ALGORITHM FOR AUTOMATED GLOBAL OPTIMIZATION OF LOW AND HIGH THRUST SPACECRAFT TRAJECTORIES IN A TIME CHANGING ENVIRONMENT

Bostjan Eferl , 31 January, 2011
Eferl & co. , Pri soli 45, 2354 Bresternica, Slovenia, GSM: +386 31-465-162, bostjan.eferl@gmail.com

Abstract: *This paper describes an automated method for global optimization of spacecraft trajectories with an arbitrary thrust, which is therefore suitable for spacecrafts with low or high thrust propulsion. It finds optimal trajectory by finding the best connections between the points or small areas in the state-time-space. Trajectory consists of smaller pieces of trajectory connecting the points in the state-space. Points or small areas in space can be considered as nodes and pieces of trajectory can be considered as edges, therefore this method is related to graph theory. It is useful for optimizing trajectories that may include gravity assists, perturbations caused by other bodies and irregularities in gravity field,as well as aerodynamic forces, and other conditions. Constraints like minimum distance to central body, space and velocity constraints are simple to implement. Applications: ion drive, solar sails, launch vehicles, landers, mission design, optimization and modification of existing trajectories, formation flying, station-keeping, (real-time) trajectory correction due discrepancies of thrust from desired, refueling, etc. It can be used for trajectories with an arbitrary number of revolutions around the central body. It is the second and heavily modified version of my CASTRO algorithm – Computational Automated Space Trajectory Revolution Optimizer. Innovations were added to reduce the amount of computational resources required by some orders of magnitude.*

Keywords: *CASTRO algorithm,global trajectory optimization, low thrust trajectory, graph theory*

1. Introduction

CASTRO is the method for global trajectory optimization for the spacecrafts with an arbitrary (low or high) thrust, that I developed. More broadly speaking it works for any object with any kind of propulsion, taking into the account the changing environment and the environmental influences. It is based on the robotics algorithms I developed. Due to the special nature of the problem and enormous amount of information these algorithms have been heavily modified and some additional approaches were introduced. Most of techniques described in this paper can also be used on other problems or systems. It is useful for most astrodynamical tasks requiring finding and optimizing trajectories for spacecraft with arbitrary strong propulsion and can be parallelized to a high degree to run on a high number of computational machines. It is automated and doesn't require human assistance. CASTRO addresses the issue of lack of automated methods and is especially suited for optimizing trajectories in a time changing environment and where various constraints need to be implemented, techniques were introduced to deal even with trajectories with high number of revolutions.

Rather than a single monolithic method it can be seen as an algorithm, where central theme are states that can be represented by nodes and pieces of trajectories, that can be represented by edges. The task of finding the optimal trajectory is to find the best connection between the possible states, where connections(edges) represent the way(information) how two states are connected. There are several ways in which this can be done and a number of algorithms and modifications can be used to achieve that, which way is the most suitable depends on the thrust to mass ratio of the spacecraft, environmental influences, mission objectives, available time and computational and memory resources available and their tradeoffs.

The basic concepts behind CASTRO are quite simple, it basically rests on idea, that at different times spacecraft can be in different states, and that it's possible to connect those states i.e. how to stear the spacecraft from one to the other state,since those states can be connected, it's possible to apply fast i.e. polynomial time algorithms to search how to get from states representing the spacecraft at start to those representing the spacecraft at the mission's end in a most efficient way. By efficient it's most often meant by lowest propellant consumption. Because time, propellant and environmental conditions for the goal states that can be reached from the start states are known, these can be used to calculate or determine how good a particular goal point satisfies optimality criteria, for example mass to time ratio, state that best meets this criteria is selected and the path/trajectory leading to it is determined including in which starting point it has it's beginning. Environmental and other constraints are easy implementable by not allowing any connections to states pertaining to them, this also eliminates the need for storing non-reachable states. Because of large number of states, connections requiring vast computational resources, methods are introduced to reduce the number of states, while still retain accuracy, and also the methods to deal with more states that can fit into the memory. The idea that states represent states of an actual spacecraft can be dropped in some cases for performance reasons.

2. Premises and nomenclature

Working of the algorithm is best described, if concept of state-time-space is introduced. State-time space is a state-space with additional dimension time(sometimes instead of time it can be a sequence), where state space is a space where each designated point in it represent a state that a spacecraft might have at a specific time. State-space is a space, where each point represents state of a spacecraft. State of a spacecraft is simply a collection of parameters that are needed for completing the task of trajectory optimization. In a typical case these are position, velocity, time and mass or Kepler's parameters the spacecraft would have at some time had the thrust been turned off for an infinitesimally small amount of time, or some other parameters that provide sufficient information needed for the process of spacecraft trajectory optimization. Since spacecraft can be in different states at different time, additional dimension time is added to the state space thus forming a state-time -space.

Because complete information on each trajectory contains informations of all parameters of state time space, each trajectory has its equivalent trajectory(curve) in a state time space. Therefore trajectory optimization can be seen as a task of finding the curve in as state-time-space, such that spacecraft “flying on that path” would best satisfy the optimality criteria.

CASTRO relies on the fact that trajectory or curve in state-time-space can be seen as interconnection of states. CASTRO thus finds the optimal trajectory by finding the “best possible-available” connections between the possible states, thus trajectory consists of smaller trajectory(curve) segments joined together in the states. In one view connection(represented by an edge) is a set of points forming the curve connecting the two points, but equivalently it can also be seen as the necessary information on how to “drive” a spacecraft from one state to another. Example: If two states are (x1,y1,z1,vx1,vy1,vz1,m1,t1) and (x2,y2,z2,vx2,vy2,vz2,m2,t2), than that information would be how to apply the thrust(magnitude and direction) with respect to time to get from the first to the second state.

Since points or sometimes small areas of the state-time space can be seen as nodes and connections can be seen as edges to whom weights can be assigned CASTRO algorithm may be seen as closely related to graph theory and graph theory algorithms can be applied. A graph is formed in as state-time space in which paths(weighted or unweighted) exists, that connect starting points to other points that can be reached across the connections(edges). Another important feature that makes search for the optimal trajectory feasible is the existence of the fast - polynomial time algorithms for searching the path in the graph.

Edges of the graph are directed, they're pointing from states with lesser value of time parameter towards the states with higher values of time parameter. Situation(edge direction) is reversed, when techniques that search graph by going/propagating backward in time are used. Directed edges are necessary since any cycle in that graph would also represent traveling back in time. However cycling in the state-space are possible, their corresponding analogue in the state-time space would be a spiral. Edges are weighted and usually relate to mass, except in the case of solar sail or some other currently more exotic propulsions, where they can be unweighted.

When we search the optimal trajectory, we assume, that whatever trajectory it might be it has to start in one state and it has to end in one state. Since except specifically defined, the beginning/start of the trajectory is not known(launch window), and likewise except specifically defined the end of the trajectory in not known.

2.1 Nomenclature of points in space-time

Node. State, point: these terms can be used interchangeably, node represent state of the spacecraft, that was chosen as a point to and from which connections represented by edges can lead.

Initial nodes (also initial states or initial points) represent those states that are among the candidates for the beginning of the optimal trajectory. They may be isolated points or form areas.(when propagating forward in time each point in graph has its start in at least one starting point). They typically represent states permitted at a mission start(launch window), for example at a time belonging to a launch window, there exist at least one, but usually many points(nodes) that represent spacecraft at the beginning of the mission. In a launch-window time-interval, spacecraft can start at different times, from different location at different velocities and sometimes with a different amount of propellant, parameters describing such state of a spacecraft in state-time space are refered to as the starting points, initial states or the initial nodes. Every complete(not a trajectory piece) trajectory in a graph has its beginning in a state from the set of the initial states.

Goal nodes (also goal states or goal points) are the end points of the trajectory, representing those points that are among the candidates for being the end of the optimal trajectory. It's possible that some goal points are not reachable at all, because it's theoretically impossible for a spacecraft to reach them.

Edge is a graph theory equivalent of a trajectory connecting two nodes.

Weights are typically related to mass, but can also partially depend on other factors.

To search for optimal trajectory, we first need to designate the starting points or the area formed by the starting points and then find the least(or most) weighted path across the edges leading from that points to (every) point that may be the goal point, due to the nature of (most) search algorithms the least(or most) weighted path to every reachable node is determined in the process.

2.2 Cost, weighted path and optimality criteria

At least in general cost(the sum of weights on the path) assigned to a node and optimum are not the same thing. Most often we'll be interested how to get to a specific state belonging to a specific time with a minimum consumption of propellant. Optimality criteria for node is then computed or determined and can be a function of mass, time, state parameters, environment and its influences, sometimes it may include information of other reached nodes. It is assumed that nearly always more mass satisfies criteria better than less mass, when all other parameters are the same. It makes sense to determine optimality only to nodes reached in the process that are the candidates for goal nodes. Cost of the path through an incoming edge to a specific node in conjunction with costs of other paths leading to that node determine which of these paths are “allowed” to go through that node, in the search for the trajectory. Sometimes other factors may also influence that decision.

3. Basics

Though the idea that the trajectory consists of connected states is common to all versions of CASTRO it does differ in respects of how nodes(states) are designated and how connections are made, what the state parameters and weights of the edges are, and what search algorithm is used. Structure of the graph can also differ and can be either a net, tree or a collection of trees.
In this paper, though it's possible otherwise, it's assumed that state parameters are position components, velocity components, mass and time. Other will be briefly discussed (ratio mass to thrust, Keplers's parameters, logarithms of (inverse) values), but because nearly everything is analogous for other representations it's sufficient to base description mainly on one representation.

For beginning it's convenient to think that at least in general that to “every” node(state/point) many (i.e. enough) connections are leading from other nodes having other values of time parameter(lesser by forward propagation) and that from every point many outgoing connections connected to points having other values of time parameter than this point(higher by forward propagation). Such graph consists of large number of interconnected points and forms a net. If this net is dense enough, then it's possible to find a very similar trajectory to any other trajectory (that might exist) by connecting the states that lie very close to that trajectory(edges are close too) , thus if there exists an optimal trajectory(true optimum), than it is possible to find a trajectory running very close to it, since such a trajectory is very similar to the most optimal, it satisfies optimality criteria almost as good as true optimal trajectory. It might happen that the algorithm finds a trajectory with a very different characteristics, but this is not a problem, because it satisfies the optimality criteria nearly as well as true optimal trajectory, had that not been the case it would be possible to find a better trajectory running close to a truly optimal one (or possibly many most optimal).

Full state-time space have 8 dimensions. Though in principle it's possible to search in such a space, there are problems regarding feasibility of implementation of an algorithm that would search in such a space, because the amount of states and connections grows exponentially with the number of dimensions. To mitigate this, dimension representing mass is removed from state-time, that is done under the assumption, that of states with all parameters except mass being the same, the state with the highest mass(representing most propellant) is the most perspective, and that trajectories emanating from it are more favorably weighted than from those states with lower mass. By usually it's meant, that it might be that the “same state” with the lower mass can lead to a trajectory with “more favorable cost”, but we may assume that in general this is not an issue. An example of advantage of otherwise equal state having lower mass over that with higher would be a case where an opportunity of significant gain using gravity assist would be lost due to insufficient acceleration of spacecraft to “catch the opportunity”. In such cases this would mean that it's meaningful to have less propellant in the tank at start, since any propellant that should be later be used in uneconomic manner, would justify having less propellant at the start, since it usually makes little sense to propel the propellant that would latter be dropped or sacrificed by being spent in uneconomic manner.

Reference frame is not specifically defined by the algorithm itself, and can be arbitrary, but in general the “inertial reference frame” with its center in the body that most dominantly affects the motion of spacecraft could be considered as the most appropriate.

Since we have removed mass-dimension from the state-time-space, it's not possible to determine in advance how two states are connected and if they can be connected at all – because mass of the spacecraft is unknown(actually there is a way to do it). Therefore connections will be made dynamically in the process of search. For a classical case(not used in the standard version of CASTRO) of a graph search algorithm, a node that is not a state would be designated(just for convenience), only from that node edges that do not represent pieces of trajectories would be connected to all the starting states(nodes) having weight values corresponding to the mass of the spacecraft at the initial state. Since search algorithms in the process of search determine the cost, (a sum of edge weights on the path leading to that node) to the nodes they reach, and since that cost determines mass, it's possible to find the edges(connections) and their weights to other states from that state, and use that connections to continue the search process to reach other states(nodes) etc...

For further discussion in the paper an algorithm closely related to Bellman-Ford shortest path algorithm will be used. Since graph is directed and does not have a cycles, longest path algorithm can be used equally well and it's slightly preferred as it “directly favors higher mass”. Example of the shortest path would be to find the path with least consumption of propellant to nodes, and example of the longest path algorithm would be to find the path with highest mass(negative weights, except that of edges from the first - non-state node). Because both types of algorithm are useful for CASTRO the term “more favorable cost” will be used instead of “smaller than” or “higher than”.

Since connecting the two predetermined states is most likely non-trivial and time(processor) consuming process and it's probably even more slow to find the connection with close to lowest propellant consumption. Therefore it's convenient to designate small areas of state-space or state-time space instead of points, this is done because, if instead of trying to connect two predetermined states we start from one state and propagate(with some level of propulsion – variable or not) several trajectories from from one state for some time, than at least some end states of those trajectories will quite likely be the points belonging to some small areas. Likelihood of a particular trajectory “hitting” the small area (when areas don't intersect) is roughly proportional to the percentage of space(state/state-time) being occupied by this small areas. Since parameter values of such a small area are very similar, such area can be to a good approximation considered as a single point and due to its bigger than infinitesimally small size it has a “real chance” to be hit by several trajectory end states”. Most of search algorithms assign the cost to a node during the search process therefore the end-state of the edge belonging to the most favorably weighted path leading to that node can be assigned to the node, and that state whose parameter values are more precisely defined than parameter values of the small area representing that state, these parameter values than become the state(node) from which further search continues. It makes sense to keep only the edge (and its end state) leading to a particular node/state that belongs to the path with the most favorable cost, other can be discarded, that saves memory and in case when we assign precise state values to small areas during the search produces the graph in which each possible path represents trajectory whose accuracy is limited only by the accuracy of the propagation method, because all outgoing connections are propagated from a precisely defined state. If all but the one incoming edge belonging to the path with most favorable cost of all incoming edges leading to each particular node are discarded during the progression of search process, then there is only one path leading to each particular state, therefore the information about the whole “history” of path leading to each reached state can be carried and updated as search progresses. Graph in such case is a tree or a collection of trees.

4. Standard algorithm

Is a version of CASTRO based on the principles already described, is based on the search algorithm that may be considered as closely related to Bellman-Ford, is simple, intuitive, easy to implement and can be easily modified in such a way that only a small part of graph needs to be stored in memory at any given time.(longest path modification is preferred)
In standard version state-time-space is “sliced” in such a way that each slice represents a state-time space at some moment in time (or alternatively some time interval). Slices are densely populated with nodes i.e. states or small areas - that can be due to similar parameter values of their points (that is a consequence of their small size) sometimes treated as points.

Algorithm runs in steps, it starts at from the initial(starting) state/s, of the “slice” belonging to the lowest time. Connections are computed between initial(start) states and the states of the some other slice with higher time. Although connection may jump over the subsequent slices, the description will first concentrate on progression between adjacent slices. Algorithm starts by connecting the initial-state(s) with the lowest time to the states of the next “slice”, typically several connections are made, but it's also possible that no connection can be made. Initial states of the next slice are overridden, if there exists a connection(s) whose end state is the same as that state, but have more favorite cost Alternatively, if not overridden, then that initial state is not assigned in the next step, i.e. no connection is made if in the second “slice”, if there exists an initial state representing the spacecraft with more propellant(higher mass) with other parameters being the same as the state representing the end of connection or alternatively connection can be made, but is deleted in the next step, because it can be substituted by the initial-state with lower cost.

In the second slice we have the states, that were via connections reached from the first slice and the remaining initial-states, that were not overridden by the end states of the incoming connections from the previous slice. In the next step - from these states connections are made to the states of the next “slice”, again initial states of the next slice are overridden by the states , that are end-states of the connections with the same parameters but more favorable cost. This process of connecting the slices (in steps) continues, until the slice denoting the end of the time interval in which algorithm searches is reached.

In the process cost is assigned to each state to which connection leads. Because parameters of the states, most notable time, and cost are known, these can be used to determine how well these states/nodes meets optimality criteria, it makes sense to determine this only for those points that are among for goal-states. The one(theoretically there can be more) that best satisfies the optimality criteria is picked. (Optimality can be computed “on the fly”, only the best result so far needs to be stored.) Then the trajectory leading to that state is obtained by tracing the path back to the origin, which is one of the initial-states, that were not overridden(by other states). Tracing the path back is easy, the connection across which the path leads to the last-state(last st.=goal) is known, that connection leads to its preceding state which now becomes known. Since every state, except the first, have its preceding state, there is a connection leading to its preceding state, that is known, in such a way by following connections(backtracking) from the goal-state, across other connections trajectory can be obtained.

4.1 Modifications for performance

When there are more than one incoming connections to some node only the connection arriving at the path of a more favorite cost is left, others are discarded. That saves memory and is also advantageous in improving the accuracy, when instead of points nodes are represented by small areas. Other important aspect of this approach is that only one path, the one with most favorable cost leads to every node, that as a consequence means that history of the whole path can be known, by storing the path “on the fly” i.e. as the algorithm progresses from “slice to slice”, in this way the process of backtracking can be made redundant, as well there is no need to to explicitly store “slices ” and connections between them, because the necessary information how to reach reached/reachable states of the “current-slice” was carried along “on the fly” and is in every step assigned to the corresponding states. Sufficient information on how to reach particular state, does not have to include the whole sequence of states, but can be for example due to the fact that the reached state is known just the information, how the thrust was changing with time, knowing that information its possible to reconstruct the whole trajectory. For example: When time interval between “slices” is constant and thrust in that interval is tangential and constant for the duration of interval, then only the thrust magnitude and direction needs to be stored and carried along, that saves much more memory, than in case of storing the complete information. In cases when the thrust vary “complexly” in time and storing this information would require a lot of memory, then its possible to store just the sequence of states and reconstruct their connections later. If in such case we want to store a whole graph, than just information on which nodes are connected(may contain weights) is sufficient, since to obtain trajectory the process of connecting can be repeated.

Since connecting(with low propellant consumptions) two predefined states is likely nontrivial and slow process, approach that uses small areas of state-space (or state-time) instead of nodes is used. There are two varieties of this approach:

The first is that end-states of those trajectory-pieces propagated from initial-states, whose end-points lands in a small-area-nodes of the next “slice” and have least cost in that node(s) becomes the beginning-states of the trajectory pieces propagated from that small-are-node, that ensures continuity of the trajectory and produce highly accurate trajectories.

The second is that in the process, connections are made from one point that we designate in small-area-node(for example center), from that point connection is propagated and if its end-point “falls” in some small-area-node of the next “slice”, than we treat this trajectory-piece as connection between the two nodes, where we treat connection as connection between first designated point in small-area-node and a designated point in second small-area-node, although the connection almost newer ends in the designated point of the second small-area-node. By this approach trajectories are always propagated from designated points, this reduces the accuracy, because there are discontinuities, however these are small and reducing the size of small-area-nodes improves the accuracy, and repeating the search, this time with nodes only in the vicinity of the obtained trajectory in state-time-space, that way similar, slightly modified and more accurate trajectory is obtained, since the density of nodes in that part of the s.t.-space is higher. While this approach does not seem to have advantage over the first, it is useful for some techniques of trajectory search where either backward or forward and backward propagation(algorithm and/or trajectory) in time are used, this is due to the fact that when nodes are of finite size the first approach does not necessarily produce the same path(due to size of areas) when applied backward in time. Another case is when for (real time) application of correcting the trajectories using precomputed graph.

4. Memory management

Especially when many steps of algorithm are required, for example like when optimizing the low thrust trajectory around the massive central body, then storing the information about(related to) graph becomes difficult due to the large amount of information that needs to be stored. To mitigate this approaches that stores only a small amount of information at any given time can be used. This is achieved at a cost of somewhat larger search time, but that is heavily outweighted by reduced memory consumption. However end state of the optimal trajectory, its cost and optimality are already determined in the first pass of the algorithm, if the desired trajectory exists. Most notable methods especially useful for standard version are described below.

4.1 Memory management method 1

By this method only some “slices are stored” while leaving many slices between them not stored. The approach works by storing the selected slices in the first pass of the algorithm. In the second pass algorithm is propagated many times between the nearest(time) stored slices, starting at the states of the stored slice before the “slice” with the end-state of the most optimal trajectory and that slice, rebuilding the graph(or its information), its now possible to backtrack(or read) the piece of the most optimal trajectory running between these two “slices”, thus also obtaining the beginning-state of that trajectory piece of optimal trajectory. In this way the process of propagating the algorithm between pairs of slices is repeated, starting search for the piece of the optimal-trajectory between the last two relevant stored slices from the set of the stored slices, then continue the search between the one before the last relevant and the one before it, so gradually piecewise discovering the optimal trajectory, search is done when the piece with its beginning-state being the initial state is obtained, that initial state is the beginning-state of the optimal-trajectory, which doesn't necessary belong to the stored slice and usually don't.

Speeding-the second pass: Second pass of the algorithm does not have to be “a piecewise repeating of the first”, since the end-state of the optimal trajectory,which is also the end state of the last piece of trajectory, is known from the first pass, and when the time interval between stored-”slices” is not very large, than it's possible to estimate the set of states(or area) in which beginning state of some particular piece might be. This is possible due to the fact that in short time some parameters that can be assigned to the trajectory – like orbital energy, semi-major axis, etc... can not change very much, therefore it's possible to locate the area from where it is originating, and perform the search from only these states of the stored-“slice”, others can be discarded. This way only a small part of the state-time-space is searched.

Another way of speeding-up the second pass is by exploiting the fact that the same optimal-path can be obtained when applying the algorithm forward in time, and when algorithm is applied backward in time from the end state of the optimal trajectory. Due to the short time interval graph search can not expand to a wide area, if time interval is relatively short, that means low load on computational resources. To which state of the stored-”slice”(the on with lower time) the path obtained by going back in time connects is determined by the cost, cost are the same. This is possible when states in the process of search are treated as points, whereas when the process include states that are represented as small areas(small-area-nodes), this is not necessarily the case, however while the trajectory most likely won't be exactly the same, it's possible to quite accurately locate the area where the optimal-trajectory goes through, thus that leaves only a narrow “channel” in the state-time-space in which forward search needs to be applied.

4.2 Memory management method 2

This approach exploits the fact that if graph is such(or made such) only one path leads to each state(otherwise there may be very many paths), than its possible to carry the history of that path further with the path as the search(paths) continues. In that way it's possible to assign information on where the path were ”somewhere” on the path like where it was at some specific time. That way states that optimal trajectory goes through are obtained in some pass of the algorithm. On further passes only the search for trajectory joining these states needs to be searched, like in the first pass information on states where trajectory goes through on these smaller intervals is obtained. In this way more and more smaller and smaller intervals, which are passed by the algorithm faster and faster(due to shorter length), passes of the algorithm are applied until interval is so short that state on the beginning is directly connected via a single edge to the states on the end of that interval. Some intervals might be longer than others, so algorithm might find some parts of the optimal trajectory sooner than other. Since the number of intervals grow exponentially, there is a small number of passes needed, and since the in the first pass the largest number of edges needs to be traversed and that number goes smaller in subsequent pases, this increases the search time only for a relatively small factor. There is a tradeoff between search time and the required memory - more points where trajectory goes through per interval shortens the search time while increases the required memory. Time needed to find the trajectory grows only logarithmically with the number of steps, when using algorithm that progresses in steps.

When it's possible to apply the algorithm backward in time, it's not necessary to have the information on where the trajectory goes through at some time, instead first pass is needed to determine the end-state of the most-optimal trajectory, while in the later passes algorithm is applied forward in time and backward in time, at some time the two progressions meet, thus obtaining the state on the path the optimal-trajectory goes through. This process is than repeated until all parts of the optimal-trajectory are known.

4.3 Memory management method 3

This approach is still under investigation, but it seems that it should be possible to obtain the trajectory path in a graph, by starting at the “slice” containing the end point of the optimal trajectory, obtained in the pass of the algorithm running forward, since the paths connecting the end-points to the initial states are the same if the graph is searched from the end to beginning, and also use the other reached states in the slice for building edges leading to previous “slices” and use information on those edges to isolate those belonging to the optimal-trajectory-path. It's possible to work the path back, since the cost was determined in the first pass of the algorithm. We try to connect reached nodes, that have cost assigned, to the nodes, of the previous layer and assign possible costs to them. Of all the possible costs(due to many possible connecting edges) we chose the most favorable one, because if this node is reachable from initial node, than it was reached across most favorably weighted path, therefore the only candidate is the edge “producing” most favorable cost when going backward. In such a way it's possible to obtain path to the initial node, other paths going backward (not to initial nodes) and starting in the “optimal node” may exist, but if there is a lot of other reached nodes in that plane, this represent relatively little information due to overwhelming number of other paths starting from other nodes. Though this approach might pose some problems when nodes are small-areas, since there(due to non-infinitesimal size of node) path back might not be exactly the same as the path forward in time. This approach - reconstructing the graph backward using the information about reached nodes, and reconstructing the graph, and the same least weighted path backward may also be used for other graph search algorithms, but it's due to memory management requiring storing information pertaining to algorithm step most useful for “standard-algorithm”.

4.4 Memory management method 4

It's possible to search for trajectory by propagating the algorithm, and not storing the past slices, to the “slice” where the end-state of the optimal trajectory is, look for its incoming edge, store it, rerun the algorithm to find the incoming edge to the beginning-node of that edge and so on, running it for shorter and shorter intervals, until the initial-state is reached. This is possible, but relationship is between time of search and number of slices is highly nonlinear, relationship between the number of slices in the time interval and total number of steps performed is quadratic, that makes this approach hardly useful.

4.5 Memory management method 5

This approach is usefull for searching the trajectory using mostly the fast memory(like RAM). For this such group of nodes is stored into RAM whose great majority of propagated “forked” trajectories will “land” on nodes of the next slice having equal or similar parameter values, these reached states together with cost are then stored in “slower memory”, and those states reached belonging to minority whose parameters lies outside that group are also stored. Then another group of nodes of the same slice is selected and stored in RAM, the same procedure as mentioned before is then applied to them. This group of nodes selection, trajectory “forking” and propagation and storing reached nodes is done until there are no more nodes in the current slice. Before algorithm moves to work from the next slice, those states reached belonging to next slice need to be updated by the information of “minority nodes” when this results in more favorable cost of particular node. “Minority nodes” are actually “regular” nodes of some group reached from another, sometimes edge from another group arrive at the path having more favorable cost. Depending on space, information pertaining to “minority nodes” can be stored exclusively in RAM or they can be stored on disk. When procedure on one slice is done, the algorithm moves on to the next slice applying the same procedure and so on.

Another approach somewhat related to this one is, where in order to conserve memory group of nodes whose edges again like before in great majority lead to group of nodes with the same parameter values in the next slice. Every n-th(n-th time interval) slice edges that leads to nodes outside or almost outside this group are stored and can be even stored with reduced density of nodes. This for example in case of group nodes belonging to some interval of orbital energy means that we have stored the information for different times when at what cost the nodes corresponding to area around border of that interval were reached. The same procedure is then applied for other groups corresponding to other intervals, when slice corresponding to previously stored nodes is reached, these are used to update the nodes, if previously stored information have more favorable cost. In that way only one group of nodes is stored at the same time, while information pertaining to reached nodes around intervals border takes relatively small amount of memory. Instead of orbital energy reached height is also useful parameter. This approach works under the assumption that the chosen parameter of the trajectory is predominantly increasing or decreasing. For example to leave Earth's orbit the height of trajectory is predominantly increasing.

5. Additional

5.1 Time tags

Sometimes state due to its somewhat different time-parameter does not belong to ”some slice”, but it's still desirable to be treated as if it does belong to it, in this case time-tag is assigned to the nodes/edges when algorithm is such that information of the path can be carried along, this is important for example when precise time is needed for other reasons that depends on precise time – like perturbations, events like position of other objects, …, tagging in this cases enables precise propagation of trajectory. Time tagging can be used for example when dealing with highly elliptical trajectories, when most of the time one time step is appropriate, but due to higher speeds closer to the central body we may still assume that time step, although trajectory propagation is done in shorter steps, that introduces the time-error, but the impact of relatively few such steps on time accuracy is relatively small. Since time tags were used the accurate time parameters of states are known, in time time discrepancies may get quite large but are still relatively small compared to mission duration. Tags can also improve accuracy by slightly variating propagation time so that trajectory that otherwise wouldn't hit the small area node hits it at the expense of time accuracy. Time tagging is also very useful at the next approach, where revolutions of similar trajectories takes similar but still slightly different amounts of time, using time-tags, accurate time can be assigned to states.

5.2 States no longer representing the spacecraft

In order to reduce the necessary computational resources(number of calculations and amount of memory), both the number of states and edges can be reduced(state-space is much more spares with states), by assuming that the states and connections/edges no longer represent the actual spacecraft. This approach is useful when when trajectory around the central body requiring large number of revolutions is optimized. It exploits the fact that, if some trajectory can be considered a low-thrust trajectory having various amounts and directions of thrust applied on different parts of revolution around the central body, and if there are k revolutions observed(analysed) and each revolution may have different thrust on the “same part” of the revolution, then the state with almost the same parameter values would be reached, if from the same beginning-state another trajectory would be propagated, whose thrust on the same parts of all k revolutions is the average of thrusts applied on the corresponding parts of the all k parts of the original trajectory. Under the assumption that these are low-thrust trajectories, where thrust applied on some part of revolution causes only small changes in orbit, it's nearly the same end-state with almost the same cost is reached upon k revolution, if edges now represent transitions i.e. changes in state-parameters on the k passes of trajectory on the same part of revolution. One edge now represents k times the change in parameters on that part of revolution in one single orbit.
Alternatively it's also possible to assume that spacecraft is capable o k times higher thrust, that also produce k times higher parameter changes on corresponding parts of the revolution.

Trajectory for “real spacecraft” can be obtained from this trajectories, either by propagating the trajectory using k times smaller thrust at corresponding parts of revolutions for k times number of revolutions (lasting in reality approximately k times more time), or use the data on the obtained trajectory, to determine where in the state-time-space the optimal trajectory may go through, and than search that relatively small area of the state-time to find the optimal trajectory.

Since the “history” of how states belonging to end of the revolution were reached from the states of belonging the beginning of the revolution can be known(or vice versa),its also possible to obtain accurate trajectory “on the fly” - as the algorithm runs, by propagating “real trajectories” using that history, to obtain states belonging to a “real trajectory”, these have almost the same parameters as their equivalents obtained before, but since they represents the states of real trajectory they're most appropriate as the beginning points from which the algorithm further propagates, thus accurate states of “real trajectory” can be obtained in the first pass/run of the algorithm. Perturbations by this approach can be represented by the superposition of forces from different trajectories acting on the “same” part of them.

5.3 Jumping over more “slices”

In addition to connection running between only the adjacent “slices”, there can also be connections “jumping over” the “slices”, these are connections that represent trajectory-pieces with longer “flight”. They differ from others only by the fact that they need to be stored for more steps, and when the step of the algorithm reaches the “slice”, whose time matches the time parameter of that trajectory piece, then connection to a state/node in the slice can be made. However allowing many connections to jump from each slice quickly leads to large amount of information needing to be stored, mainly because as the time-length of the “longest” trajectory-piece increases, so does the number of edges from that “slice” and at the same time the number of “slicer” with “longer edges” also increases, (that has multiplicative nature). This can be somewhat mitigated by allowing longer edges to “jump” only from every n-th “slice” or so .

5.4 Interpolation

It might be possible to estimate the cost of states in the “slice” that were not selected as nodes in that “slice”, by interpolating between reached states, thus in principle it would be possible to calculate the way back to initial state.

5.5 Space instances

It is possibly to have several instances of algorithm running and allow the connections to “jump” from one instance of algorithm to another. For this it's usually convenient that time of both instance is synchronized. For example: One instance deals with the edges in the Earth's sphere of influence, while the other deals with edges in the Moons sphere of influence. When there is a piece running from Earth's sphere of influence to that of the Moon the end-state(edge) is passed to the algorithm instance that operates on the edges in Moon's sphere of influence, where it might connect to a node/state there and make further connections there.

5.6 Evolution of ellipses

To speed up the process of search and first find approximatively the characteristics of the optimal trajectory, an approach could be used, where there are nodes/states relatively sparsely populated in space time. Since these nodes on unpowered flight belong to elliptical trajectories , we can find how to optimally change parameters of ellipse in various ways in one(or more) revolutions, then it's possible to scale it by multiplying the changes by some whole(might be slightly off due to some correction) number k, indicating k revolutions to bring about k times that original change. It's further possible to divide the changes by some whole number, indicating scaled-reduced thrust, to further scale the changes. Such scaling and ellipse parameter modifications allows to roughly connect one node to another, since now modified ellipse have parameters at least roughly corresponding to that of the other node. That connection is represented by edge, its “time-duration” for low-thrust trajectories in strong gravity field is roughly a product of time duration of original ellipse and the number of revolutions. Using such techniques it should be possible to narrow the search space to a small region, and than apply some other approach to find the precise optimal-trajectory. If time discrepancies are not too large such an approach allows the inclusion of perturbations. It might be possible to obtain how to change the ellipses, not by just scaling the “vector whose components are changes”, but to obtain such a “vector”, which is still useful from smaller number of vectors.

5.7 Reduced dimensions method

Reduce the number of dimensions: In some cases it's possible to reduce the number of dimensions, solve related instance of problem for that space, to obtain approximative solution, use that information to obtain information where the trajectory will be in the space with non-reduced number of dimension, and using that information to obtain a smaller part of the space with non-reduced dimensions where the search for solution is done. - Examples: use of rotational simetry to remove dimension, solving problem for planar case and then use the results to find similar trajectory for slightly inclined case(modifying the trajectory), ...
For example: Instead of “full - planar trajectory” whose states have parameters (x,y,vx,vy,t) it's possible to reduce the case by one dimension by considering rotational symmetry thus having parameters (r,vr-tangential,vr-radial,t), one example of its use is the optimization of trajectory from starting from circular trajectory, that have to reach some other trajectory, for this case it's enough to search how to get to the goal trajectory and not care about its orientation. When the solution to this problem is found, it's possible to offset the time of departure of spacecraft at initial circular orbit to shift the whole trajectory for a desired angle. If there are perturbations acting then it's useful to calculate connections using full set of parameters, and then assign them corresponding r,vrtan,vrrad parameters in the reduced-space, thus connections pointing to very different nodes in the full-space can point to the same node in the reduced space. If the orbit in the reduced space is perturbed and if rotation is needed, then that information can be used to narrow the search to reduced part of original(full dimension) state-time-space and then to search for the optimal-trajectory there, this is done under the assumption, that these two trajectories have similar properties, that reduces the search only between connections that are among the candidates candidates.

5.8 Parallelization

Parallelization is another approach to reduce the search time. Even though the search process depends on the solutions obtained previously, it's possible to parallelize the algorithm to a large degree. The simplest case is to parallelize the computation of connections(or possible connections) since this takes much more time than the graph search and weight assighnemt itself, thus one machine perform graph search operations and leaves the calculation of edges to other machines. Another approach is to see each computational device as a group of nodes, that are able to make connections(i.e. communicate) to other computational devices. To improve performance and ensure that majority of computation is done between the nodes of the same group we can exploit the fact that(at least at low-thrust) that there are parameters like orbital energy, semi-major axis, …, which are changing only gradually in time, for high thrust spatial location can be criteria. That enables to distribute the algorithm on many machines in such a way that each machine s responsible only for calculations between the states belonging to say some interval of these parameters, when some connections-end state leaves that interval, it is linked(transfered) to another machine which is responsible for states belonging to that parameters. Parallelization is also useful, in cases, when some states are known and reconstruction of the graph between them is required..

5.9 Corrections due to discrepancies of thrust from desired thrust

Fidelity with which real spacecraft follows the CASTRO computed trajectory is affected to some extent, since thrust of the spacecraft have discrepancies from desired magnitude and direction, mass is also slightly affected. This is could an issue in real time scenarios, where recomputation, or some kind of control might take too much time or result too big propellant looses. To deal with that algorithm is propagated backward in time from the end-state of the optimal trajectory, where search-space in state-time-space is a relatively narrow channel around the optimal-trajectory. This process creates connections to states/nodes around the trajectory, that are linked to end-state of the optimal-trajectory, thus if spacecraft gets slightly of course, it finds itself at or in the vicinity of some of those states, since graph(or information about paths) is stored, it's possible to quickly read the path leading to end-state-node, so every time it gets of course it lands at some state for which path to end-node is known. It is desirable to generate only such edges in the process, that direct the spacecraft towards the trajectory. Even though spacecraft almost never hits the precomputed state, the state is near enough, and if its flown the way graph suggests the discrepancy will be very small. If it happens that mass is to large, than some propellant might be sacrificed by spending it "uneconomically", dropping it, if possible or compute the original optimal-trajectory(reference) using thrust that is always slightly smaller than it's actually possible to achieve, in that way there is some thrust reserve on real "flight".

5.10 Trajectory optimization for more missions, using stored graphs

It's sometimes possible to reuse the stored graphs for other missions that don't have lower thrust to mass ratio, than the one for which it was computed.

5.11 Another view - "How to occupy the space with states"

Alternatively to approach with small-area-nodes, it can also be considered how to populate the state-time-space densely enough but not too dense, since then too many almost the same states exists on the same place, which isn't necessary. In this view small-area-nodes are one of the convenient ways to do this. Example of this alternative view would be building a graph that is a tree or collections of trees, for not to get too dense, they're pruned according to cost and parameter values of reached states in such a way that only states with enough distinct parameter values may be connected, i.e. not many connected states, that are part of the tree or collection of trees, with similar parameters and cost may exist.

5.12 Method - "forking the trajectory, when it reaches the ray"

There is another search algorithm possible, that propagates trajectories from nodes for such an amount of time until they reach the nodes, from there trajectory is forked to several "forks", that propagate until they reach the nodes, and then they're forked, etc... Nodes store information about incoming trajectories in some time interval and the one having most favorable cost is "forked", other forks are canceled, if already in existence, in that way number of trajectory pieces doesn't grow exponentially.

5.13 Search method that operates with infinite number of edges

Since a state in one instance(time) of state-space can be mapped to infinite number of states in some of the next instance of state-space and those mappings represent some(small) area, and those states also maps to areas which mostly overlaps, however boundaries of that larger area are changing it seems possible to determine or infer, how boundaries are changing in time with sufficient accuracy. After some time the most optimal state is reached, and since each state(except starting states) is mapping of at least one state of the previous instance of state-space, they are across several mappings connected to some initial state. Path to it can be obtained in a way similar to back tracing.

5.14 Optimizing the trajectories with respect to angle of revolution

It's possible to optimize the trajectory with respect to angle of revolution around central body, for this(projection to ordinary space) states are placed on rays emanating radially from central body, and trajectories are propagated from states of one ray to that of another. It's also possible to categorize “time duration of edges” to a different time categories and use that for optimization, whose optimality criteria depends on time, that can give still satisfactory results.

6. Some applications

6.1 Interplanetary mission design

CASTRO is also useful for interplanetary mission design, since it's also the problem of finding the trajectory in state-time-space. It's desirable to increase the density of nodes around planets, since precision there matters more than in “empty space”, or alternatively, Sun, planets and other bodies can be represented by their own instances(running synchronously) of algorithm operating on their part of space, when edges leaves state-time-space of particular instance they're passed to another instance. Since all that optimization process is about is finding the right sequence of edges, and because trajectories represented by them are beside thrust influenced by environmental influences like gravity and aerodynamic forces, (radiation pressure), gravity assist maneuvers and aerobraking are automatically included in the process. Technique where particular edges represent multiple revolutions are generally not useful for trajectories around the Sun, due to low number of different “looking” revolutions, or often even less than one revolution, however they're useful, if mission requires many orbits around the planets.

6.2 Solar sails

When optimizing trajectories for solar sails things are a bit simpler in that sense, that there are no weights representing propellant necessary, since the mass of the spacecraft remains constant, consequentially one edge leading to a particular state at some time is all that is needed, no “competition” between the paths arriving across different edges is needed due to no cost. Alternatively weights and cost can be used for something else , for example radiation dose received. When the goal is to find the trajectory leading from one orbit to another in the least time, things can be further simplified, by the process that somewhat resembles the “slice operating” on itself i.e. in each step representing time, edges are connected from states reached in previous step, to those states that can be reached in this step, that were not reached in some of the previous steps, search is done when the connection reaches the desired state or states, trajectory is obtained simply by tracing the edges back to the origin. Similarly other graph search algorithms can be used, like Dijkstra's, but then weights representing time would need to be introduced, and they couldn't be used for something else, - some compromise could be made by having weights being representing as a function of two or more quantities, but that would affect optimality of solution.
Preliminary computational results shows that for transfers between orbits roughly corresponding to that of Earth and Mars computation takes few minutes(in Java programming language running on average computer), when we care only for time, radial distance from the Sun, radial and tangential velocities.

If the spacecraft is of a hybrid design i.e. one type of its propulsion is solar and other is of “a propellant consuming” type, than search is done like the search for solely “propellant consuming” type, with that distinction that edge might represent “purely solar” propulsion its corresponding trajectory piece, purely propellant consuming propulsion, or both types on the same piece(at the same time or not). Since the sun is free, the only cost affecting factor is propellant consumption, thus optimization process is virtually the same as in the “normal case”.

Since there may be no cost(and mass change) in solar sail trajectory optimization, the search can begin forward in time and backward in time concurrently, and trajectories belonging to paths connecting start states to goal states will met somewhere.

Concept similar to “slice operating on itself” might be used for propellant consuming propulsions, where forked trajectories would update nodes, if new cost is more favorable or even only when it's significantly more favorable. This would be useful for spacecrafts, with low propellant to total mass ratio, with some amount of thrust reserved to always assure the same maximal thrust to mass ratio, regardless of the amount of propellant. This would reduce amount of computation and memory, and probably still produce satisfactory trajectories, assuming that perturbations are not to strong. For not having to deal with discontinuities that may arise due to updates, memory management method 2 is applicable.

6.3 Station keeping

For station keeping, where an orbiting body must have a trajectory whose orbit must be keep within some narrow boundaries, search is done only on that part of state-time-space that corresponds to that limitations(boundaries). Algorithm is then propagated from initial state and performed until there are no more states reached whose masses would indicate that there is any spare propellant left. Usually the best trajectory is that, that connects the initial-state to the the last(last in time) reached state, since this is the trajectory with longest duration.

6.4 Formation flying

Formation flying is from optimization perspective similar to station keeping, spacecraft have to follow some reference spacecraft, or some reference trajectory, and have to stay at some distance or perform some specified maneuvers, that depend on time and have to stay in some defined boundaries. Narrow channel for each of the spacecrafts is “formed” in the state space and the search is done primarily on finding the trajectory that retains propellant for the longest possible time while flying in that channel.

6.5 Rendezvouses, flybys, landings, liftoffs

To rendezvous with some object, nodes representing that object at permitted times and locations for rendezvous are sufficiently densely placed in the state-time-space, algorithm then finds the path.
To improve precision states around object can be much denser than elsewhere. That improves precision in position and velocity. Flybys are obtained in a similar way only that these nodes are some distance away from the object and denote non zero velocities relative to the object. For landings state on the surface representing place of touchdown is designated. For a liftoff a state is designated on the surface, which is an initial state.

6.6 High thrust propulsion, launch vehicles, landers

Optimizing the trajectories with high thrust propulsion is done in basically the same way, except maybe that edges often represent impulsive maneuvers, or maneuvers where thrust is applied for short time compared to the whole time interval belonging to particular edge, and that unless thrust is applied in such a way(like firing in small impulses) that it causes very small “delta-v” changes in relatively long time, techniques where edges represent several revolutions on some small part of space are not useful. Another issue relating to high-thrust is that rocket motors often have limited number of restarts, this can be solved by having several graphs, each representing states after some number n firings of motor, after motor is fired edges are linked to states of the graph representing higher number of preformed firings. alternatively to that a compromise solution can be made, where propellant and number of firings on some path determine the cost or the decision criteria that is used to determine across which incoming edges outgoing edges will be “forked”.

7. References

[1] Eferl, B., http://sites.google.com/site/eferltheory/issfd22paperfigures , figures for this paper
[2] Eferl, B., “GOTC2-Eferl-unfinished.pdf”, 2nd Global Trajectory Optimisation Competition, 2006
[3] Eferl, B., “Eferl-GTOC2-poster.pdf”, 2nd Global Trajectory Optimisation Competition