Collaborative Randomized Incremental Constructions on the Example of N-Queens

Presentation of Bachelor Results

by Tristan Menke, Supervised by Florian Fey

Westfälische Wilhelms-Universität Münster

May 10, 2020

Introduction

- Many computational tasks can be stated as Constraint Satisfaction Problems(CSPs)
  - Many AI tasks
  - Scheduling of processes and traffic
  - Many games like Sudoku
- I present an easy to implement, fast way to solve many problems, focused mainly on CSPs
- Finding fast, general and easy to implement solutions for these problems is useful in many areas
- General approaches may not be optimal for any one problem
- ...but if they are fast, that's good enough in practice

Constraint Satisfaction Problems(CSPs)

A CSP is:

- A set of variables $X_1, ..., X_n$ with domains $D_1, ..., D_n$
- A set of constraints $R_{i,j} \subseteq D_i \times D_j$ (or with more variables), that restrict which values the variables may take.
  - Example: $R(X_1, X_2, X_3)$ specifies that $X_1 + X_2 = X_3$ must be true
- A solution to a CSP is an assignment of all variables such that all constraints for each variable are satisfied
- Generally CSPs are in NP-Complete, but specific problems may be easier to solve
- I use the N-Queens problem as a specific CSP to implement all approaches presented

The N-Queens Problem as a CSP

The N-Queens problem consists of:

- An NxN chess board with the goal to place N queens
- N variables that specify the queen positions, with values in $\{1, ..., N\}$
- 3 constraints on each variable:
  - No queen may be on the same **row** as another queen
  - No queen may be on the same **column** as another queen
  - No queen may be on the same **diagonal** as another queen
- We need to eliminate as many constraints as possible to quickly check the validity of an assignment

Eliminating constraints

- Eliminate the row constraint by interpreting the indices of variables as row positions, their values as column positions
  - All queen positions can be fully specified by an array **queens** of integers with length N
  - queens[5] = 6 means that the queen on row 5 is in column 6
- Eliminate the column constraint by making sure that the variable values are a permutation of $(1, \ldots, N)$ at all times
  - Initialize the array as $(1, \ldots, N)$ and only ever change it by swapping the values of two elements within the array
- Only the diagonal constraint remains to be checked

An observation about diagonals

- The sum of row and column indices is constant on positively sloped diagonals, the difference is constant on negatively sloped diagonals

Sum:

0	1	2
1	2	3
2	3	4

Difference:

0	-1	-2
1	0	-1
2	1	0

Checking the Diagonal Constraint

- A simple observation about diagonals:
  - There are 2*N-1 diagonals with positive and negative slope each
  - All positions on the same diagonal with positive slope have the same value for $row + col$
  - ... and $row - col$ on each diagonal with negative slope
- The diagonals that a queen threatens can be calculated as $row + queens[row]$ and $row - queens[row] + N - 1$(adding N-1 to always have positive indices)
- Use two arrays of Boolean values with length 2*N-1 to save which diagonals are occupied
- Checking new queens is done in constant time by calculating their diagonal indices

Backtrack Search

- Simple and general, but inefficient algorithm used to solve CSPs
- Incrementally append new variable assignments of variable X_{i+1} to a sub-sequence $(X_1, ..., X_i)$ until all variables are set
- Check all constraints after each step, backtrack and change previous assignments if no further extensions can be made
- The simplest version iterates through all variables in a deterministic order, and tries each possible extension for that variable in a deterministic order
- Terminate if all variables are set or first variable runs out of choices

Backtrack Search pseudo-code

Backtrack search is usually implemented recursively

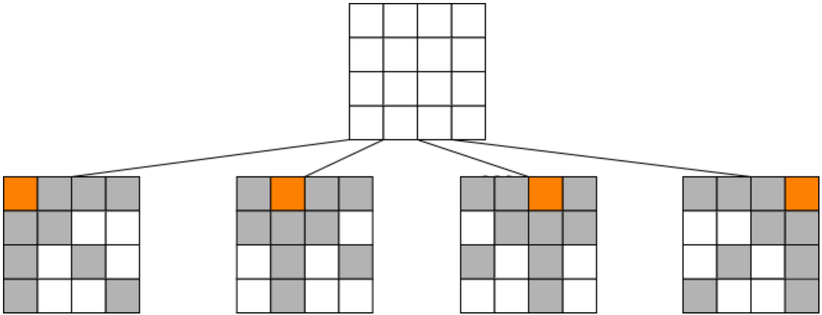
```
bool backtrack(int row)
{
  if (row == N)
    return true;                       // all queens placed

  // try choices left-to-right
  for each valid column c in row
  {
    place queen in column c;
    if (backtrack(row+1))              // next row recursively
      return true;                     // subproblem solved!
    else
      remove queen in column c;        // try next solution...
  }

  return false;                        // no solution found
}
```

Detailed implementation notes are in the thesis

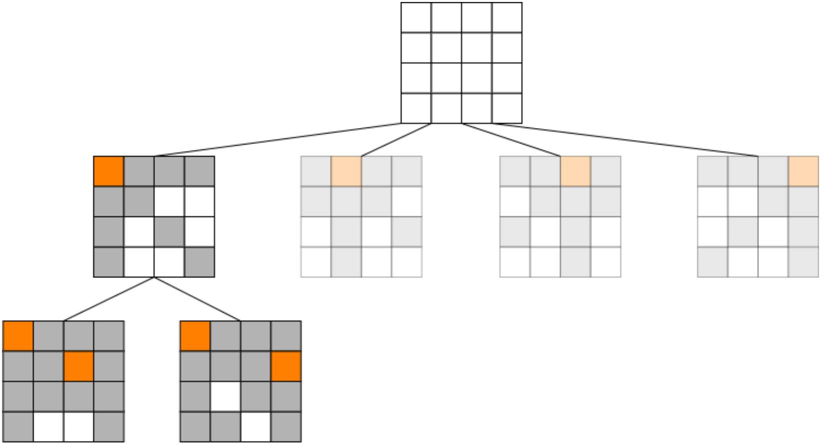
The 'search' part of Backtrack Search

- Backtrack Search can be interpreted as a depth-first search algorithm through a tree of all possible sequences of assignments
- The goal of search is to find a node with depth N (counting the root as depth 0)
- The leftmost node is searched first. If a node has no children, but is not depth N, the search algorithm "backtracks" to the previous node and checks the next child
- Backtrack Search is extremely slow for big problems, due to its exponential nature

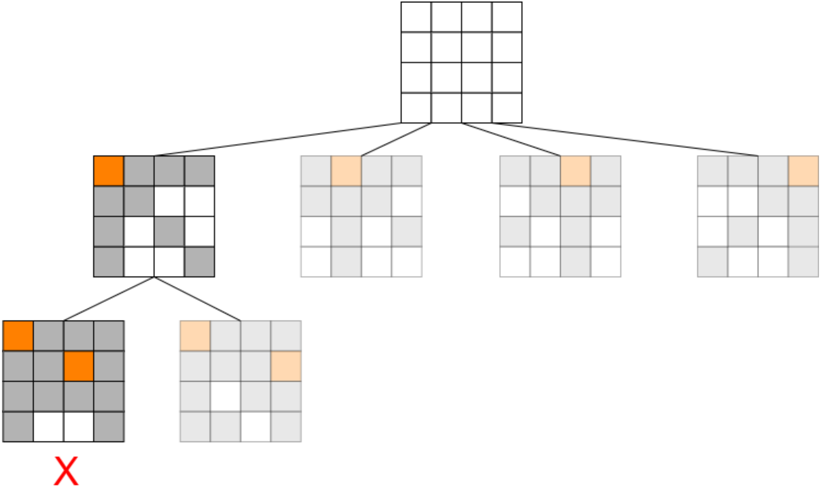
Backtrack Search deterministic order



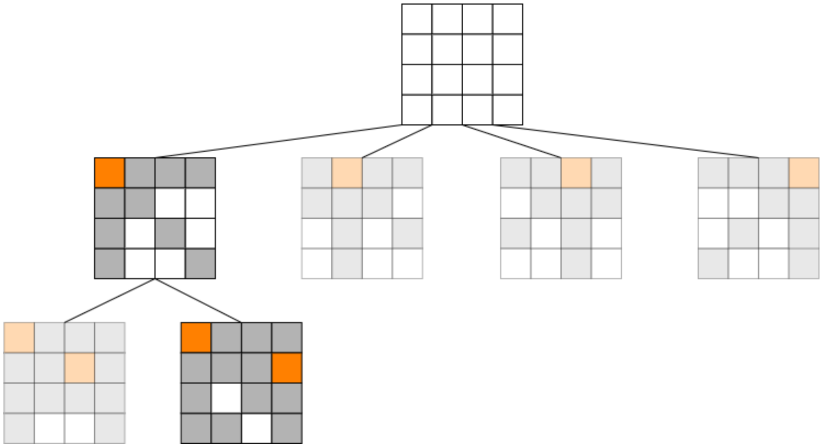
Backtrack Search deterministic order



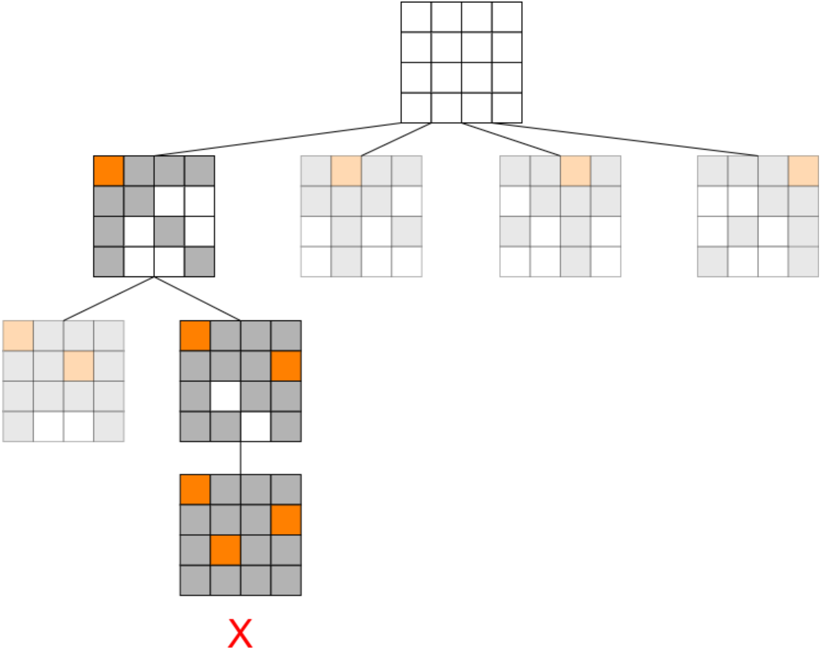
Backtrack Search deterministic order



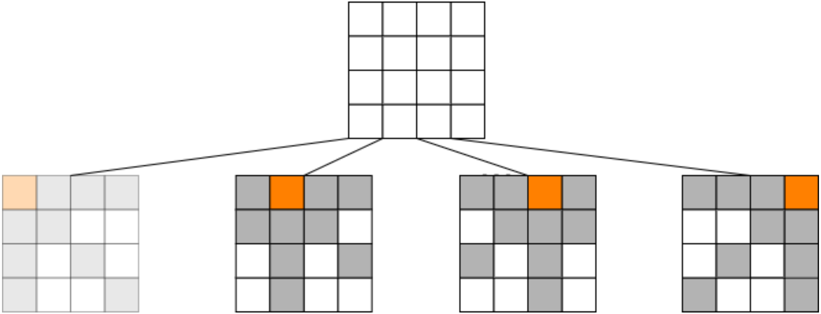
Backtrack Search deterministic order



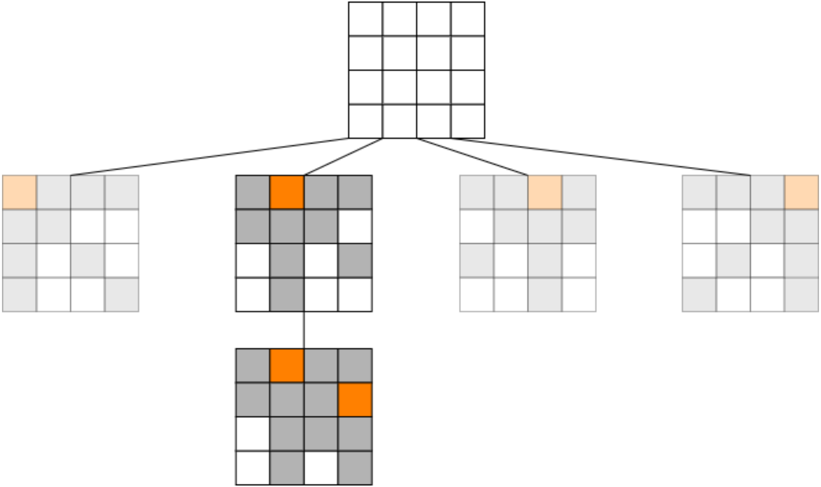
Backtrack Search deterministic order



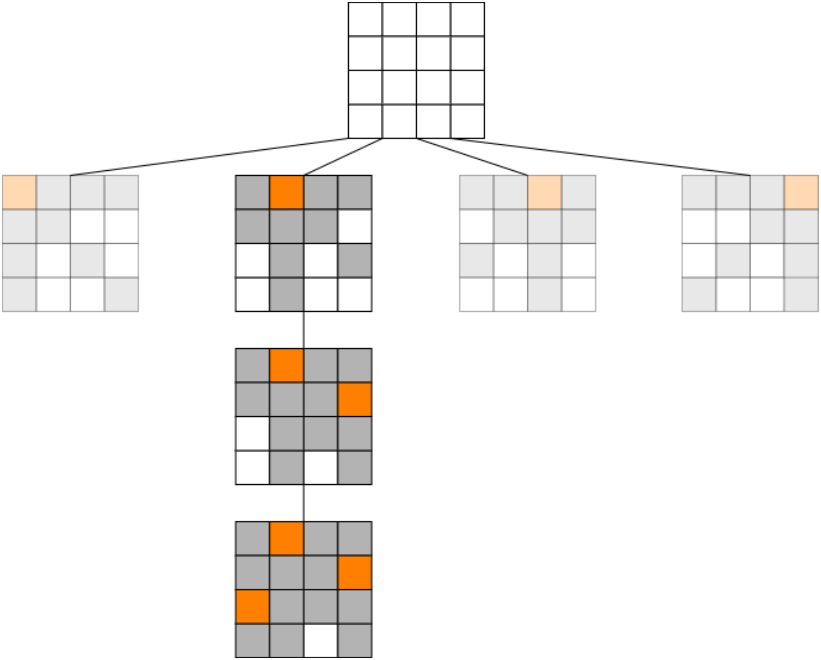
Backtrack Search deterministic order



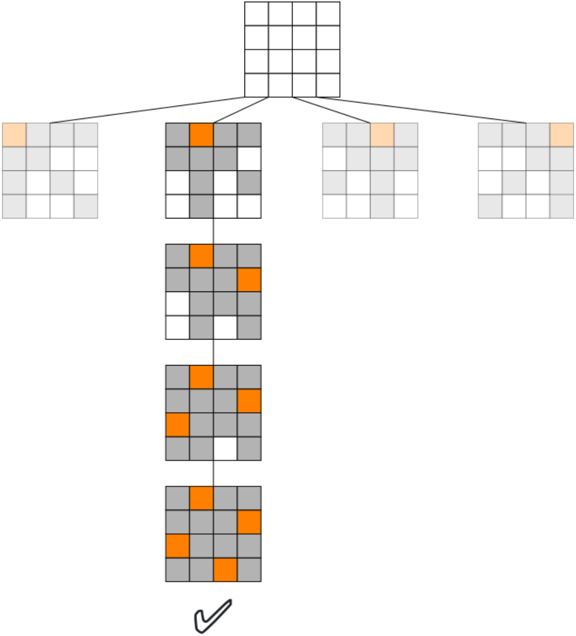
Backtrack Search deterministic order



Backtrack Search deterministic order



Backtrack Search deterministic order



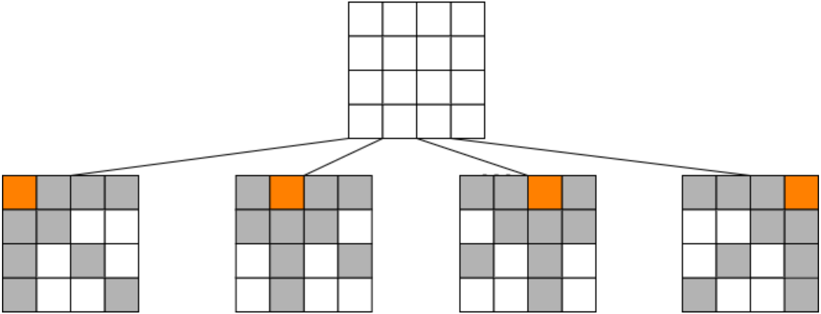
Randomized Incremental Constructions(RIC)

- Another general approach for solving CSPs
- Place a queen in each row in a random valid location
- Two central operations:
  - Choose a random possible extension for a partial solution through an **increment** operation
  - **Reset** to an empty sequence if no more extension can be made
- No attempt is made to improve a failed partial solution
- Simple to implement for specific CSPs by defining the operations
- Somewhat counter-intuitively, this is much faster in practice than Backtrack Search

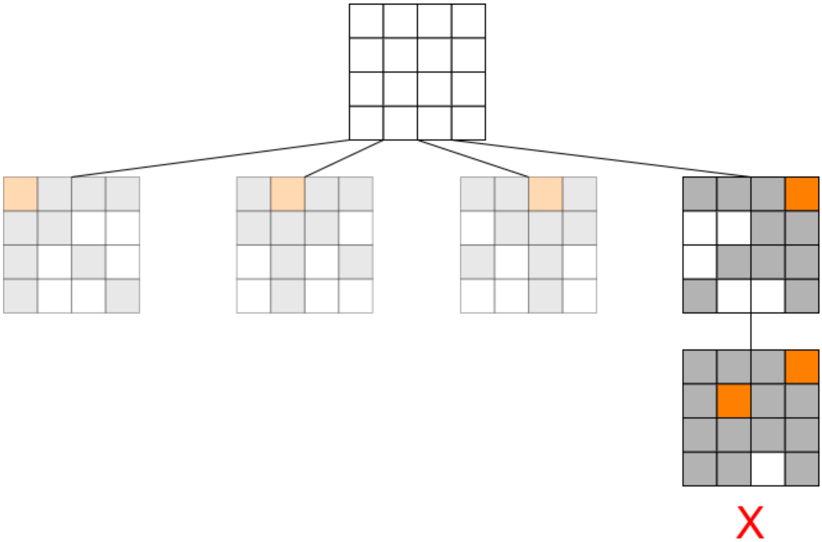
RIC operations increment and reset

- **Reset** operation
  - ... restores the solution back to the empty state
  - ... is trivial for N-Queens
- **Increment** operation
  - ... computes all possible extensions for the solution
  - ... chooses one possible extension at random
  - ... returns true only if there was a valid extension that has been chosen

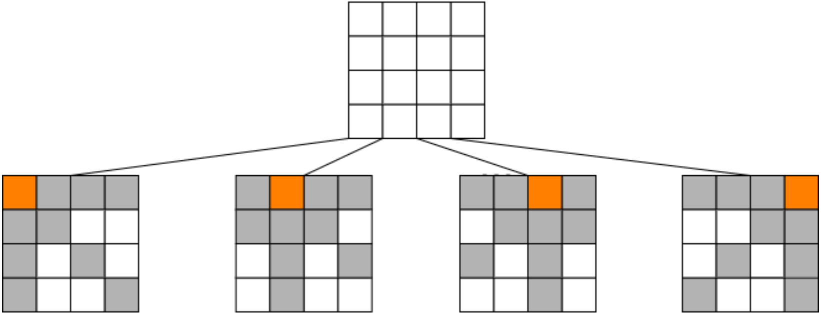
RIC randomized search



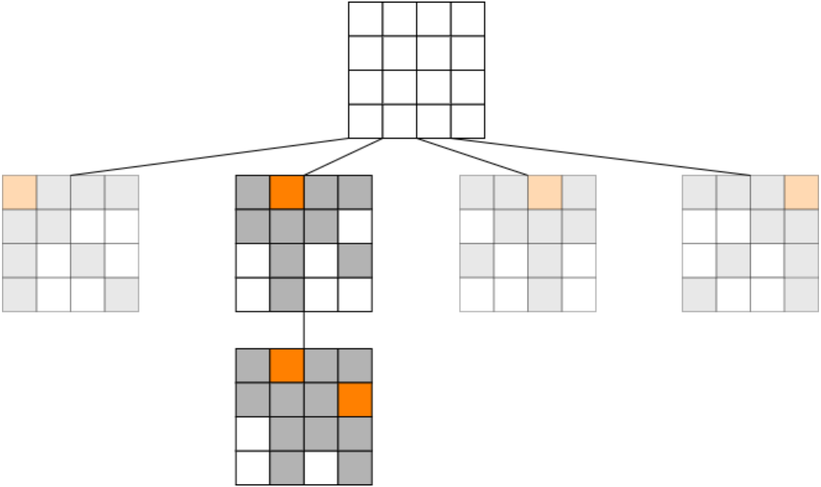
RIC randomized search



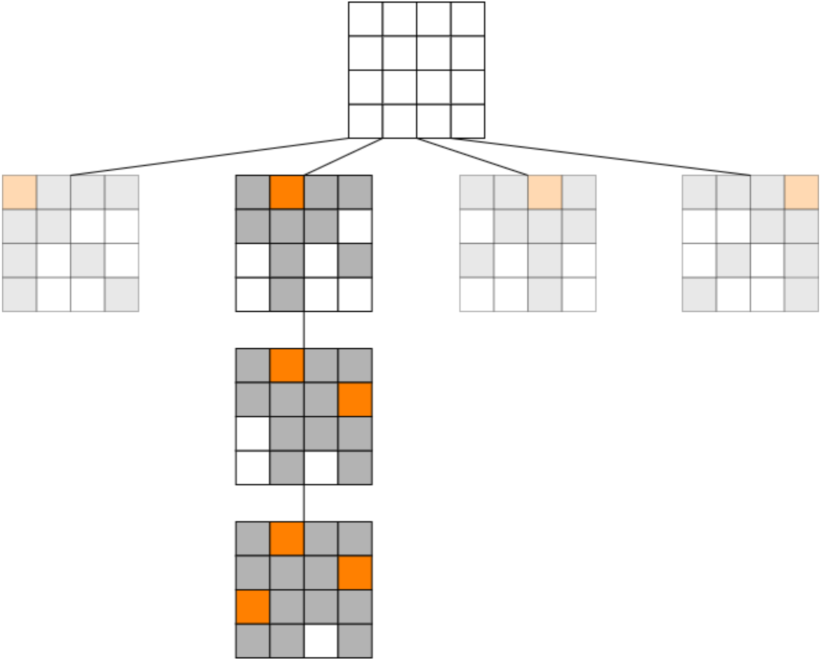
RIC randomized search



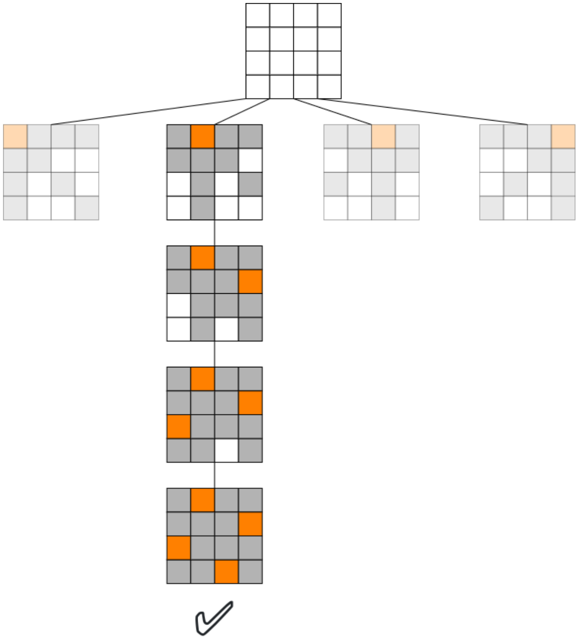
RIC randomized search



RIC randomized search



RIC randomized search



RIC pseudo-code

The goal is to increment N times without failure.

```
void RIC(int N)
{
  solution=empty solution
  while(!success)                  //Repeat until success
  {
    for row=1 to N                  //increment N times
    {
      if(!increment(solution, row))  //If increment failed
      {
        reset(solution);           //Reset and repeat
        break;
      }
    }
    if(row == N)                    //All queens placed
      success = true;
  }
}
```

Increment pseudo-code

```
bool increment(solution, int row)
{
  possible_extensions = [];
  for each column c in row  //Compute possible extensions
  {
    calculate diagonal indices row+c and row-c;
    if(both diagonals are unoccupied)
      add c to possible_extensions;
  }
  if(possible_extensions is not empty)
  {
    c = random element of possible_extensions;
    place queen in row at column c;
    return true; //Solution has been extended
  }
  return false;  //No extension can be chosen
}
```

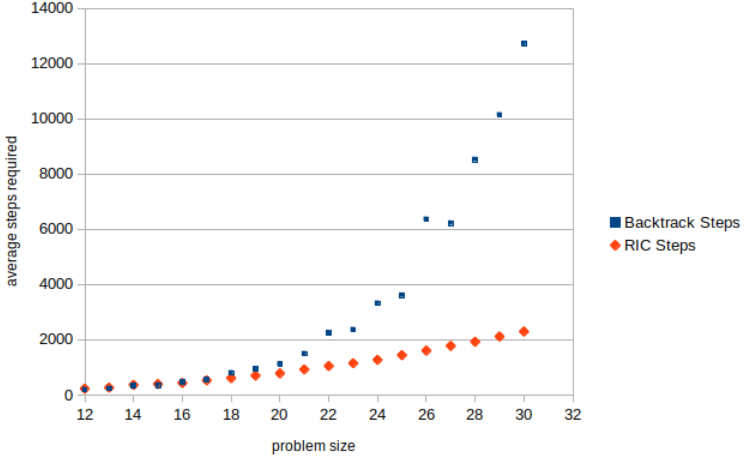
Advantages of RIC over Backtrack Search

- Cheap loop instead of recursion
- Does not need to keep track of previous choices
- Bad paths are quickly reset, not explored thoroughly like for Backtrack Search
- Worst-case rarely occurs

A comparison of Backtrack Search and RIC

- Backtrack Search may take a long time to find a solution if it traverses the tree in a bad order
- The traversal-order has been randomized and sampled 10000 times for each data-point
- We compare the steps required for problems of various board sizes
  - A step is an increment for RIC and a recursive step for Backtrack Search
  - This allows us to compare small time-frames accurately
  - ... and prevent interference of outside factors
- Backtrack Search performs much worse and has worse scaling behavior

A comparison of Backtrack Search and RIC

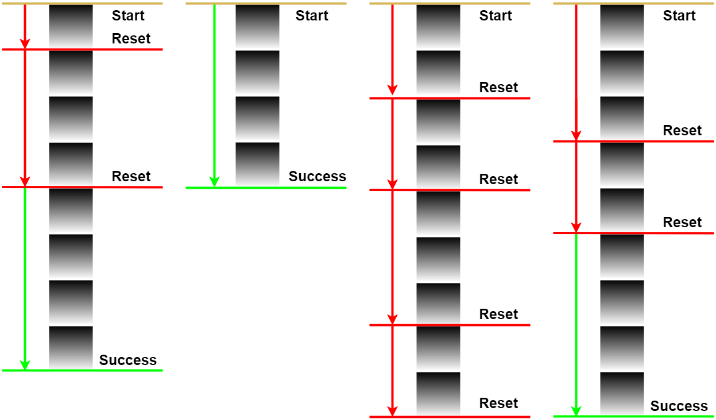


Parallel Execution of RIC

- RIC can be run in parallel due to its randomized nature
- Different threads make distinct random choices
- Terminate after one thread finds a correct solution
- No parallelisation of the workload, speedup exclusively due to faster convergence
- Early termination through a single shared variable is extremely cheap
- Implementation for the CPU is very straight forward in a parallel block with a check for success of other threads at each step
- All threads operate independently, so no restrictions of freedom are needed

Variance of steps across threads

Different threads making distinct random choices leads to variance in the required steps

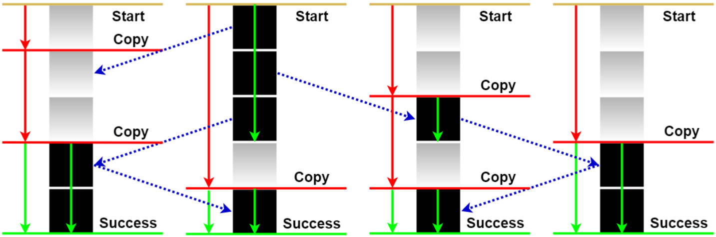


Collaborative RIC

- RIC implicitly assumes that any partial solution, regardless of how many steps have been taken, has the same probability to succeed in the end
- Instead, we assume that solutions with some progress have an inherently better chance to have at least one valid solution
- Starting a high number of threads has a high chance of at least one of them having a correct solution in their explored branch
- On this assumption, it makes no sense to reset any solution completely, if there are other solutions still active
- If there are still active threads, we replace the reset operation with a **copy** operation
- The copy operation chooses a random active thread and copies the solution of that thread to the calling thread

Solution sharing in collaborative RIC

- Copied solutions are likely to choose different extensions
- Promising branches of the tree are explored more thoroughly
- ... but still reset after at most N steps
- $\Rightarrow$ We combine the thoroughness of exploring more paths from Backtrack Search with the quick resets of RIC



Collaborative RIC pseudo-code

```
void collab_RIC(int N)
{
  execute in parallel with shared variable success
  {
    solution=empty solution
    while(!success)    //Repeat until success
    {
      for row=1 to N      //increment N times
      {
        if(success)    //Another thread succeeded
          break;

        if(!increment(solution, row)
           && !copy(solution))      //We cannot copy
        {
          reset(solution);          //Reset and repeat
          break;
        }
      }
      if(row == N)    //All queens placed
      success = true;
    }}}
```

Copy operation

- Copy operation iterates through all threads in random order
  - ... checks if that thread currently has an active solution
  - ... and copies that solution if it does
  - ... or returns false if there was no thread with an active solution
- The copy operation quickly finds an active thread when almost all threads are active
- ... which is almost always because the copy and reset operation are fast and we start with all threads being active

Copy: restricting freedom

- The copy operation depends on the state of another thread's solution
- If that solution is changed while being copied, the copying thread receives a broken solution
- Two main possibilities to restrict the freedom of threads:
  - Barriers
    - ensure that all threads finish their increment operations before any thread attempts to copy
    - force all threads to synchronize
    - are very slow, especially for high thread counts
  - Locks
    - define mutually exclusive operations that can not occur concurrently
    - limit only the freedom of threads that are affected
    - are much faster and have better scaling for high thread counts
- Locks are more well suited for this task

Copy: Locks

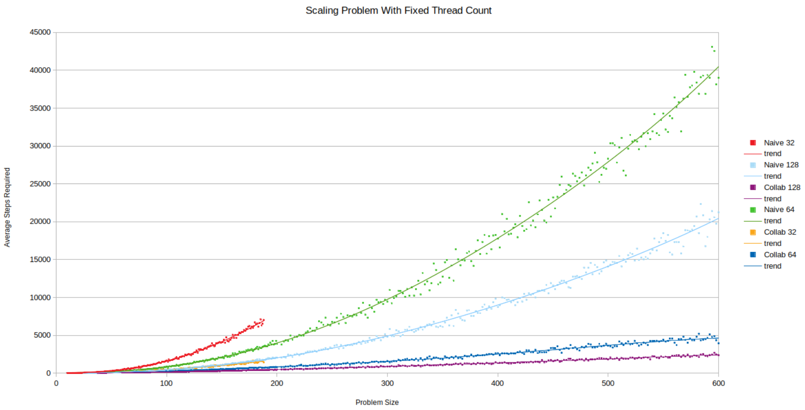
- A lock defines a critical region of code
- Each thread that wants to execute this critical region needs to acquire ownership of the lock
- Only one thread may own the lock at the same time
- All other threads need to wait for the lock to be released
- We assign a lock to each thread
  - The thread acquires its own lock when calling increment and reset
  - The thread acquires the other thread's lock when copying that thread's solution
  - $\Rightarrow$ A solution can never be changed while being copied from
  - Threads can run independently unless they are being copied from

Experimental: Counting Steps

- We utilize a global step counter, that counts the average number of increment operations across all threads
- This allows us to compare the scaling behavior of naive RIC compared to collaborative RIC in regard to the problem size, regardless of the specific hardware used
- Because both algorithms are random in nature, each data-point is the mean value of 250-500 random samples
- We test various problem sizes to compare the scaling behavior

Scaling the problem size

- The upper curves are naive execution for 32, 64 and 128 threads for problems up to 600 queens
- The lower curves are collaborative execution



Experiment interpretation

- All collaborative curves show much better scaling behavior
- Using more threads always reduces the required steps for both versions
- 32-threaded collaborative execution is still preferable to even 128-threaded naive execution
- The collaborative curves appear almost linear. Likely to be the shallow beginnings of a higher-order curve
- Collaborative execution is clearly superior
- Further experiments concerning high thread counts, real-time and alternative implementations are in the thesis

Further scaling

- The prior implementation is limited to execution on a CPU
- We observed a significant speedup for both versions by using a high thread count, even far beyond the CPU hardware capabilities
- It could be nice if we could scale the thread count even higher
- For this, we use a GPU, which has far more computing units
- However, each unit is slower than a CPU core, and has other special restrictions that may destroy any speedup gained from increased thread count

GPU considerations: Warps

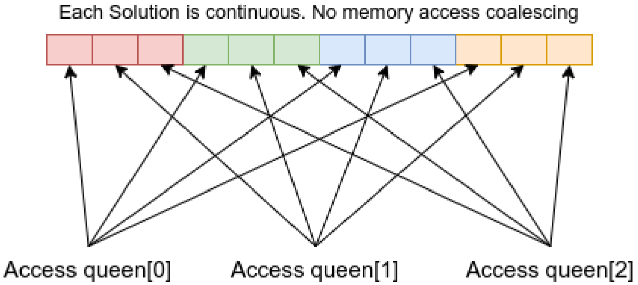
- GPU threads operate best when they compute data independently from other threads
- Warps are groups of usually 32 threads, that must always execute the same instruction at the same time, though they may operate on different data
- Branches in code force execution within a warp to be sequential
- Both RIC versions operate mostly independently from other threads, but there are many branches within the central loop and the increment operation

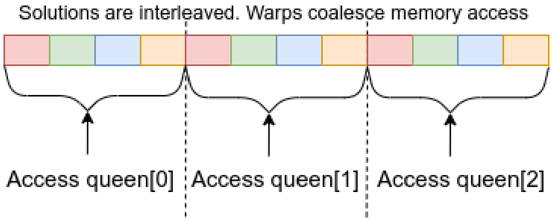
GPU considerations: Memory

- Solutions are too big to be stored locally in registers of a thread or in shared memory
- ... so we store them in global memory
- Global memory access is comparatively slow
- SIMD operations allow for access to 32 Bytes of continuous memory that is distributed across a warp
- This requires threads within a warp to access continuous memory in the same instruction
- If the solutions of each thread are stored in continuous memory, a warp would access 32 disparate memory locations
- $\Rightarrow$ The solutions for all threads must be stored in an interleaved manner

Coalescing global memory access

4 Threads in a warp attempt to access 3 queens





GPU considerations: restricting freedom

- The lock-based freedom-restrictions can not be used efficiently on a GPU
- The coupled nature of a warp would force all threads within a warp to block if any of them is copied from
- CUDA also offers no build-in lock implementation
- $\Rightarrow$ We must use a barrier-based implementation

Barriers: Phases

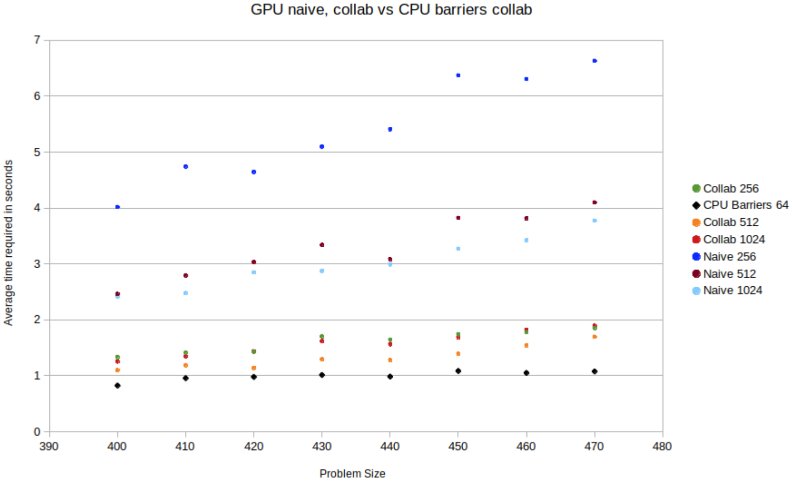
- All threads must reach a barrier before any of them continue execution
- Split execution into three distinct phases that are separated by barriers
- Phase 1: All threads increment their solutions
- Phase 2: Compute, whether all of the threads failed
- Phase 3:
  - If all threads failed: Reset all threads
  - If not all threads failed: copy from active to inactive threads
- Repeat until a solution is found

Barrier RIC pseudo-code

```
execute in parallel with shared variable failAll
{
  solution = empty solution;
  while(failAll)    //Repeat until success
  {
    for i=row to N      //increment N times
    {
      barrier;
      extended = increment(solution, i);
      failAll = true
      barrier;
      if(extended)    //This thread extended
        failAll = false;
      barrier;
      if(failAll) //All threads failed
            {
              reset(solution);  //Reset all threads
              break;  //Repeat from empty solutions
            }
      else if(!extended) //Not all threads failed
        copy(solution);  //Copy from other threads
  }}
```

A GPU test

- Slower execution times force us to test fewer values



GPU test interpretation

- All collaborative data-series still perform better than their naive counterparts on a GPU
- Comparatively low-threaded naive execution (256 threads) performs extremely badly
- ... while collaborative data-series are closer together
- A 64-threaded CPU barriers implementation performs slightly better than all GPU versions
- The fact that the results are comparable is great
- GPU execution is unlikely to be better than CPU execution but might be, given the right hardware

Conclusion

- RIC is an easy-to-implement, general algorithm for solving CSPs
- RIC can be sped up by massively parallel execution
- Collaborative RIC is an improved, just as general, and almost as easy to implement version of RIC
- Collaborative RIC is preferable to naive RIC on a CPU as well as a GPU
- CPU execution is likely preferable to GPU execution, but might not be, dependent on the available hardware

Questions?

Questions?