Różne podejścia w rozwiązywaniu problemu spełnialności formy 3 SAT z naciskiem na wykorzystanie algorytmów genetycznych.

1. Wstęp

Niniejsza praca ma na celu analizę niektórych algorytmów, w szczególności algorytmów genetycznych takich jak algorytm ewolucyjny i PSO w kontekście rozwiązywania problemu spełnialności 3SAT formy 3CNF.

Problem spełnialności to zagadnienie z dziedziny rachunku zdań polegające na określeniu czy dla danej formuły logicznej istnieje takie podstawienie zmiennych zdaniowych, że cała formuła również jest prawdziwa.

Forma 3CNF (Conjunctive Normal Form – Koniunkcyjna Postać Normalna) jest szczególnym przykładem, gdzie formuła logiczna jest koniunkcją pewnej liczby zmiennych zdaniowych składających się z maksymalnie 3 zmiennych (literałów) połączonych znakiem alternatywy.

Na przykład: $\varphi = (x_1 \vee x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee x_4)$

$\wedge (x_2 \vee \neg x_3 \vee \neg x_4) \wedge (\neg x_1 \vee \neg x_2 \vee x_5)$

$\wedge (x_1 \vee \neg x_3 \vee x_5) \quad \wedge (\neg x_3 \vee x_4 \vee \neg x_5)$

Problem 3SAT należy do zbioru problemów NP-kompletnych i zbioru 21 NP-kompletnych problemów Karpa i jest używany w dowodzeniu, NP-kompletność innych problemów.

Na przykład można wykazać, że problem zbioru niezależnego jest NP-kompletny poprzez wskazanie, że 3SAT jest wielomianowo redukowalny do tego problemu. Inne zależności i zastosowania 3SAT również wskazują na wysokie znaczenie optymalizacji rozwiązań tego problemu, ale ich głębsza analiza nie jest przedmiotem tej pracy.

W tej pracy wykorzystany został język programowania Python z użyciem bibliotek PySwarms i PyGad.

2. Opis danych używanych do analizy algorytmów.

Przykładowe formuły formy 3CNF zostały pobrane ze strony internetowej:
https://people.sc.fsu.edu/~jburkardt/data/cnf/cnf.html

W niektórych miejscach używana jest przeze mnie funkcja convert_to_boolean, która modyfikuje dane wejściowe w taki sposób, że z przykładowej listy zdań:

[[-9, 40, -68], [11, 14, -82], [-55, -56, -80]]

Tworzy ona dane w takiej formie:

[{(9, False), (40, True), (68, False)}, {(11, True), (82, False), (14, True)}, {(80, False), (56, False), (55, False)}]

Zamieniamy tutaj wartości ujemne na wartość False (negację), a wartości dodatnie na zdanie bez negacji (True).

3. Algorytm brute-force

Dobrym punktem wejściowym w analizowaniu skuteczności różnych algorytmów jest zawsze rozpoczęcie od algorytmu brute-force, co umożliwia porównywanie do niego innych algorytmów.

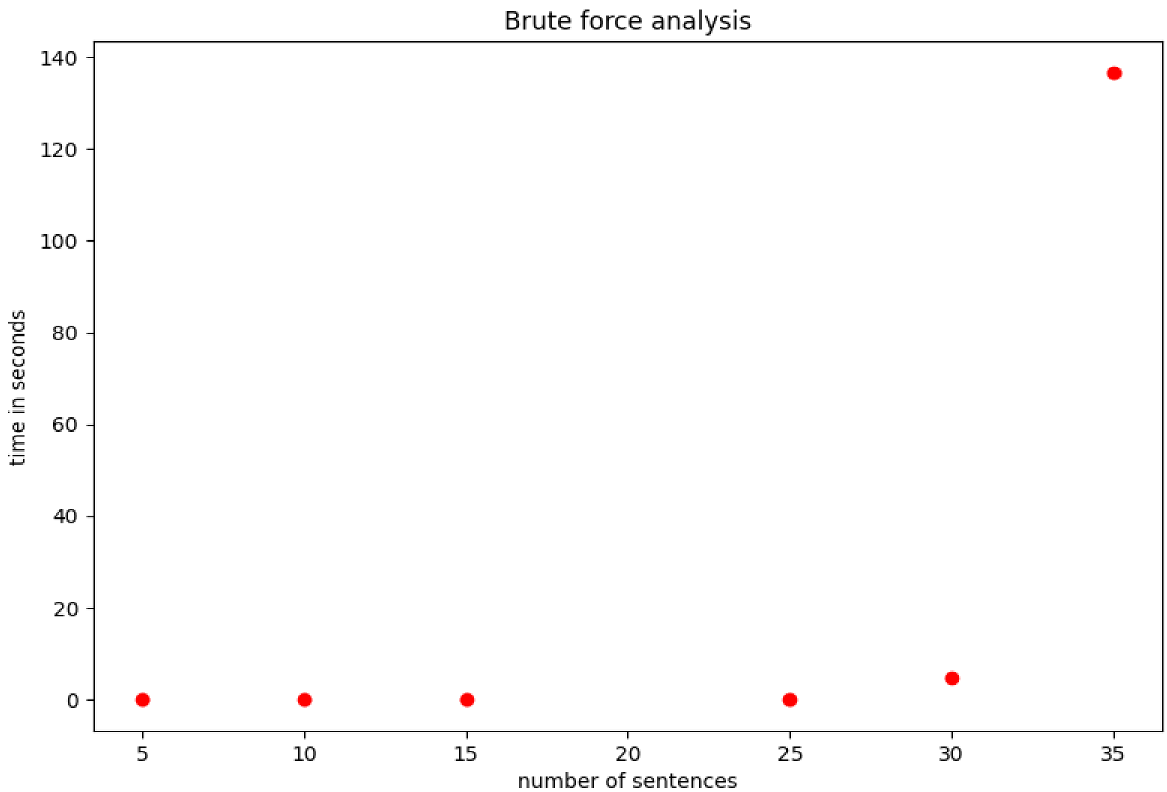
Algorytm polega na sprawdzeniu wszystkich możlwych wartości zdań składowych i sprawdzenie, czy któreś z podstawień zwraca wartość logiczną True dla całej koniunkcji.

Kod:

```
def brute_force(cnf):
    cnf = convert_to_boolean(cnf)
    literals = set()
    for conj in cnf:
        for disj in conj:
            literals.add(disj[0])

    literals = list(literals)
    n = len(literals)
    for seq in itertools.product([True, False], repeat=n):
        a = set(zip(literals, seq))
        if all([bool(disj.intersection(a)) for disj in cnf]):
            return True, a
    return False, None
```

Wykres:



Ilość zdań/literałów	5/15	10/29	15/41	25/57	30/62	35/68
Czas w sek. (średnia z 10 prób)	0.0018	0.0029	0.0037	0.0183	4.8461	136.6063

Jak widać złożoność czasowa jest funkcją wykładniczą, która rośnie wyjątkowo szybko wraz z wzrostem ilości zdań składowych. Big O Notation ma wzór - O(m^{2n}) gdzie m oznacza ilość zdań, a n ilość literałów. W związku z tym, już przy bardzo ograniczonej ilości zdań algorytm ten potrzebuje dużo czasu na znalezienie rozwiązania.

3. Algorytm DPLL

Jest to algorytm Davisa-Putnama-Logemanna-Lovelanda wykorzystujący podejście rekursywne do rozwiązania problemy 3SAT.

Przypadkami bazowymi są pusta koniunkcja i koniunkcja zawierająca pustą alternatywę.

Zaczynamy od dowolnie wybranego literału i ustawiamy jego wartość jako True.

Dzięki temu możemy wyłączyć z koniunkcji wszystkie zdania składowe zawierające ten literał, a także usunąć z kolejnych obliczeń wszystkie negacje tego literału, gdyż nie będzie od nich zależała prawdziwość innych zdań składowych.

Kod:

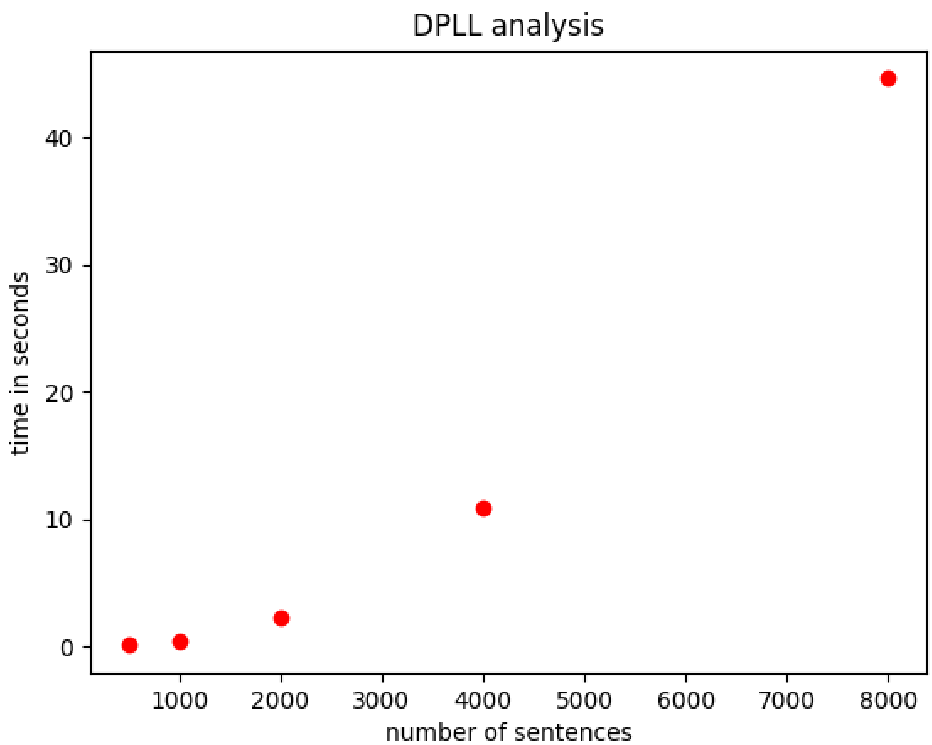
```
def select_literal(cnf):
    for c in cnf:
        for literal in c:
            return literal[0]
def dpll(cnf, assignments={}):
    if len(cnf) == 0:
        return True, assignments
    if any([len(c) == 0 for c in cnf]):
        return False, None
    l = select_literal(cnf)
    new_cnf = [c for c in cnf if (l, True) not in c]
    new_cnf = [c.difference({(l, False)}) for c in new_cnf]
    sat, vals = dpll(new_cnf, {**assignments, **{l: True}})
    if sat:
        return sat, vals
    new_cnf = [c for c in cnf if (l, False) not in c]
    new_cnf = [c.difference({(l, True)}) for c in new_cnf]
    sat, vals = dpll(new_cnf, {**assignments, **{l: False}})
    if sat:
        return sat, vals
    return False, None
```

Tutaj dostępne zdania logiczne formuły 3CNF okazały się zbyt proste dla tego algorytmu, więc używam do tego funkcji, która generuje takie zdania z dowolnie wybraną ilością klauzul i literałów.

Kod:

```
def tcnfgen(m, k, horn=2):
    cnf = []
    def unique(l, k):
        t = random.randint(1, k)
        while(t in l):
            t = random.randint(1, k)
        return t
    r = random.randint(0, 1)
    for _ in range(m):
        x = unique([], k)
        y = unique([x], k)
        z = unique([x, y], k)
        if horn:
            cnf.append([(x, 1), (y, 0), (z, 0)])
        else:
            cnf.append([(x, r), (y, r()), (z, r())])
    return cnf
```

Wykres:



Ilość zdań/literałów	500/250	1000/500	2000/1000	4000/2000	8000/4000
Czas w sekundach	0.1096	0.4191	2.2547	10.8698	44.5994

Złożoność czasowa również jest wykładnicza, jednak w tym przypadku Big O Notation jest postaci

$O(2^n)$, gdzie n znowu oznacza liczbę literałów. Jak widać ten algorytm jest nieporównywalnie skuteczniejszy w rozwiązywaniu problemu 3SAT.

4. Algorytmy ewolucyjne

Pierwszym podejściem do próby rozwiązania problemu 3SAT z użyciem algorytmów genetycznych jest użycie algorytmu ewolucyjnego. W takiej sytuacji najważniejsze jest określenie funkcji fitness. W tej pracy zostaną zaprezentowane dwie jej wersje.

Wersja 1.

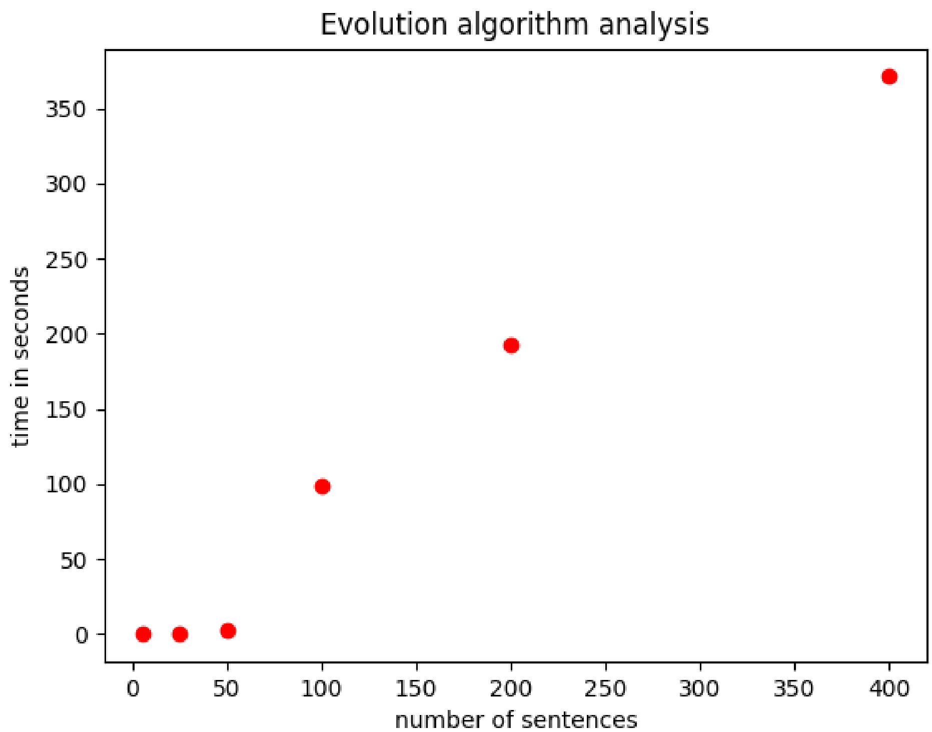
Kod:

```
def fitness(cnf, solution):
    cnfFlat = flatten_and_slice(cnf)
    hashMap = {}
    fitness = 0
    for x in range(0, len(solution), 3):
        if 1 not in solution[x:x+3]:
            fitness  -= 3
    for x in range(len(solution)):
        literal = cnfFlat[x]
        if -literal in hashMap.keys():
            if solution[x] == hashMap[-literal]:
                fitness -= 1
        elif literal in hashMap.keys():
            if solution[x] != hashMap[literal]:
                fitness -= 1
        else:
            hashMap[literal] = solution[x]
    return fitness
```

Funkcja ta przyjmuje listę 0 i 1 obrazujących wartości logiczne wszystkich literałów po kolei, umożliwiając nadanie różnych wartości tym samym literałów. Zaczynamy od wartości fitness równej 0 i karzemy dane rozwiązanie poprzez odejmowanie od tej wartości, gdy:

- W zdaniu składowym nie ma żadnego literału o wartości logicznej True
- Przypisana wartość logiczna do pewnego literału nie zgadza się z wcześniej ustaloną wartością
- Przypisana wartość logiczna do pewnego literału jest taka sama jak wcześniej ustalona jego negacja

Wykres:



Ilość zdań/literałów	5/15	25/57	50/81	100/96	200/100	400/100
Czas w sek. (średnia z 10 prób)	0. 0349	0. 3663	3.1462	99.0962	193.042	371.3892

Jak widać wyniki w porównaniu do algorytmu brute force są raczej zadowalające, jednak nie są one nawet blisko skuteczności algorytmu DPLL. Ponadto trzeba zaznaczyć, że do 100 zdań funkcja zwracała prawidłowy rezultat, w przedziale od 100 do 200 zdań nie można już było zagwarantować prawidłowej odpowiedzi za każdym razem, jednak były one dosyć zbliżone. Powyżej 200 zdań niestety nie można już używać tej funkcji jako wiarygodnego sposobu rozwiązywania problemu 3SAT. Funkcja zatrzymywała się przy znalezieniu rozwiązania, co umożliwiło bardziej wiarygodne mierzenie czasu jej działania.

W tym przypadku obliczenie złożoności czasowej jest znacznie bardziej skomplikowane, ale również możemy założyć wykładniczą naturę funkcji ilości zdań składowych od czasu, która jednak rośnie zdecydowanie wolniej od algorytmu brute force.

Wersja 2.

Kod:

```
def fitness_v2(cnf, solution):
        cnfS = convert_to_boolean(cnf)
        hashMap = {}
        fitness = -(len(cnfS))
        cnf = flatten_and_slice(cnfS)
        literals = []
        for x in cnf:
            if x[0] not in literals:
                literals.append(x[0])
        for index in range(len(solution)):
            hashMap[literals[index]] = solution[index]
        for k in cnfS:
            for l in k:
                if l[1] == False and hashMap[l[0]] == 0:
                    fitness += 1
                    break
                elif l[1] == True and hashMap[l[0]] == 1:
                    fitness += 1
                    break
        return fitness
```

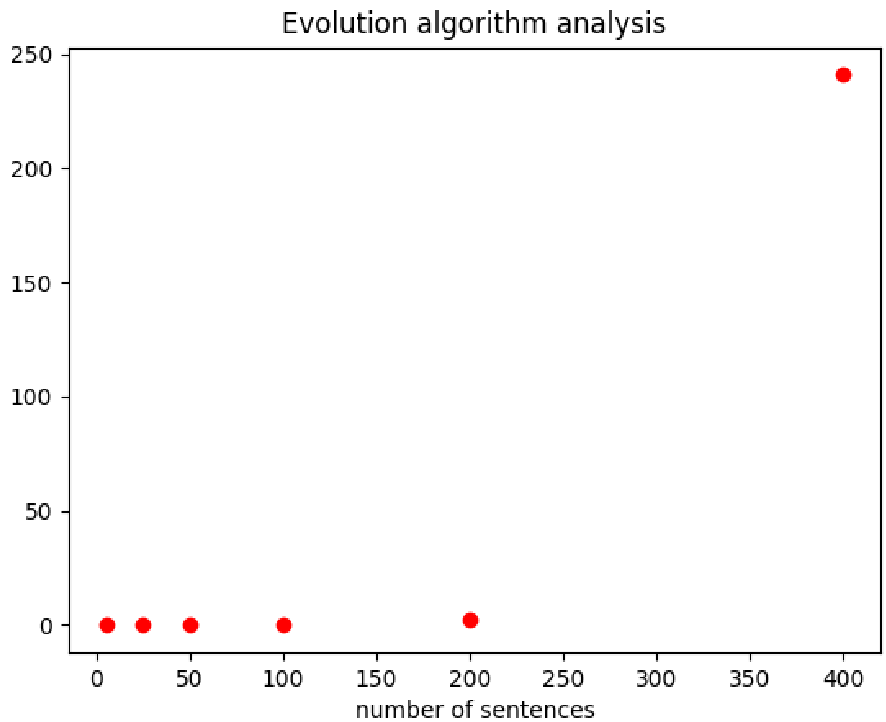
Funkcja ta również przyjmuje listę 0 i 1, jednak tym razem ich liczba jest równa liczbie literałów (bez znaczenia czy jest to negacja czy nie). Wartością początkową fitness jest tym razem liczba równa -1 * ilość zdań składowych. Następnie funkcja tworzy mapę literałów do ich wartości, a potem sprawdza czy w każdej z alternatyw:

- Negacji literału została przypisana wartość 0
- Literałowi została przypisana wartość 1

i wtedy dodaje 1 do wartości fitness i przestaje sprawdzać daną alternatywę.

Ta funkcja wydaje się bardziej odpowiednia do rozwiązywania tego problemu, gdyż nie musimy już przejmować się błędnymi przypisaniami wartości do wcześniej już określonych literałów.

Wykres:



Ilość zdań/literałów	5/15	25/57	50/81	100/96	200/100	400/100
Czas w sek. (średnia z 10 prób)	0.0262	0.0759	0.1269	0.5087	2.724	240.8363

Jak można łatwo zauważyć ta funkcja fitness spisała się o wiele lepiej, jeśli chodzi o czas wykonania, co widać zwłaszcza na przykładzie z 200 zdaniami, jednak co ważniejsze wyniki zwracane przez funkcję praktycznie zawsze były prawidłowe.

5. Algorytm PSO

Algorytm PSO – Particle Swarm Optimization polega na optymalizacji problemu poprzez iteratywne ulepszanie kandydującego rozwiązania poprzez populację rozwiązań, która poruszając się po określonej przestrzeni szuka optymalnego rozwiązania.

Kod:

```
def swarm(x):
    n_particles = x.shape[0]
    j = [fitness(x[i], x) for i in range(n_particles)]
    return np.array(j)


options = {'c1': 0.5, 'c2': 0.3, 'w':0.9, 'k':2, 'p':1}
optimizer = ps.discrete.BinaryPSO(n_particles=2000, dimensions=howManyLiterals(cnfFile),
options=options)
result = optimizer.optimize(swarm, iters=100, verbose=False)
```

Niestety algorytm ten okazał się zupełnie nieskuteczny w znajdywaniu rozwiązań problemu 3SAT, niezależnie od próbowanych opcji, ilości cząstek etc.

Jego wyniki były tak dalekie od prawdy, że mierzenie ich czasu mija się z celem.

Podejrzewam, że problemem jest niewystarczająca ilość opcji takich jak velocity w paczce PySwarms.

6. Podsumowanie.

Rezultatem tej pracy jest konkluzja, że algorytmy genetyczne, a w szczególności algorytm PSO nie są skuteczną metodą rozwiązywania problemów 3SAT, zwłaszcza w porównaniu do takiego algorytmu jak DPLL. Jednak algorytmy ewolucyjne wykazały pewien potencjał, co potwierdza znaczna ich przewaga nad algorytmem brute force.

Uznałem, że dysproporcja w jakości tych algorytmów, a co za tym idzie skala jakiej używałem do przeprowadzenia testów uniemożliwia czytelne zobrazowanie różnic między nimi na jednym wykresie.

Na podstawie przeprowadzonej analizy można ułożyć ranking wyżej przedstawionych algorytmów, w kolejności od najlepszego do najgorszego:

1. DPLL
2. Algorytm ewolucyjny wersja 2
3. Algorytm ewolucyjny wersja 1
4. Brute force
5. PSO