I’d like to introduce you to the world of **Constraint Satisfaction Problems,**

algorithm selection strategies—methods that choose a suitable solving method for a particular problem.

This is a key area in computer science and artificial intelligence.

I'll start with an overview, then discuss specific solving techniques.

Then, to bring these concepts to life, I will present some concrete cases I worked on.

To enhance algorithm performance beyond theoretical complexity (big-O notation), what really matters? MY answer is what I call/**named 'hidden constant values' and 'practical measurements.'**

It's crucial to examine their impact in real-world scenarios—how constants and lower-order terms influence runtime, particularly with large datasets or in situations with low resourves(mem) demanding high throughput within tight time constraints.

Minor in theory, Importance of empirical testing and performance tuning discover that - the devil is often hidden in the details 😊

**CSP Solving Techniques**

It’s crucial to recognize that no single strategy offers a **one-size-fits-all** solution. The complexity and diversity of problems within any given domain necessitate a flexible and informed approach to selecting algorithms.

**Backtracking**: A technique where the algorithm tries to incrementally build a solution, reverting ("backtracking") when it encounters a dead-end. It inspects all possible branches and solutions.

**Forward Checking**: This method helps in reducing the scope of the search quickly by checking the constraints of upcoming variables right after a value is assigned to one

**Constraint Propagation**: This reduces the search area by figuring out new rules from the current ones. AC-3 (Arc Consistency Algorithm 3) is a classic algorithm example.

**Heuristic-Based Search**: Uses smart guesses to speed up the search. This can mean picking the order to assign variables or choosing values that cause the least restriction.

**Metaheuristic Techniques**: For problems where conventional algorithms struggle, metaheuristic approaches like Genetic Algorithms and Tabu Search offer robust alternatives. These techniques explore the search space probabilistically rather than deterministically, frequently delivering good solutions for highly complex or large-scale issues.

**Algorithm Selection with Machine Learning**: Since no one algorithm works best for everything, newer methods use machine learning to find the best fit based on what the problem looks like. Mixing traditional methods with AI can make choosing the right algorithm smarter.

**A CSP Application - The Sliding Problem -**

Given a 4x4 grid Puzzle and the objective is to reach a given target configuration by sliding tiles left/right/top/bottom in empty space. The challenge is to move the tiles in the fewest number of moves to transition from an initial configuration to a given target configuration. This problem highlights the specific challenges associated with large state spaces and the necessity for efficient solving strategies.

A\* algorithm with heuristics demonstrates effective strategies to solve that problem. It addresses the challenge of finding the optimal sequence of moves to achieve the target configuration.

A\* is considered a variant or an extension of branch and bound strategy.

**The A\* Algorithm – and Branch and Bound technique**

**Branch and Bound:** It explores the search space through division (branching) and evaluates partial solutions using a bounding function (bound). If a bound of a partial solution exceeds the best solution found so far, that branch is removed from the search.

In Branch and Bound, partial solutions are typically stored in a data structure like a priority queue or a heap, where the priority is given by the bound (limit) of the partial solution. This bound might be the estimated/ measurement minimum cost to complete the partial solution indicates the solution's promise of leading to an optimal outcome.

**The A\*** algorithm is a heuristic search algorithm used to find the shortest path between two nodes in a graph, considering the costs associated with the edges of the graph.

In A\*, partial solutions are managed using two sets (or lists): an open list and a closed list. The open list contains nodes that have been discovered and need to be explored, meaning those nodes that have been reached but whose neighbors haven't been examined yet. Each node in the open list is evaluated based on a function. The closed list contains nodes that have already been explored.

Cost function f = g + h used to prioritize selection, where g is the cost from the start node to the current node (counting steps to current node ,for example), and h is a heuristic that estimates the cost of the cheapest path from the current node to the target.

**A**\*: Guarantees an optimal solution as long as the heuristic function ℎ it uses is admissible, meaning it never overestimates the cost to reach the goal from any node.

**From Theory to problem solving - Exposure Detector**

* **Objective problem:**

Design a program that identifies the topset 30 pixel values in a grayscale image, provided by an image sensor, where each pixel value corresponds to a 16-bit word in memory. description of the problem is intentionally fuzzy.

* ***Key Challenges:***

1. **Analysis for Optimal Solution:**
   1. Rigorous evaluation to ascertain the most effective method for high-value pixel detection.
   2. performance, accuracy, and operational efficiency in a production environment.
2. **Production Quality Assurance:**
   1. Ensuring the solution is robust, error-free, and ready for deployment in real-world setting.
   2. Focus on maintainability and scalability to adapt to future requirements or enhancements.
3. **Managing Ambiguities:**
   1. Accommodating for any ambiguities in problem description, while still delivering precise results.
   2. Implementing a flexible approach that can handle variations in image quality or format(for example pixel size).
4. **Optimization of Resources:**
   1. Balancing space and time complexity to optimize performance without compromising the quality of the solution.
   2. Designing an architecture that not only solves the problem efficiently but is also modular and reusable for similar next future challenges.

**Identification and Solving Challenge**

1. **Assumptions Made:**
   1. **Positive Value Range:** It's assumed all pixel values are positive, simplifying the search for maximum values.
   2. **Non-Sorting Requirement by values or positions:** The algorithm does not sort the entire set of pixel values; it is optimized to find the highest values directly.
   3. **Top Distinct vs. Top N Values:** The solution assumes that finding the top 'N' values does not necessarily mean they must be distinct.
   4. **Handling Identical Values, so multiple solutions are available:** In cases where multiple pixels share the same value within the top range, any solution is right no matter their order of occurrence in the data.
   5. **Fixed Data Set**: The algorithm operates under the assumption that the input data set remains unchanged during its entire execution. This means that no updates or modifications to the pixel values are made while the search for the top values is underway, ensuring consistency in the results.
2. **Performance Remarks:**
   1. **Efficiency in Memory Utilization:** Direct in-memory operations are employed to avoid unnecessary data duplication, enhancing performance.
   2. **Real-Time Data Consideration:** The solution ensures that image data remains unaltered during processing to prevent inconsistencies.
   3. **Solution Extensibility:** The modular design of the image processing logic allows for reusability in various contexts with minimal adjustments.
   4. **Performance Optimization Strategies:**
      1. For **multiple queries on the same image**: Consider pre-sorting the pixel values for repeated efficient top 'N' value extractions.
      2. For **single queries across multiple images**: A direct heap data structure approach is utilized for optimal performance.
      3. Considering the **ratio of top 'N' values to total pixel count**: Adjust the strategy based on whether the 'N' value is a small or large proportion of the total pixels, optimizing for either case.
      4. **Multi-Core Processing**: The algorithm is adaptable for execution in multicore environments, distributing the workload for concurrent processing.

**Applied Solution**

1. **Infrastructure/Architecture design**
   * **The IImage** template interface manages the image data access, ensuring flexibility for future enhancements​
   * The **ImageWrapper** - classes provide concrete implementations first for real-life image data manipulation and second for simulations
   * The **VectorImage** - facilitates testing, simplifies and make easy testing process due to vector format, respectively ​​.
   * second facilitate testing considering vector format, respectively​​.
   * The **ImageProcessor** is templated to work with any data type and relies on the IImage interface for accessing pixel values. This design allows it to be flexible and adaptable to various image formats.
   * In-memory processing for high performance.

**Algorithm Selection**

* **Heap-based Algorithm:** Efficiently manage pixel values using a heap data structure to identify the top pixels​​.
* **Optimized Heap Variations:** Adjusted heap algorithm for improved performance in general scenarios.
* **Counting Sort Algorithm:** A non-comparison-based sorting algorithm well-suited for sorting pixel values with a smaller range​​.
* **Set-based Algorithm:** Unique pixel values management using a set data structure(binary search red/black tree) to quickly retrieve the highest pixel values​​.
* **Parallel/distributed Processing:** Exploring parallel algorithm variants to determine the most efficient approach for processing large images​​.

**Verification and Simulation Overview**

**Functional Unit Testing**

* Utilized the Google Test framework for comprehensive unit testing.
* Validated the handling of edge cases, ensuring robust performance across images with both uniform and extreme pixel values and positions.

**Performance** **Unit Testing**

* Conducted performance testing on execution times/memory and assess algorithm efficiency.

**Solution Evaluation & Systematic Benchmarking**

* Performed simulations across a variety of image scenarios to rigorously benchmark the solution.
* Tested against diverse pixel distributions including uniform, random, and those following gradient and Gaussian patterns.
* Created matrices with variable regions to represent different imaging conditions realistically.

**Performance Analysis and Data Generation**

* Generated CSV data sets for a granular analysis of execution time and memory usage.
* Benchmarked using data sets reflective of practical imaging conditions, such as uniformly distributed pixel values and sorted matrices, to ensure comprehensive solution evaluation.

**Optimization Techniques**

**…………………………**

**Optimization**

**IIMage:**

The **getNextPixelValue()** simplifies pixel access by avoiding repetitive and random position calculations(multiply, etc). It progresses through pixels sequentially – keeps track- using a pointer, delivering the current pixel value, and advancing to the next, speeding up the process.

Normally, to find a specific pixel, you might need to calculate its position every time.

**Virtual func. - C**uriously Recurring Template Pattern – overhead in speed and memory

It enables polymorphic behavior to be resolved at compile time rather than at runtime. This can lead to performance improvements since it avoids the overhead associated with virtual function calls, such as **dynamic dispatch.**

**ProcessImage:**

1. **Efficient Pixel Addition**: Instead of adding and comparing each pixel individually with the heap's minimum element, the code initially adds a set of pixels from complete rows and from a partial row if necessary, based on **topN**. This reduces the number of comparisons and reorganizations of the heap at the beginning of the process.
2. **Using std::make\_heap**: After the initial pixel set is added, **std::make\_heap** is used to create a valid heap from the vector, replacing the need to insert and reorder the heap for each added pixel at the start with **std::push\_heap**
3. **Optimized Pixel Access**: Using **moveToStart(topN)** and **getNextPixelValue()** to jump directly to the relevant pixel reduces the number of pixel accesses that do not contribute to the top **N**, improving image memory access and iteration efficiency.
4. **Reducing Calls to getPixelValue**: By calculating the heap's minimum value only after updates (and not for every checked pixel), the number of function calls is significantly reduced, which is more efficient.
5. **Utilizing \_\_builtin\_prefetch and \_\_builtin\_expect**: These intrinsics suggest to the compiler the expected behavior of the code, potentially improving branch prediction and data prefetching, though these optimizations are more subtle and hardware/compiler-dependent.
6. **Avoiding Recalculation of heapMinValue If the Heap Is Not Updated**: This check reduces vector access when the heap hasn't been modified, lowering overhead in cases where pixels do not change the heap's content.

**Optimization - Graphic comparation**

Graphic representing execution time data for two variations of an image processing function. Image suggests that the optimizations can have a notable impact on execution time.

**Validating Testing Environment Consistency**

The graph depicts two execution runs for the same method, both illustrating closely aligned execution times across various datasets, confirming the consistency of the testing environment.

The minor variations between the first and second runs indicate a high degree of reproducibility, suggesting reliable performance metrics for the algorithms tested.

**Synthetic data and data collection**

1. **Data Generation:**
   * I have generated data that covers a wide spectrum of pixel distributions and even extreme scenarios. This approach enhances the relevance of the performance tests, making benchmarking more applicable to real-world situations.
   * Created diverse datasets, matrices with variable regions, to cover a broad spectrum of test cases​​:
     1. Random image regions, Image region Sorted Ascending and or Descending
     2. With Specific Distribution, Uniform Distributions
     3. Image with controlled region for pixel color distributions like uniform, Gradient, Gaussian(normal), or constant.​​...

TODO:!!!!

**Performance-Driven Algorithm Experimentation**

1. **Performance Benchmarking:**
   * Executed each algorithm variant under various image conditions to get their performance and scalability.
   * Gathered metrics such as execution time and memory usage to inform the decision on the optimal algorithm which is on TODO list 😊!​​.
2. **Experimentation with Variants**
3. Executed a suite of algorithms, including heap-based, counting sort, set-based, and parallel processing to address the pixel identification challenge.
4. Each variant was stress-tested under specific use cases to evaluate their performance edges and limitations.

* **Heap-Based Innovations:**
* **Standard Heap:** Implemented as a baseline for performance comparison.
* **Optimized Heap Variations:** Introduced heap unrolling and priority queue merging for enhanced speed in different processing conditions.
* **Non-Heap Alternatives:**
* **Counting Sort:** Deployed for its linear time complexity in scenarios with limited pixel value range.
* **Set-Based Approach:** Utilized for scenarios prioritizing the elimination of duplicate values and quick access to unique high-value pixels.
* **Concurrent Processing Techniques:**
* Explored multi-threading to take advantage of multicore processing capabilities, significantly cutting down processing time for large images.

**Data Analysis**

CSV data generation was used for performance analysis, enabling a detailed examination of execution time and memory usage​​.

The execution times show that different algorithms excel in different scenarios. Counting Sort is fast but uses more memory, and while heap strategies work well in some cases, there's no single best option for all types of data.

The unpredictable varying performance across different data types indicates a broader Constraint Satisfaction Problem (CSP), where selecting the optimal algorithm is contingent on specific characteristics of the data.

The preliminary idea of modeling with a polynomial formula for algorithm selection has encountered significant obstacles, primarily due to the NP-complexity nature of the problem. This complexity makes it impractical to accurately model the decision space with a simple polynomial equation.

Inference from Research and Publications

**For example** some research publication [Applied Sciences | Free Full-Text | A General Framework Based on Machine Learning for Algorithm Selection in Constraint Satisfaction Problems (mdpi.com)](https://www.mdpi.com/2076-3417/11/6/2749) suggested AI model as a more dynamic and adaptable approach to algorithm selection, leading to a significant decrease in search costs

Random Forest Classifier, could be trained to predict the best algorithm by learning from the performance patterns observed in the benchmark data.

**ML - predict best strategy.**

Defines features to be used for model training, which include image size, 'topN' ,and image type.

I write ML coding to predict the most efficient algorithm for processing data based on execution time. It loads training and test datasets, selects the best-performing algorithm for segments of the training data to labeled best algorithm, and then uses a Random Forest Classifier to learn these selections. The model predicts the best algorithm on the test data, and the script visualizes the actual versus predicted execution times for each type of data matrix. Red dashes depict the prediction.

This process also offers a demonstration into accuracy of the ML model through visual comparison letting you validate the machine learning model.

Saving the model in a format compatible with C++ for subsequent predictions. If the model's complexity is low, it will result in minimal overhead, thus justifying its use for practical prediction applications.

In production, when prediction is running, the image type is ascertained through a sampling method from the dataset, categorizing data according to its characteristics.

**RandomForest**

Random Forest is a machine learning algorithm that operates by constructing a multitude of decision trees at training time.

It is an ensemble learning method, which combines predictions from multiple machine learning algorithms to make more accurate predictions than any individual model.

When you ask them to make a decision, each one looks at the data differently and comes up with its own answer. Then, they get together and vote to decide on the most popular answer among them. This way, you get a more well-rounded decision than if you just asked one person (or in this case, used one decision tree).

1. **Creates Many Trees:** The Random Forest algorithm starts by creating lots of decision trees, each made from a random selection of data points and features from your dataset.
2. **Makes Predictions:** Each tree in the forest works out its prediction based on the data it was given.
3. **Combines Predictions:** The algorithm then looks at all the different answers from the trees and picks the most common one (for classification tasks) or the average (for regression tasks).

**Recall = TP / (TP + FN), Precision = TP / (TP + FP)**

**F1 Score = (2 \* Precision \* Recall) / (Precision + Recall)**

1. **Recall** is a measure of how many of the actual positive cases we were able to predict correctly with our model. To calculate it, you take the number of true positives and divide that by the sum of the true positives and the false negatives. In other words, it's the proportion of actual positives that were identified correctly.
2. **Precision** tells us the proportion of positive identifications that were actually correct. You calculate precision by dividing the number of true positives by the sum of true positives and false positives. Essentially, it's how many of the predicted positives are true positives.
3. **F1 Score** is a way of combining the precision and recall of the model, and it's the harmonic mean of precision and recall. The F1 score can be calculated by taking twice the product of the precision and recall, and then dividing that by the sum of the precision plus recall. This gives you a single score that balances both the concerns of precision and recall in one number.

Support refers to the number of actual occurrences of the class in the specified dataset. It's used in classification reports to show the size of the dataset for each class. For example, if you're trying to predict whether an email is spam or not, the support for the spam class would be the total number of spam emails in the dataset.

In terms of a formula or calculation, support is not something you calculate from other metrics like precision or recall, but rather, it is a count of how many times the true label appears in the dataset.

* **Support for a Positive Class**: This is the sum of the True Positives (TP) and the False Negatives (FN), which essentially gives you the total number of actual positives in the dataset. **Support (Positive) = TP + FN**
* **Support for a Negative Class**: Similarly, this is the sum of the True Negatives (TN) and the False Positives (FP), giving you the total number of actual negatives. **Support (Negative) = TN + FP**

key points

functional testing /unit testing/regresion testing

car's surroundings provisioning/orchestration/ProofofConcept

MPI (Message Passing Interface)/HPC hadoop/spark

**Objectives Slide Content:**

1. **Introduce CSP Challenges**:
   * Briefly define CSPs and their importance in various fields such as operations research, artificial intelligence, and computer science.
   * Highlight the primary challenges in solving CSPs, such as the vast search spaces, the complexity of constraints, and the need for efficient problem-solving strategies.
2. **The Evolution of Solving Strategies**:
   * Mention traditional approaches like backtracking, forward checking, and heuristic-based search, underscoring their limitations.
   * Introduce the concept of algorithm selection as a strategic response to these challenges, emphasizing the difficulty of choosing the best algorithm for a given CSP instance.
3. **Machine Learning as a Game-Changer**:
   * Explain how machine learning offers a dynamic and adaptable approach to algorithm selection, significantly enhancing efficiency and reducing search costs.
   * Present the idea of leveraging ML-based algorithm selectors that learn from data to predict the most suitable solving strategy for diverse CSP instances.
4. **The Impact of ML on CSP**:
   * Illustrate the transformative potential of integrating ML in CSP solving, from theoretical advancements to practical applications.
   * Highlight a case study or research finding (e.g., from the provided materials) that demonstrates the superiority of ML-based selectors over traditional state-of-the-art met