

# Evolving New State-of-the-Art for Voronoi Diagram and Distance Transform From JFA Using AutoML

MACIEJ A. CZYZEWSKI, Poznan University of Technology, Poland  
KAMIL PIECHOWIAK, Poznan University of Technology, Poland

This paper studies a practical usage of machine learning (AutoML) to automate research towards discovering efficient Voronoi Diagram and Distance Transform algorithms. As the baseline we used the Jump Flooding Algorithm (JFA) - by finding new mutations which works best for specific data, and then ensembling them into one, we create new state-of-the-art algorithm in this field named **Vorotron** with best-case  $O(1)$  time complexity and  $O(N)$  work complexity - in addition, we introduce **JFAStar**, a single variant that in most cases outperforms original JFA with proven worst-case  $O(\log^* N)$  time complexity. The algorithm is faster and produces more accurate approximations. It could be extended into 3D space in a slice-by-slice manner. We started from the assumption that JFA has potential for improvement - some benefits can be observed for specific data by adding random noise and adjusting the step size in JFA. This showed us that, AutoML could examine this space, and find the best possible algorithm in each case. In the further part of the work, we discuss the results, compare the variants and ensemble for creating the final algorithm.

CCS Concepts: • **Computing methodologies** → **Computer graphics**; **Parallel computing methodologies**; • **Theory of computation** → *Randomness, geometry and discrete structures*; **Data structures design and analysis**.

Additional Key Words and Phrases: Voronoi Diagram, Distance Transform, Code Generation

## 1 INTRODUCTION

This paper<sup>1</sup> studies a practical usage of machine learning to automate research towards discovering efficient Distance Transform algorithms (utilizing technique known as AutoML). Thus, by finding mutations which works best for specific data, and then ensembling them into one, we create new state-of-the-art algorithm in this field named **Vorotron** with best-case  $O(1)$  time complexity and  $O(N)$  work complexity. In addition, we introduce **JFAStar**, a single variant that in most cases outperforms original JFA with proven worst-case  $O(\log^* N)$  time complexity.<sup>2</sup>

Notable contribution to the quick algorithm that makes Distance Transform (DT) using graphics hardware includes [5] that creates a cone for each input (point/seed) and renders those cones to obtain the Voronoi diagram as the lower envelope of these cones. [4] use planes tangent to a paraboloid and thus avoid the errors caused by the tessellation of the cones. Unfortunately, the drawback of this approach is the significant amount of computation and the implementation complexity.

Jump flooding algorithm (JFA)<sup>3</sup> is an interesting way to utilize the graphics processing unit to efficiently compute Voronoi diagrams and distance transforms [9]. This method is faster and produces more accurate results [10], and furthermore, it could be extended into 3D space in a slice-by-slice manner. This is more effective than the previous research carried out by [12], because the speed of JFA is almost independent to the number of seeds [10].

<sup>1</sup>the original title for this paper was “Lord Vorotron: Finding the Best JFA Variant for the Coming Winter”

<sup>2</sup>source code is available at [https://github.com/maciejczyzewski/fast\\_gpu\\_voronoi](https://github.com/maciejczyzewski/fast_gpu_voronoi)

<sup>3</sup>a novel pattern of communication

Based on this research and findings, several efficient GPU-based algorithms which are either work optimal or time optimal have been proposed including SKW [11], PBA [2], FastGPU [3], Honda's algorithm [6] and WTO [7]. The main question that needs to be addressed is whether JFA has potential for improvement. We found some benefits for specific data by adding random noise and adjusting the step size in JFA. Therefore, this shows that, AutoML could examine this unknown space, and find the best possible algorithm in each case.

For convenience, this work focus on the Voronoi diagram only - because this problem can be translated to DT [9]. The algorithm would be an approximation of the output, thus we suggest using WTO [7] for exact DT (EDT). The major contributions of this paper are thus:

**UWAGA:** napisac ze podczas real-time processingu, czyli np. w grach gdzie przejrze pomiedzy kolejnymi klatkami jest dosc podobne (kolejne przyklady maja male roznice w punktach) - mozna re-uzyc poprzedniego wyniku zamiast szumu i zredukowac step, wtedy ma sie duzo fps-ow?

- (1) Presenting new envolved state-of-the-art variants of algorithm for Voronoi Diagram and Distance Transform: **JFAStar** - Multi-domain single variant, replacement of JFA; **Vorotron** - Ensemble of domain-specific variants, production ready; and
- (2) Analyzing all possible variants of JFA: comparing error and speedup relative to bruteforce method in different domains.

## 2 RELATED WORK

Several efficient GPU-based algorithms which are either work optimal or time optimal have been proposed including JFA [9], SKW [11], PBA [2], FastGPU [3], Honda's algorithm [6] and WTO [7].

Reference	Algorithm	Exactness	Time	Work
[3]	FastGPU	Exact	$O(n^3/p)$	-
[2]	PBA	Exact	$O(n)$	$O(mN)$
[6]	based on SKW	Exact	$O(n)$	$O(N)$
[7]	WTO	<b>Exact</b>	$O(\log n)$	$O(N)$
[11]	SKW	Approximate	$O(n)$	$O(N)$
[9]	JFA	Approximate	$O(\log n)$	$O(N \log n)$
In this paper	JFAStar	<b>Approximate</b>	$\sim O(\log^* n)$	$\sim O(N \log^* n)$
	Vorotron	<b>Approximate</b>	$\sim O(1)$	$\sim O(N)$

Table 1. Different GPU algorithms for computing EDT

The original author of JFA<sup>4</sup> defined some variants and modifications [10], in this work we continue on this subject and examine the relationship between them. We chose this algorithm because it is the simplest to implement and it has a wide variety of improvements.

### 2.1 Jump Flooding

Algorithm for Voronoi Diagram that uses jump flooding as communication pattern propagates the information (2D coordinates and ID) of all sites to all the pixels in the matrix. It is based on the

<sup>4</sup>Guodong Rong

observation that while flooding an area with a seed, each seeded pixel can transmit its information, instead of just the ones on the boundaries of the Voronoi cell.

In each round, the site information stored in each pixel  $(x, y)$  is propagated to at most eight other pixels at  $(x + i, x + j)$  where  $i, j \in \{-k, 0, k\}$ , and  $k$  is the step length of the current round. In the first round, we use  $n/2$  as the initial step length to ensure that each pixel is reached by at least one site. The step length  $k$  is halved in each of the following round. All pixels are processed in parallel by the GPU. This ensures an exponential increase in the number of seeded pixels in a grid and thus, can be flooded in  $O(\log n)$  rounds, for an  $n \times n$  grid [13].

## 2.2 AutoML

Program synthesis is a class of regression problems where one seeks a solution, in the form of a source-code program, mapping the inputs to their corresponding outputs errorless [8]. If we have a broad range of acceptable solutions, another aspect is how efficient the produced code is.

There are two key ingredients to a synthesis problem: a domain specific language (DSL for short) and a specification. The DSL defines a space of candidate programs which serve as the model class. DSL-based models create different grammar rules for common code statements (e.g., control flow, comments, and brackets). The specification is commonly expressed as a set of input-output examples which the candidate program needs to fit exactly. In our problem, these examples are produced by brute-force algorithm.

Given the precise and combinatorial nature of synthesis, gradient-descent based approaches perform poorly and an explicit search over the solution space is required [1].

## 3 PROPOSED METHOD

In this section we outline the general approach that we follow in this work. When testing a JFA on different domains, we found some performance improvements by adding random noise and adjusting the step size between flooding rounds. In order to systematize our research, we adopted the following scheme:

- Developing manually new modifications that can be made to the JFA (such as using a circle instead of interacting with neighbours in a square pattern) and designing them as parameters.
- Heuristic search algorithm<sup>5</sup> - to examine enormous space of possible variants:
  - Implements a new variant from a combination of several modifications (mutation).
  - Tests current variant on various domains and scores the results to update gradient descent.
- Knowing the best variant in a specific domain, we can use it in ensemble (when entering the input, ensemble algorithm will decide which algorithm will solve it).

### 3.1 Domain Space

The Voronoi Diagram has a stronger relationship with the initial density of input seeds than the matrix shape. Intuitively, if we have a dense matrix, there is no need to deliver information about a particular seed to every pixel of the matrix, only to its surrounding neighbours. Hence, less passes of information can be performed, therefore the algorithm can be terminated earlier.

<sup>5</sup>Bayesian optimization using Gaussian Processes.

<div>Density \ Shape</div>	Small (32-128×)	Medium (256-448×)	Large (512-1536×)
Low ( $\rho=0.00005-0.001$ )	Bruteforce	JFA	JFA
High ( $\rho=0.01-0.1$ )	JFA	JFA	JFA

Table 2. State-of-the-art for specific domains (before our work)

In order to find out which variants are domain-specific, we have to divide the domain into subdomains as in the Table 2. Unfortunately, we can already identify that the jump flooding algorithm would not be efficient in cases with low density of seeds. If there are less than 8 seeds to iterate<sup>6</sup>, the bruteforce algorithm is ideal for use.

### 3.2 Search Space

In order to have a consistent loss landscape, search space was divided into subspaces containing only those parameters where each pair is strongly dependent on each other.

**UWAGA:** opisac ze szukamy 4 podprzestrzenie w turach! (moze tez o nowym wzorku który szuka tylko prawidłowych bridge z score function gdziekolwiek to bedzie obliczyc ile jest mniej wiecej wersji algosow (bez special?))

jakie modyfikacje, na to osobna sekcja? wiec co tu napisac chyba tylko o zlozonosci problemu i ze kod jest skladany i testowany a niektore wersje sa pomijane (Bayesian optimization using Gaussian Processes).

### 3.3 Score Function

JEST PROBLEM - zdazaja sie warianty ktore 0zeruja rzadkie i sa bardzo szybkie na duzy matrycach - trzeba do ponownie zbalansowac roznicza w pikselach pomiedzy bruteforce a algorytmem - napisac o tym / tez ze to wszystko to ilorazy do bruteforce napisac jak score jest liczony w dziedzinie - srednia geom? czy cos

dla voronoi-a interesuja nas 2 parametry Error oraz Szybkosc, aby wyniki byly wiarygodne porownujemy je z bruteforcem (a wiec bedzie to iloraz). aby ocenic dana mutacje musimy przypisac jakis Score danej wersji, wiec uzylismy wzor ponizej

$$S(x, y) = \max\{0, \sqrt{x} \cdot (100 - y^2)\}, \tag{1}$$

$$0 \leq y \leq 100, 0 < x \tag{2}$$

ktory kaze za zbyt wysokie errors, dajac zerowy wynik - skladnik przy y rosnie szybciej niz x wiec gdy przekroczy 100 da nam ujemny wynik - czyli 0.

### 3.4 Optimizer

opisac dwie osobne taktyki optymalizacji dla best single vs. ensemble

mozemy napisac ze korzystalismy z forest/gp minimize, ale tez wspomniec ze aby miec najlepszy best single to trzeba bylo optymalizowac rownoczesnie cala przestrzen (od malych do duzych, gestych po rzadkie), a zeby miec najlepszego Vorotrona - czyli ensemble to trzeba bylo dla kazdej domeny z optymalizowac a pozniej jedynie zrobic balancera!!!!!!!!!!!!!!

<sup>6</sup>each thread in original JFA needs 8 iterations to communicate with neighbours

trzeba to przeszukiwać tak aby nie sfiksował na zadanych parametrach - bo niechcący ocenia na początku ze szum/dual jest nie fajny i później go już nie rozważa

### 3.5 Ensemble

można zapisać tabele dla każdej domeny (shape) / i zrobić przewidywania parametrów (słownik wariantu) - bo dla małych opłacalne są Circle6 a dla większych Circle12 i tak dalej - jak to zrobić?

patrzeć na rezultaty możemy znaleźć jaki algorytm najlepiej sprawdza się w zadanej domenie. np. widac że dla małej ilości seedów (mało gęstych przypadków, które mają małą powierzchnię) opłaca się użyć brute force. Dla kolejnych większych przypadków innych wariantów JFA. Jak wybrać algorytm? Każdy przypadek ma 'shape' oraz 'num' więc można na CPU wyselektować parę punktów albo od razu obliczyć gęstość i wybrać odpowiedni algorytm. To takich ensemble'acji najlepiej sprawdzi się drzewo decyzyjne (może być boostowane).

## 4 VARIANTS

ZROBIC ŁADNE RYSUNEKZKI w Google Slides - eksport to pdf!

In this paper, we present additional modifications that can be applied to create new JFA variants. Some modifications result from an alternative approach from previous works [9] (anchor<sup>7</sup> type), [10] (propagation/step function), [13] (anchor number), [4] (anchor double).

informacje można wstępnie rozpropagować losowo - w nadziei że pozwoli nam skończyć algorytm w mniejszej ilości kroków.

Aby badania były bardziej przejrzyste trzymaliśmy się pewnej konwencji nazewnictwa:

[*anchor\_type*][*anchor\_num*][*anchor\_double*] - [*step\_function*] + [*noise*]

dla przykładu Circle11(1/3)Dual(1/4)-Factor3+Noise które można przeczytać jako:

[*anchor\_type*] = *Circle*,

[*anchor\_num*] = 11,

[*anchor\_number\_ratio*] = 1/3,

[*anchor\_double*] = *True*,

[*anchor\_distance\_ratio*] = 1/4,

[*step\_function*] = *Factor3*,

[*noise*] = *True*

wariant gdzie step jest mnożony razy error z metric? w kernelu

### 4.1 Noise

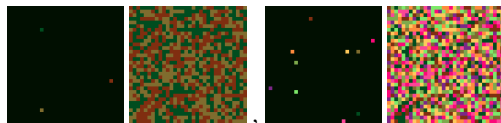


Fig. 1. Noise for 32 × 32

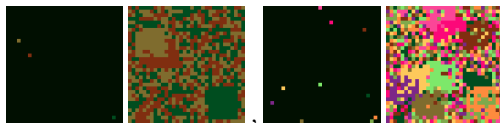


Fig. 2. Local Noise for 32 × 32

FIXME: figure z przykładami szumu (+local) i jak to wygląda i jak wyglądała instancja!

<sup>7</sup>anchor is a method of communication between neighbors

Zamiast zaczynac od pustej macierzy z seed-ami początkowymi można ją losowo uzupełnić szumem - tworząc przypadkowe short-cuty. Można tego dokonać osobnym kernelem który zostanie wywołany przed wykonaniem głównej części algorytmu. Interpretacja jest taka że pewne rejony która w JFA są wypełnione zerami podczas pierwszych iteracji nie podejmują żadnych decyzji. Uzupełniając szumem mogą one przypadkowo ustawić się na prawidłową wartość i propagować w kolejnej rundzie najlepszą wartość w swoim otoczeniu (zgodnie ze stepem).

**dowód kamila tutaj????????????????**

#### 4.1.1 Local Noise. **przesunięta ciężkości?? czyli średnia z wagami poprawiła? a może uzależnić wagę od rozmiaru pierwszego stepu?**

Można też szum uzupełniać nie losowo tylko w otoczeniu. Więc gdy w punkcie  $(x, y)$  wylosujemy losowego seed-a o wartości  $(x_{rand}, y_{rand})$  to wyliczamy nową pozycję  $(x', y')$  która znajduje się w połowie drogi w następujący sposób:  $x' = \frac{x + 3x_{rand}}{4}$ , analogicznie dla  $y'$ . Dodatkowo jeśli  $(x', y')$  jest pusta to też uzupełniamy to pole tą informacją. Nie przejmujemy się wyciągiem w dostępie do danych. Napięcia będą losowe - a szum też.

## 4.2 Anchor Type

### **losowane punkty na okręgu??**

Zamiast pobierać informacje od 8 sąsiadów o step size from grid points at  $(x + i, y + j)$  where  $i, j \in \{-step, 0, step\}$ . Można zastosować okrąg - otwiera nam to nowe możliwości na swobodną modyfikację ilości punktów od których będziemy pobierać informacje. Naturalnie wydaje się że mała ilość punktów w anchorze spowoduje wzrost błędu, a duża ilość punktów spowoduje zmniejszenie błędu.

4.2.1 Anchor Number. Dlatego kolejnym parametrem będzie możliwość kontrolowania ilości punktów. Niestety nie rozważaliśmy wariantu kwadratów o dowolnej ilości punktów (ponieważ byłyby to wielokrotności  $2 \times 2 = 4$ ,  $3 \times 3 = 9$ ,  $4 \times 4 = 16$ ,  $5 \times 5 = 25$ ) bo i tak nie dało by się wybrać uniwersalnej tej wartości. Dla okręgu punkty sąsiadujące  $(x_i, y_i)$  były liczone następująco:

$$x_i = x + step \cdot \cos\left(\frac{2\pi}{[anchor\_num]} \cdot i\right), y_i = y + step \cdot \sin\left(\frac{2\pi}{[anchor\_num]} \cdot i\right)$$

## 4.3 Anchor Double

Oprócz pojedynczego anchora, możliwe jest użycie podwójnej warstwy anchorów (czyli np. małe kolko i większe). Idea za tym stojąca to że małe kolko wewnętrzne jest dokładne (działa jak w JFA) - a wielkie zewnętrzne jest skautujące lub aby poprawić error wynikający np. z mniejszej ilości anchor\_num (w sumie to podobny mechanizm jak w Lookahead - wolny/szybki)

4.3.1 Anchor Distance Ratio. Parametr mówi o stosunku długości step size od wewnętrznego anchora do zewnętrznego.

4.3.2 Anchor Number Ratio. Parametr mówi o statusku ilości detektorów od wewnętrznego anchora do zewnętrznego.

## 4.4 Step Function

Gdy nasza informacja propaguje się szybciej lub jest bardziej zagęszczona dlatego sąsiedzi szybciej dostają prawidłową informację - to oznacza że można skrócić ilość round wykonania algorytmu.

Step size jak i ich ilość można określić za pomocą 2 podstawowych parametrów: shape and number of points - z których później możemy określić np. średnią gęstość. Zaimplementowaliśmy 2 warianty które są uzależnione jedynie od shape: defaultowy z JFA, z JFA o podstawie 3; oraz jeden

uzaleniony od shape oraz od num: logstar. Jednak aby wygeneralizowac problem stworzyliśmy też możliwość wygenerowania dowolnego polinomialu.

4.4.1 *Special Polynomial*. powinno być ograniczone do 3 PARAMETROW! wymyślisz nową funkcję problem z Special - on overfituje przykłady zmieniając 5 miejsc po przecinku aby 2 zamieniało się np. w 1

Implementacja nie jest ważna - chodzi o ideę związania shape oraz num. Oraz modyfikowanie wartości, szybkości spadku, kształtu (np. pilokształtne) - jakimiś parametrami. Wada tego rozwiązania jest że trzeba optymalizować tę funkcję na całej dziedzinie (małej, dużej, gestej, zadkiej) - bo inaczej z overfitu ona ilość kroków i wielkość stepu pod rozmiar.

---

```
def mod_step_function__special(shape, num=None, config=None):
    # [EXAMPLE]
    # Special(1.51/0.92/0.92/1.08/0.42)
    # ----- A -- B -- C -- D -- X ---

    A = config["A"] # <1, 2>
    B = config["B"] # <0, 1>
    C = config["C"] # <0, 1>
    D = config["D"] # <1, 2>
    X = config["X"] # <0.2, 1>

    q = num / (shape[0] * shape[1])
    qm = ((shape[0] + shape[1]) / 2) * q**(1 / 2)
    S = B * qm + (1 - B) * (max(shape) / 2)
    St = math.log2(S)

    steps = []
    for i in range(1, int(X * St * 2), 1):
        f = round(1 / (D**(i**A) + i % max(1, int(C * St))), 4)
        ffm = int(f * S)
        if ffm >= 1:
            steps.append(ffm)
    if len(steps) == 0:
        return [1]

    return steps
```

---

## 5 RESULTS

przenieść legendę? JAKO OSOBNY PDF? i podać w tej sekcji - tak się nie robi ale było by ok i czytelnie + więcej miejsca na wykresy a przypadków będzie więcej czy wykres loss oraz score dla przypadków powinny być nałożone? albo połączone subfigurem tak aby osie były sync. i dało się porównać

performance plot<sup>8</sup>

**UWAGA:** usunąć z tabelki PODOBNE ALGORYTMY i ich słabsze rezultaty

---

<sup>8</sup>wykres został zrobiony poprzez posortowanie score'ów - dzięki temu widac różnice w przyroście i łatwo dostrzec który algorytm ma najwyższy score lub jaka ma charakterystykę (np. jest bardzo skuteczny dla wąskiej grupy przykładów)



<div>Density \ Shape</div>	Small (32-128×)	Medium (256-448×)	Large (512-1536×)
Low ( $\rho=0.00005-0.001$ )	Bruteforce	?	?
High ( $\rho=0.01-0.1$ )	?	?	?

Table 3. Found in this paper state-of-the-art for specific domains

5.1 Multi-domain Variant (JFAStar)

- **shapes:** {32x32, 64x64, 96x96, 128x128, 256x256, 320x320, 384x384, 448x448, 512x512, 768x768, 1024x1024, 1536x1536}
- **cases:**
  - gen\_uniform: seeds=1,
  - gen\_uniform: seeds=3,
  - gen\_uniform: density=0.0001,
  - gen\_uniform: density=0.001,
  - gen\_uniform: density=0.01,
  - gen\_uniform: density=0.02,
  - gen\_uniform: density=0.03,
  - gen\_uniform: density=0.04,
  - gen\_uniform: density=0.05,
  - gen\_uniform: density=0.1,





Fig. 3. bla bla bla

Algorithm	$\rho=0.0003$	$\rho=0.0002$	$\rho=0.0003$	$\rho=0.0004$	$\rho=0.0008$	Score
Square Special(1.92/0.9/0.68/1.2/0.67)+Noise	40.4	51.1	74.8	95.1	<b>113.6</b>	74
Square Star+Noise	44.5	50.5	71.2	87.9	<b>115.8</b>	73
Square Special(1.98/0.8/0.95/1.2/0.46)+Noise	40.8	51.5	74.7	95.5	<b>102.6</b>	72
Square Special(1.47/0.68/0.75/1.52/0.41)+Noise	41	51.9	75.5	96.7	98.3	69
Square Special(1.46/0.81/0.68/1.33/0.5)+Noise	39.6	50.9	73	88.1	<b>112.1</b>	69
Square Special(1.98/0.69/0.74/1.22/0.49)+Noise	37.4	49.5	71.2	91.7	<b>102.6</b>	66
Square Special(1.14/0.8/0.61/1.38/0.62)+Noise	34.6	44.9	64.7	80.4	<b>108.7</b>	66
SquareDual(1/3) Special(1.56/0.88/0.57/1.64/0.39)+Noise	39.1	48.5	67.5	85.7	98.2	65
Square Special(1.97/0.79/0.61/1.24/0.61)+Noise	37.9	49.5	72.8	93.4	88.2	64
Square Special(1.92/0.55/0.61/1.1/0.53)+Noise	33.7	41.8	60.9	77.5	<b>105.4</b>	63
Square Special(1.83/0.69/0.81/1.24/0.45)+Noise	38.3	49.5	72.5	92.7	<b>102.5</b>	62
Square Factor3+Noise	31	39.4	58.9	76.2	<b>105.3</b>	61
Square Special(1.84/0.65/0.84/1.26/0.52)+Noise	37.2	48.8	70.3	89.7	<b>101.9</b>	60
Square Special(1.8/0.65/0.58/1.27/0.59)+Noise	36.9	46	67.5	86.8	98.0	59
Square Special(1.94/0.64/0.8/1.28/0.4)+Noise	40.1	52	77.7	96.3	88.2	58
Square Special(1.82/0.71/0.73/1.2/0.42)+Noise	40.4	51.5	75.1	94.4	82.5	58
Square Special(1.33/0.5/0.7/1.26/0.46)+Noise	35	46.1	65.8	80.9	81.3	57
SquareDual(1/3) Special(1.71/0.06/0.95/1.46/0.34)+Noise	32	38	54.2	68.8	92.2	56
Square Special(1.93/0.64/0.92/1.21/0.98)+Noise	34.6	43.9	65.6	85.4	96.7	56
JFA (original)	30.2	37.2	54.2	69.2	94.4	56
SquareDual(1/3) Special(1.51/0.46/0.97/1.88/0.57)+Noise	31.7	39.1	56.7	72.8	83.4	55
SquareDual(3/4) Star+Noise	34.8	38.6	52.9	65.7	84.8	54
SquareDual(1/3) Star+Noise	35.4	38	52.5	65.1	83.5	54
SquareDual Star+Noise	34.6	38	52	65.2	83.3	54
SquareDual(1/3) Special(1.2/0.9/0.84/1.96/0.44)+Noise	38.6	47.2	65.7	73.5	95.8	53
Square Default+Noise	26.6	33.6	49.9	64.1	88.7	52
Circle10(1/4) Special(1.58/0.9/0.64/1.16/0.6)+Noise	29.3	34.7	46.8	57.1	72.9	47
SquareDual(1/3) Special(1.45/0.85/0.74/1.95/0.34)+LNoise	40.2	44.2	45.2	54.6	56.7	45
Circle11(1/3) Star+Noise	30	32.5	44.5	51.7	67.4	44
SquareDual Special(1.48/0.39/0.84/1.34/0.72)+Noise	24	29.9	42.8	54.7	74.6	44
Circle11(3/4)Dual(2/3) Factor3+Noise	24.7	30	43.1	56.1	69.9	44
Circle9(1/4) Star+Noise	31.8	35.1	47.6	49.9	61.9	44
Circle9(3/4) Star+Noise	31.9	34.6	47.1	50.8	61.8	44
Circle12(1/3) Star+Noise	28.6	32	42.7	51.2	65.5	43
Circle14(1/4) Special(1.38/0.71/0.93/1.77/0.95)+Noise	24.3	29.4	41.5	51.5	62.2	41
Square Special(1.56/0.7/0.63/1.29/0.43)+Noise	40.9	51.6	70.9	90.9	103.2	40
SquareDual(3/4) Default	22.5	27.6	39.1	49.7	66.9	40
SquareDual Default	22.6	27.3	38.8	49.2	65.7	40

442  
443  
444  
445  
446  
447  
448  
449  
450  
451  
452  
453  
454  
455  
456  
457  
458  
459  
460  
461  
462  
463  
464  
465  
466  
467  
468  
469  
470  
471  
472  
473  
474  
475  
476  
477  
478  
479  
480  
481  
482  
483  
484  
485  
486  
487  
488  
489  
490

## 5.2 Domain-specific Variants

5.2.1 *Small Shape*:  $32 \times 32$ ,  $64 \times 64$ ,  $96 \times 96$ ,  $128 \times 128$ . **Square-Special(1.44/0.96/0.17/1.63/0.86)+Noise**  
and **Square-Special(1.07/0.24/0.9/1.88/0.64)**

--- (62) Square—Special(1.44/0.96/0.17/1.63/0.86)+Noise  
 --- (61) Square—Special(1.32/0.96/0.28/1.68/0.76)+Noise  
 --- (61) Square—Special(1.21/0.96/0.3/1.23/0.8)+Noise  
 --- (60) Square—Special(1.17/0.95/0.55/2.0/0.89)+Noise  
 --- (60) Square—Special(1.61/0.92/0.37/1.74/0.75)+Noise  
 --- (60) Square—Special(1.15/0.94/0.02/1.93/0.75)+Noise  
 --- (60) Square—Special(1.24/0.94/0.36/1.93/0.78)+Noise  
 --- (60) Square—Special(1.34/0.9/0.67/1.78/0.97)+Noise  
 --- (60) Square—Special(1.3/0.92/0.57/1.89/0.72)+Noise  
 --- (60) Square—Special(1.39/0.91/0.46/1.98/0.88)+Noise  
 --- (60) Square—Special(1.69/0.93/0.69/1.64/0.87)+Noise  
 --- (60) Square—Special(1.4/0.96/0.06/1.72/0.88)+Noise  
 --- (44) JFA (original)  
 --- bruteforce

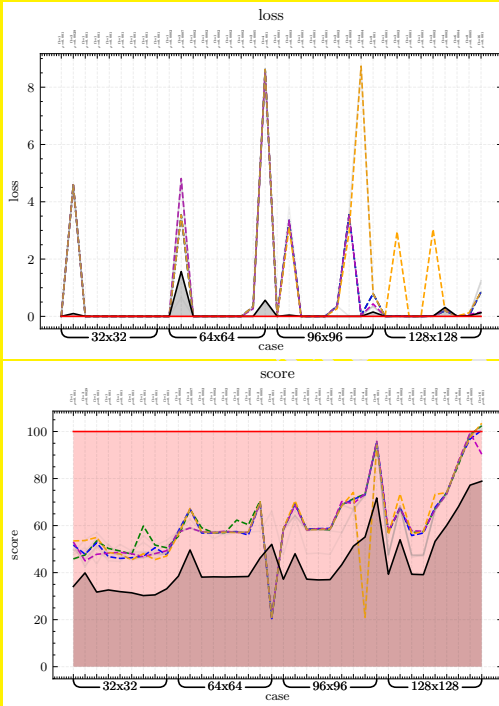


Fig. 4. Low Density

--- (128) Square—Special(1.07/0.24/0.9/1.88/0.64)  
 --- (120) Square—Star  
 --- (112) Square—Special(1.15/0.08/0.13/1.61/1.0)  
 --- (112) Square—Star+Noise  
 --- (111) Square—Factor3+Noise  
 --- (110) Square—Special(1.26/0.4/0.74/1.42/0.76)  
 --- (104) Square—Special(1.02/0.71/0.94/1.55/0.78)+Noise  
 --- (102) Square—Special(1.1/0.76/0.54/1.46/0.85)+Noise  
 --- (97) Square—Special(1.1/0.48/0.04/1.55/0.82)+Noise  
 --- (97) Square—Default+Noise  
 --- (96) Square—Factor3  
 --- (76) SquareDual(1/3)—Special(1.05/0.87/0.37/1.92/0.99)+Noise  
 --- (111) JFA (original)  
 --- bruteforce

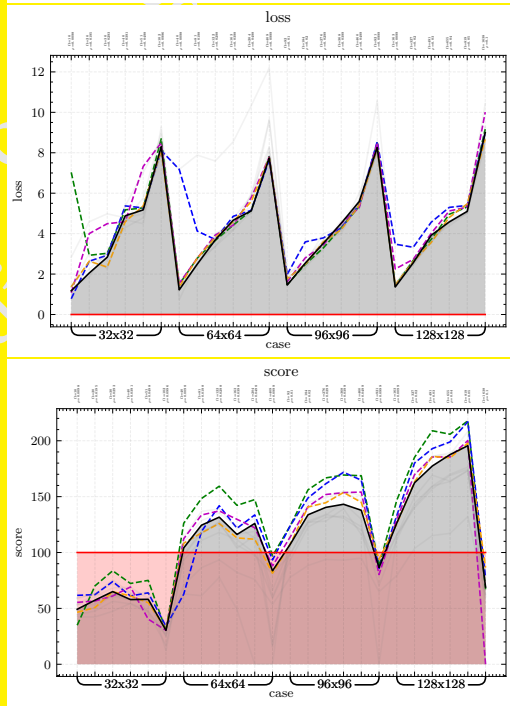


Fig. 5. High Density

5.2.2 *Medium Shape*:  $256 \times 256$ ,  $320 \times 320$ ,  $384 \times 384$ ,  $448 \times 448$ .

5.2.3 *Large Shape*:  $512 \times 512$ ,  $768 \times 768$ ,  $1024 \times 1024$ ,  $1536 \times 1536$ .

5.2.4 *Low Density*.

5.2.5 *High Density*.

5.3 Ensemble of Domain-specific (VoroTron)

bla bla

5.4 Objectives

naprawic generowanie tego wykresu napisac co nie moze byc uzyte z czym?  
czyli co ma wplyw na co (w sumie to najwazniejsze mialo byc w pracy)

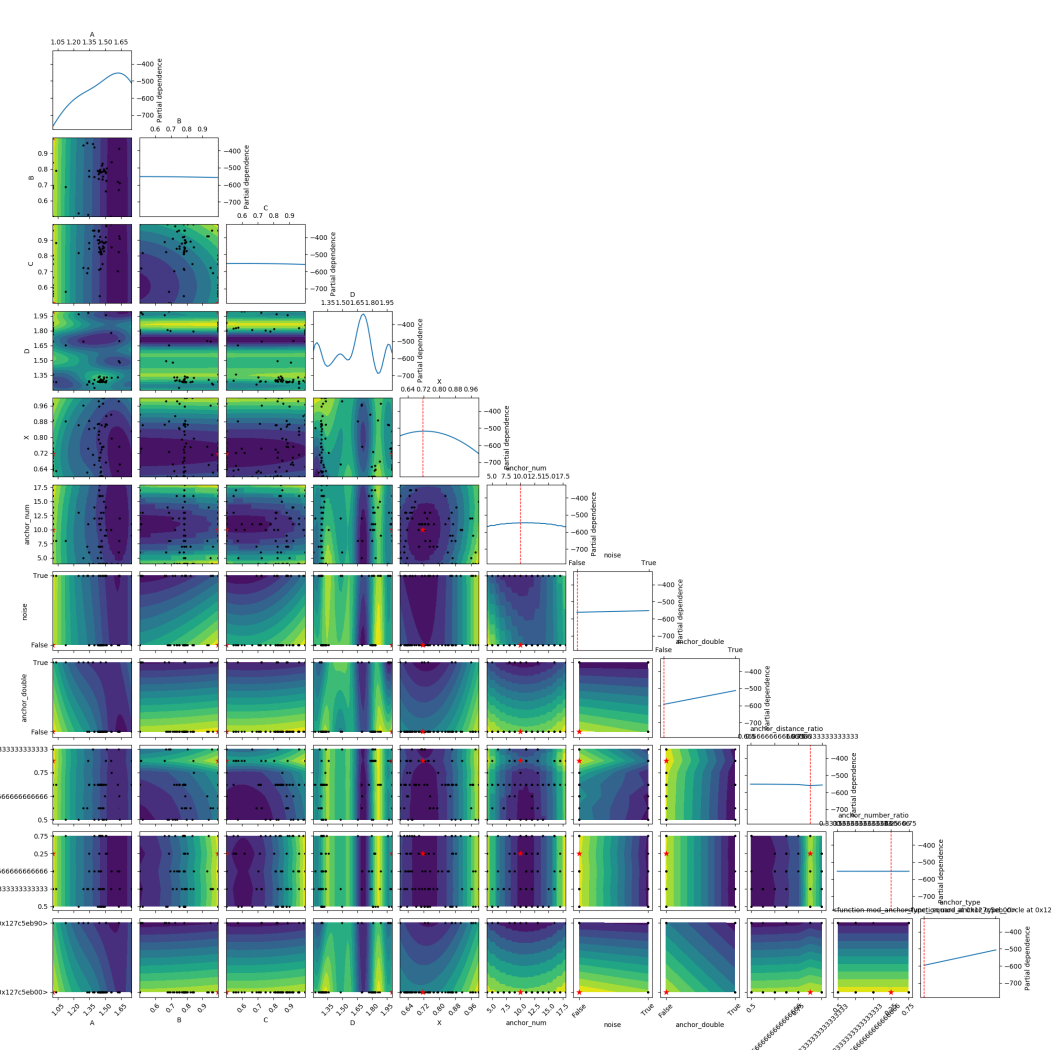


Fig. 6. bla bla bla

6 PRACTICAL USAGE

polaczyc z Conclusions

Jest wiele projektow ktore potrzebuje DT lub voronoi-a. Jedyne dwa praktyczne przyklady z tej pracy to SOTA dla JFA - czyli JFAstar, oraz praktyczny Ensemble (uwzgledniajacy np. bruteforce dla malych instancji).

## 8 ACKNOWLEDGEMENTS

## REFERENCES

- 2020-09-22 02:12. Page 13 of 1-13.