CSC 510 MILESTONE 2: BRUTE-FORCE ROUTING AND COMPLEXITY ANALYSIS

DANIEL SUNGKYUN CHA

1. Introduction

We present a brute-force strategy for our smart routing problem in the context of a road navigation system for the California based DriveBot labs. Specifically, we aim to identify the most efficient path between two vertices in a directed graph $G = (V, E)$ by exhaustively enumerating all simple paths and selecting the one that minimizes the sum cost. Though this approach is prohibitively expensive for larger graphs – it establishes a theoretical benchmark for future optimization.

Problem Formulation. To recap, let $G = (V, E)$ denote a directed graph where V represents intersections or road nodes in California, and E is the set of directed and traversable roads excluding major freeways. Let $w : E \to \mathbb{R}^+$ be a weight function where $w(e)$ denotes the cost of edge e; representing real-world quantities such as distance, traffic density, and energy expenditure.

Given source $s \in V$ and target $t \in V$, our goal is to find a simple path $P = (v_1 = s, v_2, \ldots, v_k = t)$ that satisfies the following:

(1) $(v_i, v_{i+1}) \in E$ for all i (i.e., P is a valid path),
(2) $v_i \neq v_j$ for all $i \neq j$ (i.e., P is simple),
(3) The total edge weight is given by:

$$d(P) = \sum_{i=1}^{k-1} w(v_i, v_{i+1})$$

(4) The route cost or pricing function $\pi(P)$ is computed by evaluating a pricing function $\pi : \mathbb{R}^n \to \mathbb{R}$ over a feature vector x_P associated with path P, such that:

$$\pi(P) = a_1 x_1^{c_1} + a_2 x_2^{c_2} + \cdots + a_n x_n^{c_n}$$

Date: June 2025.

Here, $x_P = (x_1, \ldots, x_n)$ includes features like $d(P)$ (total distance), traffic index, demand, and vehicle availability. This distinction separates the structural weight of the path from its economic cost.

Let $\mathcal{S}(s,t)$ be the set of all simple paths from s to t. Our brute-force approach computes:

$$P^* = \arg \min_{P \in \mathcal{S}(s,t)} \pi(P)$$

1.1. **Metric Considerations.** We define the *distance* between two vertices $u, v \in V$ as:

$$d(u,v) = \min_{P \in \mathcal{S}(u,v)} d(P)$$

This function satisfies non-negativity and the triangle inequality but not symmetry; thus, (V, d) forms a *quasi-metric space*.

2. Brute-Force Algorithm

Given $s, t \in V$, we define the brute-force strategy as the process of enumerating all simple paths $P \in \mathcal{S}(s,t)$, computing the pricing function $\pi(P)$ for each, and returning the optimal solution:

$$P^* = \arg\min_{P \in \mathcal{S}(s,t)} \pi(P)$$

Definition 2.1. A *simple path* is a finite sequence of distinct vertices $v_1, v_2, \ldots, v_k$ such that $(v_i, v_{i+1}) \in E$ for $1 \leq i < k \in \mathbb{Z}^{++}$.

The Algorithm.

Let $G = (V, E)$ be a finite directed graph with edge weight function $w : E \to \mathbb{R}^+$. Let $s, t \in V$ be the source and target vertices, respectively. Let $\mathcal{S}(s,t)$ be the set of all simple (loop-free) paths from s to t in G.

(1) Define the search space $\mathcal{S}(s,t)$ as the collection of all vertex sequences $(v_1, \ldots, v_k)$ such that:
   - $v_1 = s$, $v_k = t$,
   - $(v_i, v_{i+1}) \in E$ for all $1 \leq i < k$,
   - $v_i \neq v_j$ for all $i \neq j$ (i.e., paths are simple).

(2) For each path $P \in \mathcal{S}(s,t)$, compute its associated feature vector $x_P = (x_1, \ldots, x_n)$, where each component encodes a domain-relevant metric such as: $x_1 = d(P) = \sum_{i=1}^{k-1} w(v_i, v_{i+1})$ [path length], $x_2 =$ expected congestion index, $x_3 =$ current vehicle availability,

(3) Evaluate the pricing function $\pi(P)$ for each $P \in \mathcal{S}(s,t)$:

$$\pi(P) = a_1 x_1^{c_1} + a_2 x_2^{c_2} + \cdots + a_n x_n^{c_n}$$

where $a_i, c_i \in \mathbb{R}$ are fixed coefficients determined by system policy or business logic.

(4) Return the path $P^* \in \mathcal{S}(s,t)$ minimizing $\pi(P)$:

$$P^* = \arg\min_{P \in \mathcal{S}(s,t)} \pi(P)$$

Although the construction of $\mathcal{S}(s,t)$ is inherently exponential in the number of vertices (as each simple path must avoid cycles), we achieve the objective of optimization of the brute-force algorithm: a global minimum over the discrete combinatorial search space of simple paths.

Pseudocode Draft.

Input: Directed graph $G = (V, E)$ with edge weights w, source s, target t
Output: Optimal simple path $P^* \in \mathcal{S}(s, t)$ minimizing $\pi(P)$

(1) Initialize stack with $(s, [s], 0)$.
(2) While stack not empty:
- Pop $(u, \text{path}, \text{cost})$.
- If $u = t$: append $(\text{path}, \text{cost})$ to results.
- Else, for each neighbor v of u not in path:
  - Push $(v, \text{path} + [v], \text{cost} + w(u, v))$.

(3) Return path in results with minimal pricing value.

Canonical Example: The Five-Node Graph. Consider the directed graph G below, where our goal is to compute the optimal path from node A to node E under the brute-force strategy.

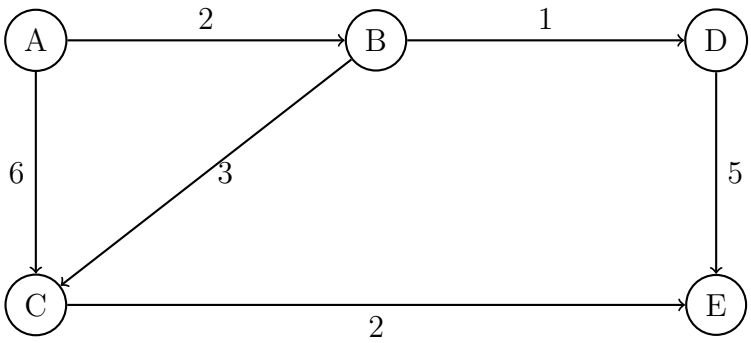

FIGURE 1. Five-node weighted graph illustrating multiple simple paths from A to E

Exhaustive Search. We simulate the algorithm's behavior with the initial stack `[(A, [A], 0)]`, expanding each node as follows:

- **Pop** $(A, [A], 0)$:
  - Push $(B, [A, B], 2)$
  - Push $(C, [A, C], 6)$
- **Pop** $(C, [A, C], 6)$:
  - Push $(E, [A, C, E], 8)$
- **Pop** $(E, [A, C, E], 8)$: *store path*
- **Pop** $(B, [A, B], 2)$:
  - Push $(C, [A, B, C], 5)$
  - Push $(D, [A, B, D], 3)$
- **Pop** $(D, [A, B, D], 3)$:
  - Push $(E, [A, B, D, E], 8)$
- **Pop** $(E, [A, B, D, E], 8)$: *store path*
- **Pop** $(C, [A, B, C], 5)$:
  - Push $(E, [A, B, C, E], 7)$
- **Pop** $(E, [A, B, C, E], 7)$: *store path*

Result. The brute-force algorithm finds three simple paths:

$$\begin{aligned} &(1)\ A \to C \to E && \text{Cost: } 8 \\ &(2)\ A \to B \to D \to E && \text{Cost: } 8 \\ &(3)\ A \to B \to C \to E && \text{Cost: } 7 \end{aligned}$$

Thereafter, the algorithm returns the optimal path:

$$P^* = [A, B, C, E], \qquad \pi(P^*) = 7$$

3. Theoretical Context

Theorem 3.1. *The number of simple paths between two nodes in a directed graph with n vertices is upper-bounded by $O(n!)$.*

Proof. Let $G = (V, E)$ be a directed graph with $|V| = n$, and suppose that we wish to compute the number of simple paths from a fixed source s to the target t.

Consider the directed acyclic graph (DAG) G that is densely connected. Thus, for every pair of distinct vertices (u, v) such that u precedes v in some topological order, the edge (u, v) is present in E. Let $V' = V \setminus \{s, t\}$ be the set of intermediate vertices. A simple path from s to t is uniquely determined by choosing a subset of V' (since we must avoid revisiting vertices) and specifying a linear ordering of that subset consistent with the edge directions in G. In a complete DAG, any such ordering is feasible.

Thus there are at most 2^{n-2} subsets of V', and each subset of size k has $k!$ permutations. So, the total number of simple paths is bounded by:

$$\sum_{k=0}^{n-2} \binom{n-2}{k} \cdot k! = \sum_{k=0}^{n-2} \frac{(n-2)!}{(n-2-k)!}$$

Our expression is asymptotically dominated by the term when $k = n - 2$, yielding:

$$\frac{(n-2)!}{0!} = (n-2)!$$

Therefore, the number of simple paths is $O(n!)$ in the worst case. The bound is tight up to a constant factor since constructing a complete DAG yields approximately $(n-2)!$ paths from s to t. □

Remark 3.2. The task of *counting* the number of simple paths between two nodes is known to be #P-complete [3], a class of problems believed to be even harder than NP-complete problems. Therefore, brute-force enumeration not only requires exponential time, but even understanding how many paths exist is itself intractable.

Additionally, we emphasize that our graph $G = (V, E)$ induces a *quasi-metric space* via the function:

$$d(u, v) = \min_{P \in \mathcal{S}(u,v)} d(P)$$

This quasi-metric structure arises due to the lack of symmetry in directed graphs (i.e., $d(u, v) \neq d(v, u)$), which is a natural fit for real-world road networks.

4. Time and Space Complexity Analysis

Let $n = |V|$ be the number of vertices and $m = |E|$ the number of edges in the graph $G = (V, E)$. We exhaustively explore all simple paths from source s to target t.

4.1. **Time Complexity.** The time complexity of our algorithm is dominated by the number of simple paths from s to t, each of which must be explored and evaluated. In the worst-case scenario—when the graph is maximally connected and acyclic—the number of simple paths grows factorially in the number of vertices.

Theorem 4.1. *Let $G = (V, E)$ be a directed acyclic graph (DAG) with n vertices. Then the number of simple paths from a source vertex $s \in V$ to a target vertex $t \in V$ is at most $(n-2)!$.*

It immediately follows, our brute-force algorithm runs in $O(n \cdot (n-2)!) = O(n \cdot n!)$ time.

Proof. We consider the worst case: a complete DAG where, for any pair of distinct vertices $u, v \in V$, if u precedes v in some topological ordering, then $(u, v) \in E$.

Let $V' = V \setminus \{s, t\}$ denote the set of intermediate vertices, with $|V'| = n - 2$. A simple path from s to t consists of an ordered list of intermediate vertices (with no repetition) placed between s and t. Each such ordering yields a unique simple path.

The number of such permutations is:

$$|\mathcal{S}(s,t)| = \sum_{k=0}^{n-2} \binom{n-2}{k} \cdot k! = \sum_{k=0}^{n-2} \frac{(n-2)!}{(n-2-k)!}$$

This sum is asymptotically dominated by the term where $k = n - 2$, which gives:

$$(n-2)! \in O(n!)$$

Therefore, in the worst case, the number of simple paths is bounded above by $O(n!)$.

Each path contains at most n vertices and hence takes $O(n)$ time to evaluate the pricing function $\pi(P)$. Thus, the overall time complexity is:

$$T(n) = O(n!) \cdot O(n) = O(n \cdot n!)$$

□

Remark 4.2. This bound reflects the fact that the algorithm visits every simple path from s to t and computes a cost for each. Because the graph is directed and simple paths cannot revisit vertices, the number of paths is finite and strictly less than all permutations of V. However, in dense acyclic graphs, this number still grows super-exponentially.

4.2. **Space Complexity.** The space complexity of the brute-force algorithm is determined by two primary factors:

(1) The maximum depth of the call stack or iterative path expansion mechanism.
(2) The number of paths stored in memory for evaluation or comparison.

Let $n = |V|$ be the number of vertices. Since we consider only simple paths, any valid path from s to t contains at most n vertices. Therefore, the depth of any recursive or stack-based traversal is at most n, resulting in an *active working memory* of $O(n)$.

If we discard each path immediately after computing its pricing value $\pi(P)$ and maintain only the currently optimal path, then we require:

$$S(n) = O(n)$$

This includes the current path under construction and the best path found so far; however, if the algorithm is configured to store all valid simple paths then in the worst case we may store up to $O(n!)$ such paths, each of which consumes $O(n)$ space. This implies:

$$S(n) = O(n \cdot n!) \quad \text{(if storing all paths)}$$

Remark 4.3. The space complexity of the brute-force algorithm is therefore conditional: linear in the size of the graph when discarding intermediate results, and factorial when full path enumeration is retained. In practical implementations, the linear-space variant is preferred unless full path history is explicitly required.

Policy	Time Complexity	Space Complexity	Description
Naive Enumeration	$O(n \cdot n!)$	$O(n \cdot n!)$	Stores all simple paths ory. Evaluates each prici tion $\pi(P)$ after full path ation.
In-Place Evaluation	$O(n \cdot n!)$	$O(n)$	Discards each path after ing $\pi(P)$. Stores only the best path and the active path.
Partial Retention	$O(n \cdot n!)$	$O(k \cdot n)$	Stores only a fixed numb $n!$ of best paths (e.g., top- ful for approximate mu planning.

TABLE 1. Time and space complexity under different brute-force storage policies.

Complexity Summary Table.

5. Comparative Analysis: Brute-Force, Backtracking, and Branch-and-Bound

While our initial implementation leverages a brute-force strategy, alternative approaches such as backtracking and branch-and-bound offer pathways to mitigate the exponential time cost inherent to exhaustive enumeration. In the following, we distinguish these three strategies in both theoretical and practical contexts.

5.1. **Brute-Force Search.** The brute-force method systematically enumerates all valid simple paths from source to target, evaluating each path against the pricing function $\pi(P)$. This guarantees optimality but at a computational cost of $O(n \cdot n!)$ in the worst case. The algorithm has the virtue of simplicity and determinism, but it scales poorly as the graph grows and quickly becomes infeasible for large-scale navigation systems.

5.2. **Backtracking.** Backtracking improves on brute-force by abandoning unpromising partial solutions early. During path exploration, if a partial path violates constraints (e.g., exceeds known cost bounds or revisits a node), it is pruned and not extended further. This leverages the recursive structure of the search tree and can dramatically reduce the effective search space. However, backtracking does not necessarily guarantee optimal pruning unless guided by good heuristics or bounding criteria.

5.3. **Branch-and-Bound.** Branch-and-bound extends backtracking with a formal bounding mechanism. As the algorithm explores paths, it maintains a record of the best solution found so far. Any path whose cost exceeds this bound is immediately discarded. Bounding functions can be problem-specific—e.g., estimating the minimal remaining cost based on heuristics or lower-bound cost functions. This method provides both correctness and efficiency, and when equipped with strong bounding strategies, can solve problems previously deemed intractable. However, it is more complex to implement and heavily depends on the quality of the bounding function.

Remark 5.1. In summary, brute-force provides a correctness benchmark and a pedagogical baseline; backtracking introduces structural pruning; and branch-and-bound injects optimization via intelligent cost constraints. All three are stages on a spectrum of search sophistication, with increasing reliance on domain-specific knowledge and problem structure.

6. Discussion: Limitations and Profitability Implications

While brute-force routing ensures correctness by identifying the globally optimal route with respect to the pricing function $\pi(P)$, its utility in a commercial or real-time setting is limited. Below we examine two interrelated concerns: computational intractability and profitability inefficiency.

6.1. **Limitations of Brute-Force in Real-World Systems.** The factorial time complexity $O(n \cdot n!)$ makes brute-force infeasible in practice for large graphs, especially under real-time constraints. Autonomous vehicle routing systems must operate under soft time windows, energy constraints, and user service guarantees. Waiting several seconds, or worse, minutes, to calculate a route renders the solution operationally irrelevant.

Moreover, brute-force assumes perfect information and static graph conditions. In reality, road conditions, traffic patterns, and service demands are dynamic. A brute-force solution calculated in advance may already be outdated by the time it's applied.

6.2. **Profitability as a System Metric.** In a smart routing context (e.g., ridesharing, delivery fleets), profitability is paramount. The pricing function $\pi(P)$ reflects multiple features beyond distance—such as traffic delays, demand priority, and driver availability. These factors can fluctuate in real-time and directly affect profit margins.

A brute-force algorithm, even if correct, does not account for opportunity cost. It may select a marginally better path for a single vehicle, while ignoring system-wide throughput or high-value requests elsewhere in the network. In this sense, brute-force is "selfish" and myopic—it optimizes the local minimum of $\pi(P)$ without regard to the global revenue picture.

6.3. **Path Bottlenecks and Structural Inflexibility.** A major drawback of naive enumeration is that it treats all subpaths as equally viable, even when the graph exhibits clear structural bottlenecks. These bottlenecks—nodes or edges through which many paths must pass—create concentrated points of congestion and economic inefficiency.

Let $b \in V$ be a bottleneck node such that a majority of simple paths from s to t include b. If b exhibits:

- high traversal cost (e.g., traffic, risk),
- or low post-traversal flexibility (i.e., limited outgoing edges),

then paths through b should be deprioritized. Yet brute-force expends equal effort on them.

Definition 6.1. A node $b \in V$ is a *routing bottleneck* for (s, t) if:

$$\frac{|\{P \in \mathcal{S}(s,t) \mid b \in P\}|}{|\mathcal{S}(s,t)|} > \delta$$

for some threshold $\delta \in (0, 1)$.

Remark 6.2. The identification of bottlenecks enables pre-processing optimizations, such as route rejection heuristics or weight penalties in $\pi(P)$. These are orthogonal to the brute-force logic but essential for real-world feasibility.

7. Limitations and Theoretical Refinements

While our brute-force method establishes a correct baseline, its computational inefficiency raises the need for refinement. Importantly, we distinguish between types of brute-force approaches:

- **Naive brute-force:** Exhaustively enumerates all possibilities without leveraging structure.
- **Structured brute-force:** Uses problem-specific properties to limit the search intelligently. The Euclidean algorithm, for instance, avoids full enumeration by reducing divisibility via greedy steps.
- **Pruned brute-force:** Rejects paths early based on dynamic constraints or heuristics.

Our current method is naive; it treats each possible path equally and evaluates them regardless of economic or structural infeasibility.

Definition 7.1. A node $b \in V$ is a *routing bottleneck* for (s, t) if:

$$\frac{|\{P \in \mathcal{S}(s,t) \mid b \in P\}|}{|\mathcal{S}(s,t)|} > \delta$$

for some threshold $\delta \in (0, 1)$. We refer to δ as the *bottleneck threshold*.

Such nodes, when associated with high traversal costs or limited outbound edges, reduce routing flexibility and economic feasibility. Yet, in our naive implementation, they are processed with the same priority as more efficient alternatives.

Remark 7.2. By precomputing structural bottlenecks or penalizing them in the pricing function $\pi(P)$, we may discard large subsets of infeasible routes early. These bottlenecks can also be represented as constraints in future optimization models.

8. Toward Milestone 3: Optimized Algorithm Design

Our critique of the brute-force strategy sets the stage for more sophisticated approaches:

- **Backtracking** introduces early pruning based on feasibility, abandoning paths as soon as they violate constraints.
- **Branch-and-Bound** maintains a global lower bound on cost, eliminating entire classes of paths whose pricing cannot improve on the current best.
- **Heuristic methods** such as A* or greedy best-first search may further limit the search space while maintaining near-optimal performance.

We aim to carry forward the insights from our brute-force model—not as a method to be discarded, but as a foundation for understanding the structure of the problem and motivating better algorithms. In Milestone 3, we will formalize these optimizations with both theoretical rigor and practical pseudocode.

References

1. T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 3rd ed., MIT Press, 2009.
2. R. E. Tarjan, *Data Structures and Network Algorithms*, Society for Industrial and Applied Mathematics (SIAM), 1983.
3. L. G. Valiant, "The Complexity of Enumeration and Reliability Problems," *SIAM Journal on Computing*, vol. 8, no. 3, pp. 410–421, 1979.
4. R. K. Ahuja, T. L. Magnanti, and J. B. Orlin, *Network Flows: Theory, Algorithms, and Applications*, Prentice Hall, 1993.
5. C. H. Papadimitriou, *Computational Complexity*, Addison-Wesley, 1994.
6. M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman, 1979.
7. J. L. Gross and J. Yellen, *Graph Theory and Its Applications*, 2nd ed., Chapman and Hall/CRC, 2006.
8. R. Bellman, "On a Routing Problem," *Quarterly of Applied Mathematics*, vol. 16, no. 1, pp. 87–90, 1958.
9. E. W. Dijkstra, "A Note on Two Problems in Connexion with Graphs," *Numerische Mathematik*, vol. 1, pp. 269–271, 1959.

San Francisco State University
Email address: dcha@sfsu.edu