Report, Project 2

Name: Yuuzen Shen UIN: 434000618

Algorithm

Our algorithm has three parts:

1. The generator that is piecewise for generating the matrix from brute force (for m<=3) and simulated annealing (for m>3)
2. The checker that checks the validity of the generated matrix based on given LP solution
3. A annealing generator that uses simulated annealing to generate the matrix from a seed

The part 1 provided preliminary seeds for the annealing generator.

Bound, and Complexity (if any)

Brute Force: Exact (within element range), Exponential time.

Annealing (Random Start): Heuristic (no guarantee), Time depends on annealing schedule and parallel runs, but does converge depends on ratio of temperature drops.

Random Search: Heuristic (no guarantee, weakest), Time depends on number of samples. Hard set by parameter.

Annealing (Seed Start): Heuristic (no guarantee, depends on seed), Time depends on annealing schedule and parallel neighbor evaluation. Returns second-best result as well for measure or reseeding.

Main Idea

For a small m, we can brute force the solution by checking all possible values of the matrix.

For a larger m, we can use simulated annealing to generate a good seed, and then use the annealing generator to generate a good matrix.

Why not one run? Because this problem shows local maximum and global maximum, even though the simulated annealing is a heuristic made to combat that, it would be much faster to parallelize the runs via generating high quality seeds and annealing from them, rather than parallelizing the search, because the entropy introduced by the random seed is much more effective than the entropy introduced by the random search (if any).

Very brief pseudocode

```
Input: k, n, m, element_range
Output: Best P matrix found

if m <= 3: // Threshold for 'small' m
  // Use Brute Force Generator
  P_best = BruteForceSearch(k, n, m, element_range)
else:
  // Manual Seed Generation Phase (Conceptual)
  // 1. Run Annealing Generator multiple times
  seeds_annealing = []
  for run in 1 to NUM_ANNEALING_RUNS:
    P_candidate = AnnealingGenerator(k, n, m, element_range, annealing_params)
    seeds_annealing.append(P_candidate)

  // 2. Run Random Generator multiple times
  seeds_random = []
  for run in 1 to NUM_RANDOM_RUNS:
```

```
    P_candidate = RandomGenerator(k, n, m, element_range, num_samples)
    seeds_random.append(P_candidate)

  // 3. Manually inspect/evaluate seeds_annealing and seeds_random
  //    Select the most promising seed based on m-height or other criteria.
  P_seed = HandpickBestSeed(seeds_annealing + seeds_random)

  // Seeded Annealing Phase
  // Write P_seed to a file (e.g., "seed.txt") along with n, k, m
  SaveSeedToFile("seed.txt", n, k, m, P_seed)

  // Run Seeded Annealing Generator using the chosen seed file
  P_best = SeededAnnealingGenerator(seed_file="seed.txt", annealing_params)

return P_best
```

Brute Force Generator

```
function BruteForceSearch(k, n, m, element_min, element_max):
    I_k = identity_matrix(k)
    p_cols = n - k
    total_candidates = (element_max - element_min + 1)^(k * p_cols)
    best_P = null
    min_cost = infinity

    # Distribute work among multiple threads
    for each chunk of indices in total_candidates:
        # Each thread does:
        for index in assigned_chunk:
            # Generate P matrix from index (maps index to unique matrix
configuration)
            P = get_p_matrix_from_index(index, k, p_cols, element_min, element_max)

            # Skip if any column in P is all zeros
            if not is_valid_P(P):
                continue

            # Construct G = [I|P]
            G = horizontally_stack(I_k, P)

            # Calculate m-height cost
            cost = compute_m_height(G, m)

            # Update best if better
            if cost < min_cost:
                min_cost = cost
                best_P = P

    return best_P, min_cost
```

The brute force generator exhaustively searches through all possible P matrices within the given element range. It's parallelized using multiple threads, with each thread responsible for a chunk of the search space. For each candidate matrix, it calculates the m-height cost and keeps track of the best solution found. Thus it is only used for small cases because it runs exponentially in time.

Random Generator

```
function RandomSearch(k, n, m, element_min, element_max, num_samples):
    I_k = identity_matrix(k)
    p_cols = n - k
    best_P = null
    min_cost = infinity

    # Distribute samples among multiple processes
    for each worker process:
        # Each worker evaluates multiple samples
        for i in range(assigned_samples):
            # Generate random P matrix
            P = random_integer_matrix(k, p_cols, element_min, element_max)

            # Skip if any column in P is all zeros
            if not is_valid_P(P):
                continue

            # Construct G = [I|P]
            G = horizontally_stack(I_k, P)

            # Calculate m-height cost
            cost = compute_m_height(G, m)

            # Update best if better (with synchronization)
            if cost < global_min_cost:
                update_global_best(P, cost)

    return global_best_P, global_min_cost
```

The random generator is a simple random search, it is used for generating seeds for the annealing generator.

Annealing Generator

```
function SimulatedAnnealing(k, n, m, element_min, element_max, T_max, T_min, alpha,
iter_per_temp):
    # Run multiple independent annealing instances in parallel
    results = []

    for each worker:
        # Generate random initial P matrix
        P = generate_valid_random_P(k, n-k, element_min, element_max)
        I_k = identity_matrix(k)
        G = horizontally_stack(I_k, P)
        current_cost = compute_m_height(G, m)
        best_P = P
        best_cost = current_cost
        T = T_max

        # Main annealing loop
        while T > T_min:
            for iteration in range(iter_per_temp):
                # Generate neighbor by changing one element
                neighbor_P = get_neighbor(P, element_range)
                neighbor_G = horizontally_stack(I_k, neighbor_P)
                neighbor_cost = compute_m_height(neighbor_G, m)
```

```
                # Decide whether to accept
                delta = neighbor_cost - current_cost

                if delta < 0 or random() < exp(-delta/T):
                    P = neighbor_P
                    current_cost = neighbor_cost

                    if current_cost < best_cost:
                        best_cost = current_cost
                        best_P = P

            # Cool down temperature
            T *= alpha

        results.append((best_P, best_cost))

    # Find overall best from all workers
    return best_result_from(results)
```

The annealing generator uses simulated annealing from a random start. Detail of annealing would be explained in the seed annealing generator.

Seed Annealing Generator

```
function SeedAnnealingGenerator(seed_file, initial_temp, cooling_rate,
steps_per_temp):
    # Load seed matrix and parameters
    n, k, m, initial_P = load_p_matrix(seed_file)
    I_k = identity_matrix(k)
    element_min = min(initial_P)
    element_max = max(initial_P)

    # Track best and second best
    current_P = initial_P
    current_G = horizontally_stack(I_k, current_P)
    current_cost = compute_m_height(current_G, m)

    best_P = current_P
    best_cost = current_cost
    second_best_P = current_P
    second_best_cost = current_cost

    temperature = initial_temp

    # Main annealing loop
    while temperature > 0.01:
        # Generate and evaluate neighbors in parallel
        neighbors = []
        for _ in range(steps_per_temp):
            neighbor_P = generate_neighbor(current_P, element_min, element_max)
            neighbors.append(neighbor_P)

        # Evaluate in parallel
        costs = parallel_evaluate(neighbors, k, m, I_k)

        for i, neighbor_cost in enumerate(costs):
            neighbor_P = neighbors[i]
```

```
            # Accept better solution always
            if neighbor_cost < current_cost:
                current_P = neighbor_P
                current_cost = neighbor_cost

                # Update best if better
                if neighbor_cost < best_cost:
                    second_best_P = best_P
                    second_best_cost = best_cost
                    best_P = neighbor_P
                    best_cost = neighbor_cost
                elif neighbor_cost < second_best_cost:
                    second_best_P = neighbor_P
                    second_best_cost = neighbor_cost
            # Accept worse solution probabilistically
            else:
                delta = neighbor_cost - current_cost
                if random() < exp(-delta/temperature):
                    current_P = neighbor_P
                    current_cost = neighbor_cost

        # Cool down temperature
        temperature *= cooling_rate

    # Return both best and second best for robustness
    return best_P, best_cost, second_best_P, second_best_cost
```

The seed annealing generator refines an existing solution (seed) through simulated annealing. Simulated annealing is the metaheuristic that is used to find a global maximum solution by having a chance (temperature) to accept worse solutions because it can be a local maximum instead of a global maximum.

There were attempts on using a surrogate model to replace the LP solver, but it was mostly bogus and not reliable.

And that's the report for Project 1.

Project 2 updates

For project 2, the only change is that we curated a surrogate model written in C++. We added that model code to the latest of code. The logic proceeds the simulated annealing search with the surrogate model, and then verify the final result with the LP solver in python (because the surrogate model has around 10% error, but way faster).

That's the update and thus report for Project 2.