APPLICATIONS OF GRAPHS IN TRAJECTORY DESIGN

Juan Arrieta
Jet Propulsion Laboratory, California Institute of Technology
4800 Oak Grove Drive, Pasadena, CA 91109, USA, +1 (818) 393–5611
Juan.Arrieta@jpl.nasa.gov

Abstract: *This paper proposes to formally adopt the graph data structure and its associated algorithms into the space mission design community. It introduces basic concepts of graph theory and their application to four problems in trajectory design: coordinate frame transformation, representation of gravity-assisted trajectories, modeling of multibody gravity, and calculation of sparse Jacobians and Hessians.*

Keywords: *Graph Data Structure, Gravity Assist Trajectory, Multibody Gravity Field, Sparse Differentiation*

1. Introduction

Graphs are mathematical entities capable of representing a binary relation between elements of a set. If the elements of that set were *people*, the binary relation *friendship between two people* could be encoded as a graph; if the elements of that set were *locations in the Solar System*, the binary relation *transfer between two locations in the Solar System* could be encoded as a graph. In practical terms they are data structures storing a set of *nodes* and a set of node pairs called *edges*.

Widely studied, they have found applications in many corners of human endeavor, and their pervasiveness has led to a strong theoretical foundation, a broad collection of algorithms, and the availability of standards and tools for their creation, manipulation, visualization, and storage.

In some domains the relationship between the problem and the graph data structure is instinctively analogous. In modeling a social network, for example, friends and friendships are naturally represented by nodes and edges. In other domains the applicability of graphs is less evident: who could find a relationship between graphs, coloring a map of the USA, and calculating a sparse Jacobian?

Reviewing existing space mission design literature, I found that the graph data structure (and some of its specializations, like trees and lists) is often relied upon for various aspects of space mission design. Examples include the internal representation of trajectories in optimization software such as CATO [1], Copernicus [2, 3], and SOCS [4], or the coordinate frame transformation subsystem of SPICE [5].

However prevalent graphs may be, their existence is seldom acknowledged and often implied as incidental, which I consider a hindrance to the full exploitation of their power and versatility: recognizing the graph as a *first-class data structure*—on the same level as matrices, for example—could bring important benefits to the mission design community, including:

- Generalization of fundamental algorithms such as frame transformations, and gravity field calculation
- Standardization of formats for external representation of spacecraft trajectories
- Exploitation of available libraries for the creation, analysis, storage, and visualization of complex graphs and networks
- Reduction in the *interpretation overhead* caused by the choice of different language to describe similar concepts

In this paper I will introduce some basic graph theory concepts including two fundamental graph algorithms: depth-first search, and breadth-first search. I will then highlight four case studies, developed during the implementation of a tool for the preliminary design of gravity-assisted trajectories, where I found graphs to be effective: coordinate frame transformation; representation of multi-segment spacecraft trajectories; multibody gravity field modeling; and sparse Jacobian and Hessian evaluation.

2. Basic Concepts and Algorithms

A graph $\mathcal{G}$ is a pair $(\mathcal{N}, \mathcal{E})$ where $\mathcal{N}$ is a finite set and $\mathcal{E}$ is a binary relation on $\mathcal{N}$ [6]. The set $\mathcal{N}$ is called the *node set* of $\mathcal{G}$, and its elements are called *nodes*; the set $\mathcal{E}$ is called the *edge set* of $\mathcal{G}$, and its elements are called *edges*.

When the binary relation represented by $\mathcal{E}$ is independent of the ordering between elements of $\mathcal{N}$ then $\mathcal{G}$ is called an *undirected graph*; otherwise it is called a *directed graph* or *digraph*. In the pictorial representation of a graph, nodes are often represented by circles, and edges by lines or arrows depending on whether the underlying graph is directed or undirected (cf. Figure **1**).

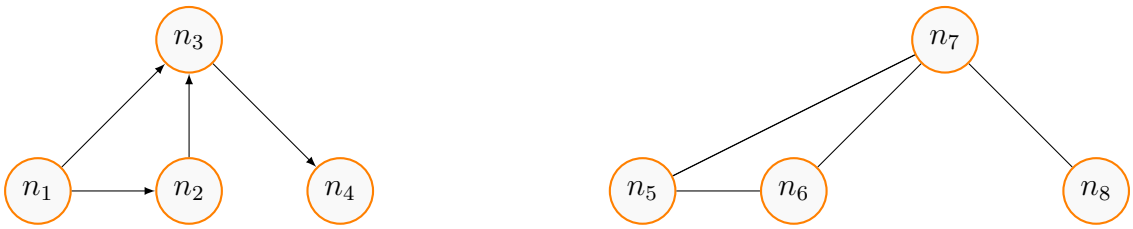

Figure 1. Graph nodes are often represented by circles, and edges by lines or arrows depending on whether the underlying graph is directed (left) or undirected (right).

If (u, v) is an edge in a graph $\mathcal{G} = (\mathcal{N}, \mathcal{E})$ we say that node u is adjacent to node v. We also say that (u, v) is *incident from*, or *leaves* node u and is *incident to*, or *enters* node v. For directed graphs the relation (u, v) is not necessarily symmetric, but for undirected graphs the edge (u, v) represents the same as the edge (v, u); in an undirected graph, self-loops are forbidden, so every edge consists of two distinct nodes. The cardinality of the set of nodes is denoted by $\|\mathcal{N}\|$, and the cardinality of the set of edges $\|\mathcal{E}\|$.

The *degree* of a node, $\deg(\mathcal{N}_k)$, is the number of edges incident on it; a node of degree 0 is *isolated*. In a directed graph the *outdegree* of a node is the number of edges leaving it, and the *indegree* is the

number of edges entering it; the degree is the sum of the *indegree* and *outdegree*.

In practice, graphs are generally represented by one of two data structures: the *adjacency matrix* representation encodes a graph by means of a $\|\mathcal{N}\| \times \|\mathcal{N}\|$ square matrix with elements (i, j) non-nil when there is an edge between nodes i and j; this representation can be wasteful in terms of memory when $\|E\| \ll \|\mathcal{N}\|^2$ because most elements of the matrix would be stored unnecessarily. In this case, a better representation may be an *adjacency list*, which encodes a graph by means of a list of $\|\mathcal{N}\|$ entries, with entry k containing a list of the $\deg(\mathcal{N}_k)$ nodes adjacent to $\mathcal{N}_k$. The choice of either representation will depend upon the structure of the graph and the kind of operations that will be performed.

A basic operation on a graph is the traversal (more commonly denoted *search*) of its nodes starting from a given location; the result of a search is a tree or a collection of trees[1] encoding the order in which the nodes were discovered. Below I outline two basic search algorithms that often serve as building blocks for other algorithms.

The first algorithm is *breadth-first search* (BFS), which visits all nodes adjacent to the current node before visiting any of their adjacent nodes. In this manner, BFS *fans out* from the starting node and continues by following edges incident on nodes as close to the starting node as possible [7]. When implemented as in Algorithm 1, BFS associates a *parent*, π, and *distance*, d, to every node found by the search; every reachable node will be visited exactly once. The resulting tree (called the breadth-first tree) has the remarkable property that it encodes the minimum distance path[2] between the node which originated the search and all nodes visited by the search.

The second algorithm is *depth-first search* (DFS), which selects a frontier edge incident on the most recently discovered node of the tree grown so far. When that is not possible, the algorithm backtracks to the next most recently discovered node and tries again. This process continues until all nodes reachable from the node that initiated the search are discovered. When implemented as in Algorithm 2, DFS associates a *parent*, π, and two *timestamps*: d recording when node N_k is first discovered, and f recording when the search finishes examining N_k's adjacency list. These timestamps have important applications. For example: they can be used to sort the nodes in a graph encoding precedence relationships in a manner such that pre-condition constraints are satisfied[3].

3. Coordinate Frame Transformation

A coordinate frame encodes a known orientation in space. For example: the International Celestial Reference Frame (ICRF) defines an inertial orientation relying on precise equatorial coordinates of extragalactic radio sources [9]. A vector specified in a given reference frame can be expressed relative to another via a frame transformation (often a rotation matrix).

A given study may entail dozens of reference frames, thus requiring the ability to find the *frame*

[1] Called a *forest*.

[2] In this context *distance* refers to the number of edges between nodes. Other algorithms, such as the one due to Dijkstra [8], are available for graphs with alternative definitions of distance.

[3] This arrangement is known as a *topological sort*.

Algorithm 1 Breadth-First Search [6, p. 595]

```
1:  procedure BFS(G, s)
2:      for each node u ∈ G except s do
3:          u.color ← WHITE
4:          u.d ← ∞
5:          u.π ← NIL
6:      s.color ← GRAY
7:      s.d ← 0
8:      s.π ← NIL
9:      Q ← ∅
10:     ENQUEUE(Q, s)
11:     while Q ≠ ∅ do
12:         u ←DEQUEUE(Q)
13:         for each node v ∈ G.adj[u] do
14:             if v.color = WHITE then
15:                 v.color ← GRAY
16:                 v.d ← u.d + 1
17:                 v.π ← u
18:                 ENQUEUE(Q, v)
19:         u.color = BLACK
```

Algorithm 2 Depth-First Search [6, p. 604]

```
1:  procedure DFS(G)
2:      for each node u ∈ G do
3:          u.color ← WHITE
4:          u.π ← NIL
5:      t ← 0
6:      for each node u ∈ G do
7:          if v.color = WHITE then
8:              DFS-VISIT(G, u)
9:  procedure DFS-VISIT(G, u)
10:     t ← t + 1
11:     u.d ← t
12:     u.color ←GRAY
13:     for each node v ∈ G.adj[u] do
14:         if v.color = WHITE then
15:             v.π ← u
16:             DFS-VISIT(G, v)
17:     u.color = BLACK
18:     t ← t + 1
19:     u.f ← t
```

transformation path that will transform between the current frame and a desired frame. As illustrated in Figure **2**, it is possible to encode all available frame transformations in a *frame transformation graph*, where a node denotes a given coordinate frame, and an edge exists between two nodes if and when[4] there is a known frame transformation between them.

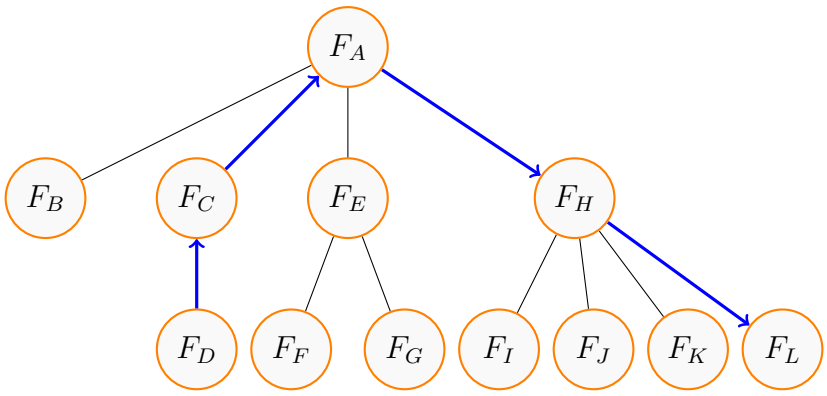

Figure 2. **A frame transformation graph can encode the available frame transformations. A breadth-first traversal starting at the desired frame (F_L in this case) and ending at the current frame (F_D in this case) can yield a frame transformation path containing the minimum number of transformations ($F_D \rightarrow F_c \rightarrow F_A \rightarrow F_H \rightarrow F_L$, highlighted in blue).**

With frame transformations encoded as a graph, it becomes straightforward to devise an algorithm capable to find the optimal frame transformation path between two nodes (cf. Algorithm 3) based on breadth-first search.

The properties of the breadth-first algorithm guarantee that a path will be found if it exists, and it will contain the minimum number of frame transformations; the time complexity of the search is $O(|N| + |E|)$ and the space complexity is $O(|N|)$. If the graph is of constant structure[5] it is possible to store the breadth-first tree rooted in a given node; any frame transformation relative to that node will be available at once without needing to perform a search. Furthermore, the *result* of transforming between inertial frames can be stored for later reuse.

It is tempting to encode the frame transformations using a *tree* instead of a *graph*[6] . However, such encoding imposes an artificial taxonomy and various conceptual limitations including: frame transformations have a *base* or *parent* frame, and are the base or parent of another; a *root* frame is at the top of the taxonomy; frame transformations must be eventually connected (no *islands* are allowed); cycles are handled as special cases. After imposing these limitations, one is left with an ad hoc data structure for which new algorithms have to be studied, developed, and tested.

[4]Some frame transformations are time dependent, and are not defined for every time instant.

[5]Which is the case for time-invariant coordinate transformation graphs.

[6]In fact, leading space mission analysis software libraries offer frame transformation facilities which are implemented in terms of trees.

Algorithm 3 Given a frame transformation graph $\mathcal{F}$, returns the optimal transformation path from frame f to a different frame t if one exists.

```
function FRAMETRANSFORMATIONPATH(F, f, t)
    for each node u ∈ F except t do
        u.color ← WHITE
        u.π ← NIL
    t.color ← GRAY
    t.π ← NIL
    Q ← ∅
    ENQUEUE(Q, t)
    while Q ≠ ∅ do
        u ←DEQUEUE(Q)
        for each node v adjacent to u do
            if v.color = WHITE then
                v.π ← u
                if v is f then
                    return BUILDPATH(v)
                else
                    v.color ← GRAY
                    ENQUEUE(Q, v)
        u.color = BLACK
    return NIL
function BUILDPATH(u)
    P ← u
    π ← u.π
    while π ≠NIL do
        APPEND(P, π)
        π ← π.π
    return P
```

4. Representation of Trajectories

The design of gravity-assisted trajectories for missions such as the Voyagers, Galileo, Cassini, and the proposed Europa mission often entails a complex piece-wise construction process aiming to *join* various flybys together. A mission designer typically starts from given incoming hyperbolic conditions at some body, generates various possible subsequent flybys, selects a specific one, and continues in this manner until a satisfactory multiple-flyby trajectory has been created. The manner in which subsequent flybys are generated depends upon various *transfer strategies* such as resonant, non-resonant, and π-transfers; these can be subject to different modeling considerations, such as the fidelity of the gravity field or the use of maneuvers. The piece-wise construction of trajectories is not limited to those relying on gravity assists; it can also be as a consequence of the numerical methods used for their design and optimization.

It seems natural to represent spacecraft trajectories as graphs: a set of locations in the Solar System

and a binary relation between them (a transfer between two locations in the Solar System). In fact, it is standard practice to *divide* spacecraft trajectories, and the leading software for trajectory optimization such as MALTO, CATO, Cosmic, SOCS, and Copernicus are all based on similar concepts: *break points* and *control points*; *nodes* and *stages*; *constraints* and *phases*. The relation of these concepts to graphs, however, is often secondary and frequently aggregated in terms of *lists*, as illustrated in Figure **3**.

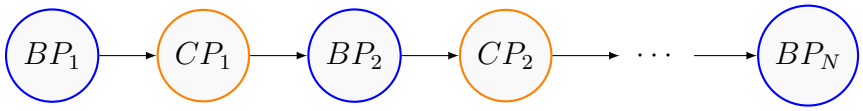

Figure 3. **Representation of a trajectory as a list of two different kinds of nodes: *break points* (BP_k) and *control points* (CP_k).**

The list arrangement is rigid because it entails a linear, sequential arrangement of events of specific types[7], thus being limited to a single trajectory that can be *extended from the last element*; it also imposes conceptual barriers because it encodes the transfer information *implicitly* in the nodes.

In contrast, the *graph* representation can offer a more cohesive methodology for representing different phases of a trajectory as a directed graph where the nodes denote the junction points between different phases of the trajectory and the edges represent the dynamical model joining the nodes; both nodes and edges can contain arbitrary information (cf. Figure **4**).

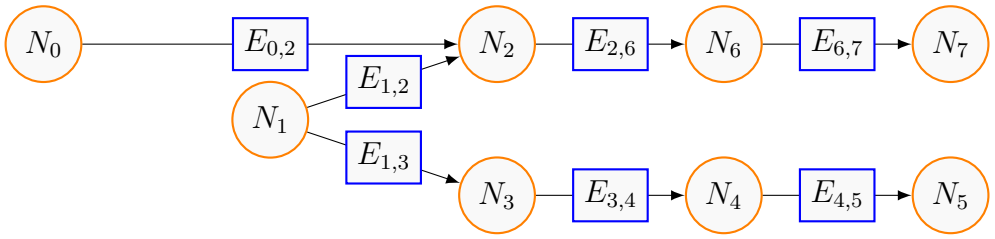

Figure 4. **Representation of a trajectory as a graph, where nodes denote the junction points between different phases of the trajectory, and edges represent the dynamical model joining the nodes.**

To illustrate the flexibility of the graph representation, consider the incremental design of a gravity-assisted trajectory based on the patched-conic approximation [10, 11]. The graph can represent available ballistic, multiple-flyby trajectories centered on a planet such as Jupiter; a node contains a flyby body identifier, an epoch, and an incoming hyperbolic excess velocity vector $\boldsymbol{v}_{\infty i}$; an edge contains references to its source and destination nodes, and an outgoing hyperbolic excess velocity vector $\boldsymbol{v}_{\infty o}$ This representation enables the implementation of complex behaviors such as the generation of nodes in either depth-first or breadth-first manner; the isolation of *tours* within the graph as a list of edges; and the *serialization* of the graph to various standard formats for storage and visualization.

[7] A *control point* followed by a *break point*, for example.

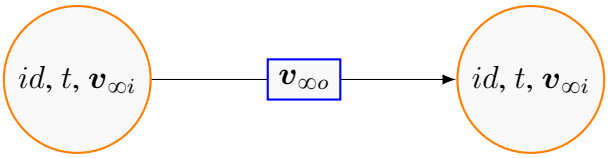

Figure 5. The incremental design of gravity-assisted trajectories can be represented by nodes containing incoming flyby conditions, and edges representing outgoing hyperbolic excess velocity. In this manner, various outgoing trajectories can be associated with a given node, and all available transfers can be considered simultaneously.

In the graph representation, the edges represent the transfer itself; it is not possible for an edge to exist unless it has both an incoming and an outgoing node. For this reason, the information about the transfer joining any two nodes is fully available in the edge joining them[8]. For example: the time of flight is obtained from the difference between the epochs of the destination and source nodes; a trajectory tabulation can be obtained by means of propagating the state derived from the incoming and outgoing $\boldsymbol{v}_\infty$ vectors (available via the source's $\boldsymbol{v}_{\infty i}$ and the edge's $\boldsymbol{v}_{\infty o}$); the minimum time-of-flight path to a given body or condition can be obtained from a breadth-first traversal starting at an arbitrary node.

Another advantage of the graph representation is that its storage and retrieval can be handled in terms of well-understood techniques. For example, a basic `JSON` representation of a trajectory graph could look similar to the following text:

```
"graph" {
  "nodes" : [
    {"node_id":1, "body_id":"Europa",   "t":x, "vinc":[x, y, z]},
    {"node_id":2, "body_id":"Io",       "t":y, "vinc":[x, y, z]},
    {"node_id":3, "body_id":"Callisto", "t":z, "vinc":[x, y, z]},
    ... more nodes...
  ],
  "edges" : [
    {"edge_id":1, "src_id":1, "dst_id":2, "vout":[x, y, z]},
    {"edge_id":2, "src_id":2, "dst_id":3, "vout":[x, y, z]},
    {"edge_id":3, "src_id":2, "dst_id":4, "vout":[x, y, z]},
    {"edge_id":4, "src_id":3, "dst_id":5, "vout":[x, y, z]},
    ... more edges...
  ],
  ... more graph data...
}
```

Notice that, unlike in the naïve storage of a list, the order in which nodes and edges are stored is not important. It is possible to build all nodes first (in any order), and edges second (once nodes are available). The `node_id` and `edge_id` fields are illustrated as integers, but an arbitrary identifier can be used instead. Detecting structural errors or peculiarities in the graph is straightforward, as

[8] And, of course, relying on the access to databases for parameters such as μ, or planetary and satellite ephemerides.

standard algorithms (again, based on breadth-first and depth-first searches) are available that can detect cycles, islands[9], edges referring to a non-existing node, or nodes without associated edges.

Given that the graph format accommodates for additional information to be associated to nodes and edges, it now becomes possible to consider more complex transfer strategies without modifying the underlying data structure. For example: ballistic transfers in the Saturnian system tend to be inaccurate due to the effects of Saturn's J_2[10]. In this case, it is desirable to add a maneuver between two flybys before proceeding any further. In this case, a Δv vector executed Δt time units from the source flyby time could be added to some edges to denote maneuver locations for non-ballistic transfers, as illustrated in Figure **6**. Finally, because the graph format contains all the information

Figure 6. Some edges can contain additional information, such as maneuver information encoded as a Δv vector executed Δt time units from the source flyby time, and enable non-ballistic transfers to be considered without having to modify the underlying data structure.

simultaneously, it is possible to investigate various paths (that is: specific trajectories) without having to commit to one in particular as illustrated in Figure **7**.

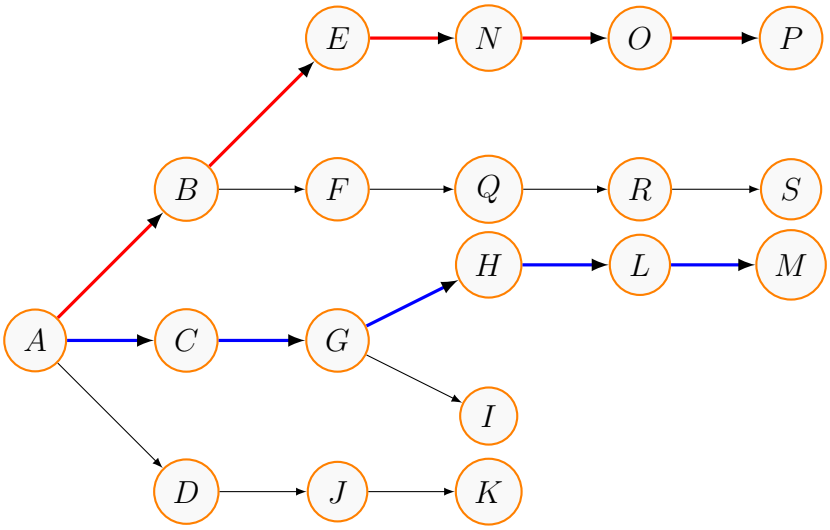

Figure 7. Simultaneous consideration of various paths within a given session: one may be interested in the main path (red), or a promising alternative (blue). While the other paths are not being considered, their information is still available and can be considered at any time, or deleted from the graph.

[9] Connected components, in graph theory language.

[10] All patched conic trajectories are inaccurate to some extent, but their inaccuracies can often be removed by a high-fidelity trajectory optimizer. On the other hand, the effect of Saturn's J_2 can be so large that the resulting patched-conic approximations are *incorrect*, and cannot be converged to a full fidelity solution. A similar situation arises in the presence of untargeted close approaches to other satellites or planets.

5. Modeling Multibody Gravity Using Undirected Graphs

When the dynamical model of a point mass incorporates the gravitational disturbance of more than one natural body, it is often necessary to *adapt* the calculation of the disturbance acceleration. For example: it is reasonable for the simulation of an Earth orbiter to consider the perturbations due to the Jovian system taken as the barycenter of Jupiter and the four Galileans; this calculation entails one—and not five—ephemerides evaluations. Similarly, it may be reasonable to consider the Moon's distributed potential, but less reasonable to consider, for example, Saturn's oblateness. The decision of what gravity sources to treat in what manner at a given time is delegated to a *gravity model*, which can also be used to determine the appropriate integration center. Depending on their distance and potential, the various gravity sources acting on the point mass can be treated as barycenters, point masses, or distributed masses; the distinction can be important both for numerical and performance reasons.

From the numerical perspective, the adaptation is necessary due to the large dynamic range of the distances involved. Consider, for example, the numerical integration of an EVEEGA[11] interplanetary trajectory towards Jupiter. In an imaginary world of infinitely precise computer arithmetic, any frame and any *center of integration* would yield exactly the same result. In the real world of limited precision, a center should be chosen to reflect the body exerting the largest gravitational influence at any given instant[12].

From the performance perspective, in the majority of cases and past a certain distance, it makes no practical difference to calculate separately the individual contributions of each satellite in a planetary system. It often suffices to augment their planet's gravitational parameter by the sum of their individual masses, and treat this conglomerate as a single point mass located at the barycenter.

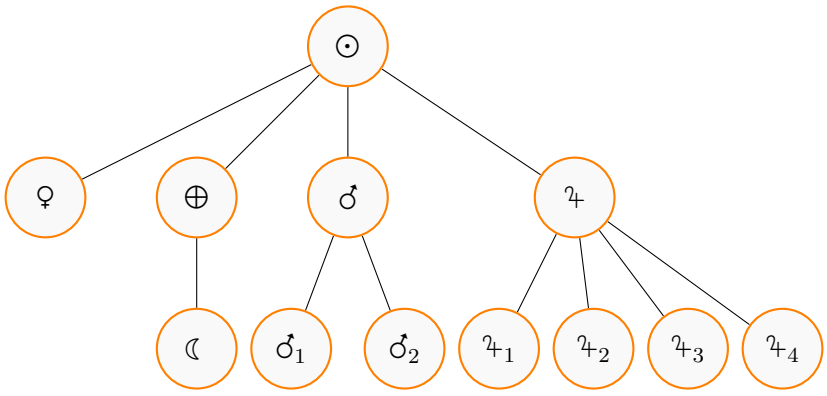

Figure 8. A graph representation of a multigravity model containing the Sun; Venus; Earth and Moon; Mars, Phobos, and Deimos; and Jupiter, Io, Europa, Ganymede, and Callisto.

The adaptation can be automated by representing the gravitational system as a graph connecting the bodies their satellites, as illustrated in Figure **8**. This representation[13] is able to *recenter* itself based

[11] An interplanetary trajectory with gravity assists from Earth, Venus, Earth, and Earth again.

[12] One would not choose the center of the Sun to investigate with precision the motion of Charon around Pluto!

[13] It was inspired by the manner in which gravity is modeled in the MONTE library using k-ary trees, structurally

on the current position, and notify when the point mass has entered or left a gravitational region such as the sphere of influence or the oblateness sphere[14].

Starting at the current center (which can be any node in the graph), the aggregation is a BFS with the following set of rules:

- If the point mass is within the sphere of influence of the visited node, mark the visited node for *central aggregation*.
- If the point mass is not within the sphere of influence of the visited node, mark the visited node for *barycentric aggregation*. Also mark it BLACK, and skip ENQUEUE (to stop BFS from visiting its adjacent vertices).

In the previous example, assume the point mass is within 250 km of Europa. The algorithm, visualized in Figure **9**, would first visit Jupiter and mark it for central aggregation; then visit Io, Ganymede, and Callisto; mark them for barycentric aggregation, and stop further traversal. Then it would discover the Sun and mark it for central aggregation. Finally, it would discover Venus, Earth, and Mars; mark them for barycentric aggregation, and stop further traversal. At this point there are no more nodes to visit and the search stops. Another example is illustrated in Figure **10**, where the center of integration is the Moon, Earth and Sun are centrally-aggregated, and the remaining planets are aggregated as barycenters.

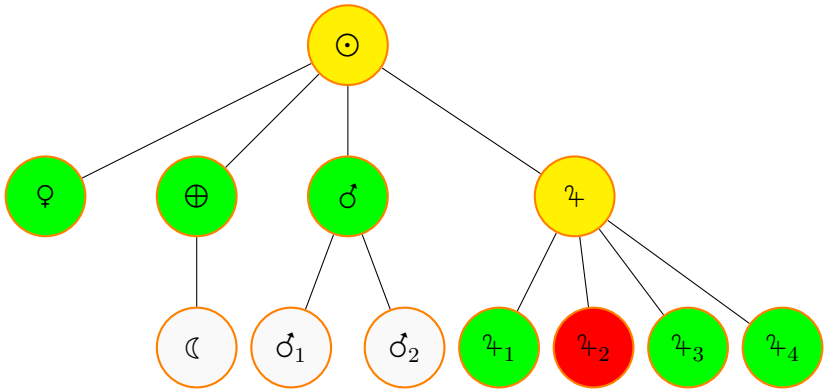

Figure 9. The result of the algorithm on a point mass within 250 km of Europa (the integration center, marked in red). Jupiter and the Sun are centrally aggregated (marked in yellow), whereas Venus, Earth, Mars, Io, Ganymede, and Callisto are baricentrically aggregated (marked in green). When a body has no satellites, barycentric aggregation is equivalent to central aggregation. Notice that the search did not *find* the Moon, Phobos, and Deimos; they will be considered during the barycentric aggregation of their parents.

6. Graph Coloring for the Evaluation of Sparse Jacobians and Hessians

The *graph coloring problem* consists of assigning a color[15] to every node in a graph in a prescribed manner. For example: *two adjacent nodes must have different colors* denotes a distance–1 coloring

identical but implemented differently.

[14]The sphere within which the disturbances due to a distributed mass are non-negligible.

[15]The term *color* is an abstract concept used to denote any distinguishable property.

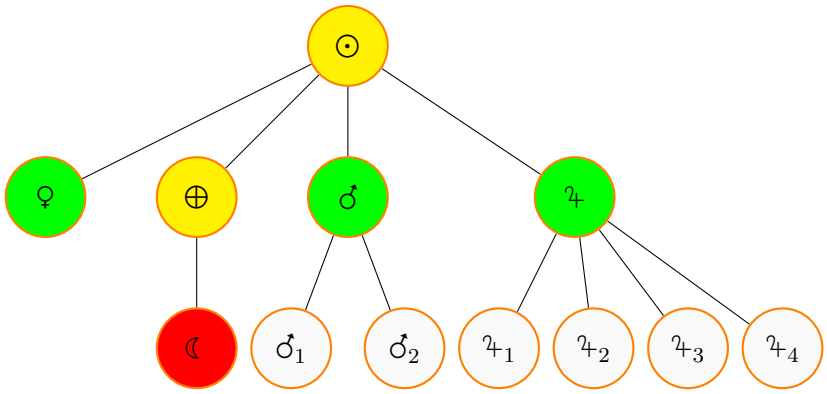

Figure 10. **Another example considering a Lunar orbiter. Earth and Sun are centrally aggregated (yellow), and Venus, Mars, and Jupiter are barycentrically aggregated (green). The satellites of Mars and Jupiter will be considered by the aggregation of their corresponding planet.**

problem; *any two nodes separated by exactly two edges must have different colors* denotes a distance–2 coloring problem[16]. The determination of the chromatic number $\chi(\mathcal{G})$—the minimum number of required colors—for an arbitrary graph $\mathcal{G}$ is known to be an NP-hard problem [7].

The relationship between graph coloring techniques and the evaluation of sparse Jacobians and Hessians traces its roots to the work of Curtis, Powell, and Reid [12], who noted that in approximating the Jacobian $\boldsymbol{J}$ of a vector function $\boldsymbol{f}(\boldsymbol{x})$ via finite differences, considerable savings in the number of function evaluations are possible if $\boldsymbol{J}$ has a large number of elements that are known constant. For example, when $\boldsymbol{J}$ is sparse with a known sparsity pattern, a group of columns can be determined with one perturbation if no two columns in such group have a non-zero in the same row.

Coleman and Moré [13] were the first ones who noticed that the problem could be attacked from the perspective of graph coloring: it is possible to reduce the number of function evaluations required to approximate a sparse Jacobian or Hessian if its columns are separated into *structurally orthogonal groups*, and finding such groups is equivalent to coloring a graph[17]. Later work by Gebremedhin, Manne, and Pothen [14] extended Coleman and Moré's approach by introducing an efficient distance–2 coloring algorithm based on the variable-dependency graph and, most importantly, extending the coloring approach to automatic differentiation.

As an example, consider the differential equations modeling the two body problem

$$\frac{\mathrm{d}^2\boldsymbol{r}}{\mathrm{d}t^2} = -\mu\frac{\boldsymbol{r}}{r^3} \tag{1}$$

[16]There are many applications of graph coloring techniques in practical engineering problems. For example: the assignment of frequencies to radio stations must be done in a manner such that no two radio stations with overlapping broadcast area transmit in the same frequency. In this case, a graph is created with radio stations at the nodes and an edge linking those with overlapping broadcast area; the colors correspond to radio frequencies, and the spectrum authority is interested in minimizing the number of allocated frequencies.

[17]From which it follows that minimizing the number of function evaluations needed to estimate sparse Jacobians and Hessians is equivalent to finding $\chi(\mathcal{G})$.

and observe that their Jacobian can be separated into three groups (red, green, and blue) of structurally-orthogonal columns:

$$\text{structure}(\boldsymbol{J}) = \begin{bmatrix} 0 & 0 & 0 & \partial_{v_x} & 0 & 0 \\ 0 & 0 & 0 & 0 & \partial_{v_y} & 0 \\ 0 & 0 & 0 & 0 & 0 & \partial_{v_z} \\ \partial_{r_x} & \partial_{r_y} & \partial_{r_z} & 0 & 0 & 0 \\ \partial_{r_x} & \partial_{r_y} & \partial_{r_z} & 0 & 0 & 0 \\ \partial_{r_x} & \partial_{r_y} & \partial_{r_z} & 0 & 0 & 0 \end{bmatrix}. \tag{2}$$

Given a nominal value, one could evaluate the six columns in the Jacobian using three (as opposed to six) additional function evaluations: perturbing red (r_x, v_x, v_y, and v_z), then green (r_y), and finally blue (r_z). The distance-2 coloring visualization of the variable-dependency graph is presented in Figure **11**.

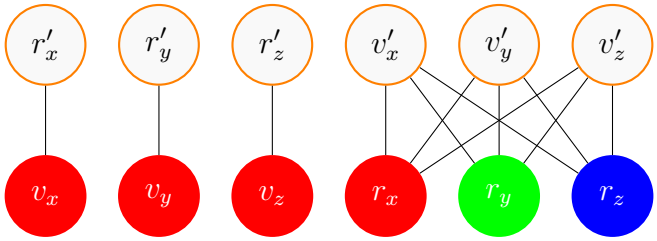

Figure 11. The variable-dependency graph for $\boldsymbol{f}(\boldsymbol{x})$ contains one node for every function and one node for every variable; f_i is connected to x_j if x_j appears in the expression for f_i. The nodes are colored according to their *group*, which can be obtained from a distance-2 coloring of the nodes assigned to variables.

Algorithm 4 outlines a sequential coloring process: let $\mathcal{N}_1, \mathcal{N}_2, \ldots, \mathcal{N}_n$, be an ordering of the nodes in $\mathcal{G}$; for $k = 1, 2, \ldots, n$ assign to $\mathcal{N}_k$ the smallest possible color. The performance[18] of the sequential algorithm depends entirely on the ORDERNODES function: the simplest alternative is to return the nodes in whatever order they happen to be (called CPR[19] ordering). As illustrated in Figure **12**, even such simple heuristic can lead to reasonable colorings. In fact, it is capable of finding a four-color map of the United States in a handful of runs when the nodes are sampled in random order[20] (most runs find five- and six-color maps).

Other orderings are possible that, while more complex to implement, lead to better overall performance:

Largest First ordering selects the nodes so that $\deg(\mathcal{N}_k)$ is non-increasing.

[18] Performance in this context refers to the number of colors used; fewer is better, and equal to $\chi(\mathcal{G})$ is optimal.

[19] After Curtis, Powell, and Reid because relying on such ordering is equivalent to using the algorithm outlined in their original paper [12].

[20] According to the Four Color Theorem, four colors are sufficient to color any planar map. In addition, it is known that $\chi(\text{USA}) = 4$

Algorithm 4 Graph coloring sequential algorithm [12, 13]

```
1:  procedure GCS(𝒢)
2:      for each node u ∈ 𝒢 do
3:          u.color ← NIL
4:      for each node u in ORDERNODES(𝒢) do
5:          u.color ← MINIMUMCOLOR(u)
6:  procedure MINIMUMCOLOR(u)
7:      F ← ∅
8:      for each node v adjacent to u do
9:          if v.color ≠NIL then
10:             APPEND(F, v.color)
11:     k ← 0
12:     loop
13:         if k not in F then
14:             return k
15:         else
16:             k ← k + 1
```

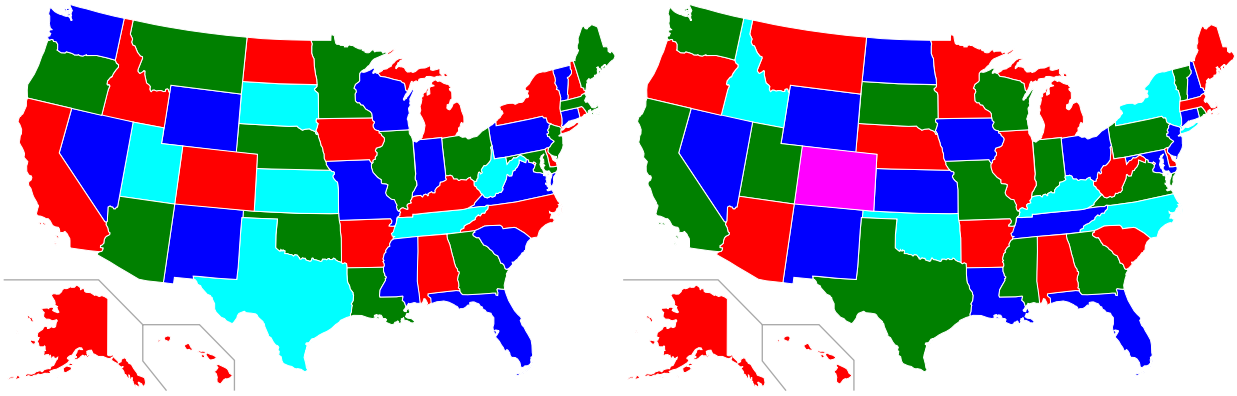

Figure 12. A map of the USA colored using four colors (left) and five colors (right). The colorings were obtained using the sequential algorithm with random node sampling.

Smallest Last ordering assumes that nodes $\mathcal{N}_{k+1}, \ldots, \mathcal{N}_n$ have been selected, and chooses $\mathcal{N}_k$ so that the $\deg(\mathcal{N}_k)$ in the subgraph with nodes $\mathcal{N} - \{\mathcal{N}_{k+1}, \ldots, \mathcal{N}_n\}$ is minimal.

Incidence Degree ordering assumes that $\mathcal{N}_1, \mathcal{N}_2, \ldots, \mathcal{N}_{k-1}$ have been selected, and chooses $\mathcal{N}_k$ so that $\deg(\mathcal{N}_k)$ in the subgraph with nodes $\{\mathcal{N}_1, \mathcal{N}_2, \ldots, \mathcal{N}_{k-1}\}$ is maximal.

Coleman and Moré present a comparison of these ordering algorithms when applied to a collection of over 50 representative and randomly-generated problems; they conclude that Smallest Last and Incidence Degree orderings tend to perform best overall, and are nearly optimal.

7. Conclusions

Graph theory provides a versatile approach to dealing with various problems directly relevant to space mission design. In this paper I provided a basic introduction to the concepts, terminology and basic algorithms of graph theory. In addition, I proposed a graph-based approach to:

- Creating a coordinate frame transformation system capable of generating an optimal transformation path between two coordinate frames.
- Representing multi-segment spacecraft trajectories in a manner that enables one to consider various alternatives simultaneously, augment the underlying data structure with arbitrary properties, and serialize/deserialize the resulting graph to a simple `JSON` format.
- Managing multi-body gravity models in a manner that enables aggregation as center or barycenter, and can trigger center switching.
- Reducing the number of function evaluations needed to estimate sparse Jacobians and Hessians by means of exploiting sparsity information under a graph coloring framework.

8. Acknowledgments

The research described in this paper was carried out at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration.

9. References

[1] Hatfield, J. N. and Rinderle, E. A. User's Guide for CATO Computer Algorithm for Trajectory Optimization. NASA Jet Propulsion Laboratory, 2001.

[2] Williams, J., Senent, J. S., Ocampo, C., Mathur, R., and Davis, E. C. "Overview and software architecture of the Copernicus trajectory design and optimization system." "International Conference on Astrodynamics Tools and Techniques," 2010.

[3] Ocampo, C. "An Architecture for a Generalized Trajectory Design and Optimization System." "Proceedings of the International Conference on Libration Points and Missions," 2002.

[4] Betts, J. T. Practical Methods for Optimal Control and Estimation Using Nonlinear Programming. Advances in Design and Control. SIAM, 2nd edn., 2010.

[5] Acton, C. H. "Ancillary Data Services of NASA's Navigation and Ancillary Information Facility." Planetary and Space Science, Vol. 44, No. 1, pp. 65–70, 1996.

[6] Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Clifford, S. Introduction to Algorithms. The MIT Press, third edn., 2009.

[7] Gross, J. L. and Yellen, J. Graph theory and its applications. Discrete Mathematics and its applications. Chapman & Hall/CRC Taylor & Francis Group, 2nd edn., 2006.

[8] Dijkstra, E. W. "A Note on Two Problems in Connexion with Graphs." Numerische Mathematik, Vol. 1, pp. 269–271, 1959.

[9] Fey, A. L., Gordon, D., and Jacobs, C. S. "The Second Realization of the International Celestial Reference Frame by Very Long Baseline Interferometry." IERS Technical Note 35, International Earth Rotation and Reference System Service (IERS), Germany, 2009.

[10] Cesarone, R. J. "A gravity assist primer." AIAA Student Journal, pp. 16–22, January 1989.

[11] Uphoff, C., Roberts, P. H., and Friedman, L. D. "Orbit design concepts for Jupiter orbiter missions." Journal of Spacecraft and Rockets, Vol. 13, No. 6, pp. 348–355, 1976.

[12] Curtis, A. R., Powell, M. J. D., and Reid, J. K. "On the Estimation of Sparse Jacobian Matrices." IMA Journal of Applied Mathematics, Vol. 13, No. 1, pp. 117–119, 1974.

[13] Coleman, T. F. and Moré, J. J. "Estimation of sparse Jacobian matrices and graph coloring problems." SIAM Journal on Numerical Analysis, Vol. 20, No. 1, pp. 187–209, 1983.

[14] Gebremedhin, A. H., Manne, F., and Pothen, A. "What color is your Jacobian? Graph coloring for computing derivatives." SIAM Review, Vol. 47, No. 4, pp. 629–705, 2005.