for example, if a subset F of G corresponds to an obstacle, we can redefine G, by removing F from G and by letting $q(x)$ be very large at the boundary of F.

Although several numerical methods for trajectory optimization are available, their computational complexity is not fully satisfactory, as will be discussed shortly. In contrast, we devise serial and parallel algorithms with optimal running time.

Interest in algorithmic efficiency can be motivated from certain situations in which the trajectory optimization problem has to be solved repeatedly and on–line; this is the case, for example, if the terrain conditions are uncertain and the remaining trajectory is reoptimized each time that new information becomes available. Of course, algorithmic efficiency is a worthy objective even when computations are carried out off–line.

Related research

Problems of this type have been considered by several different research communities. The robotics and theoretical computer science community has extensively studied the case where r is identically equal to 1, G contains several obstacles, and there is a fixed destination. Under the further assumption that the obstacles admit a finite description (in particular, if they are polygons), the problem can be transformed to a shortest path problem on a graph (the so–called "visibility graph"). Then, special shortest path algorithms can be developed that exploit the structure of the problem and reduce algorithmic complexity [M]. A more general version, the "weighted region problem", has been considered in [MP]. Here, the region G is partitioned into a finite number of polygons, and r is assumed to be constant in each polygon. The algorithms in [MP] are geared towards the case where the partition of G is fairly coarse. However, if we let the partition become arbitrarily fine, we are led to our formulation, with the function r having an arbitrary functional form.

Our problem is also a special case of deterministic optimal control. As such, variational techniques can be applied leading to a locally optimal trajectory [AF, FR]. However, in the presence of obstacles, or if the cost function r is not convex, the problem acquires a combinatorial flavor and can have several local minima that are far from being globally optimal. For this reason, other methods, of the dynamic programming type, are required. The solution to the problem is furnished, in principle, by the Hamilton–Jacobi (HJ) equation. Since an exact solution of the HJ equation is usually impossible, the problem has to be discretized and solved numerically. After discretization, one needs to solve a system of nonlinear equations whose structure resembles the structure of the original HJ equation. This approach raises two types of issues:

(a) Does the solution to the discretized problem provide a good approximation of the solution to the original problem?

(b) How should the discretized problem be solved?

Questions of the first type have been studied extensively and in much greater generality elsewhere – see, e.g. [KD, S] and references therein. We bypass such questions and focus on the purely algorithmic issues.

The usual approaches for discretizing the HJ equation are finite–difference or, more generally, finite–element methods [CD, CDI, GR, KD, S]. Furthermore, solving the discretized problem is equivalent to solving a *stochastic* optimal control problem for a finite state controlled Markov chain; the number of states of the Markov chain is equal to the number of grid points used in the discretization [KD]. Thus, the discretized problem can be solved by standard methods such as successive approximation or policy iteration [B1]. This is somewhat unfortunate: one would hope that the discretized version of an optimal trajectory problem would be a deterministic shortest path problem on a finite graph, that can be solved efficiently, say using Dijkstra's algorithm. In contrast, a method such as successive approximation can require a fair number of iterations, does not have good guarantees on its computational complexity (because the number of required iterations is not easy to bound), and can be much more demanding than Dijkstra's algorithm. The contribution of this paper is to show that, for the particular problem under consideration and for certain discretizations, Dijkstra–like methods can be used, resulting to fast algorithms. In particular, we will show under mild assumptions, that there is an algorithm whose complexity is proportional to the number of grid–points.

We close by mentioning another approach to the discretization of trajectory optimization problems. In [MK] the region G is discretized by using a regular rectangular grid, and the vehicle is only allowed to move along the edges of the grid (horizontally or vertically). Then, the shortest path problem on the resulting grid–graph is solved using Dijkstra's algorithm. The solution via Dijkstra's algorithm is certainly efficient, but the employed discretization does not lead to an accurate approximation of the solution to the original problem, no matter how fine a grid is used. The reason is that the set of allowed directions of motion is discretized very coarsely: only 4 directions are allowed. The inadequacy of the naive discretization is sometimes referred to as the *digitization bias.* It can be remedied by allowing diagonal motion [MK], but only partially. Our results establish that the digitization bias can be overcome without sacrificing the algorithmic efficiency of Dijkstra–like methods.

Summary of the paper

In Section 2, we state the HJ equation corresponding to our problem and define the standard finite–difference discretization.

In Section 3, we exploit certain properties of the discretized HJ equation to show that it can be solved in time $O(n \log n)$, where n is the number of grid points. In particular, we show that even though the discretized HJ equation does not correspond to a shortest path problem, it is still possible to mimick Dijkstra's shortest path algorithm.

In Section 4, we present a variation of Dial's shortest path algorithm. We show that, under certain assumptions on the arc costs, it has optimal computational complexity, and has good parallelization potential.

In Section 5, we explain why the algorithmic ideas of Section 4 cannot be applied to the discretized HJ equation of Section 2. We are thus led to the development of an alternative discretization. With this new discretization, we show that the algorithmic ideas of Section 4 lead to an $O(n)$ algorithm, which is the best possible.

In Section 6, we show that the algorithms of Sections 4 and 5 can be efficiently parallelized. In particular, we show that linear speedup is obtained: the running time in a shared memory parallel computer with p processors is only $O(n/p)$, as long as the number of processors is not too excessive; e.g., for two–dimensional problems, if $p = O(\sqrt{n}/\log n)$. We compare our results to those achievable by the successive approximation method.

Finally, in Section 7, we refer to some preliminary numerical experiments that strongly support our results, provide some conclusions, and discuss some possibilities for extending our results to more complex trajectory optimization problems.

II. PROBLEM FORMULATION AND A FINITE–DIFFERENCE DISCRETIZATION

Let G be a bounded connected open subset of $\Re^m$ and let ∂G be its boundary. We are also given two cost functions $r : G \mapsto (0, \infty)$ and $q : \partial G \mapsto (0, \infty)$. A *trajectory* starting at $x_0 \in G$ is a continuous function $x : [0, T] \mapsto \Re^m$ such that $x(t) \in G$ for all $t \in [0, T)$ and $x(T) \in \partial G$. A trajectory is called *admissible* if there exists a measurable function $u : [0, T] \mapsto \Re^m$ such that $x(t) = x(0) + \int_0^t u(s)\, ds$ and $\|u(t)\| \leq 1$ for all $t \in [0, T]$, where $\|\cdot\|$ stands for the Euclidean norm. The cost of an admissible trajectory is defined to be $\int_0^T r(x(t))\, dt + q(x(T))$. The optimal cost–to–go function $V^* : G \cup \partial G \mapsto \Re$ is defined as follows: if $x \in \partial G$, we let $V^*(x) = q(x)$; if $x \in G$, we let $V^*(x)$ be the infimum of the costs of all admissible trajectories that start at x.

A formal argument [FR] indicates that V^* should satisfy the Hamilton–Jacobi equation

$$\min_{\{v \in \Re^m \mid \|v\| \leq 1\}} \{r(x) + \langle v, \nabla V^*(x) \rangle\} = 0, \qquad x \in G. \tag{2.1}$$

Furthermore, for any $x \in \partial G$, V^* should satisfy

$$\limsup_{y \to x} V^*(y) \leq V^*(x), \tag{2.2}$$

where the limit is taken with y approaching x from the interior of G. If the problem data are smooth enough and if V^* is differentiable, it can be argued rigorously that V^* must satisfy Eqs. (2.1)–(2.2). Furthermore, V^* can be characterized as the maximal solution of Eqs. (2.1)–(2.2). Unfortunately, the assumptions needed for V^* to be differentiable are too strong for many practical problems. The validity of these equations can be still justified, under much weaker assumptions, if V^* is interpreted as a "viscosity" solution of Eq. (2.1) [CL, FS].

We now describe a discretized version of the HJ equation. While this discretization is a special case of the discretizations described in [KD], we provide a self–contained motivation based on Bellman's principle of optimality.

Let h be a small positive scalar representing the fineness of the discretization (the discretization step). Let S and B be two disjoint finite subsets of $\Re^m$, all their elements being of the form (ih, jh), where i and j are integers. The sets S and B are meant to represent a discretization of the sets G and ∂G, respectively.

Let $e_1, \ldots, e_m$ be the unit vectors in $\Re^m$. For any point $x \in S$, we define the set $N(x)$ of its *neighbors* by letting $N(x) = \{x + h\alpha_i e_i \mid i \in \{1, \ldots, n\},\ \alpha_i \in \{-1, 1\}\}$. The assumption that follows states that B contains the "boundary" of S, in keeping with the intended meaning of these sets.

Assumption 2.1: For every $x \in S$, we have $N(x) \subset S \cup B$.

Let α be an element of $\mathcal{A} = \{-1, 1\}^m$. To every $\alpha = (\alpha_1, \ldots, \alpha_m) \in \mathcal{A}$, we associate a *quadrant*, namely, the cone generated by the vectors $\alpha_1 e_1, \ldots, \alpha_m e_m$. Let Θ be the unit simplex in $\Re^m$; that is, $(\theta_1, \ldots, \theta_m) \in \Theta$ if and only if $\sum_{i=1}^m \theta_i = 1$ and $\theta_i \geq 0$ for all i.

We assume that we have two functions $f : B \mapsto (0, \infty)$ and $g : S \mapsto (0, \infty)$ that represent discretizations of the cost functions q and r in the original problem. The function g can be usually defined by $g(x) = r(x)$ for every $x \in S$. The choice of f can be more delicate because B can be disjoint from ∂G even if B is a good approximation of ∂G.

We finally introduce a function $V : S \cup B \mapsto \Re$ which is meant to provide an approximation of the optimal cost–to–go function V^*. The discretized HJ equation is the following system of equations in the unknown V:

$$V(x) = \min_{\alpha \in \mathcal{A}} \min_{\theta \in \Theta} \left[hg(x)\tau(\theta) + \sum_{i=1}^{m} \theta_i V(x + h\alpha_i e_i) \right], \qquad x \in S, \tag{2.3}$$

$$V(x) = f(x), \qquad x \in B, \tag{2.4}$$

where

$$\tau(\theta) = \|\theta\| = \sqrt{\sum_{i=1}^{m} \theta_i^2}, \qquad \theta \in \Theta. \tag{2.5}$$

We now explain the form of Eqs. (2.3)–(2.4). Suppose that the vehicle starts at some $x \in S$ and that it moves, at unit speed, along a direction d. This direction is determined by specifying the quadrant α to which d belongs and by then specifying the relative weights θ_i of the different vectors $\alpha_i e_i$ that generate this quadrant. Assume that the vehicle moves along the direction d until it hits the convex hull of the points $x + h\alpha_i e_i$, $i = 1, \ldots, m$. At that time, the vehicle has reached point $x + h\sum_{i=1}^{m} \theta_i \alpha_i e_i$. Since the vehicle travels at unit speed, the amount of time it takes is equal to

$$\left\| h \sum_{i=1}^{m} \theta_i \alpha_i e_i \right\| = h\tau(\theta);$$

see Fig. 2.1. Since $g(x)$ represents travel costs per unit time (in the vicinity of x), the traveling cost is equal to $hg(x)\tau(\theta)$. To the traveling cost we must also add the cost-to-go from point $x + h\sum_{i=1}^{m} \theta_i \alpha_i e_i$ and, invoking the principle of optimality, we obtain

$$V^*(x) \approx \min_{\alpha \in \mathcal{A}} \min_{\theta \in \Theta} \left[hg(x)\tau(\theta) + V^*\Big(x + h\sum_{i=1}^{m} \theta_i \alpha_i e_i\Big)\right]. \tag{2.6}$$

We approximate V^* by a linear function on the convex hull of the points $x + h\alpha_i e_i$, to obtain

$$V^*\Big(x + h\sum_{i=1}^{n} \theta_i \alpha_i e_i\Big) \approx \sum_{i=1}^{n} \theta_i V^*(x + h\alpha_i e_i). \tag{2.7}$$

Using the approximation (2.7) in Eq. (2.6), we are led to Eq. (2.3).

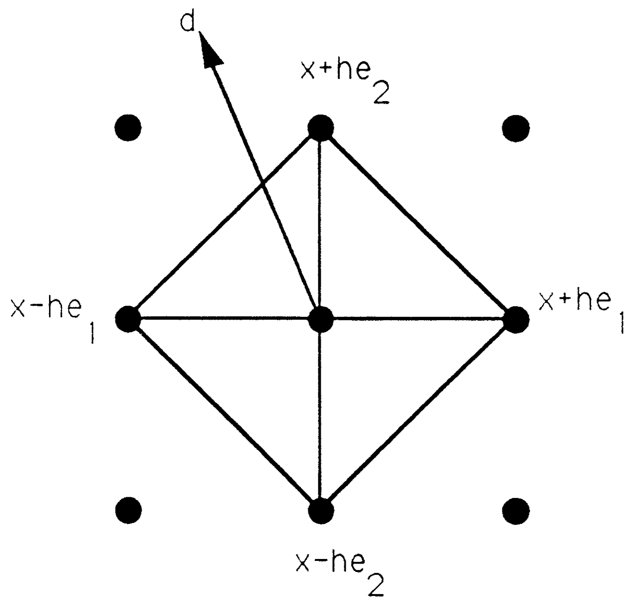

Figure 2.1. Illustration of the discretization of the HJ equation. Here, the vehicle moves along the direction d, in the quadrant defined by $-e_1$ and e_2.

The above discussion gives some plausibility to the claim that the solution V of Eqs. (2.3)–(2.4) can provide a good approximation of the function V^* and serves to motivate our objective: providing an efficient algorithmic solution of Eqs. (2.3)–(2.4).

As pointed out in [KD], Eqs. (2.3)–(2.4) are the Dynamic Programming equations for the following Markov Decision Problem: if we are at state $x \in S$ and a decision $(\alpha, \theta) \in \mathcal{A} \times \Theta$ is made, the cost $hg(x)\tau(\theta)$ is incurred and the next state is $x + h\alpha_i e_i$, with probability θ_i; if we enter a state $x \in B$, the terminal cost $f(x)$ is incurred and the process stops. Since the cost per stage is bounded below by the positive constant $h \min_{x \in S} g(x)$, standard results of Markovian Decision theory [B1, BT] imply that Eqs. (2.3)–(2.4) have a unique solution which is equal to the optimal expected cost. Furthermore, either the successive approximation or the policy iteration algorithm will converge to the solution of (2.3)–(2.4).

References [GR] and [KD] suggest the use of the successive approximation method, possibly an accelerated version. The computational complexity of each iteration is proportional to the number of grid–points. However, even for deterministic shortest path problems, the number of iterations is proportional to the diameter of the grid–graph, which is usually of the order of $1/h$. The number of iterations can be reduced using Gauss–Seidel relaxation (as in [GR], for example), but no theoretical guarantees are available. This is in contrast to Dijkstra–like algorithms that solve deterministic shortest path problems with essentially a single pass through the grid points.

In the next section, we show that even though Eqs. (2.3)–(2.4) correspond to a Markovian Decision Problem, they still have enough structure for the basic ideas of Dijkstra's algorithm to be applicable, leading to an efficient algorithm.

III. A DIJKSTRA–LIKE ALGORITHM

Dijkstra's algorithm is a classical method for solving the shortest path problem on a finite graph. Its running time, for bounded degree graphs, is $O(n \log n)$, where n is the number of nodes, provided that it is implemented with suitable data structures [B2]. The key idea in Dijkstra's algorithm is to generate the nodes in order of increasing value of the cost–to–go function. This is done in n stages (one node is generated at each stage) and the $O(\log n)$ factor is due to the overhead of deciding which node is to be generated next. We will now show that a similar idea can be applied to the solution of Eqs. (2.3)–(2.4) and that the elements of $S \cup B$ can be generated in order of increasing values of $V(x)$.

Throughout this section, we reserve the notation $V(x)$ to indicate the unique solution of Eqs. (2.3)–(2.4). The key to the algorithm is provided by the following lemma that states that the cost–to–go $V(x)$ from any node x can be determined from knowledge of $V(y)$ for those nodes y with smaller cost–to–go.

Lemma 3.1: Let $x \in S$, and let $\alpha \in \mathcal{A}$, $\theta \in \Theta$, be such that $V(x) = hg(x)\tau(\theta) + \sum_{i=1}^{m} \theta_i V(x + h\alpha_i e_i)$. Let $\mathcal{I} = \{i \mid \theta_i > 0\}$. Then, $V(x + h\alpha_i e_i) < V(x)$ for all $i \in \mathcal{I}$.

Proof: To simplify notation, and for the purposes of this proof only, let $A = hg(x)$ and $V_i = V(x + h\alpha_i e_i)$. The assumptions of the lemma and Eq. (2.3) yield

$$V(x) = A\tau(\theta) + \sum_{i=1}^{m} \theta_i V_i = \min_{\zeta\in\Theta}\{A\tau(\zeta) + \sum_{j=1}^{m} \zeta_j V_j\}. \tag{3.1}$$

Notice that the function minimized in Eq. (3.1) is convex and continuously differentiable. We associate a Lagrange multiplier to the constraint $\sum_{i=1}^{m} \zeta_i = 1$. Then, the Kuhn–Tucker conditions show that there exists a real number λ such that

$$A\frac{\partial\tau(\theta)}{\partial\theta_i} + V_i = \lambda, \tag{3.2}$$

for all $i \in \mathcal{I}$. Using the functional form of $\tau(\theta)$, we obtain

$$\frac{A\theta_i}{\tau(\theta)} + V_i = \lambda, \qquad \forall i \in \mathcal{I}. \tag{3.3}$$

We solve Eq. (3.3) for V_i and substitute in Eq. (3.1), to obtain

$$V(x) = A\tau(\theta) + \lambda - \frac{A\sum_{i\in\mathcal{I}} \theta_i^2}{\tau(\theta)}.$$

Thus, it remains to show that

$$A\tau(\theta) + \lambda - \frac{A\sum_{j\in\mathcal{I}} \theta_j^2}{\tau(\theta)} > \lambda - \frac{A\theta_i}{\tau(\theta)}, \qquad \forall i \in \mathcal{I},$$

or, equivalently, that

$$\tau(\theta) - \frac{\sum_{j\in\mathcal{I}} \theta_j^2}{\tau(\theta)} > -\frac{\theta_i}{\tau(\theta)}. \tag{3.4}$$

Using the definition of $\tau(\theta)$, we see that the left hand side of Eq. (3.4) is equal to zero. On the other hand, for $i \in \mathcal{I}$, we have $\theta_i > 0$ and the right hand side of Eq. (3.4) is negative, thus establishing the desired result. **Q.E.D.**

We now proceed to the description of the algorithm. Let x_1 be an element of B at which $f(x)$ is minimized. Using the Markov Decision Problem interpretation of Eqs. (2.3)–(2.4), it is evident that $V(x) \geq f(x_1) = V(x_1)$, for all $x \in S \cup B$. Thus, x_1 is a point with a smallest value of $V(x)$, and this starts the algorithm.

We now proceed to a recursive description of a general stage of the algorithm. Suppose that during the first k stages ($1 \leq k < n$) we have generated a set of points $P_k = \{x_1, \ldots, x_k\} \subset S \cup B$ with the property

$$V(x_1) \leq V(x_2) \leq \cdots \leq V(x_k) \leq V(x), \qquad \forall x \notin P_k.$$

Furthermore, we assume that the value of $V(x)$ has been computed for every $x \in P_k$. (The set P_k is like the set of *permanently labeled* nodes in Dijkstra's algorithm.)

We define $\overline{V}_k(x)$ by letting $\overline{V}_k(x) = V(x)$ for $x \in P_k \cup B$, and $\overline{V}_k(x) = \infty$, otherwise. We then compute an estimate $\hat{V}_k$ of the function V by essentially performing one iteration of the successive approximation algorithm, starting from $\overline{V}_k$. More precisely, let $\hat{V}_k(x) = V(x)$ for $x \in B$ and

$$\hat{V}_k(x) = \min_{\alpha \in \mathcal{A}} \min_{\theta \in \Theta} \Big[hg(x)\tau(\theta) + \sum_{i=1}^{m} \theta_i \overline{V}_k(x + h\alpha_i e_i) \Big], \qquad x \in S. \tag{3.5}$$

In this equation, and throughout the rest of the paper, we use the interpretation $0 \cdot \infty$. Since $\overline{V}_k(x) \geq V(x)$, a comparison of Eqs. (3.5) and (2.3) shows that

$$\hat{V}_k(x) \geq V(x), \qquad \forall x \in B \cup S. \tag{3.6}$$

The variable $\hat{V}_k(x)$, for $x \notin P_k$ is similar to the *temporary labels* in Dijkstra's algorithm.

We now choose a node with the smallest temporary label to be labeled permanently. Formally, we choose some x_{k+1} that minimizes $\hat{V}_k(x)$ over all $x \notin P_k$. The following lemma asserts that this choice of x_{k+1} is sound.

Lemma 3.2: (a) $V(x_{k+1}) = \hat{V}_k(x_{k+1})$.
(b) For every $x \notin P_k$, we have $V(x_{k+1}) \leq V(x)$.

Proof: Let $y \notin P_k$ be such that $V(y) = \min_{x \notin P_k} V(x)$. We will show that $V(y) = \hat{V}_k(y)$. If $y \in B$, this is automatically true. Assume now that $y \in S$. Let $\alpha \in \mathcal{A}$ and $\theta \in \Theta$ be such that $V(y) = hg(y)\tau(\theta) + \sum_{i=1}^{m} \theta_i V(y + h\alpha_i e_i)$. Let $\mathcal{I} = \{i \mid \theta_i > 0\}$. Lemma 3.1 asserts that $V(y + h\alpha_i e_i) < V(y)$ for every $i \in \mathcal{I}$. In particular, $y + h\alpha_i e_i \in P_k$ for every $i \in \mathcal{I}$. Therefore, $V(y + h\alpha_i e_i) = \overline{V}_k(y + h\alpha_i e_i)$, for every $i \in \mathcal{I}$. Consequently,

$$V(y) \leq \hat{V}_k(y) \leq hg(y)\tau(\theta) + \sum_{i=1}^{m} \theta_i \overline{V}_k(y + h\alpha_i e_i) = hg(y)\tau(\theta) + \sum_{i=1}^{m} \theta_i V(y + h\alpha_i e_i) = V(y).$$

(The first inequality follows from Eq. (3.6); the second from Eq. (3.5); the last one from the definition of α and θ.) The conclusion $V(y) = \hat{V}_k(y)$ follows.

This, together with the fact $V(x) \leq \hat{V}_k(x)$, for all x, shows that x_{k+1} which mimimizes $\hat{V}_k(x)$ over all $x \notin P_k$ also minimizes $V(x)$ over all $x \notin P_k$ and $\hat{V}(x_{k+1}) = V(x_{k+1})$. **Q.E.D.**

The description of the algorithm is now complete. The algorithm terminates after n stages and produces the values of $V(x)$ for all $x \in S \cup B$, in nondecreasing order. In order to determine the complexity of the algorithm, we will bound the complexity of a typical stage. Throughout this analysis, we view the dimension m of the problem as a constant, and we investigate the dependence of the complexity on n.

Let us first consider what it takes to compute $\hat{V}_k(x)$. There are $O(1)$ different elements α of $\mathcal{A}$ to consider and for each one of them, we have to solve, after some normalization, a convex optimization problem of the form

$$\min_{\theta \in \Theta} \left[\sqrt{\sum_{i=1}^{m} \theta_i^2} + \sum_{i=1}^{m} \theta_i V_i \right] \tag{3.7}$$

No matter what method is used to solve the problem (3.7), the computational effort is independent of the number n of grid points; it depends, of course, on the dimension m, but we are viewing this as a constant. Thus, we can estimate the complexity of computing $\hat{V}_k(x)$, for any fixed x, according to Eq. (3.5), to be $O(1)$.

How would we solve (3.7) in practice? We can use an iterative method, such as a gradient projection method or a projected Newton method. For small dimensions m (which is the practically interesting case), such a method would produce an excellent approximation of the optimal solution after very few iterations. Furthermore, it is not difficult to show that small errors in intermediate computations only lead to small errors in the final output of our overall algorithm. Finally, for theoretical reasons, it is useful to notice that the problem (3.7) can be solved exactly with a finite number of operations, if the computation of a square root counts as a single operation; the details are provided in the appendix.

We now notice that $\overline{V}_k(x) = \overline{V}_{k+1}(x)$ for every $x \neq x_{k+1}$. This means that if x is not a neighbor of x_{k+1}, then $\hat{V}_k(x) = \hat{V}_{k+1}(x)$. Thus, $\hat{V}_{k+1}(x)$ only needs to be computed for the $O(1)$ neighbors of x_{k+1}. We conclude that once $\hat{V}_k$ is computed, the evaluation of $\hat{V}_{k+1}$, at the next stage of the algorithm, only requires $O(1)$ computations.

At each stage, we must also determine the next point x_{k+1}, by minimizing $\hat{V}_k(x)$ over all $x \notin P_k$. Comparing $O(n)$ numbers takes $O(n)$ time, which leads to $O(n)$ time for each stage, and a total $O(n^2)$ running time. In a better implementation, the values $\hat{V}_k(x)$ can be maintained in a binary heap, in which case x_{k+1} can be determined in $O(\log n)$ time; see [B2, CLR] for the use of binary heaps in shortest path algorithms. We conclude that each stage of the algorithm can be implemented with $O(\log n)$ computations. We now summarize:

Theorem 3.1: The algorithm of this section solves the system of equations (2.3)–(2.4). Assuming that square roots can be evaluated in unit time, it can be implemented so that it runs in time $O(n \log n)$.

Some more comments are in order. We have been using a uniform grid. If we were to use a nonuniform grid instead, there would be some minor changes in the form of Eq. (2.3). The general structure would still be the same. However, Lemma 3.1 would cease to hold. Similarly, if the cost function $g(x)$ were to become direction dependent, e.g., of the form $g(x, \alpha, \theta)$, Lemma 3.1 would again fail to hold.

Finally, we note that the algorithm of this section is inherently serial. This is because the elements of S are generated one at a time, in order of increasing values of $V(x)$. To obtain a parallelizable algorithm, we should be able to generate the values of $V(x)$ for several points x simultaneously. To gain some insight into how this might be done, we first consider, in the next section, an algorithm for the classical shortest path problem.

IV. A VARIATION OF DIAL'S SHORTEST PATH ALGORITHM

We are given a directed graph $G = (N, A)$. Here, $N = \{1, \ldots, n\}$ is the set of nodes, and A is the set of directed arcs. For each arc $(i,j) \in A$, we are given a positive arc length a_{ij}. The objective is to find, for every node i, a shortest path from node i to node 1. We will use the following assumptions:

Assumption 4.1: (a) For every i, there exists a path from i to node 1.
(b) For every $(i,j) \in A$, we have $a_{ij} \geq 1$.

Let $V(i)$ be the length of a shortest path from node i to node 1. For notational convenience, we let $V(1) = 0$ and $a_{ij} = \infty$ if $(i,j) \notin A$. For $k = 1, 2, \ldots$, let $Q_k = \{i \mid k-1 \leq V(i) < k\}$ and $R_k = \cup_{i=0}^{k} Q_i = \{i \mid V(i) < k\}$.

The algorithm starts with $R_1 = Q_1 = \{1\}$. Suppose that after k stages of the algorithm, we have determined the sets Q_k and R_k, and have computed $V(i)$ for every $i \in R_k$. We may call the nodes in R_k *permanently labeled.* We then define *temporary labels* by letting

$$\hat{V}_k(i) = \min_{j \in R_k} \{a_{ij} + V(j)\}. \tag{4.1}$$

Notice that $V(i) = \min_j \{a_{ij} + V(j)\}$, which implies that $V(i) \leq \hat{V}_k(i)$ for all i.

Lemma 4.1: Suppose that $V(i) \geq k$, i.e., $i \notin R_k$.

(a) If $V(i) < k+1$, then $\hat{V}_k(i) = V(i)$.

(b) If $V(i) \geq k+1$, then $\hat{V}_k(i) \geq k+1$.

(c) We have $i \in R_{k+1}$ if and only if $\hat{V}_k(i) < k+1$ and, if this is the case, then $\hat{V}_k(i) = V(i)$.

Proof: (a) Let ℓ be the first node on a shortest path from i to 1. Then, $V(i) = a_{i\ell} + V(\ell)$. If $V(i) < k+1$, then $V(\ell) < k$ and $\ell \in R_k$. Thus, $\hat{V}_k(i) \leq a_{i\ell} + V(\ell) = V(i)$. On the other hand, we have already noted that $V(i) \leq \hat{V}_k(i)$, which shows that $V(i) = \hat{V}_k(i)$.

(b) This is trivial because $V(i) \leq \hat{V}_k(i)$.

(c) This is just a restatement of (a) and (b). **Q.E.D.**

Lemma 4.1 shows that $Q_{k+1} = \{i \notin R_k \mid \hat{V}_k(i) < k+1\}$, from which the set Q_{k+1} can be determined and this completes the description of a typical stage of the algorithm. The algorithm terminates after at most $L+1$ stages, where $L = \lceil \max_i V(i) \rceil$. We now describe an efficient implementation.

As in Dial's shortest path algorithm, we store the temporary labels $\hat{V}_k(i)$ in "buckets". (As is well known [B2], buckets can be implemented so that insertion and deletion of an item takes $O(1)$ computations.) We will use L buckets and at the kth stage of the algorithm, the jth bucket will contain a list of all nodes i such that $j - 1 \leq \hat{V}_k(i) < j$. On the side, we will also maintain an array whose ith entry will contain the value of $\hat{V}_k(i)$. The algorithm is initialized by computing $\hat{V}_1(i)$ for all i, and by placing each i in the appropriate bucket.

Suppose that $\hat{V}_k$ has been computed, and each i is stored in the appropriate bucket. Note that Eq. (4.1) can be written as

$$\hat{V}_{k+1}(i) = \min\{\hat{V}_k(i), \min_{j \in Q_{k+1}} \{a_{ij} + V(j)\}\}. \tag{4.2}$$

Let us consider a typical node i. If there exists no $j \in Q_{k+1}$ such that $(i,j) \in A$, then Eq. (4.2) shows that $\hat{V}_{k+1}(i) = \hat{V}_k(i)$, i stays in the same bucket and nothing needs to be done.

If on the other hand, there exists some $j \in Q_{k+1}$ such that $(i,j) \in A$, then $\hat{V}_{k+1}(i)$ has to be evaluated according to Eq. (4.2). Let Z_{k+1} be the total number of arcs leading into some element of Q_{k+1}. (Note that $\sum_{k=1}^{L} Z_k = |A|$.) Then, the computation required to evaluate $\hat{V}_{k+1}(i)$ for all i, is $O(Z_{k+1})$. This leads to a total of $O(|A|)$ computations throughout the course of the algorithm. For every i for which $\hat{V}_{k+1}(i) \neq \hat{V}_k(i)$, we also need to move i to a new bucket and this takes $O(1)$ time. By a similar argument, the total amount of work is still $O(|A|)$.

We now summarize:

Theorem 4.1: Let Assumption 4.1 hold and suppose that $V(i) \leq L$ for all i. Then, the above described algorithm computes $V(i)$ for all i in time $O(L + |A|)$.

Remarks:

1. If all a_{ij} are integer, the algorithm of this section is identical with Dial's algorithm. Our development here shows that the assumption $a_{ij} \geq 1$, rather than the integrality assumption, is the essential one.

2. If $L = O(|A|)$, the running time of the algorithm is simply $O(|A|)$, which is the best possible. Suppose that the graph G is a square mesh in m–dimensional space, with a total of n points. We then have $|A| = O(mn)$. Suppose that $a_{ij} \leq K$ for some constant K. Then, the length L of any shortest path is bounded by K times the diameter of the graph. Thus, we can let $L = Kmn^{1/m}$. Recall that we have an optimal algorithm if $L = O(|A|)$. This will happen if $Kmn^{1/m} = O(mn)$, or, equivalently, if $K = O(n^{(m-1)/m})$. Even in two dimensions ($m = 2$), we obtain an $O(n)$ algorithm while allowing a fairly large amount of variability of the arc lengths (a factor of $n^{1/2}$). Notice that this is exactly the type of shortest path problems that one obtains from the naive discretization of trajectory optimization problems mentioned in the end of Section 1.

3. The algorithm has excellent parallelization potential. At each stage, we can let a different processor compute $\hat{V}_k(i)$ for a different node i. Thus, the parallel time seems to be limited only by the number L of stages in the algorithm. If L is much smaller than the number n of nodes, then we can hope that parallelism leads to a substantial speedup. So, for the case of a two–dimensional mesh (see remark 2), if we have $L = O(n^{1/2})$ and $K = O(1)$, we can hope for $O(n^{1/2})$ parallel running time. We will see in Section 6 that we can come fairly close to this optimistic estimate.

V. AN ALGORITHM WITH OPTIMAL COMPLEXITY

The algorithm of Section 3 achieved $O(n \log n)$ running time by mimicking Dijkstra's shortest path algorithm. In order to reduce the complexity to $O(n)$, we will mimick the algorithm of Section 4. The key to that algorithm was the following elementary fact: if $V(i)$ depends on $V(\ell)$, in the sense that $V(i) = a_{i\ell} + V(\ell)$, then $V(i) \geq V(\ell) + 1$. An analogous property that would lead to a fast solution of Eqs. (2.3)–(2.4) is the following:

Property P: There exists a constant $\delta > 0$ such that if $V(x) = hg(x)\tau(\theta) + \sum_{i=1}^{m} \theta_i V_i(x + h\alpha_i e_i)$ and $\theta_i > 0$, then $V(x) \geq V_i(x + h\alpha_i e_i) + \delta$.

Lemma 3.1 established that property P holds with $\delta = 0$. Unfortunately, property P is not true for Eqs. (2.3)–(2.4) when we let δ be positive. In this section, we show that property P becomes true if a somewhat different discretization is used. Then, based on this property, we mimick the algorithm of Section 4, to solve the trajectory optimization problem in $O(n)$ time. Unfortunately, the discretization that we introduce is more cumbersome and is unlikely to be useful when the dimension is higher than 3. For this reason, we will only describe our method when the dimension m is 2 or 3. The reader should have no difficulty in generalizing to higher dimensions.

Let us first consider 2–dimensional problems. Let H be the boundary of a square centered at the origin and whose edge length is equal to $2h$. We define the vectors $w_1, \ldots, w_8$ as shown in Fig. 5.1. We use $x + H$ to denote the translation of H so that it is centered at x.

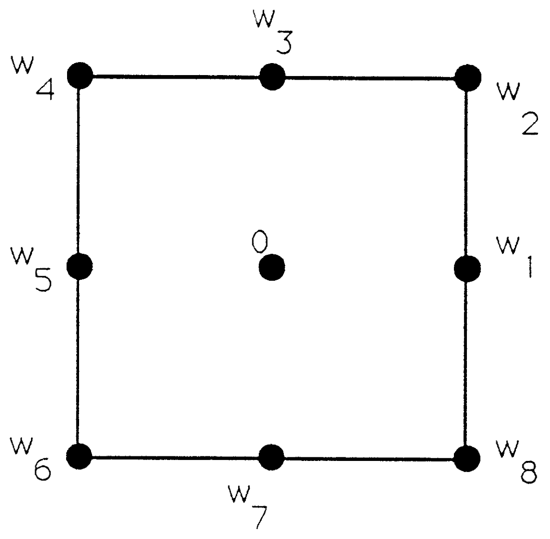

Figure 5.1. A square of size $2h \times 2h$ centered at the origin and the definition of the vectors $w_1, \ldots, w_8$.

As in Section 2, let S and B be two disjoint finite subsets of $\Re^m$, all of their elements being of the form (ih, jh), where i and j are integers. We assume that we are given functions $f : S \mapsto (0, \infty)$ and $g : B \mapsto (0, \infty)$. For any point $x \in S$, let $N(x)$, the set of its be neighbors, be $N(x) = \{x + w_i \mid i = 1, \ldots, 8\}$. As in Assumption 2.1, we assume that for every $x \in S$, we have $N(x) \subset S \cup B$.

We now motivate the discretization of the HJ equation that will be used in this section. Suppose that the vehicle starts at some $x \in S$ and moves along a direction d, for some time τ, until it hits the set $x + H$. The direction d is in the cone generated by w_α and $w_{\alpha+1}$ for some suitable choice of α. The point at which the vehicle meets H is of the form $(1-\theta)w_\alpha + \theta w_{\alpha+1}$, for some $\theta \in [0,1]$. We will thus parametrize the choice of direction d by a parameter $\alpha \in \{1, \ldots, 8\}$ that specifies a particular cone and then by a parameter $\theta \in [0,1]$ that picks a particular element of that cone. Let $h\tau_\alpha(\theta)$ be the travel time along the direction determined by α and θ, until the set $x + H$ is reached. It is easily seen that

$$\tau_\alpha(\theta) = \|(1-\theta)w_\alpha + \theta w_{\alpha+1}\| = \begin{cases} \sqrt{1+(1-\theta)^2}, & \text{if } \alpha \text{ is even}, \\ \sqrt{1+\theta^2}, & \text{if } \alpha \text{ is odd}. \end{cases}$$

Using the principle of optimality, as in Section 2, and by approximating V by a linear function on the segment joining w_α and $w_{\alpha+1}$, we obtain the following system of equations:

$$V(x) = \min_{\alpha=1,\ldots,8} \min_{\theta\in[0,1]} \Big[hg(x)\tau_\alpha(\theta) + (1-\theta)V(x+w_\alpha) + \theta V(x+w_{\alpha+1})\Big], \qquad x \in S, \quad (5.1)$$

$$V(x) = f(x), \qquad x \in B. \quad (5.2)$$

Equations (5.1)–(5.2) are again a special case of the finite element discretizations studied in [KD]. Once more, they admit a Markov Decision Process interpretation, have a unique solution, and we reserve the notation $V(x)$ to denote such a solution.

Recall that the cost per stage g in the discretized problem has been assumed to be positive. In the following, we assume a lower bound of unity for g and proceed to establish property P.

Assumption 5.1: For every $x \in S$, we have $g(x) \geq 1$.

Lemma 5.1: Let α and θ be such that

$$V(x) = hg(x)\tau_\alpha(\theta) + (1-\theta)V(x+w_\alpha) + \theta V(x+w_{\alpha+1}).$$

If $\theta < 1$, then $V(x) \geq V(x + w_\alpha) + (h/\sqrt{2})$. If $\theta > 0$, then $V(x) \geq V(x + w_{\alpha+1}) + (h/\sqrt{2})$.

Proof: We only consider the case where $\alpha = 1$. The argument for other choices of α is identical. Suppose that $\theta = 0$. Then, $V(x) = hg(x) + V(x + w_1) \geq (h/\sqrt{2}) + V(x + w_1)$, as desired. Suppose that $\theta = 1$. Once more, $V(x) = hg(x)\sqrt{2}+V(x+w_2) \geq (h/\sqrt{2})+V(x+w_2)$.

Suppose now that $0 < \theta < 1$. The first order optimality condition for θ yields

$$\frac{hg(x)\theta}{\sqrt{1+\theta^2}} + V(x + w_2) - V(x + w_1) = 0.$$

Therefore,

$$\begin{aligned} V(x) - V(x + w_2) &\geq V(x) - V(x + w_1) \\ &= hg(x)\sqrt{1+\theta^2} + \theta(V(x + w_2) - V(x + w_1)) \\ &= hg(x)\sqrt{1+\theta^2} - \frac{hg(x)\theta^2}{\sqrt{1+\theta^2}} \\ &= hg(x)\frac{1+\theta^2-\theta^2}{\sqrt{1+\theta^2}} \\ &\geq \frac{h}{\sqrt{2}}. \end{aligned}$$

Q.E.D.

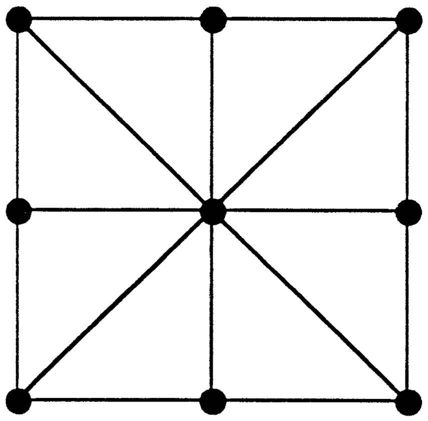

Figure 5.2. A triangulation of each face of the cube H.

We now continue with the 3–dimensional case. Let H be the boundary of a cube centered at the origin and with edge–length equal to $2h$. We triangulate each face of H as shown in Fig. 5.2. We use a similar triangulation for every face of $x + H$. The rest is very similar to the two–dimensional case. A direction of motion can be parametrized by specifying a triangle on some face of the cube, and by then specifying a particular point in that triangle. Let α be a parameter indicating the chosen triangle. (There are six faces with 8 triangles each; thus, α runs from 1 to 48.) For a given triangle α, let $y_{\alpha,1}, y_{\alpha,2}, y_{\alpha,3}$ be its vertices. In particular, let $y_{\alpha,1}$ be the point closest to the center of the cube, and let $y_{\alpha,3}$ be the one furthest away. We define the set $N(x)$ of neighbors of x, as the set of all points in the set $x + H$ whoe coordinates are integer multiples of h. As in the two–dimensional case, we require that $N(x) \subset S \cup B$ for all $x \in S$.

Let $\Theta = \{(\theta_1, \theta_2, \theta_3) \mid \theta_i \geq 0,\ \sum_{i=1}^{3} \theta_i = 1\}$. Every point in the triangle corresponding to some α is of the form $\sum_{i=1}^{3} \theta_i y_{\alpha,i}$, where $\theta \in \Theta$. Let $h\tau(\theta)$ be the distance from the center of the cube to the point determined by α and θ. It is easily seen that

$$\tau(\theta) = \|(\theta_1 + \theta_2 + \theta_3, \theta_2 + \theta_3, \theta_3)\| = \sqrt{1 + (1 - \theta_1)^2 + \theta_3^2}.$$

Once more, the principle of optimality yields

$$V(x) = \min_{\alpha} \min_{\theta \in \Theta} \Big[hg(x)\tau(\theta) + \sum_{i=1}^{3} \theta_i V(x + y_{\alpha,i}) \Big], \qquad x \in S, \tag{5.3}$$

$$V(x) = f(x), \qquad x \in B. \tag{5.4}$$

We reserve again the notation $V(x)$ to indicate the unique solution of Eqs. (5.3)–(5.4). The following is the 3–dimensional analog of Lemma 5.1.

Lemma 5.2: Let α and $\theta \in \Theta$ be such that

$$V(x) = hg(x)\tau(\theta) + \sum_{i=1}^{3} \theta_i V(x + y_{\alpha,i}).$$

If $\theta_i > 0$, then $V(x) \geq V(x + y_{\alpha,i}) + h/\sqrt{3}$.

Proof: Suppose that α corresponds to the triangle whose vertices are the points $y_{\alpha,1} = x + (h, 0, 0)$, $y_{\alpha,2} = x + (h, h, 0)$, $y_{\alpha,3} = x + (h, h, h)$. The proof for any other choice of α is identical, due to the symmetry of the triangulation we are using. Let $V_i = V(y_{\alpha,i})$. Using the formula for $\tau(\theta)$, we have

$$\begin{aligned} V(x) &= hg(x)\sqrt{1 + (1 - \theta_1)^2 + \theta_3^2} + \sum_{i=1}^{3} \theta_i V_i \\ &= \min_{\zeta_1 \geq 0, \zeta_3 \geq 0, \zeta_1 + \zeta_3 \leq 1} \Big[hg(x)\sqrt{1 + (1 - \zeta_1)^2 + \zeta_3^2} + \sum_{i=1}^{3} \zeta_i V_i \Big]. \end{aligned} \tag{5.5}$$

Suppose that $\theta_i > 0$ for all i. Then, the first-order optimality conditions yield

$$hg(x)\frac{1-\theta_1}{\tau(\theta)} = V_1 - V_2 \tag{5.6}$$

and

$$hg(x)\frac{\theta_3}{\tau(\theta)} = V_2 - V_3. \tag{5.7}$$

In particular, we have $V_3 < V_2 < V_1$ and it suffices to find a positive lower bound for $V(x) - V_1$. We use Eqs. (5.6) and (5.7) to eliminate V_1 and V_3, respectively, from Eq. (5.5) and obtain

$$V(x) = hg(x)\tau(\theta) + \theta_1 V_2 + hg(x)\frac{\theta_1(1-\theta_1)}{\tau(\theta)} + \theta_2 V_2 + \theta_3 V_2 - hg(x)\frac{\theta_3^2}{\tau(\theta)}.$$

We then subtract Eq. (5.6) to obtain, after some algebra,

$$V(x) - V_1 = hg(x)\tau(\theta) + hg(x)\frac{\theta_1(1-\theta_1) - \theta_3^2 - (1-\theta_1)}{\tau(\theta)} = \frac{hg(x)}{\tau(\theta)} \geq \frac{h}{\sqrt{3}}.$$

The argument for the case where some component of θ is zero is similar and is omitted.
Q.E.D.

Having established an analog of property P for two- and three-dimensional problems, we discuss how it leads to efficient algorithms and estimate their complexity. The basic ideas are the same as for the shortest path algorithm of Section 4 and we only discuss the three-dimensional case.

Let $\delta = h/\sqrt{3}$. Let $Q_k = \{x \mid (k-1)\delta \leq V(x) < k\delta\}$ and $R_k = \cup_{i=0}^{k} Q_i = \{x \mid V(x) < k\delta\}$. Suppose that at some stage of the algorithm, we have computed $V(x)$ for all $x \in R_k$. We define $\overline{V}_k(x)$ to be equal to $V(x)$ if $x \in R_k$ and infinity otherwise. Let

$$\hat{V}_k(x) = \min_{\alpha} \min_{\theta \in \Theta} \left[hg(x)\tau(\theta) + \sum_{i=1}^{3} \theta_i \overline{V}(x + y_{\alpha,i})\right], \qquad x \in S, \tag{5.8}$$

where we are again following the convention $0 \cdot \infty = 0$. We then argue as in Lemma 4.1. If $V(x) \geq (k+1)\delta$, then $\hat{V}_k(x) \geq V(x) \geq (k+1)\delta$. If on the other hand $V(x) < (k+1)\delta$, Lemma 5.2 shows that for every i such that $\theta_i > 0$ we must also have $V(x + y_{\alpha,i}) < k\delta$ and therefore $V(x + y_{\alpha,i}) = \overline{V}_k(x + y_{\alpha,i})$. This implies that $\hat{V}_k(x) = V(x)$. Thus, we have computed $V(x)$ for every $x \in R_{k+1}$, and we are ready to start the next stage of the algorithm.

We implement the algorithm by using buckets, exactly as in Section 4, except that the "width" of each bucket is $\delta = h/\sqrt{3}$ instead of unity. The complexity estimate is essentially the same as in Section 4, because the underlying algorithmic structure is almost the same. Since each x has a bounded number of "neighboring points" $x + hy_{\alpha_i}$, a point x may move from one bucket to another and the value of $\hat{V}_k(x)$ may need to be recomputed only $O(1)$ times. Each time that $\hat{V}_k(x)$ is recomputed, we need to solve the optimization problem in Eq. (5.8). Following an approach similar to the one in the Appendix, this can be done with a finite number of operations, provided that square root computations are counted as single operations. Thus, the complexity estimate becomes $O(n)$ plus the number of buckets employed. The number of buckets can be bounded in turn by $O(L/\delta) = O(L/h)$, where L is an upper bound on $\max_{x \in S} V(x)$.

For the two–dimensional case, there are no essential differences, except that the bucket "width" should be $h/\sqrt{2}$. We summarize below.

Theorem 5.1: Let Assumption 5.1 hold and assume that square roots can be evaluated in unit time. Then, a solution of Eqs. (5.1)–(5.2) in the two–dimensional case, or Eqs. (5.3)–(5.4) in the three–dimensional case, can be computed in time $O(n + L/h)$, where L is an upper bound for $\max_{x \in S} V(x)$.

We now interpret the complexity estimate of Theorem 5.1 in terms of the original continuous trajectory optimization problem. We assume that the underlying cost function r (cf. Section 1) is bounded below by some positive constant. For a problem involving trajectories in $\Re^m$, the number of grid–points is $n = O(h^{-m})$, where h is the grid–spacing. On the other hand $V(x)$ should converge to $V^*(x)$, the cost–to–go for the original continuous problem, which is independent of h. In particular, the factor L in Theorem 5.1 can be taken independent of h. For $m > 1$, the term $O(n)$ is the doiminant one in the complexity estimate $O(n + L/h)$. We conclude that, as long as the problem data in a trajectory optimization problem are regular enough for our discretizations to be justified, we have algorithms whose complexity is proportional to the number of grid–points involved, which is the best possible.

VI. PARALLEL IMPLEMENTATION

In this section, we comment on the parallelization potential of the algorithm of Section 5 and compare it with the parallel implementation of relaxation methods. In order to avoid discussing the effects of architecture–dependent features, we frame our discussion in the context of an idealized shared memory parallel computer; similar results are possible for some message–passing architectures like hypercubes.

Let us concentrate on the computations required during a typical stage of the algorithm. Suppose, for example, that $\hat{V}_k(x)$ is available for all points x, so that Q_{k+1} can be determined. Let N_{k+1} be the set of points that have a neighbor belonging to Q_{k+1}. For every $x \notin N_{k+1}$, we have $\hat{V}_{k+1}(x) = \hat{V}_k(x)$ and no computation is required to obtain $\hat{V}_{k+1}(x)$. Thus, a high–level description of a typical stage of the algorithm of Section 5 is as follows:

1. Use the values of $\hat{V}_k(x)$ to determine the set Q_{k+1}.
2. Determine the set N_{k+1}.
3. For every $x \in N_{k+1}$, compute, in parallel, the value of $\hat{V}_{k+1}(x)$.

If a different processor were assigned to every point $x \in S$, then step 3 would be carried out in $O(1)$ parallel time. However, such an implementation would be wasteful because the processors associated to points $x \notin N_{k+1}$ would be idle; for most stages, the majority of the processors would be idle and the parallelization would be inefficient. In order to obtain an efficient implementation, it is important to use a smaller number, say p, of processors, certainly no more than the average size of N_{k+1}. Then, at each stage, we need to allocate more or less the same number of elements of N_{k+1} to each processor. Such load balancing can be accomplished by running a parallel prefix algorithm at each stage [L].† The running time of a parallel prefix algorithm is $O(\log p)$. Once the load of the different processors is balanced, the parallel time for that stage is $O(|N_{k+1}|) = O(|Q_{k+1}|)$.

Putting everything together, the total parallel running time is $O((L/\delta)\log p + \sum_k |Q_k|/p)$ $= O((L/\delta)\log p + n/p)$. By the argument in the end of the preceding section, L should be viewed as a constant independent of n. Then, for two–dimensional problems, the parallel complexity becomes $O(n^{1/2}\log p + n/p)$. With $p = O(n^{1/2}/\log n)$, the running time is $O(n^{1/2}\log n)$. A similar calculation shows that, for three–dimensional problems, the parallel running time is $O(n^{1/3}\log n)$, using $O(n^{2/3}/\log n)$ processors.

Note that no parallel implementation of the algorithm of Section 5 could have much better running time. This is because we have to deal with one bucket after the other and in

† The details of how to do this are somewhat uninteresting and fairly common in the parallel algorithms field; we therefore choose to omit them.

two (respectively, three) dimensions there will be $O(n^{1/2})$ (respectively, $O(n^{1/3})$) buckets. In addition, the proposed implementation is efficient, in the sense that the processor–time product is of the same order of magnitude as the serial running time.

It could be argued that the successive approximation algorithm is more suitable for parallelization because all points can be simultaneously iterated: using $O(n)$ processors, the parallel time is of the order of the number of iterations. However, the number of iterations cannot be less than $O(n^{1/2})$ or $O(n^{1/3})$ for two– or three–dimensional problems, respectively. We conclude that parallel successive approximation cannot be much faster than the algorithm described here in terms of running time, even though it uses a much larger number of processors. The number of iterations in the successive approximation algorithm can be reduced by using the Gauss–Seidel technique, maybe with some heuristics guiding the choice of the next point to be iterated, but the resulting methods are usually much less parallelizable.

VI. DISCUSSION

The Dial–like algorithm of Section 5 has the best possible order of magnitude of running time, namely $O(n)$. On the other hand, the constant factor is likely to be larger than the constant factor in the $O(n \log n)$ estimate for the Dijkstra–like algorithm of Section 3. Thus, it is not clear which algorithm would be better in practice. We expect that the Dial–like algorithm could be better for two–dimensional problems and fairly fine discretizations.

We also expect that the Dijkstra–like algorithm would significantly outperform the classical successive approximation algorithm. Successive approximation is likely to be competitive only if its Gauss–Seidel variant is used and if the points are sweeped in more or less the same order as they appear on optimal trajectories. In other words, successive approximation becomes competitive only if it manages to mimick the Dijkstra—like method.

We report here on some preliminary numerical experiments carried out by L. C. Polymenakos. The Dijkstra–like algorithm was compared with a reasonable implementation of the Gauss–Seidel (GS) successive approximation algorithm, in which states were scanned row–by–row. Test runs involved a square 100×100 grid. The costs per stage $g(x)$, $x \in S$, were chosen either at random or as quadratic functions of x. For problems involving no obstacles, the ordering of the states used by the Gauss–Seidel algorithm was reasonably well aligned with the general direction of optimal trajectories; still, the GS algorithm was slower by about a factor of 10–15. When several obstacles were introduced, forcing the optimal trajectories to go back and forth several times, the Dijkstra–like algorithm was faster by a factor of 50–300. Even though more extensive testing with a larger variety of problems is needed, these preliminary results suggest that the suggested algorithms are fast both in theory and in practice.

In this paper, we have not addressed the more general trajectory optimization problems in which the cost $r(x)$ also depends on the control variable u and is of the form $r(x, u)$. One–pass algorithms do not seem possible and genuinely iterative methods, like successive approximation, seem to be necesseary. Nevertheless, there are reasons to believe that similar methods that try to propagate "wavefronts" (or level sets) of the function V will be much better than naive iterative methods.

ACKNOWLEDGMENTS

The author wishes to thank Mr. Lazaros Polymenakos for carrying out the computational experiments mentioned in the last section.

REFERENCES

[AF] Athans, M., and Falb, P. L., *Optimal Control*, McGraw Hill, New York, 1966.

[B1] Bertsekas, D. P., *Dynamic Programming: Deterministic and Stochastic Models*, Prentice Hall, Englewood Cliffs, New Jersey, 1987.

[B2] Bertsekas, D.P., *Linear Network Optimization*, Prentice Hall, Englewwod Cliffs, NJ, 1991.

[BT] Bertsekas, D. P., and Tsitsiklis, J. N., "An Analysis of Stochastic Shortest Path Problems", *Mathematics of Operations Research*, Vol. 16, No. 3, August 1991, pp. 580–595.

[CD] Capuzzo Dolcetta, I., "On a discrete approximation of the Hamilton–Jacobi equation of dynamic programming", *Applied Mathematics and Optimization*, Vol. 10, 1983, pp. 367–377.

[CDI] Capuzzo Dolcetta, I., Ishii, H., "Approximate solutions of the Bellman equation of deterministic control theory", *Applied Mathematics and Optimization*, Vol. 11, 1984, pp. 161–181.

[CL] Crandall, M. G., Lions, P.–L., "Viscosity solutions of Hamilton–Jacobi equations", *Trans-*

actions of the American Mathematical Society, Vol. 277, No. 1, 1983, pp. 1–42.

[CLR] Cormen, T.H., Leiserson, C.E., Rivest, R.L., *Introduction to Algorithms*, McGraw Hill, New York, 1990.

[D] Dial, R., "Algorithm 360: Shortest path forest with topological ordering", *Communications of the ACM*, 12, 1969, pp. 632-633.

[FR] Fleming, W., and Rishel, R., *Deterministic and Stochastic Optimal Control*, Springer–Verlag, New York, 1975.

[FS] Fleming, W. H., Soner, H. M., *Controlled Markov Processes and Viscosity Solutions*, Springer–Verlag, New York, 1993.

[GR] Gonzalez, R., and Rofman, E., "On deterministic control problems: an approximation procedure for the optimal cost, I, the stationary problem", *SIAM J. on Control and Optimization,* 23, 2, 1985, pp. 242-266.

[KD] Kushner, H. J., Dupuis, P. G., *Numerical Methods for Stochastic Control Problems in Continuous Time*, Springer–Verlag, New York, 1992.

[L] Leighton, F. T., *Introduction to Parallel Algorithms and Architectures*, Morgan Kaufmann, San Mateo, CA, 1992.

[M] Mitchell, J. S. B., "Planning shortest paths", PhD thesis, Dept. of Operations Research, Stanford University, Stanford, California, 1986.

[MK] Mitchell, J. S. B., and Keirsey, D. M., "Planning strategic paths through variable terrain data", *SPIE Volume 485: Applications of Artificial Intelligence*, 1984, pp. 172-179.

[MP] Mitchell, J. S. B., and Papadimitrioy, C. H., "The weighted region problem", to appear in the *J. of the ACM.*

[S] Souganidis, P. E., "Approximation schemes for viscosity solutions of Hamilton–Jacobi equations", *J. of Differential Equations*, Vol. 59, 1985, pp. 1–43.

APPENDIX

We explain here how the problem (3.7) can be solved with a finite number of operations, if the evaluation of a square root is counted as a single operation.

Let us frst assume, without loss of generality, that $V_1 \geq V_2 \cdots \geq V_m$. Suppose that the optimal value of θ_1 is positive. It is then apparent from the structure of the problem (3.6) that the optimal value of θ_i is positive for all i. Then, by the Kuhn–Tucker conditions, there exists a scalar λ such that

$$\frac{\theta_i}{\tau(\theta)} + V_i = \lambda, \qquad \forall i. \tag{A.1}$$

We thus have

$$1 = \sum_{i=1}^{n} \frac{\theta_i^2}{\tau(\theta)^2} = \sum_{i=1}^{n} (\lambda - V_i)^2. \tag{A.2}$$

We can solve this quadratic equation to determine λ (this requires a square root computation). Furthermore, the relation

$$1 = \sum_{i=1}^{n} \theta_i = \tau(\theta) \sum_{i=1}^{n} (\lambda - V_i) \tag{A.3}$$

can be used to determine the value of $\tau(\theta)$. The value of each θ_i can be then computed from Eq. (A.1). If after doing all these calculations, we find that $\theta_i > 0$ for all i, then we have an optimal solution of the problem (3.7). If some θ_i is negative or zero, or if (A.2) has no real roots, then our assumption $\theta_1 > 0$ was erroneous. In that case, we can let $\theta_1 = 0$ and optimize with respect to the remaining variables. This is a problem with the same structure, but in one dimension less, and the same procedure can be used. By repeating these steps at most m times, the optimal solution of (3.7) will have been determined.