Optimal Graph Structures: A Search and Problem Solving Approach

Tazmeen Afroz
22P-9252

March 10, 2024

1 Problem Description

The problem at hand involves finding optimal graph structures, a significant optimization problem with applications in various areas, including machine learning. Specifically, the problem can be simplified as the vertex ordering problem. Given a set of vertices, the goal is to determine a vertex ordering with the minimum cost. Each vertex has a set of possible parent sets, each with an associated cost. A parent set of a vertex is consistent with an ordering if all of the parents come before the vertex in the ordering. The task is to find an ordering of vertices that minimizes the total cost of the network.

2 Approach

To solve this problem, we can use search algorithms to explore the space of all possible orderings and find the one with the minimum cost. We will implement and experiment with different search algorithms, including breadth-first search (BFS), depth-first search (DFS), hill climbing, simulated annealing, and A* search.

3 Search Algorithms

Search algorithms are a fundamental tool in artificial intelligence for navigating through the state space of a problem. In our case, the state space is the set of all possible orderings of the vertices, and the goal is to find an ordering with the minimum cost.

3.1 Breadth-First Search (BFS) and Depth-First Search (DFS)

BFS and DFS are basic search algorithms that explore the state space by traversing the graph of all possible orderings. BFS explores all the neighboring states

at the current level before moving on to states at the next level, while DFS explores as far as possible along each branch before backtracking.

3.1.1 Breadth-First Search (BFS) Implementation

The function ‘bfs‘ takes as input a dictionary ‘data_dict‘ where the keys are the nodes and the values are the possible parent sets and their associated costs. The function returns the ordering of nodes that gives the minimum cost and the minimum cost itself.

Here is the Python code for the ‘bfs‘ function:

```
def bfs(data_dict):
    nodes = list(data_dict.keys())
    permutations_generator = permutations(nodes)
    visited = set()
    min_cost = float('inf')
    min_cost_ordering = None

    for ordering in permutations_generator:
        if ordering in visited:
            continue
        visited.add(ordering)
        cost = calculate_cost(ordering, data_dict)
        if cost < min_cost:
            min_cost = cost
            min_cost_ordering = ordering

    return min_cost_ordering, min_cost
```

The ‘bfs‘ function generates all possible orderings of the nodes and calculates the cost of each ordering. It keeps track of the ordering with the lowest cost. The function uses a set to keep track of the orderings it has already visited to avoid visiting the same ordering multiple times.

3.1.2 DFS Implementation

The function ‘dfs‘ takes as input a dictionary ‘data_dict‘ where the keys are the nodes and the values are the possible parent sets and their associated costs. The function returns the ordering of nodes that gives the minimum cost and the minimum cost itself.

Here is the Python code for the ‘dfs‘ function:

```
def dfs(data_dict):
    nodes = list(data_dict.keys())
    permutations_generator = permutations(nodes)
    visited = set()
    min_cost = float('inf')
```

```
    min_cost_ordering = None

    for ordering in permutations_generator:
        if ordering in visited:
            continue
        visited.add(ordering)
        cost = calculate_cost(ordering, data_dict)
        if cost < min_cost:
            min_cost = cost
            min_cost_ordering = ordering

    return min_cost_ordering, min_cost
```

The ‘dfs‘ function generates all possible orderings of the nodes and calculates the cost of each ordering. It keeps track of the ordering with the lowest cost. The function uses a set to keep track of the orderings it has already visited to avoid redundant calculations. The function returns the ordering with the minimum cost and the minimum cost itself.

3.2 Hill Climbing

Hill climbing is a heuristic search algorithm that starts with an arbitrary solution and iteratively makes local modifications to improve the solution. In our case, a local modification could be swapping two vertices in the ordering.

3.2.1 Hill Climbing Implementation

The function ‘hill_climbing‘ takes as input an initial ordering and a dictionary ‘data_dict‘ where the keys are the nodes and the values are the possible parent sets and their associated costs. The function returns the ordering of nodes that gives the minimum cost and the minimum cost itself.

Here is the Python code for the ‘hill_climbing‘ function:

```
def hill_climbing(initial_ordering, data_dict):
    current_ordering = initial_ordering
    current_cost = calculate_cost(current_ordering, data_dict)

    while True:
        neighbors = []
        for i in range(len(current_ordering)):
            for j in range(i+1, len(current_ordering)):
                new_ordering = current_ordering[:i] +
                current_ordering[i:j+1][::-1] + current_ordering[j+1:]
                new_cost = calculate_cost(new_ordering, data_dict)
                neighbors.append((new_ordering, new_cost))

        min_cost = float('inf')
```

```
        next_ordering = None
        for ordering, cost in neighbors:
            if cost < min_cost:
                min_cost = cost
                next_ordering = ordering

        if min_cost < current_cost:
            current_ordering, current_cost = next_ordering, min_cost
        else:
            return current_ordering, current_cost
```

The ‘hill_climbing‘ function starts with an initial ordering and iteratively generates all neighboring orderings by reversing all possible sublists of the current ordering. It then moves to the neighboring ordering with the lowest cost. If no neighbor has a lower cost than the current ordering, it returns the current ordering and its cost.

3.3 Simulated Annealing

Simulated annealing is a probabilistic technique for approximating the global optimum of a given function. It uses a random search instead of a greedy search to avoid getting stuck in local optima.

3.3.1 Simulated Annealing Implementation

The function ‘simulated_annealing‘ takes as input an initial ordering and a dictionary ‘data_dict‘ where the keys are the nodes and the values are the possible parent sets and their associated costs. The function returns the ordering of nodes that gives the minimum cost and the minimum cost itself.

Here is the Python code for the ‘simulated_annealing‘ function:

```
def simulated_annealing(initial_ordering, data_dict, T=1.0, T_min=0.00001, alpha=0.9):
    current_ordering = initial_ordering
    current_cost = calculate_cost(current_ordering, data_dict)

    while T > T_min:
        indices = random.sample(range(len(current_ordering)), 2)
        indices.sort()
        i, j = indices

        new_ordering = list(current_ordering)
        sublist = new_ordering[i:j+1]
        sublist.reverse()
        new_ordering[i:j+1] = sublist

        new_cost = calculate_cost(new_ordering, data_dict)
```

```
        delta = new_cost - current_cost

        if delta < 0 or random.uniform(0, 1) < math.exp(-delta / T):
            current_ordering, current_cost = new_ordering, new_cost

        T = T * alpha

    return current_ordering, current_cost
```

The 'simulated_annealing' function starts with an initial ordering and iteratively generates a new ordering by reversing a random sublist of the current ordering. It then decides whether to move to the new ordering based on the difference in cost and the current temperature. If the new ordering has a lower cost, it moves to it. If the new ordering has a higher cost, it still might move to it with a certain probability that decreases as the difference in cost increases and as the temperature decreases. The function returns the final ordering and its cost.

3.4 A* Search

3.4.1 A* Search Implementation

The function 'a_star' takes as input an initial ordering and a dictionary 'data_dict' where the keys are the nodes and the values are the possible parent sets and their associated costs. The function returns the ordering of nodes that gives the minimum cost and the minimum cost itself.

Here is the Python code for the 'a_star' function:

```
import queue

def a_star(initial_ordering, data_dict):
    pq = queue.PriorityQueue()
    pq.put((calculate_cost(initial_ordering, data_dict), initial_ordering))
    visited = set()

    while not pq.empty():
        cost, ordering = pq.get()
        ordering_tuple = tuple(ordering)

        if ordering_tuple in visited:
            continue

        visited.add(ordering_tuple)

        if len(ordering) == len(data_dict):
            return ordering, cost
```

```
        for i in range(len(ordering)):
            for j in range(i+1, len(ordering)):
                new_ordering = ordering[:i] + ordering[i:j+1][::-1] + ordering[j+1:]
                new_cost = calculate_cost(new_ordering, data_dict)
                pq.put((new_cost, new_ordering))
```

The 'a_star' function starts with an initial ordering and uses a priority queue to keep track of the orderings to be explored, with the ordering with the lowest cost at the front. It generates all neighboring orderings by reversing all possible sublists of the current ordering and adds them to the priority queue. The function uses a set to keep track of the orderings it has already visited to avoid redundant calculations. The function returns the final ordering and its cost.

4 Results and Discussion

The implemented algorithms were tested on four datasets, referred to as 'data0', 'data1', 'data2', and 'data3'. The results of the algorithms on these datasets are as follows:

- For 'data0', all algorithms except A* returned the same optimal ordering with a cost of 465.434. A* returned a different ordering with a higher cost of 490.194.
- For 'data1', Hill Climbing and Simulated Annealing returned similar orderings with costs around 3197, while A* returned the ordering [1, 2, ..., 18] with a slightly higher cost of 3211.685.
- For 'data2', Hill Climbing and Simulated Annealing returned similar orderings with costs around 1976, while A* returned the ordering [1, 2, ..., 19] with a higher cost of 1995.512.
- For 'data3', Hill Climbing returned an ordering with the lowest cost of 7974.521, Simulated Annealing returned an ordering with a slightly higher cost of 8058.006, and A* returned the ordering [1, 2, ..., 19] with the highest cost of 8240.569.

BFS and DFS were not run on 'data1', 'data2', and 'data3' due to their high memory requirements. These algorithms generate all possible orderings of the nodes, which can be computationally expensive for large datasets.

The results show that Hill Climbing and Simulated Annealing often find better solutions than A*, despite A* being theoretically optimal. This discrepancy can be attributed to the heuristic function used by A*, which in this case might not provide an accurate estimate of the cost to reach the goal. On the other hand, Hill Climbing and Simulated Annealing make use of local search, which can be more effective for this particular problem where a small change in the ordering can lead to a large decrease in cost.

In conclusion, while A* guarantees to find the optimal solution, it may not always be the best choice in practice due to its reliance on a good heuristic function and its high computational cost. Hill Climbing and Simulated Annealing, despite their simplicity, can often find good solutions more efficiently.