O*: A Bivariate Best First Search Algorithm to Process Optimal Sequence Traversal Query in a Graph

Qifeng Lu
MacroSys LLC.
Arlington, United States
qilu1@vt.edu

Kathleen Hancock
Center for Geospatial Information Technology, Virginia Polytechnic Institute and State University
Alexandria, United States
hancockk@vt.edu

***Abstract*—An Optimal Sequence Traversal Query is a new query in a graph typically representing a transportation network that determines the minimum-cost path with a predefined origin-destination pair, traversing a set of non-ordered points of interest, at least once for each point. It has distinctive applications in the GoeProcessing domain in transportation where a user may query a shortest route to start from his/her office, traverse a set of consumer destinations, and then go home. Optimal Sequence Traversal Query generalizes TSP in the sense that the origin and the destination may be different in the former. This paper proposes a bivariate best first search, O*, to process such a query within a graph. Two special cases of O*, O*-SCDMST and O*-Dijkstra, are provided, their performance in a fully connected directed graph are studied through a set of experiments, and the result demonstrates that, on average, O*-SCDMST reduces computation time by one order of magnitude when compared to O*-Dijkstra.**

Keywords- Bivariate Best First Search, Heuristic, Optimal Sequence Traversal Query, O*, O*-SCDMST

I. INTRODUCTION AND BACKGROUND

The Optimal Sequence Traversal Query (OSTQ) is a query conducted to find the minimum-cost path that starts from any given origin, passes through a set of points of interest, and terminates at a given destination. It has distinctive applications in the GeoProcessing domain in transportation where a user may query a shortest route to start from his/her office, traverse a set of consumer destinations, and then go home. It may have potential applications in artificial intelligence where a robot is sent to collect data from multiple sensors and delivers the data to a given destination for downloading and analysis. The traveling salesman problem (TSP) is a special case of OSTQ, where the given origin and destination are the same [1] [2] [3] [4] [5] [6].

A graph can conceptually represent a complex network, such as a transportation network. In this context, a graph G is defined as a set of vertices, $\{V_i\}$, and a set of directed line segments, $\{E_{i,j}\}$, called arcs. $E_{i,j}$ is defined such that an arc is from vertex V_i to vertex V_j, and V_j is a successor of V_i. Each $E_{i,j}$ has an associated cost $C_{i,j}$. Only graphs with $C_{i,j}>=0$ are considered in this paper. These graphs are referred to as δ graphs.

In this paper, we propose a new type of query defined in such a δ graph: Optimal Sequence Traversal Query (OSTQ). Given a δ graph G with its vertices $\{V_i\}$ and edges $\{E_{ij}\}$, a set of vertices of interest VI from $\{V_i\}$ in graph G, a starting vertex S, and a destination vertex D, an OSTQ retrieves the minimum-cost path, traversing all vertices of interest, at least once for each vertex. In such a graph, there may exist *normal vertices* that are neither vertices of interest to traverse, nor the given origin or destination. Within this context, all necessary sub state graphs are generated implicitly. An application of this query in the GeoProcessing domain is that a consumer drives from his/her office, traverses a gas station, a coffee shop, and a post office, and gets home. Another application is that a food delivery person starts from the restaurant he works at, traverses a set of delivery points, and then returns to the restaurant, a typical example of TSP.

This paper proposes a bivariate best-first search algorithm, O*, to process OSTQ in a graph. O* is a bivariate best first search in the sense that it uses two variables to specify its state. Both theoretical and experimental analyses of the proposed algorithm are presented.

The paper is organized as follows. First, related work is discussed in Section II. In Section III, O* is presented. Section IV provides the SCDMST heuristic, a globally admissible heuristic for O*. Section V presents a set of experiments and an analysis of the results. Finally, the conclusion is presented.

II. RELATED WORK

This section provides a review of the state-of-the-art research on 1) Traveling Salesman's Problem (TSP), and 2) best first searches.

A. Travelling Salesman Problem (TSP)

The earliest research in TSP is in Euclidean space that searches for a shortest round-trip route to traverse each city exactly once with all cities directly connected to each other, forming a fully connected graph. A set of solutions, including dynamic programming [7], nearest neighbor [8], iterative algorithms such as 2-OPT, 3-OPT, and n-OPT [9], best first search [10], ant colony simulation [11], simulated annealing [12], Branch and Bound approach [13], and so on, were proposed to resolve this problem, either exactly or approximately, and the result is a Hamiltonian cycle that visits each vertex exactly once and returns to the starting vertex. These algorithms may be adjusted to process OSTQ.

B. Best First Searches

A best first search is a kind of informed search. The following two subsections provide a review of best first searches. A best first search is n-variate if it uses n variables to specify its states.

 ISBN: 978-1-61208-118-2

Single-Variate Best First Searches

Single-variate best first search is the existing best first search that searches a graph by expanding the most promising vertex chosen according to some rule. It adopts estimates to the promise of vertex n by a "heuristic evaluation function *f(n)* that, in general, may depend on the description of *n*, the description of the goal, the information gathered by the search up to that point, and most important, on any extra knowledge about the problem domain." [14], which is in prevalent used by researchers in Artificial Intelligence (AI), including Russell & Norvig [15].

Several algorithms, including A* [15][16], Dijkstra search [17], Greedy search [15], frontier search [18], and so on, extract the path of minimum cost between a predefined origin-destination vertex pair in a graph. A* uses a distance-plus-cost heuristic function as *f(n)* to determine the order, in which the search visits vertices in the graph [14]. *f(n)* is the sum of two functions: *g(n)*, the path cost function of the path from the origin to the current vertex *n*, and *h(n)*, the heuristic estimate of the distance from the current vertex *n* to the goal. For *h(n)*, two important concepts exist. The first is *admissibility*. A heuristic is *admissible* if its value is less than or equal to the actual cost [15]. The other is *consistency*. A heuristic is *consistent* when the real cost of the path from any vertex *A* to any vertex *B* is always larger than or equal to the reduction in heuristic [16]. Once a heuristic is consistent, it is always admissible [15]. A* has been shown to obtain the optimal solution, i.e., the minimum-cost solution, whenever the heuristic is admissible [15], and, indeed, is optimally efficient among all best-first algorithms guided by path-dependent evaluation functions when the heuristic is consistent [19]. Additionally, A* is complete in the sense that it will always find a solution if one exists. The spatial and time complexity of A* depends on the heuristic and, in general, is exponential. However, A* is very fast in practice. Both Dijkstra and the Greedy algorithm can be considered as a special case of A*. Dijkstra algorithm only considers g(n) as f(n). The Greedy algorithm only uses h(n) as f(n). Frontier search is similar to A* except that Frontier search only works on data sets with consistent heuristic and does not require a closed-list to implement the search algorithm, which consequently saves space at the cost of increased computation [18].

There is also a set of A* variations such as anytime A* [20], hierarchical A* [21], MA* [22], and SMA* [23], which take the same form *f(n)* as A* but adapts A* to different scenarios to reduce time or space complexity of A*.

All the *f(n)*s used in these identified best first searches are defined upon a single variable, vertex n, to estimate its promise.

One exact approach uses a Minimum Spanning Tree (MST) to provide an admissible heuristic to retrieve optimal TSP routes with A* [10]. The algorithm's performance has not been reported since then. A possible reason is that to process TSP, existing single-variate best first search is not adequate. This is because a vertex must be able to store multiple partial paths that traverse different sets of points of interest during the search process that a single-variate best first search cannot handle.

Bivariate Best First Searches

The concept of multivariate best first searches was first proposed in [24] to address the deficiency of a single-variate best first search to process multiple categories of interest. It uses multiple variables to specify a state to be evaluated and expanded. L#, a generalized best first search that evaluates the promise upon a state in a similar form as A*, was proposed, together with a set of novel concepts in best first searches, including local heuristic, global heuristic, local admissibility, and global admissibility [24]. As an instance of L#, the bivariate best-first-search C* was provided to processes Category Sequence Traversal Query (CSTQ) in a graph, which asks for a minimum cost route that starts from a given origin, traverses a set of ordered categories of interest that includes multiple objects in each category, with one selection from each category, and ends at a given destination [24]. In C*, a bivariate state instead of a single-variate state is evaluated and expanded. The state in C* is defined as the combination of a vertex and its *VisitOrder*, a discrete integer variable to indicate the order of a visiting category. A vertex may have multiple states in a graph, and not all the states of a vertex may be generated and expanded. Through its state specification, C* extends the theorems on optimality identified in single-variate A*.

As a bivariate best first search, C* substantially improves the ability of best first searches to process more complex queries. However, C*'s state specification is still not adequate to process OSTQ because it does not consider different visit orders of a vertex of interest. Therefore, C* cannot be used to process OSTQ.

Since best first search is supported with solid theories to obtain optimally efficient, optimal, and sub-optimal solutions and in nature a Branch and Bound approach that is prevalent for TSP calculation, a bivariate best first search is proposed in this paper to process OSTQ.

III. O*: A BIVARIATE BEST-FIRST-SEARCH APPROACH TO PROCESS OSTQ IN A GRAPH

This section describes the details of O*, the bivariate best first search algorithm to process OSTQs in a graph. First, for a state *s*, O* uses the same-form distance-plus-cost function *f(s)* as C*, shown in equation (1), to determine the order, in which the search visits vertices in the graph [24]. Therefore, similar to C*, O* is another instance of L#. Second, O* assures that its solution traverses all points of interest.

$$f(s)=g(s)+h(s) \quad (1)$$

where

f(s) is the estimate to the promise of a state *s* to be expanded,

g(s) is the cost from the origin state to the state *s*, and

h(s) is the estimation to the actual cost from the state *s* to the final state.

The lower the *f(s)*, the higher is the priority for a state to be expanded.

In the following subsections, first, a set of best first search concepts identified for O* is discussed, followed by the presentation of the O* algorithm. Next, how the algorithm assures the traversal of all vertices of interest is

 ISBN: 978-1-61208-118-2

discussed, followed by the analysis of its completeness. Finally, the relationship between different states of a vertex is discussed.

A. Definition

In O*, a state $S_{i,j}$ is defined as (V_i, VL_j), where VL_j is an ordered list of j vertices of interest that are traversed along the path obtained so far at vertex V_i. Since a vertex may have multiple states and each state has its own points of interest traversed, the ordered list is used to efficiently compare two states to remove a state that is not on the target route to be retrieved. A state of a vertex n describes the traversed points of interest along the partial path from the origin to the vertex n. A successor operator Γ is defined on $\{S_{i,j}\}$. Its value for each $S_{i,j}$ is a set of child states of $S_{i,j}$. Whenever a $S_{i,j}$ is of a vertex of interest, a Ψ operator will transform the state $S_{i,j}$ to $S_{i,j+1}$, (V_i, VL_{j+1}), at no cost by adding the vertex of interest to VL_j. A state graph *SG* defined on a graph *G* is the graph *G* whose vertices are of the same *VL*. The number of state graphs for *G* is the number of *VL*s. Since a *VL* is a sorted list storing traversed vertices of interest, in a problem with n vertices of interest, the number of state graphs is 2^n. These state graphs compose the state graph space $\mathcal{G}$, an item-enumeration graph space. A sub state graph *SSG* is a portion of a state graph *SG,* and a forest in nature. It is said to be *implicitly* generated by Γ operations in *SG*. A sub state graph space G_s is a set of sub state graphs, i.e., *SSG*s. In G_s, each point represents a sub state graph that contains the expanded vertices to each of which the path from the origin has traversed the same set of vertices of interest described by an item enumeration variable. It is said to be *implicitly* generated if it is initiated with a single source state $S_{0,0}$ and a set of Γ operations and some possible Ψ operations applied to it, to its successors, and so forth. A δ sub space graph space is a special G_S with edge cost always not smaller than 0. A path from a source *S* to a goal *D*, traversing a set of vertices of interest, is an ordered set of states (V_i, VL_j). In O*, all sub state graphs are generated implicitly. The concepts are illustrated by the example in Section IV.*B*.

Since OSTQ traverses multiple vertices of interest between the origin and the destination, the way to estimate its heuristic is similar to in C* but different from that in A* that is directly based on the currently generated vertex and the destination. Since O* is an instance of L#, the following concepts identified for L# [24] are also applicable to O*: *local heuristic, global heuristic, local admissibility,* and *global admissibility.* For these concepts, the only difference between in C* and in O* is the state used in their corresponding definitions. For example, in O*, a global heuristic, *hg*, is estimated based on the currently generated vertex, the remaining vertices of interest to traverse, and the final goal, while C*'s global heuristic is estimated based on the remaining categories of interest to traverse instead of the remaining vertices of interest to traverse [24]. The following describes these concepts in O*.

Local heuristic is the estimate to the actual cost from the current vertex to a *subgoal, sg*. A subgoal is a vertex that is a point of interest (PoI). For two vertices n and n', *local consistency* means the following inequality exists:

$$hl(n,sg) <= g^*(n, n') + hl(n', sg) \quad (2)$$

where
hl is a local heuristic, and
$g^*(n, ,n')$ is the actual cost from *n* to *n'* in the graph.

Global heuristic is the estimate to the actual cost from the current state to estimate to the final goal.

Global admissibility means the global heuristic is not larger than the actual cost from the current state to evaluate to the final goal state.

B. The O Algorithm*

O* incrementally searches all paths leading from the starting vertex, traversing the vertices of interest, until it finds a path of minimum cost to the goal. It first takes the paths most likely to lead towards the goal.

Similar to C*, O* also maintains a set of partial solutions, unexpanded leaf states of expanded vertices. These solutions are stored in an *open list*, also called a *priority queue*, which is a sorted queue based on each element's priority. Same as in C*, the priority is assigned to a state *s* based on the function (1).

Even though C* and O* use the same-form bivariate distance-plus-heuristic function *f(s)*, they use different state definitions.

The lower the *f(s)*, the higher is the priority for a vertex to be expanded. Whenever an equal f(s) occurs, the state with a larger *VisitList* will be the next to expand. Otherwise, one is randomly selected. State *A*'s *VisitList,* vl_A, is *larger* than State *B*'s *VisitList*, vl_B, i.e., $vl_A > vl_B$, if the length of vl_A is longer. In other words, the path from the start state to *A* traverses more vertices of interest than that to *B*.

For an N-point traversal problem, O* first generates the source state that contains the given vertex and an empty *VisitList*, and puts it into the open list. For all the states in the open list, the algorithm expands the state with the lowest *f(s)* value, and its children states are generated. A child state always inherits the *VisitList* of its parent whenever the child is not a vertex of interest; otherwise, the child's *VisitList* will be incremented by adding the vertex to it. The process continues until a goal state whose vertex is the final goal and *VisitList* contains all the vertices of interest or no solution is found. Once a goal state is reached, it will retrieve the obtained path using a data structure called *backpointer*, the combination of vertex identification and *VisitList,* to recursively obtain the parent until the origin state is reached.

1) O Pseudo Code*

Given an estimation function for *hg(s)*, the starting vertex *S*, the goal *D*, and the vertices of interest to visit, the pseudo code for O* is provided in Figure 1.

2) Time and Space Complexity

In O*, the complexity between a vertex and a subgoal is the same as in A*, whose time complexity and space complexity are dependent on the heuristic. For A*, in general, the time and space complexities are both exponential. Assume *b* is the branching factor, *d* is the largest depth to obtain the shortest path, and then both the time complexity and the space complexity are $O(b^d)$ [15]. A* is sub-exponential only when its heuristic *h(x)* and the actual cost *h*(x)* satisfies the following condition [15]:

```
Input:
Starting vertex S, Goal vertex D, VisitList: a sorted list to store the traversed vertices of interest
bAdd2PQ=false: Indicator that indicates whether a generated vertex is put into PQ, not be put into PQ by default
Priority queue PQ(=set of generated (s(vertex,VisitList), f(s),g(s))) begins empty.
Closed list CL (= set of previously visited (s(vertex,VisitList), backpointer, f(s),g(s))) begins empty.
Algorithm Process:
Put (S, VisitList =null), f=hg(S,null), and g(S,null)=0 into PQ
While (PQ is not empty)
{     remove the state with the lowest f having the largest VisitList from PQ. Name it n.
    If n is a goal, then  //a goal must be the predefined destination after all  vertices of interest are visited
    {       Succeeded, report the result, and END;
    }
    Else
    {       put n with its f,g,and backpointer in CL
          For each v' in successors(n.vertex)
           {      if v' is an unvisited sub goal, then
                        VisitList'=(n.VisitList).add(v'); // add n to the VisitList
              Else
              {         VisitList'=n.VisitList;
                        g(v',VisitList')= g(n)+Cost(n.vertex,v');
                        hg (v',VisitList')=calculateGlobalHeuristic(v',VisitList');
                        f (v',VisitList')=g(v',VisitList')+hg(v',VisitList')
                        Process((v',VisitList'), f, g) //decide to place the state in PQ and/or to remove other states from CL/PQ
              }
           }
    }
}
Process((v',VisitList'), f, g):
   bAdd2PQ=true;
    If v' not seen before, or (v',VisitList') currently in PQ with f(v',VisitList')>f Then
          Place/promote (v', VisitList') on priority queue with f ,g; END;
    If (v', VisitList*) is in PQ, Then
          If(VisitList* is a super set of  VisitList' && g(v',VisitList*)<g(v',VisitList')) Then
                bAdd2PQ=false;
          If(VisitList* is a sub set of VisitList' && g(v',VisitList*)>g(v',VisitList')) Then
                Delete (v',VisitList*) from PQ
    If(v',VisitList') previously expanded Then
          If f(v',VisitList')<=f Then
                bAdd2PQ=false;
          else
                Delete (v',VisitList*) from the closed list
    Else
         If(v',VisitList*) is in CL Then
                If(VisitList* is a super set of  VisitList' && g(v',VisitList*)<g(v',VisitList')) Then
                      bAdd2PQ=false;
                If(VisitList* is a sub set of VisitList' && g(v',VisitList*)>g(v',VisitList')) Then
                      Delete (v',VisitList*) from CL
   If(bAdd2PQ) Then Place (v',VisitList') with f,  and g on priority queue
```

Figure 1. The pseudo code of O*

$$|h(x)-h^*(x)|<=O(\log h^*(x)) \quad (3)$$

Where

h(x): the heuristic in A*, i.e., the local heuristic in O*; and

h(x)*: the corresponding actual cost

For O*, assume b is the branching factor, d is the largest depth to obtain the shortest path between two objects including the origin, the destination, and the specified vertices of interest to traverse, which is named as a *section path* in O*, m is the number of vertices of interest specified to traverse, and F_h is the worst time complexity to obtain a global heuristic, then the state length is $m+1$, the number of state graphs is 2^m. In the worst case, O* requires traverse paths resulting from all possible order combinations of vertices of interest and storage of all state graphs. Each section path is exponential in time and space complexity, and the corresponding time complexity is $O(m!F_h b^d)$, and space complexity is $O(m2^m b^d)$.

To reduce the time complexity to sub-exponential, the heuristics used in O* must be sufficiently close to the actual cost to reduce the number of candidate order combinations from *of permutation* level to *of sub-exponential* level.

3) Traversal Constraints of Vertices of Interest

The solution from O* must traverse all the vertices of interest to be a candidate solution to an OSTQ.

Lemma 1: The solution from O* satisfies the traversal constraint of vertices of interest.

Proof:

Use contradiction.

Assume the solution obtained from O* does not traverse at least one vertex of interest.

Since the solution is obtained, then the search stops. According to O*, if a subgoal, a specified vertex of interest, is not reached, then it cannot be added to the *VisitList* of any

vertex generated by O*, and thus there is no state whose *VisitList* contains the vertex of interest. Consequently, the final state will never be reached. In other words, the search will not stop to report a solution if there is one. This is contradicted with the assumption. So O* does provide a solution that satisfies the constraint to traverse the specified vertices of interest.

C. Completeness

Completeness means that an algorithm finds a solution if one exists.

Theorem 1: O* is complete.

The algorithm will not stop until either the goal is reached or there is no solution to the OSTQ.

D. Admissibility and Optimality

Admissibility is important in A* since it guarantees the solution is optimal. This is also true in O*.

Lemma 2: If any global heuristic for any vertex is admissible, then the solution is optimal.

Proof:

Use contradiction.

Suppose O* finds a suboptimal path, ending in goal state (G_1,vl_g) where vl_g contains all the vertices of interest since the search guarantees the solution traverses all the vertices of interest, i.e., $f(G_1,vl_g) > f^*$ where $f^* = hg^*$ *(origin state)* = cost of the optimal path. Let $hg^*(n,vl)$ as the actual cost from a state (n,vl) (n is the vertex and vl is its *VisitList*) to the goal state.

There must exist a state (n,vl) that is unexpanded, to which the path from the origin state is the start of a true optimal path, and $f(n,vl) >= f(G_1, vl_g)$ (else search would not have ended).

Also $f(n,vl) = g(n,vl) + hg(n,vl) = g^*(n,vl) + hg(n,vl)$ because (n,vl) is on the optimal path,

Since the global heuristic is globally admissible, then

$f(n,vl = g^*(n,vl) + hg(n,vl)) <= g^*(n,vl) + hg^*(n,vl) = f^*$

So $f^* >= f(n,vl) >= f(G_1, vl_g)$

Contradicting the assumption. So the solution is optimal.

Once the global heuristic is admissible, then the solution is optimal.

An algorithm is defined as *admissible* if it is guaranteed to find an optimal path from s to the goal state for any δ graph, traversing at least once for each specified vertex of interest.

Theorem 2: Once any global heuristic is admissible, then O* provides the optimal solution to the corresponding OSTQ, i.e., O* is admissible.

Proof:

First, based on Lemma 1, O* guarantees that each specified vertex of interest is traversed at least once. Then based on Lemma 2, global admissibility guarantees solution optimality. Consequently, an optimal solution that satisfies the vertices of interest traversal constraint is the optimal solution to the corresponding OSTQ, completing the proof.

E. Relationship between Different States of a Vertex

Since in a bivariate best first search, a vertex may have multiple states, the relationship between any two of its states can be used to prune unnecessary states.

Theorem 3: For a vertex v,

If(g(v,VL')>g(v,VL) AND VL is a super or equal set of VL', then the state (v,VL') is not on the optimal path.

Proof:

Since *VL* is a super or equal set of *VL'*, which means *VL* contains at least the same set of traversed vertices of interest as *VL'*, it is clear that $g^*(v,VL)>=g^*(v,VL')$. According to the given condition, $g(v,VL')>g(v,VL)$, then $g(v,VL')>g^*(v,VL)$, so $g(v,VL')>g^*(v,VL')$. Consequently, (v,VL') is not on the optimal path. Completing the proof.

IV. SCDMST: TO PROVIDING A GLOBALLY ADMISSIBLE HEURISTIC TO PROCESS OSTQ IN A FULLY CONNECTED DIRECTED GRAPH

According to the time complexity analysis in Section 3.2.2, it is desirable to reduce the depth to process OSTQ to reduce the computation time in the worst case, especially for a program within a graph where the majority of the vertices are not points of interest. Through Dijkstra, an algorithm using polynomial time in terms of all the number of vertices in a graph to obtain routes for a single-source multiple-destination routing problem, it is clear that an OSTQ processing in a general graph that contains both vertices of interest and general vertices can be efficiently transformed into a fully connected graph containing only the points of interest plus the origin and the destination. This is also the reason that TSP is primarily studied in a fully connected graph. In addition, directed graphs are more general in real world trip planning. Therefore, this section presents an instance of O* that uses a global heuristic to process OSTQ processing in a fully-connected directed graph.

Consider a directed graph, $G(V,E)$, where V and E are the set of vertices and edges, respectively, and a starting vertex s and an ending vertex e in E. Associated with each edge (i,j) in V is a cost $c(i,j)$. Let $|V|=n$ and $|E|=m$. A semi-connected directed spanning tree, *SCDST*, is defined as a graph that connects, without any cycle, all vertices with $n-1$ arcs, while vertex s only has outgoing arcs and vertex e only has incoming arcs. It is semi-connected in the sense that it does not necessarily connect any two points together. A Semi-Connected Directed Minimum Spanning Tree (*SCDMST*) is the graph with the minimum total edge cost among all *SCDST*s. In other words, the problem is to find a *SCDST*, $G(V,S)$ where S is a subset of E, such that the sum of $c(i,j)$ for all (i,j) in S is minimized. Figure 2 shows a *SCDMST* example.

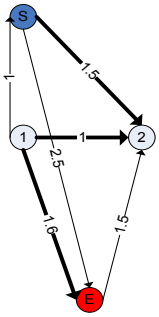
The *SCDMST* heuristic is globally admissible in a fully connected directed graph whose edge costs obey the triangle inequality. The property is proved through Theorem 4.

Theorem 4: A *SCDMST* heuristic is globally admissible in a fully connected directed graph whose edge costs obey the triangle inequality.

Proof:

Use contradiction.

Suppose the *SCDMST* heuristic is larger than the actual cost. Therefore, in the *SCDMST* there must exist at least one cost between two points is larger than their actual cost. At the same time, since edges obey the triangle equality and an



Where

S: starting vertex, E: ending vertex.
The *SCDMST* tree is highlighted in dark black. The *SCDMST* starts from S, connects vertex 1 and vertex 2, and ends at E. No cycle exists. 3 edges are used to connect 4 vertices. Only outgoing edges exist for S and incoming edges exist for E. The *SCDMST* is semi-connected since not any two points are connected through the tree. For example, 2 and E are not connected together in the obtained *SCDMST*.

Figure 2. A SCDMST example

edge on the optimal path cannot be a directed edge with its ending vertex as the starting vertex of the *SCDMST* tree or with its starting vertex as the ending vertex of the *SCDMST* tree, this means an edge with a less cost is not found by the *SCDMST*, which is contradicted with the fact that a *SCDMST* always adopts the edge of the minimum cost between two points. Completing the proof.

The O* algorithm that uses a *SCDMST* to provide a global heuristic is named as *O*-SCDMST*.

A. The D-ODPrim Algorithm to Retrieve a SCDMST from a Fully Connected Directed Graph

In this paper, D-ODPrim is proposed to obtain a *SCDMST* from a directed graph. Its pseudo code is provided in Figure 3.

Input: A connected directed weighted graph with vertices V and edges E, the starting vertex s, and the ending vertex e.
Initialize: V_{new} = {x}, where x is the starting vertex from V, E_{new} = {}
Remove incoming edges of s and outgoing edges of e in E
Repeat until V_{new} = V:
  Choose edge (u,v) from E with the minimal weight such that one vertex is in V_{new} and the other is not
  (if multiple edges with the same weight exist, choose arbitrarily)
  Add v to V_{new}, add (u, v) to E_{new}
Output: V_{new} and E_{new} describe a semi-connected directed minimal spanning tree

Figure 3. The pseudo code for D-ODPrim algorithm

D-ODPrim can be regarded as a variation of the Prim algorithm that calculates a MST for an undirected graph [25]. Similar to Prim's algorithm, D-ODPrim continuously increases the size of a tree starting with the given starting vertex until it spans all the vertices.

Next, we prove the algorithm outputs a SCDMST.

Proof:

Four steps are used to prove D-ODPrim outputs a SCDMST if a solution exists.

Assume the number of vertices is *N*. Name the obtained graph as *DG*.

Step 1: Prove that *DG* contains *N*-1 edges.

Proof: Every time a new edge connecting to a new vertex is added to E_{new}, until the *N*-1 points are added to V_{new}. Consequently, for N vertices, the algorithm will try *N*-1 times, thus *N*-1 edges will be added to form *DG*.

Step 2: Prove that *DG* does not contain a cycle.

Proof: According to the algorithm, if the edge direction is neglected in *DG*, *DG* connects any two points together with *N*-1 edges, which is impossible to have a cycle. So the directional *DG* does not contain a cycle.

Step 3: Prove that no outgoing edges for the ending vertex e and no incoming edges for the starting vertex s.

Proof: Since the algorithm removes the incoming edges of s and the outgoing edges of e from the candidate edge set, it is no way to add these edges into *DG*.

Step 4: Prove that the total edge cost is the minimum.

Proof: Use contradiction. Suppose another *SCDST*, DG_1, has a smaller total edge cost. Then there must be at least one vertex that uses an edge of smaller cost in DG_1 than in *DG*, which conflicts the fact that every time the algorithm finds the minimum cost edge for a vertex to add it to *DG*.

Based on the four steps, it is clear that D-ODPrim retrieves a SCDMST, completing the proof.

For D-ODPrim, the time complexity is $O(V^2)$ and space complexity is $O(V^2)$.

B. An Example

To illustrate how O*-SCDMST works, the following simple example is provided. Figure 4 shows the fully connected directed graph having C_{ij}=C_{ji} where i and j represents any two vertices in the graph and C_{ij} represents the cost from i to j. The OSTQ ask for a shortest route starting from *O*, traversing vertex *1* and vertex *2*, and then ending at *D*. The edge costs in the graph obey triangle inequality. Therefore, O*-SCDMST retrieves an optimal solution for the OSTQ problem, and its search process in also shown in Figure 4.

The algorithm begins with the starting point *O* with *VisitList* as null and parent (*parentpointer)* as null. Its *f* value is 3.5, the cost of SCDMST of *O*,*1*,*2*, and *D*. The state is put into the priority queue. Then, the state of the lowest *f* value, vertex *O* with *VisitList* null is expanded first and put into the closed list with a null *backpointer*, and its three children, *1*, *2,* and *D*, are generated. *D* is not a subgoal, so its *VisitList* is null, inheriting its parent. Both *1* and *2* are subgoals, so their *VisitList*s are changed accordingly. For each of these three vertices, O* calculates its heuristic based on its corresponding SCDMST, and then calculates its *f* value. All generated states are put into the priority queue.

Next, since state *(1,(1))* has the lowest *f*, 3.5, it is expanded and put into the closed list with a backpointer pointing to *(O,null)*, and its children *O,2,and D* are generated. Neither *O* nor *D* is a subgoal, so their *VisitList*s are *(1)*. *2* is a subgoal, so its *VisitList* is changed to *(1,2)*. The generated three states are put into the open list.

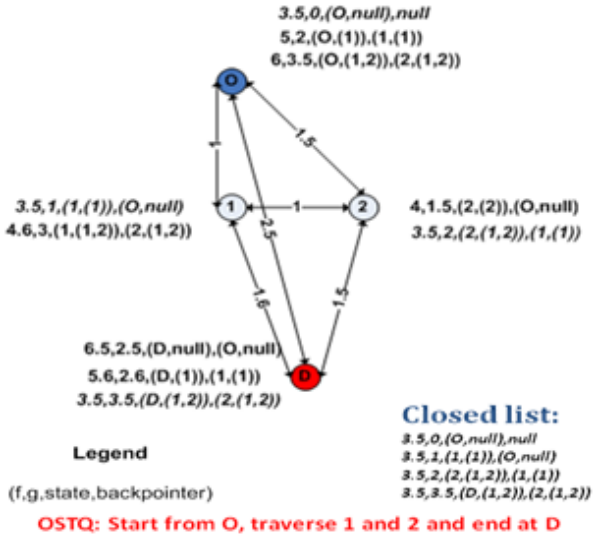


Figure 4. The OSTQ problem and O*-SCDMST search process to retrieve an optimal solution

Now *(2,(1,2))* has the lowest *f*, but it does not reach the goal state yet. So *(2,(1,2))* is expanded. Its children, *O,1,* and *D,* are generated. Now *D* is the final goal since the partial path traverses *1* and *2*. Neither *O* nor *1* is a subgoal in this case. Again, these generated states are put into the open list.

Again, based on the lowest *f*, *(D,(1,2))* is the next to be expanded. Since it is a final goal, the search stops and reports the optimal solution is O->1->2->D with 3.5 as the minimum cost.

In Figure 4, the elements in normal style are in the open list; those in italic style were in the open list first, and then moved to the closed list. The closed list in Figure 4 is its snapshot after the final goal state is expanded by O*-SCDMST. Through this example, it is clear that O*-SCDMST must store multiple states for a vertex, which a single-variate best first search cannot handle.

Figure 5 shows the corresponding sub-state-graph space representing the search space that O*-SCDMST explores to retrieve the optimal route for the OSTQ . The sub-state-graph space is composed of 2^n sub state graphs where n is the number of PoI and n=2. Note that in this example the sub-state graph with point of interest (PoI) 2 traversed has no states. The green arrow represents the Ψ operation that transforms the search space from one sub state graph to another without any cost. Each sub-state-graph is a sub graph of the original graph, with a certain set of traversed points of interest. Within each sub-state-graph, O* starts the search with a point of interest instead of the origin, O. Each PoI is traversed following a best first way. Vertices, including *O, 1, 2,* and *D,* have multiple states. These scenarios presented in O* are clearly different from the traditional single-variate best first search.

O* was implemented with C#. The experiments were performed on a Toshiba Satellite A215 Laptop with 2.0GB memory (RAM), AMD Turion™ 64*2 Mobile Technology TL-56 1.80HZ processors, and Windows Vista™ Home Premium operating system.

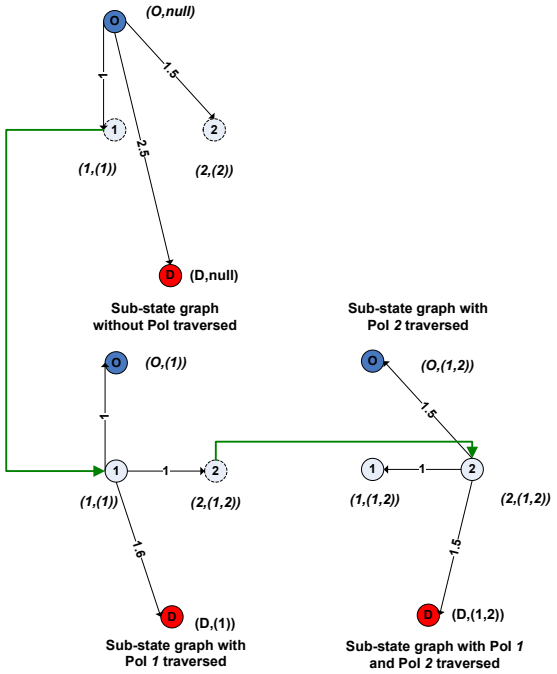


Figure 5: The sub-state-graph Space

V. EXPERIMENTS

The purpose of the experiments is to test the performance of O*-SCDMST to calculate the global heuristics in a FDG. O*-Dijkstra, a special case of O* when no heuristic is adopted, is used as the baseline. O*-Dijkstra is a consecutive network expansion algorithm to traverse multiple points of interest.

A. Data Set

An asymmetric TSP problem (Fischetti) with 34 points of interest [26], corresponding to vertices of interest in O*, is used as the data set for this experiment. The problem is a special case of Vehicle Routing Problem, and thus an asymmetric TSP [26]. The data set contains the edge costs between any two points. In this experiment, a set of OSTQ problems is generated from this data set. First, the number of points of interest consecutively changes from 2 to 15. Second, for each number of points of interest, 30 problem samples are randomly generated, i.e., the origin, the destination, and the points of interest in each problem sample are randomly selected from the 34 points. Consequently, a set of 420 problems is generated. The whole problem set is used for O*-SCDMST, the partial set with up to 12 points of interest is used for O*-Dijkstra.

B. Performance Measures

To study the performances of the two algorithms, the following performance measures are identified.

Minimum Process Time (MinPT): the minimum time required to obtain a solution for each number of points of interest (seconds);

Maximum Process Time (MaxPT): the maximum time required to obtain a solution for each number of points of interest (seconds);

Average Process Time (APT): the average time required to process a query over all runs (seconds).

C. Results and Discussion

The results are presented in Table 1. *O*-S* represents O*-SCDMST, O*-D represents O*-Dijkstra, and *NPoI* represents the number of points of interest, i.e., the number of cities to traverse. The "-" indicates that a value is not available since the time to obtain a result exceeded a reasonable expected solution time.

Figure 6 through Figure 8 are provided to visualize the performance measures provided in Table 1.

Based on MinPT, O*-SCDMST can retrieve the optimal solution within 4 seconds for an OSTQ of 15 NPoI. However, based on MaxPT, it may still require 20,858 seconds for another query with the same number of points of interest. This implies that O*-SCDMST's performance depends on how closely the selected SCDMST heuristic approaches to the actual cost.

Based on MinPT shown in Figure 6, O*-SCDMST outperforms O*-Dijkstra over all runs.

Based on MaxPT shown in Figure 7, O*-SCDMST outperforms O*-Dijkstra when NPoI larger than 2. This is due to the fact that O*-SCDMST requires additional time to compute the SCDMST heuristic, and when NPoI becomes larger, this additional time is no longer a dominant factor, instead, the obtained heuristic expedites the overall search process.

TABLE 1: PERFORMANCE RESULTS FOR O*-SCDMST, AND O*-DIJKSTRA (IN SECONDS)

	MinPT		***MaxPT***		***APT***	
NPoI	**O*-S**	**O*-D**	**O*-S**	**O*-D**	**O*-S**	**O*-D**
2	0.00	0.00	0.04	0.02	0.00	0.00
3	0.00	0.00	0.00	0.00	0.00	0.00
4	0.00	0.00	0.01	0.01	0.00	0.01
5	0.00	0.00	0.03	0.06	0.01	0.03
6	0.00	0.02	0.15	0.28	0.04	0.14
7	0.00	0.09	0.77	1.51	0.17	0.59
8	0.05	0.23	2.21	7.31	0.53	2.29
9	0.05	0.68	7.62	45.61	1.78	9.26
10	0.04	3.12	51.96	215.84	5.63	41.11
11	0.18	4.29	314.78	786.55	29.03	137.43
12	0.39	34.10	234.94	1479.48	66.82	356.69
13	3.54	-	1109.16	-	245.07	-
14	2.06	-	3900.67	-	548.08	-
15	3.37	-	20857.75	-	2204.14	-

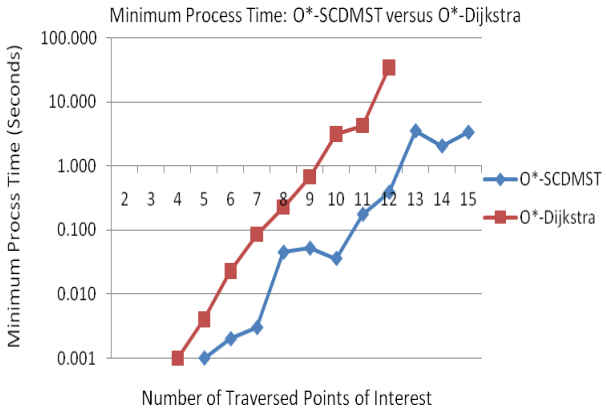


Figure 6: Minimum process time over different number of traversed points of interest: O*-SCDMST versus O*-Dijkstra

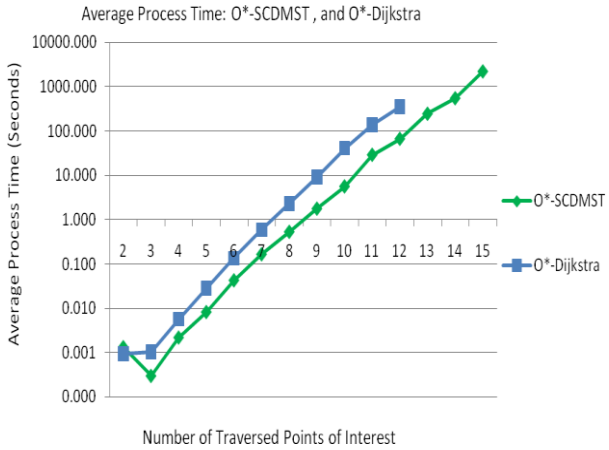


Figure 7: Maximum process time over different number of traversed points of interest: O*-SCDMST versus O*-Dijkstra

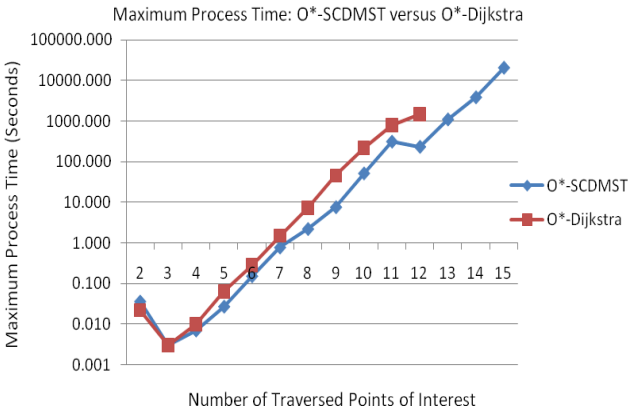


Figure 8: Average process time over different number of traversed points of interest: O*-SCDMST versus O*-Dijkstra

Based on APT shown in Figure 8, O*-Dijkstra outperforms O*-Dijkstra by an order of magnitude. On Average, O*-SCDMST can process OSTQ of up to 14 NPoI within 10 minutes.

 ISBN: 978-1-61208-118-2

In Figure 6 through Figure 8, it is noticeable that both O*-SCDMST and O*-Dijkstra are sub-exponential in time complexity.

VI. CONCLUSION

The contribution of this paper includes four components: 1) this paper proposes a novel query in a graph, OSTQ, which determines the minimum-cost path with a predefined origin-destination pair, traversing a set of vertices of interest, and provides O*, an instance of L#, to process the query in a heuristic way without explicit consideration of the order permutation; 2) A SCDMST heuristic is developed to calculate the global heuristics in O* to process OSTQ in a fully connected directed graph; 3) D-ODPrim is provided to retrieve a SCDMST for a directed connected graph; 4) The SCDMST heuristic is proved to be globally admissible if the edge costs of a fully connected directed graph obey the triangle inequality; 5) the corresponding O*-SCDMST's performance is statistically studied using a real data set. Based on the result, O*-SCDMST can retrieve an optimal solution for an OSTQ of 15 points of interest within 4 seconds at best and of 14 points of interest in 10 minutes on average. O*-SCDMST is always faster than O*-Dijkstra when the number of points of interest is larger than 2. On average, O*-SCDMST reduces the computation time by one order of magnitude compared to O*-Dijkstra.

REFERENCES

[1] S. Arora. Polynomial Time Approximation Schemes for Euclidean TSP and Other Geometric Problems. In: Proceedings of 37th IEEE Symposium on Foundations of Computer Science, Burlington, 1996, pp. 2-11.

[2] S. Arora. Approximation Schemes for NP-hard Geometric Optimization Problems: A survey. Mathematical Programming, Springer, 97 (2003) pp. 43-69.

[3] D. L. Applegate, R. E. Bixby, V. Chvátal, and W. J. Cook. The Traveling Salesman Problem: A Computational Study. Springer, 2007.

[4] N. Christofides. Worst-case Analysis of a New Heuristic for the Traveling Salesman Problem. Carnegie Mellon University, Computer Science, Tech Technical report, 1976.

[5] T. Cormen, T. Leiserson, R. Rivest, and T. Stein. Introduction to Algorithms. The MIT Press, 1997.

[6] A. Dumitrescu and J. S. B. Mitchell. Approximation Algorithms for TSP with Neighborhoods in the Plane. In: Proceedings of the 12th annual ACM-SIAM Symposium on Discrete Algorithms, Washington DC, USA, 2001, pp. 38–46.

[7] M. Held and R. M. Karp. A Dynamic Programming Approach to Sequencing Problems, Journal of the Society for Industrial and Applied Mathematics 10(1) (1962): pp. 196–210.

[8] D. J. Rosenkrantz, R. E. Stearns, and P. M. Lewis II. An Analysis of Several Heuristics for the Traveling Salesman Problem. SIAM Journal on Computing. 6 (1977): pp. 563–581.

[9] J. L. Bentley. Fast Algorithms for Geometric Traveling Salesman Problems. ORSA Journal on Computing 4, (1992), pp. 387-411.

[10] M. Held, and R. M. Karp.The Traveling-Salesman Problem and Minimum Spanning Trees. Operations Research. 18 (1970), pp. 1138-1162.

[11] Ma. Dorigo. Ant Colonies for the Traveling Salesman Problem. Université Libre de Bruxelles. IEEE Transactions on Evolutionary Computation, 1(1) (1997):pp. 53–66.

[12] E.H.L. Aarts, and J. Korst. Simulated Annealing and Boltzmann Machines: A stochastic Approach to Combinatorial Optimization and Neural Computing. John Wiley & Sons, Chichester, 1989.

[13] J. Clausen and M. Perregaard, On the Best Search Strategy in Parallel Branch-and-Bound - Best-First-Search vs. Lazy Depth-First-Search, Proceedings of the Parallel Optimization Colloquium, (1996).

[14] J. Pearl. Heuristics: Intelligent Search Strategies for Computer Problem Solving. Addison-Wesley, 1984.

[15] S. Russell and P. Norvig. Artificial Intelligence: a Modern Approach (2nd edition). Prentice Hall, 2002.

[16] P. E. Hart, N. J. Nilsson, and B. Raphael. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. IEEE Transactions on Systems Science and Cybernetics SSC4 (2) (1968) pp. 100–107

[17] E. W. Dijkstra. A Note on Two Problems in Connexion with Graphs. In Numerische Mathematik, 1 (1959), S. pp. 269–271

[18] R. E. Korf, W. Zhang, I. Thayer, and H. Hohwald. Frontier Search. Journal of the Association for Computing Machinery. 52(5) (2005) pp. 715-748

[19] R. Dechter and J. Pearl. Generalized Best-first Search Strategies and the Optimality of A*. Journal of the Association for Computing Machinery. 32(3) (1985) pp. 505-536

[20] M. Likhachev, G. J. Gordon, and S. Thrun. Ara*: Anytime A* with provable bounds on sub-optimality. In. Advances in Neural Information Processing Systems 16. MIT. Press, Cambridge, MA, 2004.

[21] A. Botea, M. Muller, J. Schaeffer. Near Optimal Hierarchical Path-Finding. In Journal of Game Development, volume 1, issue 1, (2004), pp. 7-28.

[22] S. Russell. Efficient Memory-bounded Search Methods. In Proceedings of Tenth European Conference on Artificial Intelligence, pp. 1-5. Chichester, England: Wiley, 1992.

[23] R. Zhou, and E. A. Hansen. Memory-Bounded A* Graph Search. Proceedings of the Fifteenth International Florida Artificial Intelligence Research. May 2002.

[24] Q. Lu, and K. Hancock. C*: A Bivariate Best First Search to Process Category Sequence Traversal Queries in a Transportation Network. geoprocessing, pp.127-136, 2010 Second International Conference on Advanced Geographic Information Systems, Applications, and Services, 2010.

[25] M. Fischetti, P. Toth, and D. Vigo. A branch and bound algorithm for the Capacitated Vehicle Routing Problem on Directed Graphs. Operations Research, Vol 42, pp. 846-859. 1994.

[26] Robert C. Prim: Shortest connection networks and some generalizations. In: Bell System Technical Journal, 36, pp. 1389–1401,1957

 ISBN: 978-1-61208-118-2