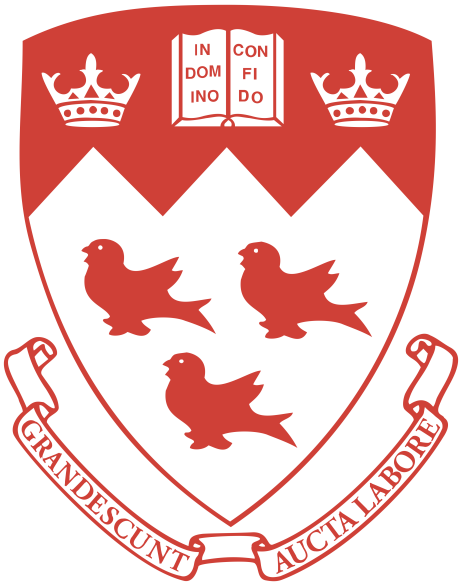

ECSE 420 Group 13
N-Queens Problem Solver

Author:
Amani Jammoul-260381641
Ankur Singh-260891708
Annabelle Dion-260928845
Bálint Nyakas-261097259

Contents

1 Introduction

1.1 Project Goal

The goal of this project was to implement solvers for the N-Queens problem and compare their efficiencies. This is a problem we had been introduced to in other courses in the past and we wanted to solve it with parallel computing. We worked on two separate solvers, one that runs computations sequentially and another that includes parallelism. This report presents the problem, the algorithms we followed, our implementation approaches, and the results.

1.2 N-Queens Problem

The N-queens problem is the problem of placing N queens on an $N * N$ chessboard such that they cannot attack each other. In other words, no pair of queens can be on the same row, column or diagonal. A sample case can be seen on Fig.1 for N=4.

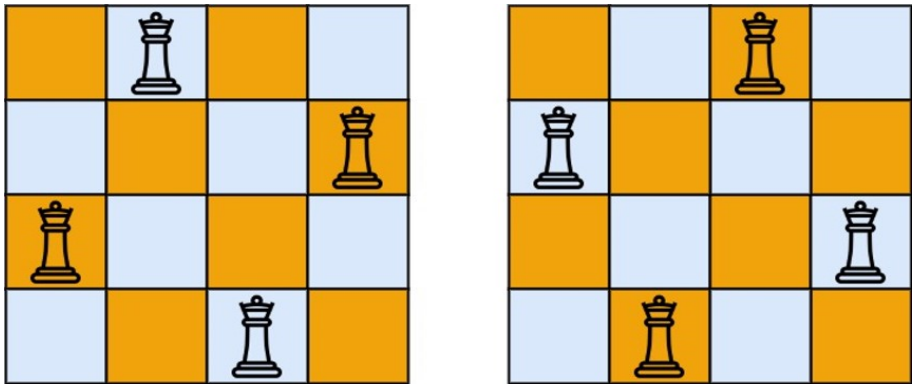

Figure 1: 4-Queens problem

1.3 Stochastic Local Search

In order to solve this problem, we initially formulated it as a Constraint Satisfaction Problem (CSP). The constraints are defined from the rules described in section 1.2. The search space for this NP-hard problem is large. A basic search for optimization algorithm that uses constructive methods would take too long to build up a solution. This is why we used stochastic local searches, an iterative improvement method.

A local search problem consists of three main components: a CSP, a neighbour relation, and a heuristic function. The neighbouring relation for our problem defines neighboring state as any board configurations in which the queens are in the same position, with the exception of one being moved to a different row within a column. A heuristic function is used to evaluate how well a state satisfies the goal (which is to find a state that satisfies all the constraints). In our case, the heuristic value of any state is defined as the number of constraint violations. Consequently, the aim of finding an assignment of queens' positions on the board that satisfies the constraints translates to finding a board state with minimal heuristic value, where the optimal solution has value of 0.

The idea is to start with a state that isn't optimal. It then moves from the current state to a neighboring state according to the heuristic values. It continues this search until stopping criteria are met or a satisfying assignment is found. With greedy descent (i.e. when the best neighbor is chosen at each step), this algorithm often gets stuck in local minima. In order to work around this, randomness should be introduced. Stochastic local searches start with a random assignment, then include a mix of greedy descent and random walk to balance exploration and exploitation.

1.4 Sequential Search with Simulated Annealing

Simulated annealing is an algorithm that incorporates random steps to help in escaping local optima. Instead of computing all neighbors and choosing the best one at each step, it chooses a random neighbor and uses heuristic and temperature values to decide the probability of accepting it even if it is a "bad" move. Temperature values start high, which means there is higher tendency to take random steps at the start (because exploration is valuable earlier on when the assignment you have is not strong). Over time, the temperature decreases and exploitation is favored more.

The probability for choosing a new worse value revolves around the Boltzmann distribution. If the temperature decreases slowly enough, simulated annealing is guaranteed to reach the optimal solution (i.e., find the global maximum). However this may take an infinite number of steps. This is why choosing good temperature and cooling values is crucial. Cooling too fast means it will converge to a sub-optimal solution quicker, and cooling too slow means it won't converge at all.

Algorithm 1: Simulated Annealing Algorithm

$X_0 \leftarrow$ random initial board configuration ;
let $E(X_0)$ be the value of X_0;
$X \leftarrow X_0$;
$E(X) \leftarrow E(X_0)$;
$E(X*) \leftarrow E(X)$;
define temperature and cooling values;
while *temp > threshold and* $E(X*) \neq 0$ **do**
    $X_i \leftarrow$ random neighbor of X;
    let $E(X_i)$ be the value of X_i;
    if $E(X_i) < E(X*)$ **then**
        $X* \leftarrow X_i$;
        $E(X*) \leftarrow E(X_i)$;
    else if $E(X_i) < E(X)$ **then**
        $X \leftarrow X_0$;
        $E(X) \leftarrow E(X_i)$;
    else
        $p = e^{(E(X)-E(X_i))/temp}$;
        with probability p: $X \leftarrow X_0$ and $E(X) \leftarrow E(X_i)$;
    decrease temp;

1.5 Parallel Search with Stochastic Beam Search

Stochastic beam search is an algorithm that is used to run searches in parallel while sharing information across searches. It runs many instances at the same time, keeping k assignments at once. At each step, it generates the neighborhoods of all k states, unions them, and keeps k next neighbors. A generic beam search always chooses the k best neighbors. However, this is too greedy. Instead, stochastic beam search chooses k nodes probabilistically. Similar to simulated annealing, the probability that a neighbor is chosen is proportional to the value of the evaluation function and temperature.

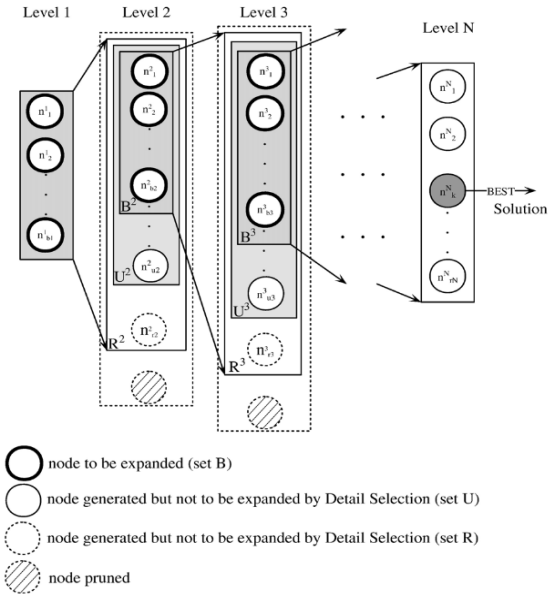

Figure 2: Stochastic Beam Search Algorithm[2]

Algorithm 2: Stochastic Beam Search Algorithm

states ← list of k random configurations;
best state ← state with min value;
define temperature and cooling values;
while *temp* > *threshold and* $E(X*) \neq 0$ **do**
    generate neighbors for all k state;
    ; `/* in parallel */`
    neighborhood ← union of all neighbors;
    $p = e^{heurisitc\ diff/temp}$;
    states ←
     randomly choose k states from *neighborhood* with prob proportional to p;

    update best state;
    decrease temp;

2 Implementation

We implemented two Python programs that use different algorithms to solve the N-Queens problem.

2.1 N-Queens Sequential Solver using Simulated Annealing

This first program defines two important data structures, a BoardState and NQueens_Problem class. These are used to formulate the program in our code. A BoardState includes information about any board configurations, including the positions of the queen on any N-sized board. The NQueens_Problem class is used to define any N-sized problem. It holds an initial state and a current state. It also has functions to generate random initial states and a random neighboring state (where a neighbor state is defined as a board configuration where a single queen changes placement within its column).

We also define a NQueens_ProblemSolver class which is used to find a best solution using simulated annealing. It has a function that computes the heuristic value of board states, which is used in the simulated annealing function. This function, given any NQueens_Problem object, follows the algorithm to return a solution is finite time.

2.2 N-Queens Parallel Solver using Stochastic Beam Search

Our second program uses similar data structures and functions as the first, with additional complexity and modifications to correspond to the stochastic beam search algorithm. One major difference is the use of the Ray Core library to run Python code in parallel. It provides core primitives for building and scaling distributed applications [1].

Any N-Queens problem is solved using the NQueens_ParallelProblemSolver class. This class is initialized with two important arguments: n (the size of the board) and k (the number of parallel processes). The main flow is found in the solveParallel() method, which itself calls ray remote tasks to perform certain computations. A ray remote task is a Python function that can be run in parallel. At the start, the solver generates k random initial states. Then it enters a while loop that continues until the optimal solution is found or the temperature cools below a certain threshold. Throughout the process of finding a solution, we always keep k states and perform computations on them in parallel. The idea is that a solution may be found faster this way. At each iteration of the loop, k processes are called to find all neighbors of each k state. Computations for computing values necessary for generating the probabilities are done in parallel as well.

3 Results

At the start of this project, we expected the parallel program to return a solution quicker than the sequential one. However, we found that the runtime for the parallel program was in fact slower. Figure 3 shows the runtime of solving an N=8 queens problem with the serial solver versus the parallel solver with k=8. The serial program runs faster or equally as fast as the parallel one. This was also true for larger N values. As N grew, the serial program performed well but the parallel one ran for a long time.

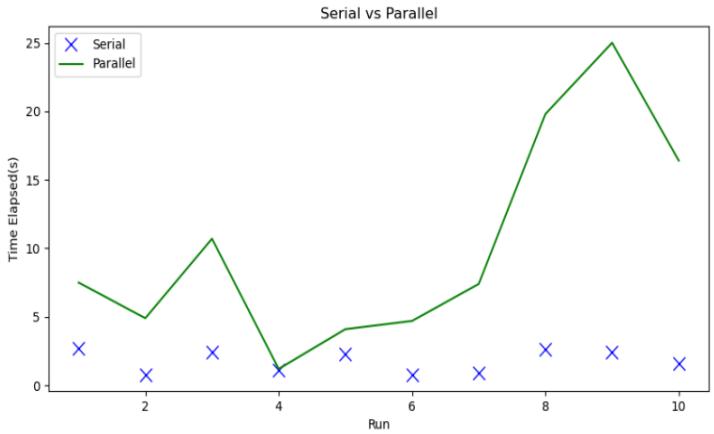

Figure 3: Runtime

One reason we suspect that the parallel program did not run as fast as expected is because of large overhead. At each iteration, k ray tasks are called and each task invocation has non-trivial overhead (including time for scheduling, inter-process communication, updating system state, etc.). This overhead seems to dominate the actual time for executing the task.

In addition, each iteration calls ray.get(neighbors) to get the resulting neighbors and ray.put(states) to store the list of states in the object store. This is necessary because any object that a ray task creates or computes over must be in the remote object store. However, every time these are called, the data must be copied back and forth from the object store. This also takes non-trivial time and adds to the run time of the program.

In conclusion, for this particular problem and when using the Ray library to parallelize Python code, it was better to run a sequential solver. The parallel solver introduced unexpectedly large overhead and had worse runtime.

4 References

References

[1] *Ray Core*. URL: https://www.ray.io/ray-core.

[2] Fan Wang and Andrew Lim. "A stochastic beam search for the berth allocation problem". In: *Decision Support Systems* 42.4 (2007). Decision Support Systems in Emerging Economies, pp. 2186–2196. ISSN: 0167-9236. DOI: https://doi.org/10.1016/j.dss.2006.06.008. URL: https://www.sciencedirect.com/science/article/pii/S016792360600090X.