Contact Multigraph Routing: Overview and Implementation

Michael Moy
Colorado State University

Robert Kassouf-Short and Nadia Kortas and Jacob Cleveland and Brian Tomko
NASA Glenn Research Center

Dominic Conricode
University of California, Berkeley

Yael Kirkpatrick
Massachusetts Institute of Technology

Robert Cardona and Brian Heller and Justin Curry
University at Albany - State University of New York

***Abstract*—In Delay Tolerant Networking (DTN), the standard routing algorithm used to navigate time-varying networks has been Contact Graph Routing (CGR). In CGR, a globally distributed list of *contacts*, periods during which two DTN nodes may communicate, is used to construct a *contact graph*, in which contacts are vertices. A version of Dijkstra's algorithm can then be used to find paths through this model of the time-varying network. However, since contact graphs may be large compared to the network, potentially growing with the square of the number of network nodes and linearly with the time interval represented, the resulting algorithm does not scale well with the size of the network or time. Any improvement to the routing algorithm will bring significant returns to scale.**

In a previous paper, we briefly introduced an alternative to the contact graph model for routing. This alternative model is based on a multigraph (a graph in which there may be multiple edges between a pair of vertices) where vertices represent network nodes instead of contacts. A version of Dijkstra's algorithm in these multigraph models reduces the time needed to perform the same routing computations done in the existing CGR algorithm. Moreover, a modified version of Yen's algorithm for multigraphs is included. Our variation of CGR, which we call Contact Multigraph Routing (CMR), provides an in-line replacement for the previously used pathfinding algorithms. This paper describes an implementation created based on the CMR approach, and experimental comparisons to traditional CGR are given.

In addition, we explore some additional modifications to the routing pipeline traditionally assumed in CGR. These modifications range from the theoretical to the practical in terms of size and scope. We step forward our understanding of sheaf-theoretic networking and describe how to model the routing pipeline using sheaves. We detail some enhanced route selection criteria that addresses some of the added complexity of DTN-based systems. We also include a future works section on future improvements and implementations that would be of service to the broader DTN community.

TABLE OF CONTENTS

1. INTRODUCTION 1
2. BACKGROUND ON CONTACT GRAPH ROUTING . 2
3. THE MULTIGRAPH-BASED ALGORITHM 2
4. COMPLEXITY OF THE CONTACT MULTIGRAPH DIJKSTRA SEARCH 4
5. CONTACT MULTIGRAPH YEN'S ALGORITHM ... 5
6. IMPLEMENTATION IN HDTN 6
7. EXPERIMENTAL RESULTS 7
8. CONCLUSION 8
REFERENCES 8

U.S. Government work not protected by U.S. copyright

1. INTRODUCTION

As the scale of space communications networks increases, the ability to make intelligent choices in navigating these networks becomes harder and harder. For satellite networks in particular, delays, disconnections, and disruptions are a natural part of interacting with the greater system. Describing these networks falls under the purview of Delay Tolerant Networking (DTN).

At NASA Glenn Research Center, the High-rate Delay Tolerant Networking (HDTN) project has been seeking means of improving the efficiency of its DTN implementation, also called HDTN. The implementation itself is publicly available on GitHub at `https://github.com/nasa/HDTN`. As part of this effort, we have been examining and evaluating alternative routing approaches for DTN in general. One of the preeminent routing algorithms is Contact Graph Routing (CGR), with its most common implementation appearing in the Interplanetary Overlay Network (ION) documentation [1]. Alternative implementations also exist, and prior to the release of this paper, the HDTN team was using pyCGR introduced by Juan Fraire [2].

As we continued to develop the HDTN implementation of DTN, we began to streamline the code to improve efficiency. Initially, we reimplemented pyCGR in a faster C++ routing library called libcgr which removes our dependency on Python and sheds the memory footprint of the Python runtime.

However, the results discussed in [3] encouraged us to pursue an implementation of Contact Multigraph Routing (CMR) as well. Now, the routing library on GitHub includes code for both CGR and CMR, which we will highlight and demonstrate here.

At their core, both CGR and CMR utilize Dijkstra's algorithm, but over very different graph structures. The graph for CMR is called a multigraph, meaning each pair of nodes is permitted to have multiple edges between them. It turns out that adapting Dijkstra's algorithm to a multigraph rather than a contact graph is enough to see significant improvements. However, using multigraphs is not entirely novel in its own right. Jain, Fall, and Patra in [4], which predates CGR, proposed the use of Dijkstra's algorithm for a multigraph in a slightly different setting. Much more recently, an approach to routing similar to ours has also been suggested in [5] based

on the same multigraph model of the network. Nevertheless, our approach is different in key ways, and our implementation differs in its ability to be compared to CGR.

In this paper, we will expand upon the initial comments on CMR in [3] as well as present some experimental results comparing CMR and CGR on simulated space networks. The interested reader may wish to run their own tests using our code available on GitHub at `https://github.com/nasa/HDTN`.

2. BACKGROUND ON CONTACT GRAPH ROUTING

We begin with a brief overview of Contact Graph Routing (CGR) and its pathfinding algorithm, following [2]. A *contact* between two network nodes is a period of time during which it is expected that data could be transmitted. A contact is written as $C_{A,B}^{t_0,t_1}$, where A is the source node, B is the destination node, and t_0 and t_1 are the start and end times of the contact. In the algorithms, the source and destination of an arbitrary contact C are written as $C.src$ and $C.dst$, and the start and end times are written as $C.start$ and $C.end$. CGR depends on a globally distributed and consistent schedule of contacts between its network constituents, called a *contact plan*. A contact plan is simply a list of contacts spanning a specific range of time, and thus contact plans need to updated regularly. They are assumed to be computed in advance by some central source and distributed to the network nodes. This is appropriate for networks in space, as contacts can be accurately predicted by orbital mechanics. Additional information associated to each contact is stored, including the average data rate and average one-way light time. The one-way light time of a contact C is written as $C.owlt$ and will appear in the algorithms below.

The approach to pathfinding in CGR begins with the definition of a *contact graph* from a given contact plan, which provides a static graph model of a time-varying network. In a contact graph, the vertices are the contacts of the contact plan – thus, "vertices" are not DTN nodes, but rather contacts between them. An edge is placed between two contacts if any amount of data could be sent successively along these contacts: specifically, there is a directed edge from $C_{A,B}^{t_0,t_1}$ to $C_{D,E}^{t_2,t_3}$ if $B = D$ and $t_3 > t_0$. If a route from A to Z is to be found starting at time t_0, a root contact $C_{A,A}^{t_0,\infty}$ and a terminal contact $C_{Z,Z}$ are added to the contact graph with appropriate edges. Then a path from $C_{A,A}^{t_0,\infty}$ to $C_{Z,Z}$ in the contact graph indicates a sequence of contacts that can be used to send data from A to Z. Thus, a contact graph turns the problem of routing through a time-varying network into the problem of finding a path in a graph. An example of a contact graph is given in the top of Figure 1.

CGR uses a version of Dijkstra's algorithm [6], [7] to find a path from the root contact to the terminal contact in the contact graph. This version of Dijkstra's algorithm is called the *Contact Graph Dijkstra Search* and is explained in detail in [2]. The search optimizes for arrival time, the earliest time the first byte of data can reach the end of a path. The basic principle behind a Dijkstra search applies here, because the arrival time along a path cannot decrease as the path is lengthened.

The driver for complexity in CGR is the size of the contact plan, since each contact becomes a vertex in the contact graph. The number of contacts in a contact plan may be much larger than the number of network nodes, even over a reasonably small time period. In fact, with 14 nodes determining line of sight contacts over 24 hours, our simulations yield 368 contacts. Hence, the algorithm does not scale well with either the size of the network or the number of contacts, limiting the size of contact plans that can be effectively used in a network. Independent of complexity, the requirement of globally consistent contact plans also imposes restrictions on the size of the network.

CGR encompasses both this pathfinding algorithm and a larger strategy for generating lists of routes, choosing a route from the list for given data, and queuing. In particular, CGR uses a version of Yen's algorithm to generate a list of routes to a given destination. Recall that in a DTN, the primary unit of data is the *bundle*, which may be of arbitrary size and features a time to live (TTL) measured in milliseconds [8]. For a given bundle, a list of candidate routes is selected from those output by Yen's algorithm, based on the bundle size, available volume of routes, and the priority assigned to the bundle. A route is chosen from this list of candidates based on the projected arrival time.

In this paper, we focus on improving the main pathfinding algorithm and the version of Yen's algorithm. We show that by using an alternative graph model of a network instead of a contact graph, the speed of these algorithms can be improved. Specifically, we provide new versions of the Dijkstra search and Yen's algorithm that can be used as replacements for the existing versions in CGR, leaving the remaining generation of candidate routes and route selection unchanged. We demonstrate theoretically and experimentally how this leads to improved scalability, while assuming the same network data.

3. THE MULTIGRAPH-BASED ALGORITHM

Our improved algorithm will be based on an alternate graph model of a time-varying network, replacing the contact graph and also constructed from the same data of a contact plan. The model is based on a *multigraph*, a graph-like structure that allows for multiple edges between a given pair of vertices. Each edge will be directed, and we will not allow loops – that is, an edge cannot start and end at the same vertex. We get an equivalent description of such directed multigraphs by assigning a multiplicity to each edge of a simple directed graph, and we will use this description later. Given a contact plan, we define a multigraph model of a network by letting the vertices be network nodes and letting each contact $C_{A,B}^{t_0,t_1}$ be an edge from vertex A to vertex B (thus, there may be multiple edges between a pair of vertices). We call the resulting multigraph a *contact multigraph*. The time intervals and any other attributes of the contacts may be thought of as labels on the edges, making the this a labeled, directed multigraph. Note that vertices once again represent the network nodes, making this model more similar to traditional graph-based models of networks than the contact graph. An example of a contact multigraph is given in the bottom of Figure 1.

Routing using this model becomes a question of finding a path through the contact multigraph such that consecutive contacts in the path have compatible time intervals. That is, we will search for paths such that if $C_{B,D}^{t_2,t_3}$ follows $C_{A,B}^{t_0,t_1}$ in the path, then $t_3 > t_0$ (this is the same requirement

on time intervals imposed by edges in a contact graph). A modified version of Dijkstra's algorithm can be used to find paths meeting this requirement, again optimizing for delivery time. This algorithm follows the classic version of Dijkstra's algorithm but only explores through edges (contacts) that are available after the arrival time to the current vertex. This algorithm was introduced in [3] by many of the same authors, and this paper expands on that introduction by providing a more complete and detailed analysis of the algorithm, as well as experimental results.

A motivating example

Before going over the formalities, we provide an example, which was also given in [3]. Consider a network with three nodes, A, B, and D, and a contact plan ($C^{0,1}_{A,B}$, $C^{2,3}_{A,B}$, $C^{4,5}_{B,D}$, $C^{6,7}_{B,D}$). The goal will be to send a message from A to D starting at time 0. Figure 1 shows a contact graph (top) and a contact multigraph (bottom), both built from the same contact plan. The contact graph has been given the appropriate root and terminal contacts.

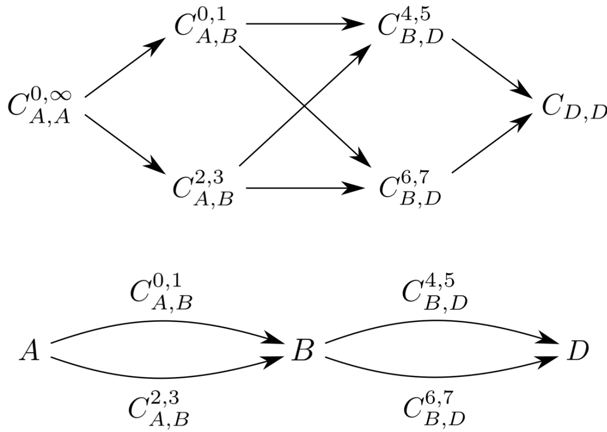

Figure 1. A contact graph (top) and a contact multigraph (bottom) modeling the same network.

We outline the steps of the respective algorithms on these two graphs, beginning with the traditional Contact Graph Dijkstra Search (see [2]). We begin at $C^{0,\infty}_{A,A}$ and explore out each edge, finding arrival times of 0 to $C^{0,1}_{A,B}$ and 2 to $C^{2,3}_{A,B}$. Next, $C^{0,1}_{A,B}$ is chosen as the current contact as it has the earliest arrival time of the unvisited contacts, and we explore its outgoing edges, finding an arrival time of 4 to $C^{4,5}_{B,D}$ and 6 to $C^{6,7}_{B,D}$. Next, $C^{2,3}_{A,B}$ is chosen as the current contact since it has the next smallest arrival time of 2, and we explore its outgoing edges, finding that we cannot improve the arrival time to either $C^{4,5}_{B,D}$ or $C^{6,7}_{B,D}$. The next current contact would be $C^{4,5}_{B,D}$, and since its destination is D, we have found an optimal path with destination D, which has a best delivery time of 4.

Next we consider the corresponding Dijkstra search in the multigraph (pseudocode is given in Algorithm 1). We begin at A at time 0, and see that both outgoing contacts are available after time 0. We explore through both outgoing contacts, finding an arrival time of 0 to B, resulting from the contact $C^{0,1}_{A,B}$. Then B becomes the current vertex, and both contacts leaving B are available after the arrival time of 0 to B. We thus explore through both of these outgoing contacts, finding an arrival time of 4 to D through the contact $C^{4,5}_{B,D}$. Since D becomes the next current vertex, we have found an optimal path with a best delivery time of 4.

These algorithms have both found the same path with the same best delivery time, but the search in the contact graph included a redundant step by using $C^{2,3}_{A,B}$ as the current contact. This step was not able to improve the arrival time to either $C^{4,5}_{B,D}$ or $C^{6,7}_{B,D}$ because they were already reached from $C^{0,1}_{A,B}$, which had an earlier arrival time than $C^{2,3}_{A,B}$. If we remove this step from the search, the contacts explored in the two algorithms are in one-to-one correspondence. This suggests the search through the multigraph will generally be faster. In fact, it can be shown that the two algorithms exhibit similar behavior on any given contact plan, thus showing that the number of steps required by the search in the multigraph will always be less than or equal to the number of steps required by the search in the contact graph. This provides strong evidence that the search in the multigraph will be faster, and in Section 4, we elaborate on this observation by comparing the time complexities of the algorithms.

Contact Multigraph Dijkstra Search – Version for Implementation

Pseudocode for the search in the multigraph, which we call the *Contact Multigraph Dijkstra Search*, was introduced in [3], in a format that allowed for easy comparison to the Contact Graph Dijkstra Search of [2]. Here we provide more practical pseudocode meant for implementation. Efficient implementations of Dijkstra's algorithm typically store the collection of unvisited vertices in a min-priority queue, sorted by lengths of the current best paths to the vertices [7] (in our context, the lengths of the paths are replaced with arrival times). Queue operations can then be used to update vertices' values and positions in the queue and choose the next current vertex. An effective queue will allow these operations to be performed quickly and with known time complexities.

Algorithms 1 and 2 give an efficient version of the Contact Multigraph Dijkstra Search, based on a priority queue of the unvisited vertices. The priority queue V is ordered by the arrival times to the vertices, which are updated throughout the algorithm. The queue operations used are written as $V.decrease_priority(u, arr_time)$, which changes the value of vertex u in the queue to arr_time, and $V.extract_min()$, which returns the vertex in the queue with minimal arrival time and removes it from the queue. Implicitly, the construction of the queue also uses an operation that adds vertices to the queue.

The while loop beginning on line 5 of Algorithm 1 is the main loop of the Dijkstra search. It repeatedly explores outward from the current vertex v_{curr}, then chooses the next current vertex to be the vertex remaining in the queue with minimal arrival time. The Multigraph Review Procedure (MRP) is called in this loop on line 6 and is described in Algorithm 2: it performs the search outward from v_{curr}. For each unvisited neighbor u of v_{curr}, line 5 of Algorithm 2 sets C to be the contact from v_{curr} to u that is available earliest after the arrival time to v_{curr} (we discuss how to find this contact in Section 4, where we find the complexity of the algorithm). If C provides an improvement to the arrival time to u, then the queue is updated and the predecessor of u is set to C. After all unvisited neighbors of v_{curr} are explored, v_{curr} is marked

Algorithm 1 Contact Multigraph Dijkstra Search

Data: Contact multigraph CM, root vertex v_r, destination vertex v_d, $initial_time$
Result: Path P from v_r to v_d with best delivery time $P.BDT$

1: for all vertices v in CM, set $v.arr_time = \infty$, $v.visited = False$, $v.pred = \{\}$
2: $v_r.arr_time = initial_time$
3: construct min-priority queue V from vertices of CM, ordered by arr_time
4: $v_{curr} = V.extract_min()$
5: **while true do**
6: $V = \mathrm{MRP(CM, V, v_{curr})}$
7: $v_{next} = V.extract_min()$
8: **if** $v_{next} == v_d$ **then**
9: break
10: **else**
11: $v_{curr} = v_{next}$
12: **end if**
13: **end while**
14: route reconstruction using predecessors to find path P
15: $P.BDT = v_d.arr_time$

Algorithm 2 Multigraph Review Procedure (MRP)

Data: CM, V, v_{curr}
Result: Revised V
1: **for** neighbor u of v_{curr} **do**
2: **if** $u.visited$ **then**
3: skip u
4: **end if**
5: $C =$ contact from v_{curr} to u with smallest $C.start$ such that $C.end \geq v_{curr}.arr_time$
6: $arr_time = \max(C.start, v_{curr}.arr_time) + C.owlt + owlt_{mgn}$
7: **if** $arr_time < u.arr_time$ **then**
8: $u.arr_time = arr_time$
9: $V.decrease_priority(u, arr_time)$
10: $u.pred = C$
11: **end if**
12: **end for**
13: $v_{curr}.visited = True$

as visited and we return to the main loop of Algorithm 1. The loop ends when the next current vertex is the destination vertex v_d, at which point an optimal path has been found. The path can be reconstructed from the recorded predecessors of the vertices, where in this case a path consists not only of the data of a sequence of vertices, but also a sequence of compatible contacts between them. We have not written the reconstruction explicitly, but recorded the step at line 14 of Algorithm 1.

As a practical note, the algorithm may perform better in practice if vertices are not all added to the queue at the beginning of the algorithm, but rather are only added once discovered during the search (as neighbors during the **for** loop of Algorithm 2). Depending on the input to the algorithm, this can lead to a smaller queue, speeding up the queue operations.

4. Complexity of the Contact Multigraph Dijkstra Search

Here we determine the complexity of Algorithm 1. We will view a multigraph as an underlying (directed) simple graph

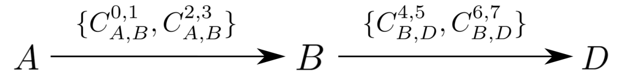

Figure 2. A contact multigraph can be viewed as a simple directed graph with a set of contacts associated to each edge. This example describes the same multigraph as in Figure 1.

with multiplicities stored for each edge. In our case, we in fact store over each edge a list of contacts with source and destination equal to the source and destination of the edge; the multiplicity of the edge is then the number of contacts associated to the edge. An example corresponding to Figure 1 is shown in Figure 2. Let $|V|$ be the number of vertices and let $|E|$ be the number of edges in the underlying simple graph. For any arbitrary ordering of the edges, let m_i be the multiplicity associated to the ith edge. Let $|CP|$ be the total number of the contacts in the contact plan, so that $\sum_{i=1}^{|E|} m_i = |CP|$.

Algorithm 1 follows an implementation of Dijkstra's algorithm using a priority queue, with an additional step of finding the optimal contact associated to a given simple edge (line 5 of Algorithm 2). Each simple edge is explored at most once over the course of Dijkstra's algorithm, and thus the time complexity of Algorithm 1 is equal to that of the implementation of Dijkstra's algorithm plus the time required to find the optimal contact associated to each edge. A straightforward implementation could use a linear search through the list of contacts associated to the ith edge. It requires a time of $O(m_i)$ for the ith edge, and thus contributes a total time of $O(\sum_{i=1}^{|E|} m_i) = O(|CP|)$.

We can improve this time complexity if we assume there are no overlapping contacts for each edge, that is, the time intervals for the contacts over a given edge are disjoint[2]. Then we may assume the contacts for each edge are stored sorted by end time, and in this case a binary search takes $O(\log(m_i))$ for the ith edge. Again, this is performed at most once for each edge, so the total time required by these steps is $O\left(\sum_{i=1}^{|E|} \log(m_i)\right)$. Since a mean of logarithms is less than or equal to the logarithm of the mean, we can bound the time as follows:

[2]Non-overlapping contacts can always be achieved by splitting and combining contacts in a contact plan. However, overlapping contacts might realistically be used to model cases of multiple channels of communication between a given pair of nodes. A similar complexity calculation could also be done allowing for multiple channels of communication between any pair of nodes, but assuming no overlapping contacts for a given channel.

$$\sum_{i=1}^{|E|} \log(m_i) \leq |E| \log\left(\frac{1}{|E|}\sum_{i=1}^{|E|} m_i\right)$$
$$= |E| \log\left(\frac{|CP|}{|E|}\right).$$

Therefore the time required to determine the optimal contact for every edge is $O(|E| \log(|CP|/|E|))$. Note that the term $|CP|/|E|$ is the average multiplicity of an edge. Adding this to the time required by Dijkstra's algorithm gives the total time complexity of Algorithm 1. The best known time complexity for Dijkstra's algorithm is $O\big(|E| + |V| \log |V|\big)$ (see [9]), giving an implementation of Algorithm 1 with a time complexity of $O\big(|E| \log(|CP|/|E|) + |V| \log |V|\big)$.

Realistically, we should expect that $|CP|$ is the largest variable, possibly many times larger than $|V|$ and $|E|$. This is because $|CP|$ grows with both the size of the network and the time covered by the contact plan. The complexity given above shows that Algorithm 1 scales better with the size of the contact plan than the Contact Graph Dijkstra Search. Indeed, since the contact graph has $|CP|$ vertices, the corresponding Dijkstra search has a time complexity of (at least[3]) $O(|CP| \log |CP|)$, as noted in [2].

Practically speaking, the improvement in complexity, and especially the logarithmic dependence on $|CP|$, will allow Algorithm 1 to run successfully with larger contact plans. This amounts to a more scalable version of CGR, and in particular allows for the flexibility of scheduling longer periods of time in contact plans.

In addition to complexity calculations, it is insightful to see how the algorithms directly compare. However, such analysis is somewhat beyond the scope of this paper. A further discussion on comparing the algorithms as well as proofs demonstrating the improved efficiency in general will be provided in an upcoming paper.

5. Contact Multigraph Yen's Algorithm

In addition to using a version of Dijkstra's algorithm to find paths, CGR also uses a version of Yen's algorithm [10] to find lists of the K best paths, for a given K [2]. This version of Yen's algorithm runs the Dijkstra search iteratively to find the list of best paths. Here we give a version of Yen's algorithm for contact multigraphs, further enabling contact multigraphs to serve as the fundamental network model in the routing strategy of CGR. As with our version of the Dijkstra Search, our version of Yen's algorithm gives improved time complexity.

Algorithm 3 describes Yen's algorithm in the multigraph, incorporating Lawler's modification [11]. In the pseudocode, slicing of lists excludes the ending index. This algorithm and the original Yen's algorithm consider *loopless* paths, meaning paths in which a vertex is not visited more than once. This is the reason it is appropriate for CGR, as DTN nodes are assumed to be able to store information (and this is assumed to be preferred over sending data in loops). Note that there can be multiple valid outputs if there is a tie for the Kth best path.

Let $CM_Dijkstra$ be Algorithm 1 that takes as an input a contact multigraph CM, a root vertex v_r, a destination vertex v_d, and a starting time $initial_time$. Let a path P be a data type with a list of contacts $P.contacts$ and a list $P.arr_times$ of arrival times to the vertices of the path in order. The arrival time to the destination vertex can be referred to as the best delivery time $P.BDT$ to match notation of [2]. Then the list $P.arr_times$ is one longer than $P.contacts$, and the arrival time to the source vertex of the ith contact of $P.contacts$ is the ith time in $P.arr_times$. Assume that $CM_Dijkstra$ outputs a path with this information recorded. Let a path P also have a spur index $P.spur_index$, to be set during the algorithm.

The algorithm is based on the observation that if the first k best paths are known, the $(k+1)^{\text{th}}$ best path must deviate from one of these paths at some vertex, called the *spur node*. The portion of the $(k+1)^{\text{th}}$ best path from the source to the spur node is called the *root path* and agrees with the beginning of one of the first k best paths. The portion of the $(k+1)^{\text{th}}$ best path from the spur node to the destination is called the *spur path*, and $CM_Dijkstra$ is used to find the spur path and its best delivery time. To find the $(k+1)^{\text{th}}$ best path, the algorithm considers all possible spur nodes in the first k best paths. By finding the best spur path resulting from each spur node, it generates candidates for the next best path. Once all candidates have been generated, the optimal one can be chosen. Throughout, the algorithm maintains a list A of the currently known k best paths and a collection B of the candidate best paths.

[3]This estimate of the complexity ignores the edges of the contact graph. There is at least a linear dependence on the number of edges, and realistically the number of edges should be greater than the number of contacts.

Algorithm 3 Contact Multigraph Yen's Algorithm

Data: Contact multigraph CM, root vertex v_r, destination vertex v_d, $initial_time$, K
Result: K optimal paths from v_r to v_d starting after $initial_time$

1: $A[0] = CM_Dijkstra(CM, v_r, v_d, initial_time)$
2: **if** $A[0] == \varnothing$ **then**
3: **return** $\varnothing$
4: **end if**
5: Initialize min-heap B
6: $A[0].spur_index = 0$
7: **for** $k = 1$ to $K - 1$ **do**
8: $prev_spur = A[k-1].spur_index$
9: remove from CM the source vertex of each contact in $A[k-1].contacts[0 : prev_spur]$
10: $root_path = A[k-1].contacts[0 : prev_spur]$
11: **for** path P in A **do**
12: **if** $P.contacts[0 : prev_spur] == root_path$ **then**
13: remove contact $P.contacts[prev_spur]$ from CM
14: **end if**
15: **end for**
16: **for** $i = prev_spur$ to $length(A[k-1].contacts) - 1$ **do**
17: $spur_node = A[k-1].contacts[i].src$
18: remove contact $A[k-1].contacts[i]$ from CM
19: $spur_path = CM_Dijkstra(CM, spur_node, v_d, A[k-1].arr_times[i])$
20: **if** $spur_path == \varnothing$ **then**
21: continue
22: **end if**
23: $total_path.contacts = root_path + spur_path.contacts$
24: $total_path.spur_index = i$
25: $total_path.BDT = spur_path.BDT$
26: assign total path source arrival times based on those of $A[k-1]$ and $spur_path$
27: add $total_path$ to B with value $total_path.BDT$
28: remove $spur_node$ from CM
29: $root_path.append(A[k-1].contacts[i])$
30: **end for**
31: restore CM
32: **if** B is empty **then**
33: break
34: **end if**
35: $A[k] = B.extract_min()$
36: **end for**
37: **return** A

Complexity

To determine the time complexity of Algorithm 3, we will assume the optimal complexity of $CM_Dijkstra$ from Section 4. The same reasoning applies if a different implementation of $CM_Dijkstra$ is used with a different complexity. Following [10], we will assume that K is small compared to the other variables. Then the time required by lines 11-15 is negligable, as A will always contain less than K paths. The dominant term is the use of $CM_Dijkstra$ in line 19. For each k, the number of iterations of the loop starting on line 16 is bounded above by V, since no loopless path has more than V vertices. Thus, $CM_Dijkstra$ is called at most KV times, so the time required by Algorithm 3 is $O\Big(K|V|\big(|E|\log(|CP|/|E|) + |V|\log|V|\big)\Big)$. This improves upon the version of Yen's algorithm in a contact graph, which has complexity of (at least) $O\big(K|CP|^2\log|CP|\big)$ [2].

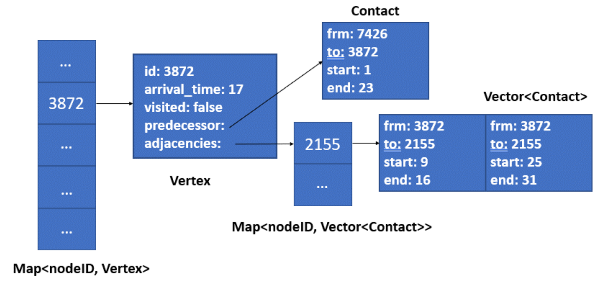

Figure 3. Contact Multigraph Structure

6. Implementation in HDTN

Algorithm 1 was recently implemented into NASA's High Rate Delay Tolerant Networking (HDTN) project, with code available at `https://github.com/nasa/HDTN`. We describe the implementation in this section. A C++ library implementing the subset of PyCGR (as in [2]) needed by HDTN —namely loading contact plans and computing best routes over them using Dijkstra's algorithm— was also recently added. In Section 7, we will provide an experimental comparison of these two algorithms to complement the theoretical comparison given in the previous sections.

Data Structures

The inputs to the implementation of Algorithm 1 are a contact plan, which is a vector of contacts; the ID of the destination node; and a pointer to the root contact, from which the root vertex and initial time can be determined. These were chosen to agree with the inputs to the Contact Graph Dijkstra Search, for easy comparison. Each contact is an instance of a class containing fields for the source node's ID, the destination node's ID, and the contact's start and end times (Figure 3).

The following representation of a contact multigraph was chosen to give an efficient implementation; see Figures 1 and 2 for visualizations of contact multigraphs. Roughly speaking, a contact multigraph will be represented as an adjacency list of vertices, with additional information recorded for each adjacency. We begin with a vertex class to represent network nodes. It will have a node ID and adjacencies, which are other vertices that can be reached by traversing a contact. For each vertex v, the adjacencies will be stored as a mapping from a node ID to a vector of contacts, recording every edge in the multigraph from v to a given adjacent vertex. We will assume contacts in this vector do not overlap and are sorted by time, as discussed in Section 4, allowing for a binary search of the contacts in line 5 of Algorithm 2. A contact multigraph class will contain an unordered map mapping node IDs to their corresponding vertex objects. This allows access to a vertex and its fields in constant time from the node ID.

Throughout the execution of the Dijkstra search, a min-priority queue is used to store the unvisited vertices, sorted by arrival time (see Section 3). The priority queue is implemented through a heap that uses a custom comparator to insert vertex objects based on their arrival time. A vertex object also contains relevant fields for the routing algorithm such as the arrival time, whether that vertex has been visited already, and a predecessor contact, to be set and updated in line 10 of Algorithm 2. At the end of the routing algorithm, the route to the destination vertex can be reconstructed by following back the chain of predecessors to the root vertex

(line 14 of Algorithm 1). The end result, a route, will be a chain of contacts from the starting node to the destination node.

The construction of the contact multigraph comes from filling up the above-mentioned structure from the contact plan. Looping through the contacts in the contact plan, a vertex is created for the source and destination node IDs of the contact. The contact is then inserted into the adjacencies from the source vertex to the destination vertex, sorted by time. One optimization in the construction of the contact multigraph is as follows: if a node ID never appears as the source of a contact, then it has an outdegree of zero in the contact multigraph, and thus is never constructed because it cannot be part of the optimal route. This results in better runtime of the construction of the contact multigraph and the routing.

Integration with HDTN router

The function implementing the Algorithm 1 in C++ was added to the new HDTN routing library as another option for routing in addition to dijkstra CGR implementation `https://github.com/nasa/HDTN/blob/master/common/cgr/src/libcgr.cpp`. It was also integrated with the HDTN router module so that if the user selects to use CMR instead of CGR, this function is invoked by the router to get the next hop for the optimal route leading to the final destination.

7. Experimental Results

The CMR implementation successfully finds the optimal route that CGR finds. CMR was integrated with HDTN and run on an HDTN simulator. CMR produced the same route within this simulator as CGR. In addition, preliminary timing tests between CMR and CGR yields positive results. On a contact plan with 6 nodes and 9 contacts, averaged over 100 trials, CMR found the optimal route in 16 microseconds while CGR found the optimal route in 22 microseconds (see Figure 4). This decrease in runtime is expected to be larger in bigger contact plans due to the algorithmic complexity differences. Further breaking down the runtime of CMR, approximately half of the runtime (8 microseconds) was taken up by the construction of the contact multigraph.

After preliminary testing, our next step was to test scalability of the algorithm. To wit, we initiated testing runtime efficiency with larger contact plants generated from real satellite data. To test the scalability of our algorithm, we elected to pull from the `STARLINK` mega constellation, whose Two-Line Elements (TLEs) are available at `https://celestrak.org/`. On these greater tests, we anticipated that the runtime of CMR in generating the multigraph would become a smaller and smaller percentage of the overall CMR runtime with larger contact plans. This would then result in even greater speedups when compared with CGR.

Simulation in SOAP

For the simulations, we used the Satellite Orbit Analysis Program (*SOAP*) developed by the Aerospace Corporation [12]. SOAP is built around a GUI, which makes manually building large simulations time consuming. In a naive workflow, one typically adds satellites to the simulation by hand and then augments them with platforms and sensors to record sunrise and sunset times for line-of-sight communication. We wanted to develop a script-based system so that one could add, ideally, all 2300 `STARLINK` satellites to a single simulation.

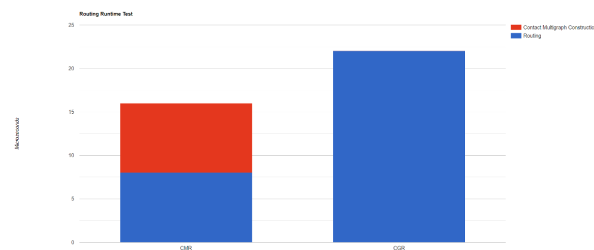

Figure 4. CMR vs. CGR runtime comparison in microseconds. The red bar corresponds to constructing the multigraph from the contact plan, while the blue bars correspond to computing the routes.

To achieve this, we analyzed the associated (ORB) simulation files to see how they were formatted. We then wrote code to programmatically generate ORB files according to specified parameters, such as (1) date of simulation, (2) duration of simulation, (3) list of satellites and ground stations to use, (4) customized reports for additional data such as the distance between any two satellites at specified time slices of the simulation.

The simulations we used for these experiments consist of a few hard coded ground stations specified by longitude and latitude coordinates, together with a random k-sample of the entire `STARLINK` network. We then coded the ORB files to generate a report of contact times (saved as a CSV file), which track the rise and set times for when any two satellite / ground stations are connected throughout line-of-sight during the simulation period, which was typically fixed to twenty-four hours. This data is then sufficient to describe the underlying time-varying graph of the simulated space network in question..

Finally, by running *SOAP* in GNU/Linux, we were able to pass in our generated ORB files as arguments and generate the contact reports as an automated batch process. The contact report CSV outputs were then parsed and translated to the JSON format required by the CMR implementation and used in this paper.

In this paper, the simulations were constructed with four imagined ground stations, at Albany, NASA Glenn Research Center, UC Berkeley, and Guam. In addition, we took samples of size 10, 50, 100, and 200 from the `STARLINK` mega constellation. With each simulation lasting for 24 hours (86400 seconds), and counting the ground stations as nodes, the contact plans consisted of 14, 54, 104, and 204 nodes corresponding to 368, 7186, 28162, and 109330 contacts respectively.

For the routing problem, we selected four longer distance connections to test, namely Albany to Berkeley, NASA Glenn to Berkeley, Albany to Guam, and NASA Glenn to Guam. For each connection tested, routes were computed using our CGR and CMR implementations within each of the four simulations. For each simulation-algorithm-connection combination the time it took to run the routing algorithm was computed in microseconds. The results of the CGR implementation appear in Table 1, and the results of the CMR implementation appear in Table 2.

CGR Route	10 sat	50 sat	100 sat	200 sat
Albany - Berkeley [10-30]	376	26286	334939	12509186
Glenn - Berkeley [20-30]	423	33965	459089	33339362
Albany - Guam [10-40]	352	25931	321582	11837432
Glenn - Guam [20-40]	244	31025	293578	14852335

Table 1. Runtime in microseconds for each simulation in CGR implementation

CMR Route	10 sat	50 sat	100 sat	200 sat
Albany - Berkeley [10-30]	254	8338	45929	154221
Glenn - Berkeley [20-30]	275	13287	59814	267133
Albany - Guam [10-40]	178	9577	40781	158381
Glenn - Guam [20-40]	222	9188	44727	155402

Table 2. Runtime in microseconds for each simulation in CMR implementation

8. CONCLUSION

While HDTN is still an experimental program, we hope to bring our software to wider adoption and use in space missions. Results and improvements like those presented in this paper are part of the reason we are confident that HDTN will mature into a successful product. As we continue experimenting and examining scalability for our algorithms, we are setting HDTN into the position to best support the widest group possible.

The increase in efficiency we achieve with CMR is only realized by a combination of theoretical research and practical implementation. We will continue testing HDTN and CMR as we build additional robustness and assurance in our capabilities. We are currently targeting several missions for implementation and testing, including connecting to the ISS and future Artemis and Mars missions. This will further establish HDTN as the performance-optimized version of DTN. Pushing HDTN forward brings to space research a new possible tool to expand the horizons of what is possible.

Future Work

In this paper we presented some first steps we can take to address complexity and scalability of contact-based routing. Implementing CMR in HDTN proved to be successful at speeding up route computation by using an updated data structure to represent all contacts and implementing the Contact Multigraph Dijkstra Search. Naturally, the next step in this process would be to implement Yen's Multigraph algorithm to find multiple shortest paths between two endpoints in the network. Given the success of Multigraph Dijkstra's algorithm, we expect Multigraph Yen's algorithm will also perform better in practice when compared to CGR.

However, even with the better runtime provided by CMR, scalability still remains a big issue in contact-based routing. In both CGR and CMR, route computation requires knowledge of the entire network. This becomes expensive and impractical in large scale networks, such as the one we expect to see in a future space internet. One way to address the scalability of these routing methods is by partitioning the network into smaller sub-networks or clusters. This would allow the use of different routing algorithms on inter and intra-cluster routing and would avoid the situation of running a search on a large scale contact multigraph. Arriving at this network partition, as well as developing distributed routing algorithms that allow the interaction of different endpoints and clusters, leave much potential future work.

Acknowledgements

This work was supported by the NASA SCaN Summer Internship Program 2022. Special thanks to Alan Hylton, Juan Fraire, Olivier De Jonckére, Gilbert Clark, Matt Piekenbrock, Billy Bernardoni, Zoe Cooperband, Prash Choksi, and Timothy Recker for the useful discussions and commentary throughout this process.

REFERENCES

[1] J. Segui, E. Jennings, and S. Burleigh, “Enhancing contact graph routing for delay tolerant space networking,” in *2011 IEEE Global Telecommunications Conference - GLOBECOM 2011*, 2011, pp. 1–6.

[2] J. A. Fraire, O. De Jonckère, and S. C. Burleigh, “Routing in the space internet: A contact graph routing tutorial,” *Journal of Network and Computer Applications*, vol. 174, January 2021.

[3] A. Hylton, R. Short, J. Cleveland, O. Freides, Z. Memon, R. Cardona, R. Green, J. Curry, S. Gopalakrishnan, D. V. Dabke, B. Story, M. Moy, and B. Mallery, “A survey of mathematical structures for lunar networks,” in *2022 IEEE Aerospace Conference*, 2022.

[4] S. Jain, K. Fall, and R. Patra, “Routing in a delay tolerant network,” in *Proceedings of ACM SIGCOMM*, 2004.

[5] O. De Jonckère and J. A. Fraire, “A shortest-path tree approach for routing in space networks,” *China Communications*, vol. 17, no. 7, pp. 52–66, 2020.

[6] E. W. Dijkstra, “A note on two problems in connexion with graphs,” *Numerische mathematik*, vol. 1, no. 1, pp. 269–271, 1959.

[7] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 3rd ed. The MIT Press, 2009.

[8] S. Burleigh, K. Fall, and E. J. Birrane, “Bundle Protocol Version 7,” RFC 9171, Jan. 2022. [Online]. Available: https://www.rfc-editor.org/info/rfc9171

[9] M. Fredman and R. Tarjan, “Fibonacci heaps and their uses in improved network optimization algorithms,” in *25th Annual Symposium on Foundations of Computer Science, 1984*, 1984.

[10] J. Y. Yen, “An algorithm for finding shortest routes from all source nodes to a given destination in general networks,” *Quarterly of Applied Mathematics*, vol. 27, pp. 526–530, 1970.

[11] E. L. Lawler, “A procedure for computing the k best solutions to discrete optimization problems and its application to the shortest path problem,” *Management Science*, vol. 18, no. 7, pp. 401–405, 1972. [Online]. Available: https://EconPapers.repec.org/RePEc:inm:ormnsc:v:18:y:1972:i:7:p:401-405

[12] D. Stodden and G. Galasso, “Space system visualization and analysis using the satellite orbit analysis program (soap),” in *1995 IEEE Aerospace Applications Conference. Proceedings*, vol. 1, 1995, pp. 369–387 vol.1.

Michael Moy is pursuing a PhD in mathematics at Colorado State University, having completed his master's there in 2021. His research is focused on applied topology. During the summers of 2020, 2021, and 2022, he worked as an intern at NASA through the SCaN Internship Project. His research areas at NASA have included machine learning and mathematical approaches to networking.

Robert Kassouf-Short earned his PhD in mathematics from Lehigh University in 2018. He worked as a Visiting Assistant Professor of Mathematics at John Carroll University until he joined the Secure Networks, System Integration and Test Branch at NASA Glenn Research Center in 2020. His research interests lie in the intersection of abstract mathematics and real world applications. Currently, his focus is on the foundations of networking theory and how to efficiently route data through a network using local information.

Nadia Kortas is a research computer engineer working on the development of software to support embedded systems, space communication systems and other Space flight and Science projects at NASA. She currently works on High-rate Delay Tolerant Networking implementation with focus on Routing and Security. She holds a Master Of Science Degree in Engineering and Computer Science from Mines ParisTech, France.

Jacob Cleveland is a PhD student studying mathematics at Colorado State University. They have a bachelors degree in mathematics from the University of Nebraska at Omaha and a bachelors degree in computer engineering from the University of Nebraska - Lincoln. They joined the Secure Networks, System Integration and Test Branch as a Pathways Intern at NASA Glenn Research Center in 2020. Since joining, they have contributed to several research projects applying pure mathematics to engineering problems in space networking, star tracking, and artificial neural networks.

Brian Tomko earned his Bachelor of Science in Computer Engineering from Ohio Northern University in 2009. He has been working as a computer programmer at the NASA Glenn Research Center since 2009. His primary computer languages are C and C++ on both desktop and embedded platforms. Currently, his focus is on contributing code to the High-rate Delay Tolerant Networking (HDTN) software project.

Dominic Conricode is currently working on his undergraduate degree in Computer Science from the University of California, Berkeley. He worked as an intern at NASA Glenn Research Center during the summer of 2022.

Yael Kirkpatrick is pursuing a PhD in applied mathematics at the Massachusetts Institute of Technology. Her research interests include graph algorithms and fine grained complexity. She worked as an intern at NASA Glenn Research Center during the summer of 2022. Her research focused on applying techniques from graph theory to improving routing algorithms in space networks.

Robert Cardona is a PhD student in applied topology. He studied computer engineering and mathematics before going on to work as a software developer. He then obtained a masters at Freie Universität Berlin and continued on to study applied topology at Albany.

Brian Heller is a PhD student studying mathematics at University of Albany. Previously he worked as a software engineer and as an advisor in digital and emerging technologies. His research interests include category theory, mathematical logic, and algebraic geometry.

Justin Curry is an Assistant Professor of Mathematics and Statistics at the University at Albany, SUNY. Before arriving at Albany in 2017, he was a Visiting Assistant Professor at Duke University. Professor Curry earned his PhD in mathematics from the University of Pennsylvania in 2014, under the direction of Robert Ghrist. His research interests include the use of category theory in applied mathematics, with particular emphasis on applied sheaf theory, and inverse problems in topological data analysis (TDA).