2024 American Control Conference (ACC)
July 8-12, 2024. Toronto, Canada

Analysis of Backtracking A* For Resource Constrained Shortest Path Problems

1st Bryce Ford
Mechanical and Aerospace Engineering
The Ohio State University
Columbus Ohio, United States
ford.1009@osu.edu

2nd Mrinal Kumar
Mechanical and Aerospace Engineering
The Ohio State University
Columbus Ohio, United States
kumar.672@osu.edu

***Abstract*—The Resource Constrained Shortest Path Problem (RCSPP) requires a minimum-cost simple path between two nodes that is subject to a resource consumption constraint. In this paper, we consider the Backtracking A^* algorithm presented in previous works applied to the general RCSPP. Backtracking A^* attempts to solve the RCSPP by iterative modification of paths generated by a shortest path algorithm such as A^*. We consider the completeness of Backtracking A^* and demonstrate that it cannot be a complete algorithm. Then we propose a complete, modified version of Backtracking A^*. Finally, we give a result for the time complexity of Backtracking A^*, and demonstrate it under worst-case conditions applied to randomly generated graphs.**

***Index Terms*—Path Planning, Resource Constrained Shortest Path**

I. Introduction

The resource-constrained shortest path problem (RCSPP) or more generally the Constrained Shortest Path Problem (CSP) is a well-studied class of path planning problems. The objective is to seek to find a minimum-cost path subject to a path-dependent constraint similar to the Knapsack Problem. The RCSPP appears in many domains including robot path-planning applications where path-dependent constraints are considered, such as fuel constraints in [1]. Quality of Service routing is commonly considered [2], [3].

Backtracking A^* has been previously considered in [4] where it is applied to a trajectory planning problem subject to a path-dependent integral constraint. No formal analysis of the time complexity or completeness of the algorithm was performed. In this work, we propose an extension of the original Backtracking A^* algorithm to the general RCSPP. We consider the modifications required to make Backtracking A^* complete, and the implications of this on the time complexity.

A. Problem Statement

A RCSPP is solved on a connected graph $G = (N, E)$ with nodes $v \in N$ and edges $e \in E$. The graph G has a cost function $c : E \to \mathbb{R}$ and a resource consumption or loading function $l : E \to \mathbb{R}$ mapping edges to real-valued costs and loads. G has $n = |N|$ nodes and $m = |E|$ edges. A path is a sequence of nodes described as $p = \{v_0, v_1, \ldots, v_i\}$. The RCSPP on G is defined as

$$p^* = \underset{p \in P(s,t)}{\operatorname{argmin}} \sum_{e \in p} c(e) \tag{1}$$

such that

$$\sum_{e \in p} l(e) \leq L \tag{2}$$

where $p \in P(s,t)$ is the set of all simple paths in G that connect a start node s to a goal node t. $L \in \mathbb{R}^+$ is a resource constraint or loading limit. This definition of the RCSPP is adopted from the linear programming formulation posed in [5]. In this work RCSPPs with a single constraint are considered. Even with a single resource constraint, the RCSPP is NP-Complete [6], so no polynomial time exact solutions exist.

The core of most approaches to the RCSPP lies in finding k shortest paths between s and t [7]–[9]. In a fully connected graph with n nodes, there will be $O(n!)$ simple paths connecting the s and t. This direct solution approach rapidly becomes intractable as the size of the graph grows due to this result.

B. Previous Work

Algorithms that attempt to solve or approximate solutions to the RCSPP can be classified under two broad categories. The first category approaches the RCSPP by posing it as an integer program and relaxing it to an unconstrained minimization problem. This technique was introduced by Handler and Zang [7], and Beasley and Christofides [8]. Handler and Zang propose solving the k-th shortest path problem and using the relaxation to reduce the number of shortest path computations. Beasley and Christofides use a sub-gradient approach to solve the relaxed problem.

This process is formalized as Lagrange Relaxation-based Aggregated Cost (LARAC) by Juttner et al. [2]. In [3] the runtime of LARAC is shown to be strongly polynomial with a runtime $O((m + n\log(n))^2)$. LARAC is generalized to handle multiple constraints in [10]. The primary advantages of LARAC are that it terminates if no solution exists, and guarantees a relaxed-cost optimal solution in polynomial time if a solution does exist.

979-8-3503-8265-5/$31.00 ©2024 AACC

The second category approaches the RCSPP using dynamic programming techniques. Joksch introduced the dynamic programming approach in [11] as a generalization of the shortest path problem. Desrochers and Soumis solve the dynamic program in [9] with a label-setting approach. The label-setting approach processes nodes in a minimum-cost first order similar to Dijkstra's Algorithm. Labels that obey Eq. 2 are added to a permanent set of labels. Unlike LARAC this approach can guarantee the optimal solution to the RCSPP, but in the worst case will label every feasible path in the graph. Label-setting is built on with preprocessing in [12] to label infeasible paths before the main optimization procedure. Modern approaches include a bi-directional heuristic search-based approach [13].

II. Proposed Solution

In this section, we re-state the Backtracking A^* [4] algorithm and propose an extension to apply it to the general RCSPP. In Dijkstra's Algorithm or more generally A^* can be applied to the RCSPP to generate a minimum cost guess. Dijkstra's is a greedy algorithm, so it assumes that the first path that it finds is the optimal path. If the cost-minimizing path violates the resource constraint (Eq. 2), the root of the constraint violation lies in the path to the constraint violation, not the node where the violation occurred. The motivating concept for Backtracking A^* is that A^* can be used to generate minimum-cost candidate paths, the a backtracking procedure can "reset", or backtrack, the search to a state prior to a constraint violation. After a backtrack, A^* is able to make a locally sub-optimal decision, and "shed" some of the resource consumption accumulated during the search. The desired effect is that repeated backtracks will nudge the search towards lower resource consumption solutions.

A. The Algorithm

In Backtracking A^*, the A^* algorithm [14] is used to generate cost-minimising candidate paths. A^* is modified to track the load of candidate nodes. It returns the resulting closed set V and open set F along with the current path p_c. The algorithm terminates if it encounters a node that violates the resource constraint (Eq. 2) and p_c is set to the path that violated the resource constraint. If p_c is a path $s \rightarrow t$ and does not violate the loading constraint then the Backtracking A^* terminates returning p_c. If p_c violates the resource constraint, the search backtracks.

The purpose of a backtrack is to reset the search to a state prior to the constraint violation. Two pieces of information are needed to backtrack in addition to V and F; the current path p_c, and the depth to backtrack k. To perform a backtrack the node p_c^k is identified, also referred to as a stop-node. A breadth-first search is used to find all of the nodes in V and F to form a sub-tree originating at p_c^k consisting of the sets V_b and F_b. The nodes identified are removed from V and F.

A key addition is this work is what to do after nodes are removed in the backtrack step. In previous work, backtracks only encompassed removing nodes from V and F. This can lead to cases where a backtrack removes all of the nodes from F terminating the search. To remediate this and expand the set of potential paths considered by the search, new paths containing the removed nodes V_b and F_b must be considered. A node removed from V or F during the backtrack can be connected to a node currently in V, thus re-adding it to F.

Algorithm 1 Backtracking A^*

Require: G, s, t, L
  $F \leftarrow s$
  $V \leftarrow \emptyset$
  $D \leftarrow \emptyset$
  while $F \neq \emptyset$ **do**
    $\{F, V, p_c\} \leftarrow ASTAR(F, V, L, s, t)$
    if $l(p_c) \leq L$ **and** $c == t$ **then**
      return p_c
    end if
    $k \leftarrow STOP(p_c)$
    $\{F, V, D\} \leftarrow BACKTRACK(F, V, D, p_c, k)$
  end while

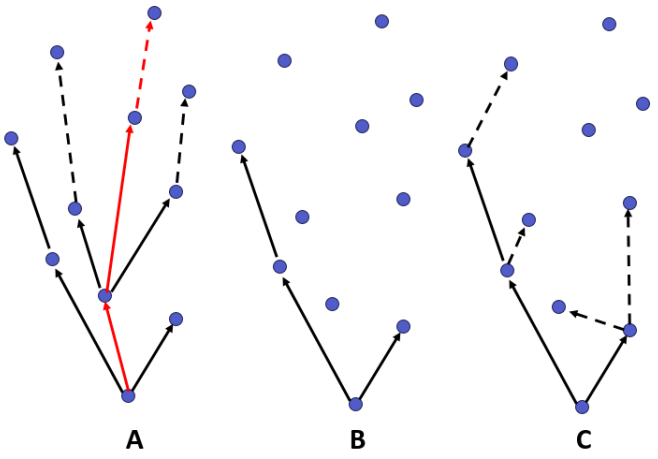

Fig. 1. Illustration of the backtracking process. Solid arrows represent connections in V and dashed arrows represent connections in F. In A, a path is encountered that violates the resource constraint. In B, nodes in the sub-tree of the stop node are removed from the open and closed sets. In C, nodes that were removed from the open and closed sets in B are rewired back into the open set.

Algorithm 2 BACKTRACK

Require: V, F, D, p_c, k
  $\{V_b, F_b\} \leftarrow BFS(V, F, p_c^k)$
  $V \setminus V_b,\ F \setminus F_b$
  $D \setminus (V_b \in D)$
  $D \cup p_c^k$
  for $v \in \{V_p, F_p\}$ **do**
    $U \leftarrow adj(v) \notin D$
    $F \leftarrow \underset{u \in U}{\operatorname{argmin}}\, c(s \rightarrow u) + c(u \rightarrow v) + h(v, t)$
  end for
  return V, F, D

This rewires the search to allow it to consider paths that are sub-optimal with respect to their cost. To prevent infinite loops, a set of stop-nodes D is maintained to prevent previously backtracked paths from being rewired back into the search. This process is detailed in Fig. 1. It should be noted that a k value of 0 results in a backtrack that does not remove any nodes, and thus represents the search neglecting the current constraint violation.

Choosing what node to backtrack to is a critical step in the backtracking process as it controls where deviations from the cost-optimal path are made. This decision is referred to as the stopping criterion and it is purely heuristic in nature. In previous work [4], a stopping criterion is proposed that compares the minimum resource consumption at a node to the current resource consumption, and stops when the ratio is below some threshold λ. This stopping criterion backtracks the search to a point where it is more similar to the minimum resource consumption solution but requires tuning the λ parameter. In this work, we propose a stopping criterion that stops at the maximum load edge in the path p_c. This can be stated as follows:

$$k = \underset{k \in p_c}{\operatorname{argmax}}\, l(p_c^{k-1} \rightarrow p_c^k) = STOP(p_c) \tag{3}$$

where the node backtracked is pointed to by the maximum resource consumption edge in p_c. This stopping criterion resets the search where the resource consumption is at a local maximum and thus will force a choice that leads to a lower path of resource consumption.

B. Completeness

To determine if Backtracking A^* is a complete algorithm we must consider the choice of stop-node p_c^k on the state of the search. In Alg. 1, after the initial call of $ASTAR$ assuming a valid solution is not found, the search is presented with a number of choices. These choices represent which node in the path p_c to backtrack to. Considering all possible choices after each subsequent backtrack, a tree structure becomes apparent as can be seen in Fig. 2. A sequence of backtracks can be represented as follows:

$$p_0 \xrightarrow{k^1} p_1 \xrightarrow{k^2} \dots \xrightarrow{k^d} p_d \tag{4}$$

where d represents the length of the sequence or the depth of the decision tree.

Suppose that for any pair paths $p_u \in P(s,u)$ and $p_v \in P(s,v)$, there exists a sequence of backtrack operations (Eq. 4) that transform p_u into p_v. In a complete graph, with prior knowledge of $p_v = \{v_0, v_1, \dots, v_i\}$, the search can backtrack to a depth of j where j is the length of p_c. The search backtracks at this depth until $p_c^0 = v_0$. This process is repeated at incrementally decreasing backtracking depths as follows:

$$\begin{array}{ll} k = j & p_c = \{v_0, \dots\} \\ k = j-1 & p_c = \{v_0, v_1, \dots\} \\ k = j-2 & p_c = \{v_0, v_1, v_2, \dots\} \\ \vdots & \vdots \\ k = j-i & p_c = \{v_0, v_1, v_2, \dots, v_i\} \end{array} \tag{5}$$

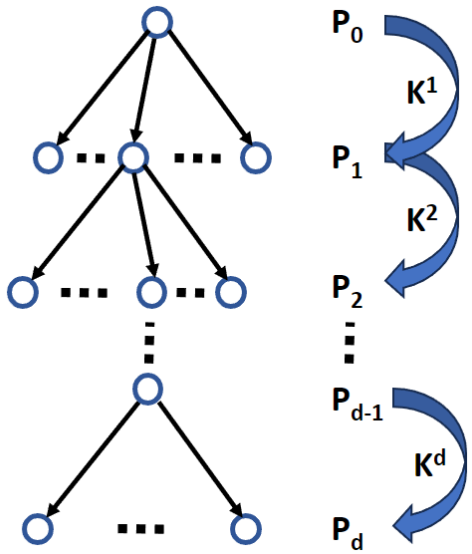

Fig. 2. Decision tree induced by repeated stop node choices during backtracks. P_0 represents the initial minimum cost partial solution. Each subsequent layer represents new partial solutions generated by each potential stop node in the previous layer.

where i is the length of p_v. This only applies if the path p_u is the initial path generated. However, this approach requires prior knowledge of the solution to the RCSPP. The only way to create a stopping criterion that guarantees a solution if one exists is to know the solution a priori, thus Backtracking A^* cannot be a complete algorithm.

To extend Backtracking A^* to be a complete algorithm it must be able to consider the entire decision tree. Backtracking A^* behaves like a depth-first search. It increases its tree depth until the backtrack at p_d results in a valid solution with respect to the resource constraint or empty frontier where it terminates without a solution. We propose the Complete Backtracking A^* to demonstrate what is required to guarantee a solution.

The Complete Backtracking A^* algorithm stores the state $\{V, F, D, p_c, k\}$ in a list $PathList$ that represents the sequence of partial solutions encountered on the current path down the decision tree. The set $PathSet$ contains all paths seen by the search, it is used to prevent infinite loops. If the search reaches the bottom of the tree, e.g. the backtrack produces an empty open set F, instead of terminating the search pops another the previous state in the tree and chooses a different stop-node. Preprocessing the graph by using Dijkstra's Algorithm to the tree of resource consumption minimizing paths is used to prune any unreachable nodes to guarantee that Alg. 3 terminates. This extension allows Complete Backtracking A^* to consider every potential sequence of backtracks in the decision tree. As shown in Eq. 5, any path can be found using a sequence of backtracks, and Complete Backtracking A^* will consider all possible backtracks sequences it will find a solution assuming it exists. Thus Complete Backtracking A^* is in fact complete.

Algorithm 3 Complete Backtracking A^*

Require: G, s, t, L
  $F \leftarrow s$
  $V \leftarrow \emptyset$
  $D \leftarrow \emptyset$
  Containers to store decision tree information :
  $PathSet \leftarrow p_c$
  $StateList \leftarrow \{V, F, \emptyset, p_c, k = 0\}$
  while True **do**
    $\{F, V, p_c\} \leftarrow ASTAR(F, V, L, s, t)$
    if $l(p_T) \leq L$ **and** $c == t$ **then**
      return p_T
    end if
    Check if the path has been visited :
    if $p_c \notin PathSet$ **and** $p_c! = \emptyset$ **then**
      $PathSet \leftarrow p_c$
      $StateList \leftarrow \{V, F, D, p_c, k\}$
    end if
    If no stop-nodes remain, discard the current path :
    if $StateList(end)(p_c) == \emptyset$ **then**
      $POP(StateList(end))$
    end if
    Pick a new stop-node and backtrack :
    $k \leftarrow STOP(StateList(end))$
    $StateList(end) \setminus p_T^k$
    $\{F, V, D\} \leftarrow BACKTRACK(StateList(end), k)$
  end while

C. Time Complexity

To determine the time complexity of Backtracking A^* the time complexity of the backtracking procedure must be determined. To describe the worst-case behavior of the algorithm, fully connected graphs are considered. Backtracking A^* will call A^* and the backtracking procedure for each level in the decision tree. This means that the worst-case depth of the decision tree must be determined as well as the time complexity of the backtracking procedure (Alg. 2).

The time complexity of A^* reduces to the time complexity of Dijkstra's Algorithm if the heuristic $h(\cdot) = 0$. Dijkstra's Algorithm will consider each edge in the graph and will place each node in the open set once. If a Fibonacci heap is used as the open set Dijkstra's time complexity is $O(m + n\log(n))$ [15].

The worst-case time complexity for a backtrack operation is determined by the size of the sub-tree enumerated and the number of nodes required into the open set. In the case that a backtrack to the depth of s occurs when the closed set encompasses all nodes in the graph, $n-1$ nodes will be enumerated by the breadth-first search resulting in a complexity of $O(n)$ in the nodes removed from the closed set. The other extreme is where all nodes are present in the open set, only possible in a complete graph. Assuming a Fibonacci heap is used for the open set, this case results in a complexity of $O(n\log(n))$. In a general depth backtrack after the sub-tree is enumerated at most n nodes will be rewired, and the adjacency list of each node will be considered. If the adjacency list of every node is considered for rewiring then the entire set of edges ($O(m)$) will be processed. Because insertion is amortized $O(1)$, the insertion into the open set will run in $O(n)$. The total time complexity for the backtracking procedure will be $O(m + n\log(n))$.

Given that A^* and backtrack operations have the same worst-case time complexity, Backtracking A^* will have a time complexity of $O(d(m + n\log(n)))$ where d is the depth of the decision tree. Each internal decision in the tree will have between 2 and n children. This is due to the fact that in a complete graph, all possible paths $s \rightarrow t$ will be between 2 and n nodes, meaning the decision tree will be a n-ary tree. The depth of an n-ary tree is $O(\log(\cdot))$ in the number of nodes it holds. Because all possible single source paths in G are reachable by traversing the decision tree, there will be at least that many nodes in the tree. A complete graph has $O(n!)$ simple paths so d will scale as $O(log(n!))$ in the size of the graph. To account for the possibility that a path can be reached with multiple sequences of backtracks and thus will have multiple instances in the tree, the bound is relaxed to $O(n\log(n))$. This results in a time complexity of Backtracking A^* of $O((n\log(n))(m + n\log(n)))$.

As Complete Backtracking A^* extends Backtracking A^* to be a brute force search of possible backtrack sequences, it will find every path between s and t in the worse case. Complete Backtracking A^* has a time complexity of at least $\Omega(n!(m + n\log(n)))$ due to this fact. This is only a lower bound as there may be multiple backtrack sequences that can produce a solution path.

III. Numerical Results

In this section, we perform a numerical study to demonstrate the time complexity of Backtracking A^*. We focus on the number of backtracks before the search terminates as we want to verify the depth of the decision tree structure scales as $O(n\log(n))$. To do this we must induce worst-case behavior in the search. This is done by using randomly generated complete graphs.

To generate a random complete graph n nodes are created with (x, y) coordinates sampled from a uniform distribution. Each node is then connected to every other node to create a set of m edges. The cost of an edge (u, v) is defined as the Euclidean distance between the nodes u and v. The load is sampled from a uniform distribution such that $l(e) \in (0, 1]$. The s and t nodes are picked to be in opposite corners of the domain (Fig. 3), this does impact the result of the search as the graph is fully connected.

The resource constraint is set to 0 so that the search always reaches the bottom of the decision tree without finding a valid solution. Complete graphs with a range of $100 - 900$ nodes are constructed using increments of 50 nodes. Backtracking A^* is run to termination on 100 different randomly generated complete graphs for each of the graph sizes. The number of

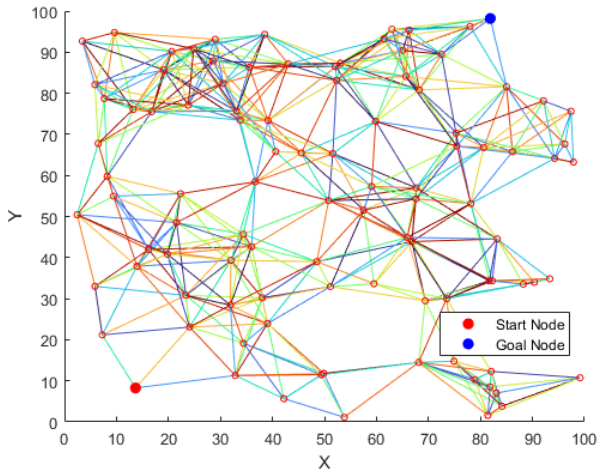

Fig. 3. Random graph generated by sampling node coordinates from $(x, y) \in [0, 100]$. This is not a fully connected graph as they do not translate well to visualization. The edge color maps to the resource consumption along each edge, with blues mapping to low resource consumption and, red edges mapping to high resource consumption. The s and t nodes are highlighted in blue and red respectively.

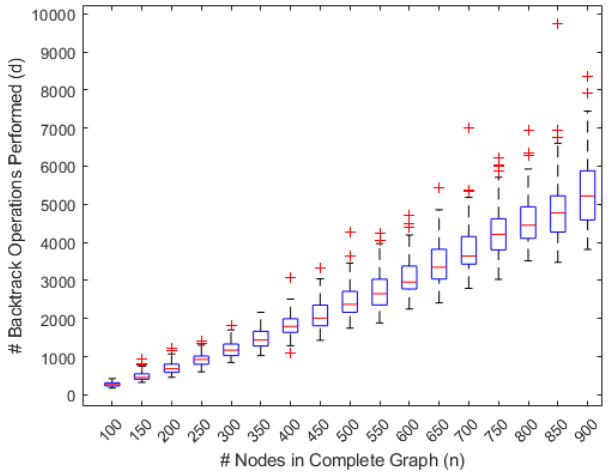

Fig. 4. Box plots represent the distribution of backtrack backtracked performed in each Backtracking A^* run at each graph size.

times Backtracking A^* backtracks before it terminates was recorded. The results can be seen in Fig. 4.

If the scaling of the number of backtracks is, in fact, $O(n\log(n))$, the ratio $(\#Backtracks)/(nlog(n))$ will tend to a constant value as n increases. This trend can be seen in Fig. 5 where $(\#Backtracks)/(n\log(n))$ is computed for the minimum, maximum, and average number of backtracks for a given graph size. In each case, the ratio increases at a slowing rate. This demonstrates that the depth of the decision tree is approximately $O(n\log(n))$, but a more accurate characterization of duplicate paths will give a tighter bound. The more important result is that the depth is not exponential in n, showing that subsequent backtracks do induce a decision

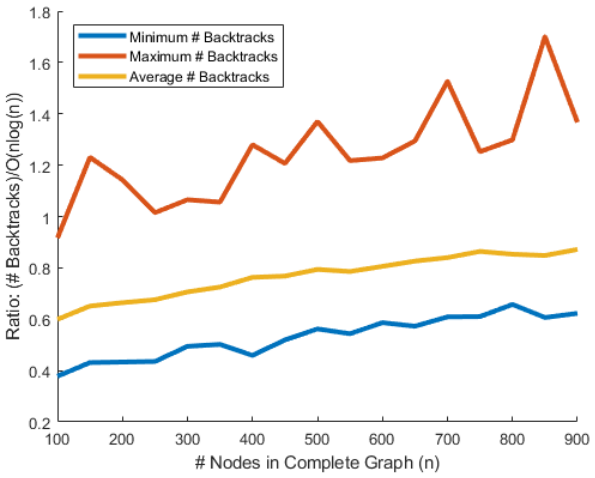

Fig. 5. The ratio of the minimum, maximum, and average number of backtracks at each graph size to the proposed complexity of $O(nlog(n))$.

tree structure in the search.

IV. Conclusion

We presented an extension to the Backtracking A^* algorithm applied to the general Resource Constrained Shortest Path Problem. We proposed a framework to analyze backtracking operations as a decision tree and used the framework to demonstrate that Backtracking A^* is not a complete algorithm. A new algorithm, Complete Backtracking A^* is proposed to show what needs to be done to make Backtracking A^* a complete algorithm. We use first-principles analysis combined with the decision tree structure to state a time complexity bound for Backtracking A^*. Finally, we perform a numerical study to show the worst-case behavior of Backtracking A^* applied to complete graphs. The Numerical study confirms our time complexity bound by showing that the number of backtracks performed matches the theoretical depth of the decision tree. Future work includes expanding our analysis of Complete Backtracking A^*, and a more exact method to model double counting paths in the decision tree.

References

[1] K. Sundar, S. Venkatachalam, and S. Rathinam, "Formulations and algorithms for the multiple depot, fuel-constrained, multiple vehicle routing problem," in *2016 American Control Conference (ACC)*. IEEE, 2016, pp. 6489–6494.

[2] A. Juttner, B. Szviatovski, I. Mécs, and Z. Rajkó, "Lagrange relaxation based method for the qos routing problem," in *Proceedings IEEE INFOCOM 2001. conference on computer communications. twentieth annual joint conference of the ieee computer and communications society (Cat. No. 01CH37213)*, vol. 2. IEEE, 2001, pp. 859–868.

[3] Y. Xiao, K. Thulasiraman, G. Xue, A. Jüttner, and S. Arumugam, "The constrained shortest path problem: Algorithmic approaches and an algebraic study with generalization," *AKCE International Journal of Graphs and Combinatorics*, vol. 2, no. 2, pp. 63–86, 2005.

[4] B. T. Ford, R. Aggarwal, M. Kumar, S. G. Manyam, D. Casbeer, and D. Grymin, "Backtracking hybrid a* for resource constrained path planning," in *AIAA SCITECH 2022 Forum*, 2022, p. 1592.

[5] L. D. P. Pugliese and F. Guerriero, "A survey of resource constrained shortest path problems: Exact solution approaches," *Networks*, vol. 62, no. 3, pp. 183–200, 2013.

[6] M. R. Garey and D. S. Johnson, "Computers and intractability," *A Guide to the*, 1979.
[7] G. Y. Handler and I. Zang, "A dual algorithm for the constrained shortest path problem," *Networks*, vol. 10, no. 4, pp. 293–309, 1980.
[8] J. E. Beasley and N. Christofides, "An algorithm for the resource constrained shortest path problem," *Networks*, vol. 19, no. 4, pp. 379–394, 1989.
[9] M. Desrochers and F. Soumis, "A generalized permanent labelling algorithm for the shortest path problem with time windows," *INFOR: Information Systems and Operational Research*, vol. 26, no. 3, pp. 191–212, 1988.
[10] Y. Xiao, K. Thulasiraman, G. Xue, and M. Yadav, "Qos routing under multiple additive constraints: A generalization of the larac algorithm," *IEEE Transactions on Emerging Topics in Computing*, vol. 4, no. 2, pp. 242–251, 2015.
[11] H. C. Joksch, "The shortest route problem with constraints," *Journal of Mathematical Analysis and Applications*, vol. 14, no. 2, pp. 191–197, 1966.
[12] I. Dumitrescu and N. Boland, "Improved preprocessing, labeling and scaling algorithms for the weight-constrained shortest path problem," *Networks: An International Journal*, vol. 42, no. 3, pp. 135–153, 2003.
[13] B. W. Thomas, T. Calogiuri, and M. Hewitt, "An exact bidirectional a* approach for solving resource-constrained shortest path problems," *Networks*, vol. 73, no. 2, pp. 187–205, 2019.
[14] R. Dechter and J. Pearl, "Generalized best-first search strategies and the optimality of a," *Journal of the ACM (JACM)*, vol. 32, no. 3, pp. 505–536, 1985.
[15] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*. MIT press, 2022.