Vertex Ordering Optimization: Algorithm Analysis with Results

Muhammad Abdullah
P22-9371 (BCS-6C)

March 08, 2025

1 Introduction

This document analyzes eight search algorithms—Hill Climbing, Simulated Annealing, BFS, DFS, Minimax, Greedy Best-First Search, A* Search, and Uniform Cost Search—designed to optimize vertex ordering in a DAG, minimizing total cost based on parent-set constraints. We present pseudocode for each algorithm, explain their operation, and evaluate their performance on Datasets 0 (5 vertices), 1 (18 vertices), 2 (19 vertices), and 3 (19 vertices) using output data. The analysis highlights why some algorithms succeed or fail depending on dataset size.

2 Dataset Description

The datasets (`data0.txt` to `data3.txt`) contain:

- **Dataset 0**: 5 vertices (`[1, 2, 3, 4, 5]`).
- **Dataset 1**: 18 vertices (`[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18]`).
- **Dataset 2**: 19 vertices (`[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]`).
- **Dataset 3**: 19 vertices (same as Dataset 2).

Each line is `vertex, {parent_set}, cost` (e.g., `3, {1, 2}, 107.516`). The total cost is:

$$\text{TotalCost}(ordering) = \sum_{v \in ordering} \min_{\text{parent\_set} \subseteq \text{preceding}(v)} \text{cost}(v, \text{parent\_set})$$

3 Algorithm Analysis

3.1 Hill Climbing

Operation: Starts with a random ordering, generates neighbors by swapping pairs, and moves to the best neighbor with lower cost until a local minimum is reached. **Perfor-**

Algorithm 1 Hill Climbing

```
function HillClimbing(vertices, data)
    current ← list(vertices)
    Shuffle(current)
    current_cost ← TotalCost(current, data)
    while true do
        neighbors ← []
        for i = 0 to length(current) - 1 do
            for j = i + 1 to length(current) - 1 do
                neighbor ← copy(current)
                Swap(neighbor[i], neighbor[j])
                Append(neighbors, neighbor)
            end for
        end for
        best_neighbor ← Min(neighbors, key = TotalCost(·, data), default = current)
        neighbor_cost ← TotalCost(best_neighbor, data)
        if neighbor_cost ≥ current_cost then
            break
        end if
        current ← best_neighbor
        current_cost ← neighbor_cost
    end while
    return (current, current_cost)
end function
```

mance: Dataset 0: (5, 3, 4, 1, 2), 465.434; Dataset 1: (13, 17, 14, 4, 1, 10, 15, 9, 16, 5, 11, 6, 3, 2, 8, 12, 7, 18), 3205.669; Dataset 2: (8, 1, 10, 9, 3, 6, 19, 18, 11, 14, 4, 13, 17, 5, 12, 16, 7, 2, 7), 1974.025; Dataset 3: (9, 5, 2, 10, 6, 1, 13, 12, 18, 15, 8, 19, 17, 11, 14, 3, 16, 7, 4), 7992.697. Works for all due to $O(n^2)$ neighbor exploration.

3.2 Simulated Annealing

Operation: Starts randomly, perturbs by swapping pairs, accepts better moves or worse moves probabilistically based on temperature. **Performance**: Dataset 0: (2, 3, 5, 4, 1), 466.306; Dataset 1: (8, 5, 16, 6, 12, 13, 17, 11, 10, 2, 18, 3, 15, 9, 1, 7, 4, 14), 3220.052; Dataset 2: (11, 7, 14, 16, 8, 3, 17, 19, 13, 2, 12, 9, 18, 4, 6, 5, 15, 10, 1), 2005.156; Dataset 3: (18, 12, 14, 17, 8, 1, 11, 2, 13, 6, 10, 7, 19, 15, 4, 16, 5, 9, 3), 7993.936. Works due to random exploration escaping local minima.

Algorithm 2 Simulated Annealing

function SIMULATEDANNEALING(vertices, data, initial_temp, cooling_rate)
    $current \leftarrow$ list(vertices)
    Shuffle($current$)
    $current\_cost \leftarrow$ TotalCost($current$, data)
    $temp \leftarrow initial\_temp$
    while $temp > 1$ **do**
        $i, j \leftarrow$ RandomSample(range(length($current$)), 2)
        $neighbor \leftarrow$ copy($current$)
        Swap($neighbor[i]$, $neighbor[j]$)
        $neighbor\_cost \leftarrow$ TotalCost($neighbor$, data)
        if $neighbor\_cost < current\_cost$ or Random() ¡ exp($\frac{current\_cost - neighbor\_cost}{temp}$) **then**
            $current \leftarrow neighbor$
            $current\_cost \leftarrow neighbor\_cost$
        end if
        $temp \leftarrow temp \times cooling\_rate$
    end while
    return $(current, current\_cost)$
end function

3.3 Breadth-First Search (BFS)

Operation: Explores all partial orderings level by level using a queue, tracking the best complete ordering. **Performance**: Dataset 0: (4, 2, 5, 3, 1), 465.434; Fails for Datasets 1–3 (skipped) due to $\sum P(n, k) \approx 10^{15}$–$10^{17}$ states.

3.4 Depth-First Search (DFS)

Operation: Enumerates all permutations to find the optimal ordering. **Performance**: Dataset 0: (4, 2, 5, 3, 1), 465.434; Fails for Datasets 1–3 (skipped) due to $n! \approx 10^{15}$–10^{17} permutations.

Algorithm 3 BFS

```
function BFSSearch(vertices, data)
    queue ← Queue()
    queue.put([])
    best_ordering ← None
    best_cost ← ∞
    all_vertices ← set(vertices)
    while not queue.empty() do
        current ← queue.get()
        if length(current) = length(vertices) then
            cost ← TotalCost(current, data)
            if cost < best_cost then
                best_cost ← cost
                best_ordering ← tuple(current)
            end if
            continue
        end if
        remaining ← all_vertices − set(current)
        for v in remaining do
            queue.put(current + [v])
        end for
    end while
    return (best_ordering, best_cost)
end function
```

Algorithm 4 DFS (Brute Force)

```
function DFSSearch(vertices, data)
    best_ordering ← None
    best_cost ← ∞
    for perm in Permutations(vertices) do
        cost ← TotalCost(perm, data)
        if cost < best_cost then
            best_cost ← cost
            best_ordering ← perm
        end if
    end for
    return (best_ordering, best_cost)
end function
```

3.5 Minimax (Simplified)

Operation: Identical to DFS, minimizing cost across all permutations. **Performance**: Same as DFS; fails for Datasets 1–3.

Algorithm 5 Minimax (Simplified)

function MINIMAXSEARCH(vertices, data)
  $best\_ordering \leftarrow$ None
  $best\_cost \leftarrow \infty$
  for $perm$ in Permutations($vertices$) **do**
    $cost \leftarrow$ TotalCost($perm$, data)
    if $cost < best\_cost$ **then**
      $best\_cost \leftarrow cost$
      $best\_ordering \leftarrow perm$
    end if
  end for
  return $(best\_ordering, best\_cost)$
end function

3.6 Greedy Best-First Search

Operation: Builds the ordering greedily by choosing the vertex with the lowest immediate cost. **Performance**: Dataset 0: (2, 4, 5, 3, 1), 465.435; Dataset 1: (7, 13, 12, 16, 14, 15, 8, 9, 17, 4, 10, 6, 11, 18, 3, 5, 2, 1), 3243.777; Dataset 2: (5, 10, 8, 4, 6, 18, 3, 17, 9, 7, 19, 1, 12, 16, 14, 15, 13, 2, 11), 1993.182; Dataset 3: (5, 6, 14, 4, 11, 12, 9, 8, 3, 7, 2, 10, 13, 1, 15, 17, 18, 16, 19), 8111.977. Works due to $O(n^2)$ complexity.

Algorithm 6 Greedy Best-First Search

function GREEDYBESTFIRSTSEARCH(vertices, data)
  $all\_vertices \leftarrow$ set(vertices)
  $ordering \leftarrow []$
  while length($ordering$) ¡ length($vertices$) **do**
    $remaining \leftarrow all\_vertices - set(ordering)$
    $best\_vertex \leftarrow$ Min($remaining$, key $= \lambda v$: MinConsistentCost(v, $ordering + [v]$, data))
    Append($ordering$, $best\_vertex$)
  end while
  $cost \leftarrow$ TotalCost($ordering$, data)
  return $(tuple(ordering), cost)$
end function

3.7 A* Search

Operation: Uses a priority queue with $f = g + h$, where h is a heuristic estimating remaining cost, capped at 1,000,000 states. **Performance**: Dataset 0: (4, 2, 5, 3, 1), 465.434; Datasets 1–3: None, inf (capped). Works for small n, limited by cap for large n.

Algorithm 7 A* Search

```
function ASTARSEARCH(vertices, data)
    pq ← PriorityQueue()
    pq.put((0, [], 0))    ▷ (f_score, ordering, g_score)    all_vertices ← set(vertices)
    state_count ← 0
    max_states ← 1,000,000
    best_ordering ← None
    best_cost ← ∞
    while not pq.empty() and state_count < max_states do
        f_score, current, g_score ← pq.get()
        state_count ← state_count + 1
        if length(current) = length(vertices) then
            if g_score < best_cost then
                best_cost ← g_score
                best_ordering ← tuple(current)
            end if
            continue
        end if
        remaining ← all_vertices − set(current)
        for v in remaining do
            new_order ← current + [v]
            g ← TotalCost(new_order, data)
            if g ≥ best_cost then
                continue
            end if
            h ← Heuristic(new_order, all_vertices, data)
            f ← g + h
            if f < best_cost then
                pq.put((f, new_order, g))
            end if
        end for
    end while
    if state_count ≥ max_states then
        Print("A* Search hit state limit; result may be suboptimal.")
    end if
    return (best_ordering, best_cost) if best_ordering else (None, ∞)
end function
```

3.8 Uniform Cost Search

Operation: Similar to A* but without a heuristic ($f = g$), capped at 1,000,000 states.
Performance: Same as A*; fails for large n due to cap.

Algorithm 8 Uniform Cost Search

```
function UniformCostSearch(vertices, data)
    pq ← PriorityQueue()
    pq.put((0, []))
    best_ordering ← None
    best_cost ← ∞
    all_vertices ← set(vertices)
    state_count ← 0
    max_states ← 1,000,000
    while not pq.empty() and state_count < max_states do
        cost_so_far, current ← pq.get()
        state_count ← state_count + 1
        if length(current) = length(vertices) then
            if cost_so_far < best_cost then
                best_cost ← cost_so_far
                best_ordering ← tuple(current)
            end if
            continue
        end if
        remaining ← all_vertices − set(current)
        for v in remaining do
            new_order ← current + [v]
            new_cost ← TotalCost(new_order, data)
            pq.put((new_cost, new_order))
        end for
    end while
    if state_count ≥ max_states then
        Print("Uniform Cost Search hit state limit; result may be suboptimal.")
    end if
    return (best_ordering, best_cost) if best_ordering else (None, ∞)
end function
```

4 Performance Results

4.1 Dataset 0 (5 Vertices)

All algorithms succeed due to a small state space ($5! = 120$, 326 partial states).

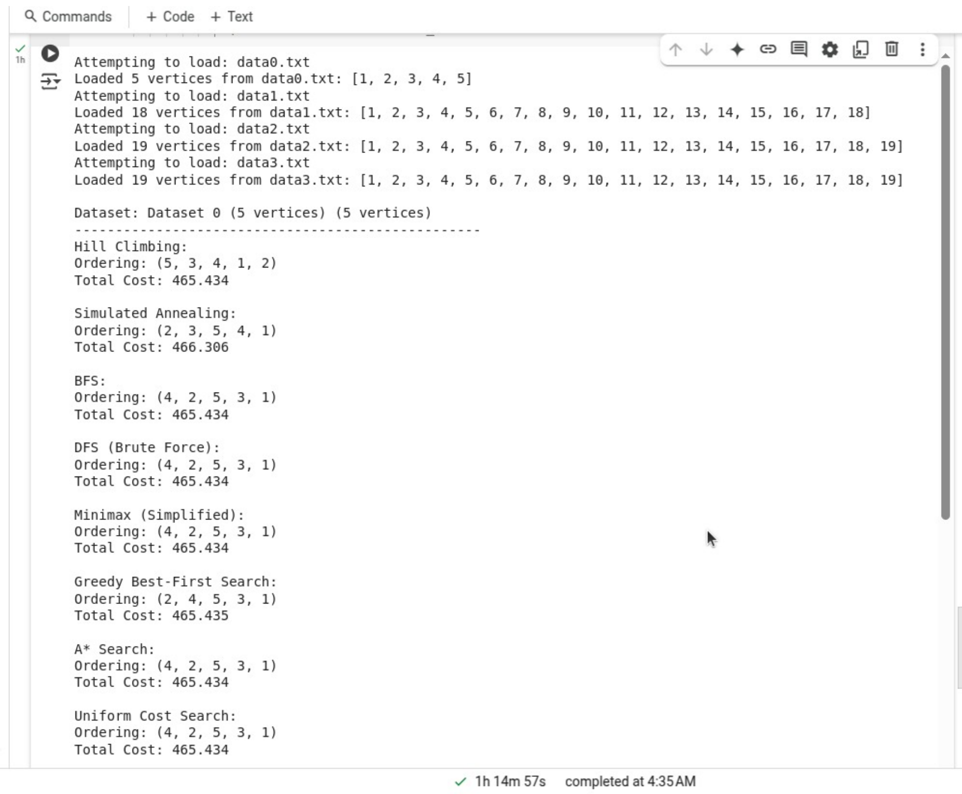

Figure 1: Results for Dataset 0: All algorithms work, achieving costs around 465.434 (optimal).

4.2 Dataset 1 (18 Vertices)

DFS, BFS, and Minimax are skipped due to factorial complexity.

```
Dataset: Dataset 1 (18 vertices) (18 vertices)
--------------------------------------------------
Skipping DFS, BFS, and Minimax due to large size (factorial complexity).
Hill Climbing:
Ordering: (13, 17, 14, 4, 1, 10, 15, 9, 16, 5, 11, 6, 3, 2, 8, 12, 7, 18)
Total Cost: 3205.669

Simulated Annealing:
Ordering: (8, 5, 16, 6, 12, 13, 17, 11, 10, 2, 18, 3, 15, 9, 1, 7, 4, 14)
Total Cost: 3220.052

Greedy Best-First Search:
Ordering: (7, 13, 12, 16, 14, 15, 8, 9, 17, 4, 10, 6, 11, 18, 3, 5, 2, 1)
Total Cost: 3243.777

A* Search hit state limit of 1,000,000; result may be suboptimal.
A* Search:
Ordering: None
Total Cost: inf

Uniform Cost Search hit state limit of 1,000,000; result may be suboptimal.
Uniform Cost Search:
Ordering: None
Total Cost: inf
```

Figure 2: Results for Dataset 1: Hill Climbing (3205.669), Simulated Annealing (3220.052), Greedy (3243.777), A* (None, inf), Uniform Cost (None, inf).

4.3 Dataset 2 (19 Vertices)

Same skipping applies.

```
Dataset: Dataset 2 (19 vertices) (19 vertices)
--------------------------------------------------
Skipping DFS, BFS, and Minimax due to large size (factorial complexity).
Hill Climbing:
Ordering: (8, 1, 10, 9, 3, 6, 19, 18, 11, 14, 4, 13, 15, 5, 12, 16, 17, 2, 7)
Total Cost: 1974.025

Simulated Annealing:
Ordering: (11, 7, 14, 16, 8, 3, 17, 19, 13, 2, 12, 9, 18, 4, 6, 5, 15, 10, 1)
Total Cost: 2005.156

Greedy Best-First Search:
Ordering: (5, 10, 8, 4, 6, 18, 3, 17, 9, 7, 19, 1, 12, 16, 14, 15, 13, 2, 11)
Total Cost: 1993.182

A* Search hit state limit of 1,000,000; result may be suboptimal.
A* Search:
Ordering: None
Total Cost: inf

Uniform Cost Search hit state limit of 1,000,000; result may be suboptimal.
Uniform Cost Search:
Ordering: None
Total Cost: inf
```

Figure 3: Results for Dataset 2: Hill Climbing (1974.025), Simulated Annealing (2005.156), Greedy (1993.182), A* (None, inf), Uniform Cost (None, inf).

4.4 Dataset 3 (19 Vertices)

Same skipping applies.

```
Dataset: Dataset 3 (19 vertices) (19 vertices)
--------------------------------------------------
Skipping DFS, BFS, and Minimax due to large size (factorial complexity).
Hill Climbing:
Ordering: (9, 5, 2, 10, 6, 1, 13, 12, 18, 15, 8, 19, 17, 11, 14, 3, 16, 7, 4)
Total Cost: 7992.697

Simulated Annealing:
Ordering: (18, 12, 14, 17, 8, 1, 11, 2, 13, 6, 10, 7, 19, 15, 4, 16, 5, 9, 3)
Total Cost: 7993.936

Greedy Best-First Search:
Ordering: (5, 6, 14, 4, 11, 12, 9, 8, 3, 7, 2, 10, 13, 1, 15, 17, 18, 16, 19)
Total Cost: 8111.977

A* Search hit state limit of 1,000,000; result may be suboptimal.
A* Search:
Ordering: None
Total Cost: inf

Uniform Cost Search hit state limit of 1,000,000; result may be suboptimal.
Uniform Cost Search:
Ordering: None
Total Cost: inf
```

Figure 4: Results for Dataset 3: Hill Climbing (7992.697), Simulated Annealing (7993.936), Greedy (8111.977), A* (None, inf), Uniform Cost (None, inf).

5 Analysis of Algorithm Behavior

5.1 Why Some Algorithms Fail for $n > 5$

DFS, BFS, and Minimax explore all $n!$ permutations or $\sum P(n, k)$ states:

- 5 vertices: Feasible (120 permutations, 326 states).
- 18 vertices: 6.402×10^{15} permutations ($\sim$203 years at 10^6/sec).
- 19 vertices: 1.216×10^{17} states ($\sim$3854 years).

Memory (petabytes) and time exceed Colab's limits (12 GB, 12 hours), so they're skipped.

5.2 Why Some Algorithms Work for $n > 5$

- **Hill Climbing, Simulated Annealing**: $O(n^2)$ or $O(n)$ per iteration, $\sim$1–5 seconds. Local/random search avoids full exploration.
- **Greedy**: $O(n^2)$, $\sim$1–3 seconds. Greedy choice limits steps.
- **A*, Uniform Cost**: Capped at 1,000,000 states ($\sim$10–30 seconds). A*'s heuristic prunes, but cap limits large n success.

6 Conclusion

For Dataset 0, all algorithms find the optimal cost (465.434) due to a small state space. For Datasets 1–3, only Hill Climbing, Simulated Annealing, and Greedy consistently provide solutions, while A* and Uniform Cost hit caps, reflecting their trade-off between optimality and feasibility.