Algorytmy heurystyczne dokumentacja wstępna projektu

“Złap je wszystkie! - Baza danych Pokemon”

Mateusz Dorobek, Tomasz Nowak

Problematyka projektu

Dane, z jakimi mamy do czynienia, to zbiór 801 pokemonów, wraz z ich statystykami, które będą wykorzystywane do rozstrzygnięcia wyniku walki między nimi. Problem, jaki został postawiony to: “Znalezienie takiej szóstki pokemonów, która będzie w stanie wygrać jak najwięcej walk”. Taki problem można interpretować na wiele różnych sposobów, między innymi tak, że wygranie walki przez drużynę polega na pokonanie przeciwnika przez co najmniej jednego z członków drużyny. Jednym ze sposobów badanie tego jak silna jest dana drużyna jest liczenie średniej liczby wygranych walk przez każdego z członków drużyny. Zdefiniowanie sąsiedztwa w tym nieciągłym zbiorze jest również kwestią wartą uwagi, ponieważ zawsze będzie nieprecyzyjne, a nawet jeżeli uda się to zaimplementować to czy z punktu widzenia funkcji kosztu takie sąsiedztwo będzie dawało jakąkolwiek korzyść. Zdecydowaliśmy się na proste rozwiązanie - drużyny sąsiadują ze sobą jeżeli różnią się dokładnie jednym pokemonem.

Analiza danych

Z danych wybieramy jedynie te, które są istotne dla rozwiązywanego problemu, czyli wybrania najlepszej szóstki pokemonów. Aby go rozwiązać, musimy stworzyć symulację, która decyduje o wygraniu walki. Dane, które mogą mają wpływ na wynik walki

- *type* (jeden lub dwa)
- statystyki (*hp, attack, defence, special attack, special defence, speed*)
- against (*against_fire, against_water*, ... etc.)
- *capture_rate*

Algorytm walki

Wzór obliczający obrażenia będzie postaci: $Damage = \frac{A * AgainsType}{k*D}$, gdzie *A* to atak atakującego, *AgainstType* to współczynnik obrońcy, gdzie *Type* to typ atakującego pokemona, *D* - obrona atakowanego, a *k* to współczynnik wyrównujący, który powoduje, że zmniejsza się średnia liczba rozgrywanych tur z 60 do 6 dla k=10. 60 tur to liczba powstała poprzez podzielenie średniego Hp przez średni Damage, a faktyczna średnia liczba tur to ~6 (poparte doświadczeniem) Motywacją do tego typu działania było zwiększenie wpływu statystyki *speed*, która będzie decydowała o wyniku pojedynku w przypadku takiej samej ilości tur. W przypadku gdy *speed* nie rozstrzyga sporu pojedynek kończy się remisem.

Funkcje celu

Przyjęliśmy najprostszy sposób oceniania walki: za przegraną pokemon dostaje 0 punktów, za remis 0.5 punkta, a za wygraną 1 punkt. Ze względu na to, że wynik walki między daną parą pokemonów będzie wykorzystywany wielokrotnie podczas przeszukiwania, obliczyliśmy wyniki wszystkich możliwych walk już wcześniej (są one w dwuwymiarowej macierzy o liczbie kolumn/wierszy równej liczbie pokemonów).

Poniżej definicja dla funkcji goal_function_max_fight_result:

q - funkcja celu
k - parametr normalizujacy liczbe tur $k = 10$
$T_{p,q}$ - liczba tur potrzebnych na pokonanie przeciwnika
$Dmg_{p,q}$ - obrazenia jakie zadaje pokemon p pokemonowi q
$P_{p,q}$ - punkty jakie zdobywa pokemon p za pokonanie pokemona q
$CaptureRateNormalized_q$ - znormalizowany do przedzialu $< 0, 1 >$ parametr pokemona q
Hp_p - punkty zycia pokemona p
$AgstType_q(type1_p)$ - parametr $AgainstType$ pokemona q wzgledem typu 1 pokemona p

$$q = \frac{1}{n} \sum_{q \subset Pok} max(P_{p,q}),\ p \subset Team$$

$$Dmg_{p,q} = \frac{A_p * max(AgstType_q(type1_p), AgstType_q(type2_p))}{k * D_q}$$

$$T_{p,q} = \left\lceil \frac{Hp_q}{Dmg_{p,q}} \right\rceil$$

$$P_{p,q} = CaptureRateNormalized_q * \begin{cases} 1 & , (T_{p,q} > T_{q,p}) \vee (T_{p,q} = T_{q,p} \wedge Speed_p > Speed_q) \\ 0 & , (T_{p,q} < T_{q,p}) \vee (T_{p,q} = T_{q,p} \wedge Speed_p < Speed_q) \\ 0.5 & , (T_{p,q} = T_{q,p} \wedge Speed_p = Speed_q) \end{cases}$$

Sposób obliczania funkcji można zdefiniować na wiele sposobów, ponieważ w opisie zadania nie była ona narzucona. Przyjęliśmy, że funkcję celu drużyny będziemy obliczać niezależnie dla każdego możliwego przeciwnika (włącznie z członkami samej drużyny), a następnie zsumować wyniki.

Sposób obliczenia funkcji celu dla pojedynczego przeciwnika zależy od dwóch czynników:

1. Ocena pojedynczego pokemona:
a) Wynik w walce (pobrany z obliczonej wcześniej macierzy, opisanej powyżej).
b) Wynik w walce pomnożony przez capture_rate znormalizowany do (0, 1]

2. Wpływ poszczególnych pokemonów na wynik drużyny:
a) Dla każdego przeciwnika wybieramy najlepszego z naszych pokemonów (przeciwko temu przeciwnikowi) i potem sumujemy wyniki tych najlepszych pokemonów.

b) Dla każdego przeciwnika zakładamy że wszystkie nasze pokemony z nim walczą, a za końcowy wynik przyjmujemy średni wynik pokemonów w drużynie.

Początkowo przyjęliśmy wariant max(wynik w walce) (w implementacji goal_function_max_fight_result). Przy tak zdefiniowanym celu otrzymujemy rozwiązanie optymalne (liczba wygranych walk = liczbie pokemonów) już przy prostym błądzeniu przypadkowym. Jest to zrozumiałe o tyle, że dla tak zdefiniowanej funkcji celu fakt że rozpatrujemy całą drużynę, a nie pojedyncze pokemony, sprawia że wybór przypadkowej drużyny daje dobre rozwiązanie (wystarczy jeden dobry pokemon w drużynie). Spodziewaliśmy się początkowo, że problem nie będzie tak prosty ze względu na to, że pokemony mają różną skuteczność zależnie od rodzaju przeciwnika (np. wodne pokemony są skuteczne przeciwko ogniowym). Jednak okazało się, że możliwych "dobrych składów" jest bardzo dużo. Przykład drużyny, dla której funkcja celu przyjmuje największą możliwą wartość - 801 (liczba pokemonów): Abomasnow, Passimian, Dragonite, Rhyperior, Rampardos, Nosepass. Optymalny wynik uzyskujemy po kilku tysiącach iteracji błądzenia przypadkowego (dla random seed 0 było to 4481 iteracji) lub po kilkuset - do kilku tysięcy iteracji symulowanego wyżarzania.

Następnie przeanalizowaliśmy wariant średnia(wynik w walce) (w implementacji goal_function_mean_fight_result). Jednak okazał się on jeszcze prostszy od poprzedniego, ponieważ wynik drużyny zależy liniowo od wyników poszczególnych pokemonów w drużynie. Przy tak postawionym problemie optymalne rozwiązanie to po prostu wybór tych 6 pokemonów, dla których "indywidualna funkcja celu" jest największa (a wyznaczenie tego nie jest problemem obliczeniowym, wynik to drużyna: Palkia, Dialga, Yveltal, Zekrom, Solgaleo, Giratina) - w implementacji realizuje to funkcja greedy_search.

Wariantu średnia(wynik w walce * capture_rate) nie zrealizowaliśmy, ponieważ jakościowo nie różniłby się on niczym od średnia(wynik w walce). Wynik pojedynczego pokemona zależałby dodatkowo od capture_rate, ale nadal “zachłanny” wybór 6 najlepszych pokemonów dałby rozwiązanie optymalne.

Wariant max(wynik w walce * capture_rate) - tutaj szukamy pokemonów, które wygrywają większość walk, ale jednocześnie są łatwe do znalezienia. Był to najciekawszy przypadek, ponieważ tylko w tym wariancie funkcji celu nie mamy 100% pewności, czy otrzymaliśmy rozwiązanie optymalne. Teoretycznie największa wartość f. celu to 801 (wygranie wszystkich walk przez pokemony ze znormalizowanym capture_rate równym 1), ale prawdopodobnie nie istnieje skład osiągający taki wynik. Wiemy tylko, że mamy rozwiązanie “bardzo dobre” - największy wynik, jaki otrzymaliśmy - po 100000 iteracji symulowanego wyżarzania - to 788.16. Dokładniejsze omówienie wyników - w kolejnym paragrafie.

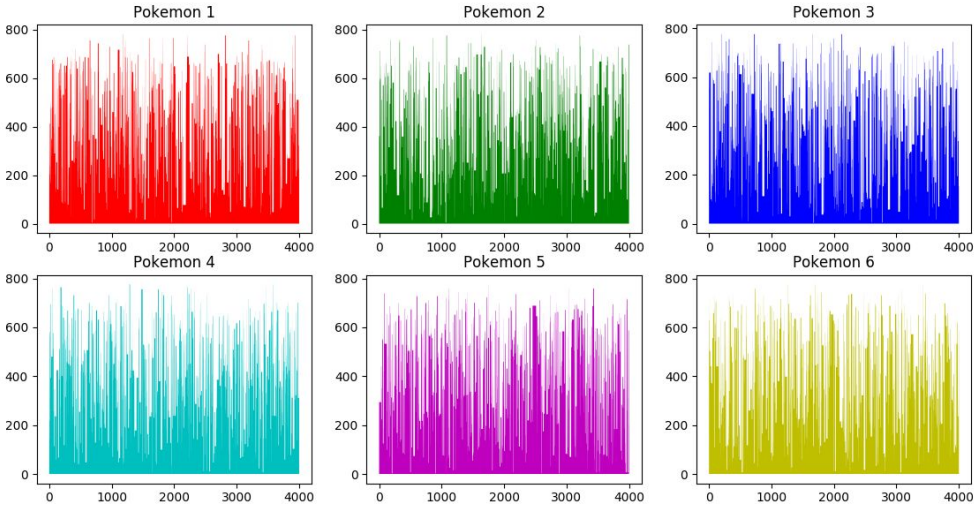
Wyniki (z wykresami i komentarzem)

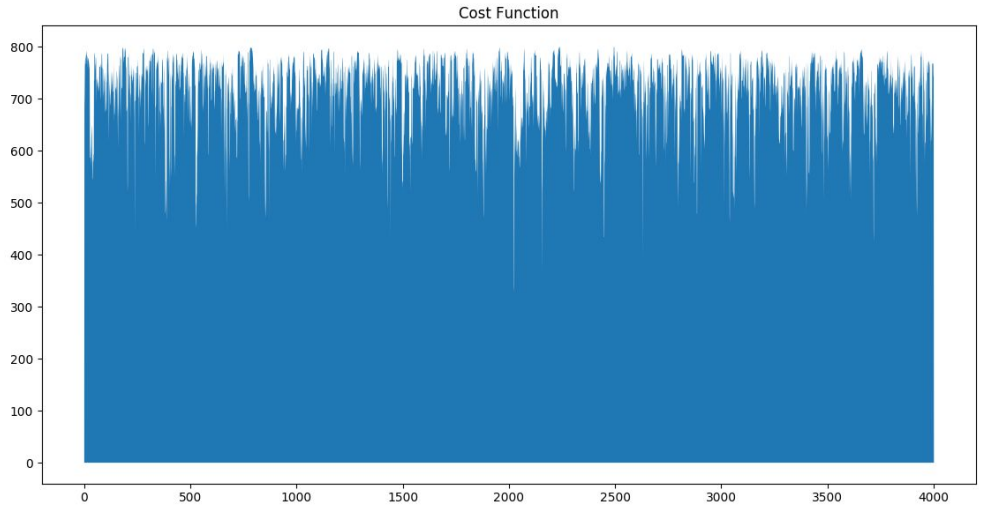
Umieszczamy po 2 wykresy w funkcji numeru iteracji: wykres z indywidualnymi wynikami poszczególnych pokemonów i wykres funkcji celu całej drużyny.

goal_function_max_fight_result, random_search:

```
best score: 801.0, best team: ['Slowbro', 'Mightyena', 'Beartic',
'Steelix', 'Vulpix', 'Rayquaza']
best team normalized capture rates: [0.29411765 0.49803922
0.23529412 0.09803922 0.74509804 0.17647059]
pokemon usage statistics: [654, 453, 635, 736, 112, 732]
best team in terms of all goal functions:
goal_function_max_fight_result: 801.0
goal_function_mean_fight_result: 554.4999999999991
goal_function_max_fight_result_with_capture_rate:
346.19607843137476
```

Tak jak wspomnieliśmy wcześniej, nawet błądzenie przypadkowe znajduje optymalne rozwiązanie (funkcja celu 801). Na wykresach widać niepoinformowanie tej metody - wartości na wykresach są zupełnie “przypadkowe”. Dla znalezionej drużyny wartości innych funkcji celu liczonych w inny sposób są niewielkie co wynika ze sposobu obliczania funkcji goal_function_max_fight_result (nie uwzględniamy capture_rate i wystarczy 1 silny pokemon w drużynie). Parametr “pokemon usage statistics” pokazuje, że największy wkład w zwycięstwo drużyny mają Steelix i Rayquaza, ale nie dominują one w drużynie (inne pokemony też mają dobre wyniki).

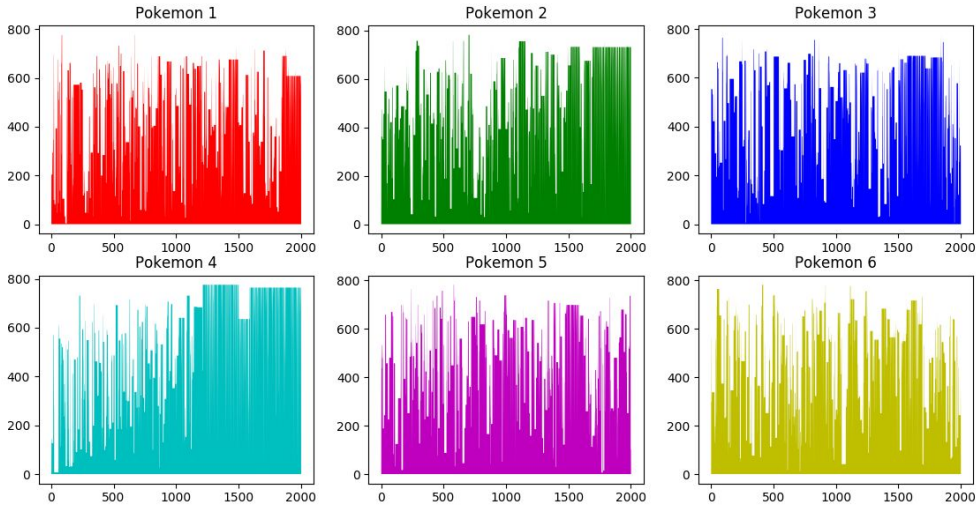


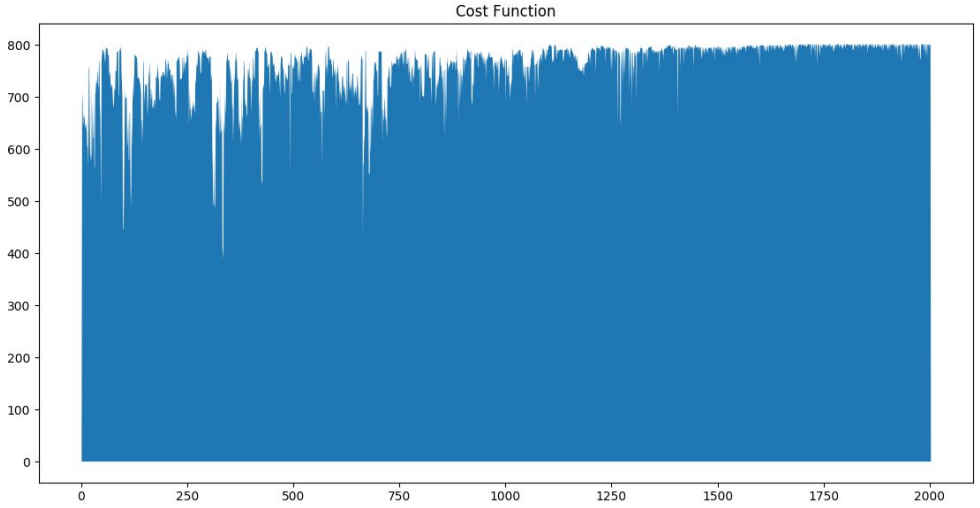


goal_function_max_fight_result, simulated_annealing:

```
best score: 801.0, best team: ['Drifloon', 'Bisharp', 'Bewear',
'Yveltal', 'Jolteon', 'Donphan']
best team normalized capture rates: [0.49019608 0.17647059
0.2745098  0.17647059 0.17647059 0.23529412]
pokemon usage statistics: [337, 674, 684, 764, 327, 618]
best team in terms of all goal functions:
goal_function_max_fight_result: 801.0
goal_function_mean_fight_result: 569.1666666666666
goal_function_max_fight_result_with_capture_rate:
286.57843137255156
```

Ostateczny wynik jest jakościowo podobny do tego z błądzenia przypadkowego, co wynika ze specyfiki funkcji celu: znaleziono rozwiązanie optymalne, co nie było trudne, ponieważ jest wiele drużyn z optymalnym goal_function_max_fight_result. Istotną różnicą jest sposób osiągania rozwiązania - wartość funkcji celu rośnie z czasem, a maleją zmiany tej wartości w kolejnych iteracjach (ze względu na malejącą temperaturę). Symulowane wyżarzanie początkowo przypomina błądzenie przypadkowe (co pozwala wyjść z optimów lokalnych), a później - algorytm wspinaczkowy (dzięki czemu szybciej osiąga optimum lokalne, które ma szansę być optimum globalnym). Nie jest również zaskakujące, że ten algorytm osiąga optymalne rozwiązanie nieco szybciej niż błądzenie przypadkowe.





goal_function_mean_fight_result, greedy_search:

```
best score: 770.33, best team: ['Palkia', 'Dialga', 'Yveltal',
'Zekrom', 'Solgaleo', 'Giratina']
best team normalized capture rates: [0.01176471 0.01176471
0.17647059 0.01176471 0.17647059 0.01176471]
best team in terms of all goal functions:
goal_function_max_fight_result: 801.0
goal_function_mean_fight_result: 770.3333333333334
goal_function_max_fight_result_with_capture_rate:
140.93529411764675
```

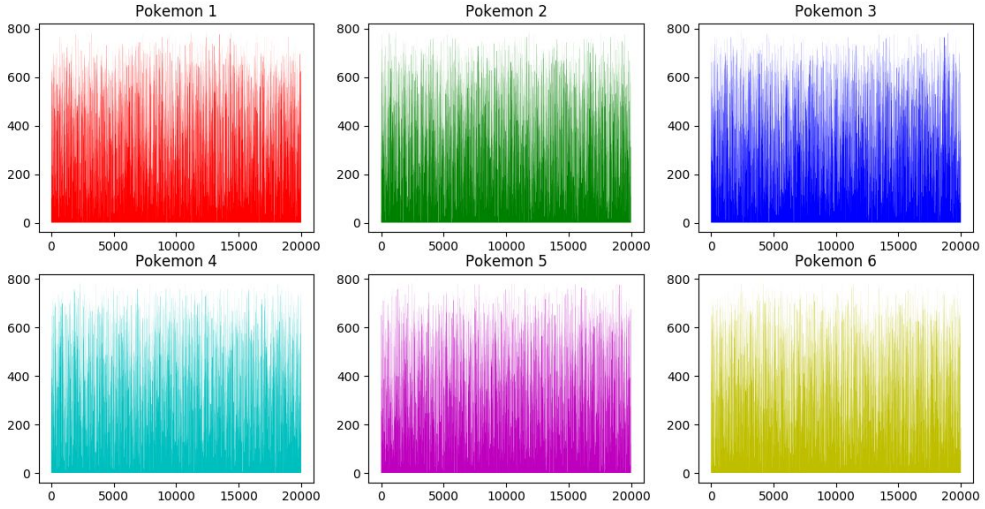
Tak jak wspomnieliśmy, dla funkcji celu goal_function_mean_fight_result problem da się rozwiązać bez użycia heurystyki. Nie jest zaskakujące, że wybrane zostało 6 bardzo silnych pokemonów, o niskim capture_rate - dlatego też goal_function_max_fight_result dla drużyny wybranej w ramach goal_function_mean_fight_result również przyjmuje optymalną wartość, natomiast goal_function_max_fight_result_with_capture_rate - bardzo małą.

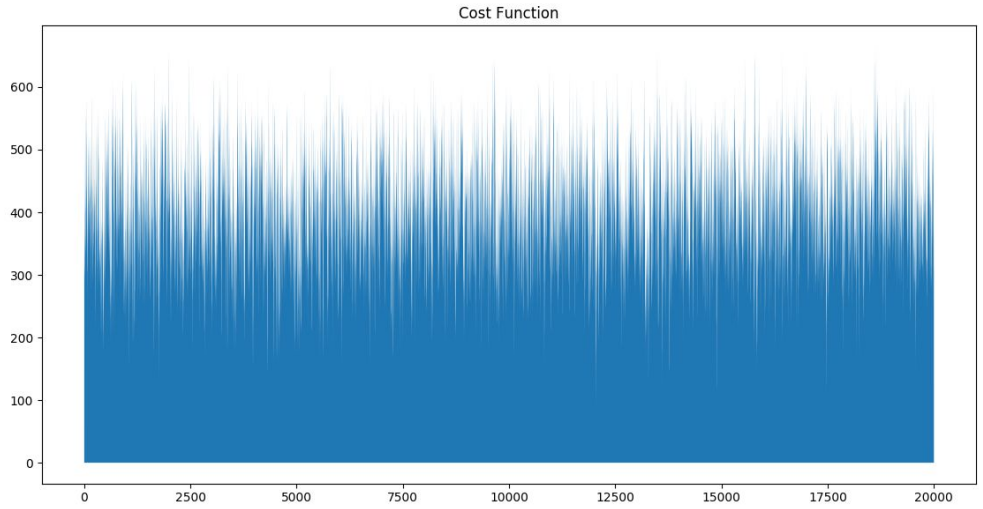
Pomimo otrzymania optymalnego rozwiązania przez greedy_search, sprawdziliśmy też, jak z tym problemem poradzą sobie inne metody:

goal_function_mean_fight_result, random_search:

```
best score: 665.25, best team: ['Samurott', 'Solgaleo',
'Darmanitan', 'Doublade', 'Araquanid', 'Metagross']
best team normalized capture rates: [0.17647059 0.17647059
0.23529412 0.35294118 0.39215686 0.01176471]
best team in terms of all goal functions:
goal_function_max_fight_result: 793.0
goal_function_mean_fight_result: 665.2499999999995
goal_function_max_fight_result_with_capture_rate:
295.21568627451217
```

Pomimo 20000 iteracji random_search zwraca wynik daleki od optymalnego. Zapewne w tym wariancie funkcji celu niepoinformowanie błądzenia przypadkowego ma większy (negatywny) wpływ na wynik niż w wariancie goal_function_max_fight_result.

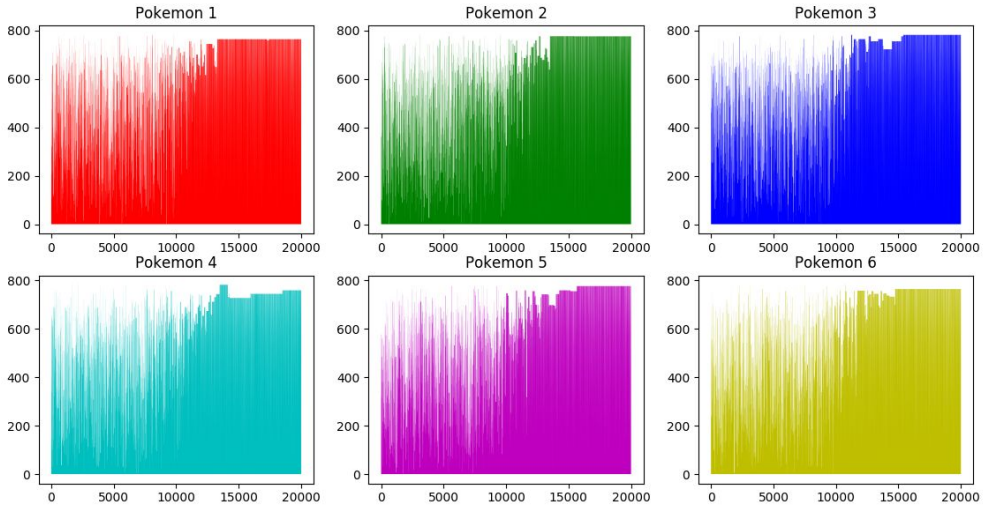


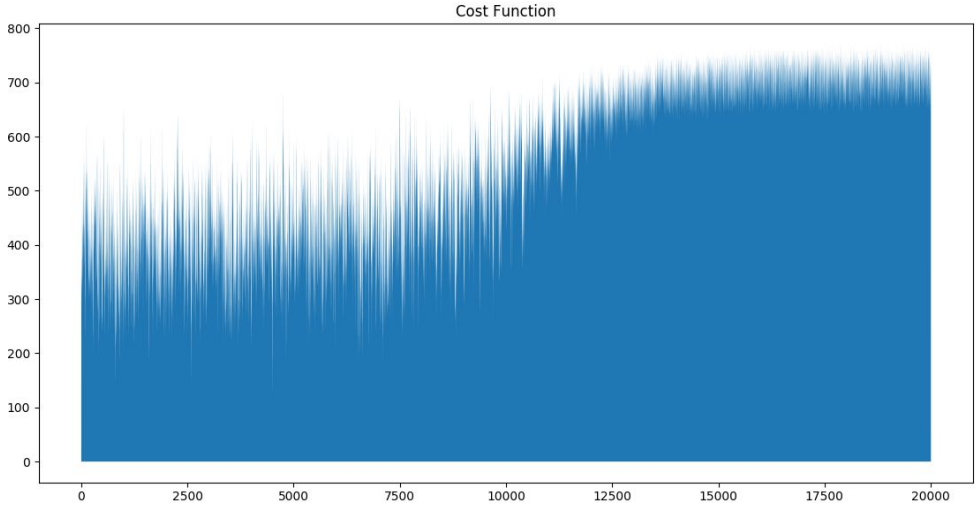


goal_function_mean_fight_result, simulated_annealing:

```
best score: 770.33, best team: ['Dialga', 'Zekrom', 'Giratina',
'Palkia', 'Solgaleo', 'Yveltal']
best team normalized capture rates: [0.01176471 0.01176471
0.01176471 0.01176471 0.17647059 0.17647059]
best team in terms of all goal functions:
goal_function_max_fight_result: 801.0
goal_function_mean_fight_result: 770.3333333333334
goal_function_max_fight_result_with_capture_rate:
140.93529411764675
```

Symulowane wyżarzanie zwróciło drużynę z funkcją celu o takiej samej wartości jak greedy_search, więc również jest to rozwiązanie optymalne. Pokazuje to skuteczność tej metody — w przypadku problemu, gdzie wybór “przypadkowej drużyny” nie daje dobrych wyników radzi sobie znacznie lepiej niż błądzenie przypadkowe. Podobnie jak wcześniej, na wykresach widać wpływ temperatury na zmienność funkcji celu w kolejnych iteracjach. Widać również, że wszystkie wybrane pokemony są silne (niemal równa “prostokątna” część wykresów z wynikami poszczególnych pokemonów w późniejszych iteracjach) - jest tak, ponieważ w tym wariancie funkcji celu wszystkie pokemony mają wpływ na wynik walki (drużyna nie może mieć “słabego ogniwa”).



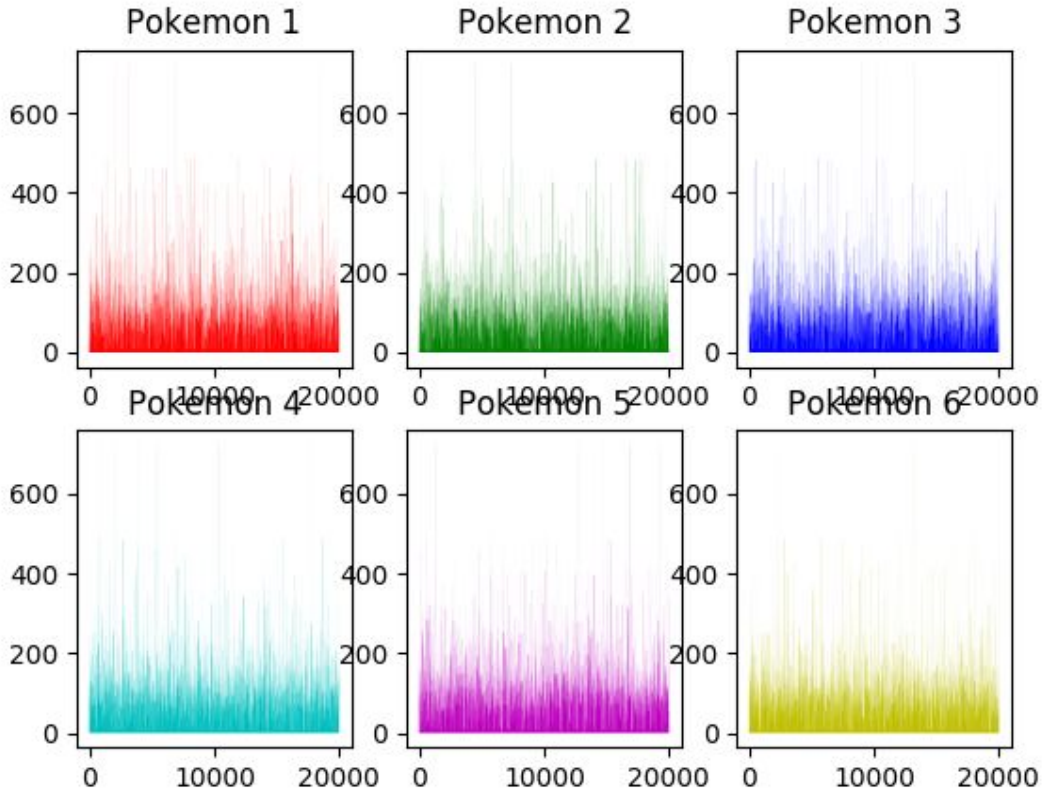


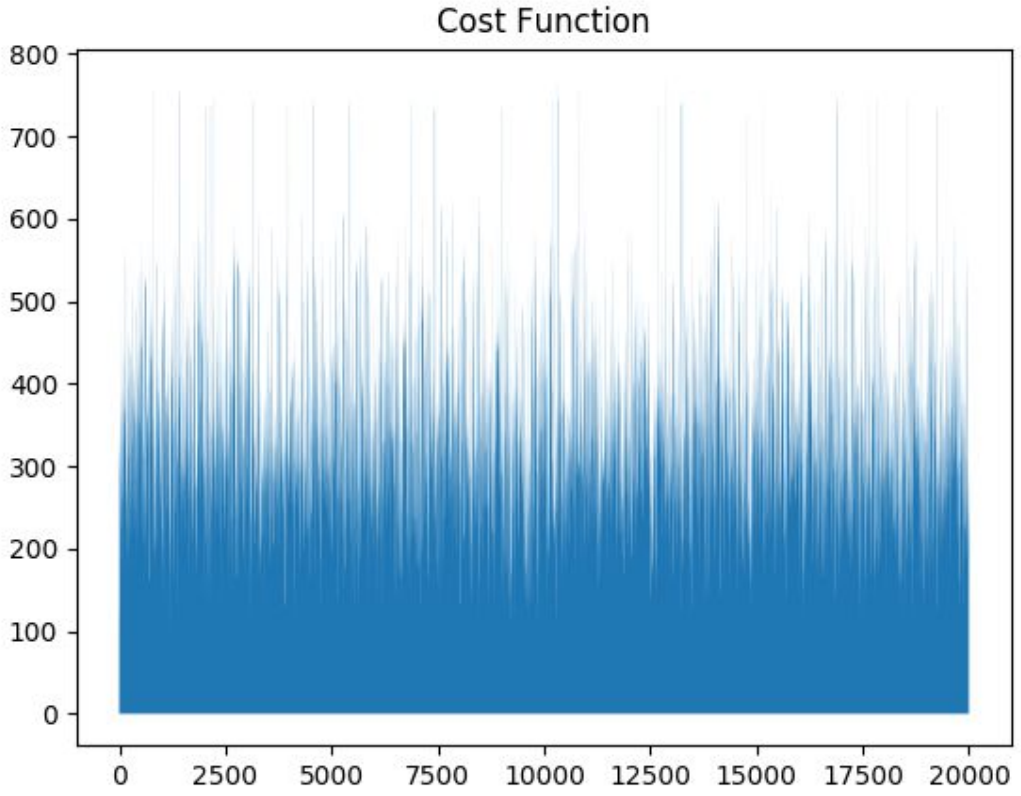
goal_function_max_fight_result_with_capture_rate, random_search:

```
best score: 768.02, best team: ['Stoutland', 'Magearna',
'Skorupi', 'Stunfisk', 'Kartana', 'Hariyama']
best team normalized capture rates: [0.17647059 0.01176471
0.47058824 0.29411765 1.         0.78431373]
pokemon usage statistics: [7, 12, 8, 19, 729, 61]
best team in terms of all goal functions:
goal_function_max_fight_result: 794.0
goal_function_mean_fight_result: 585.0833333333326
goal_function_max_fight_result_with_capture_rate:
768.0196078431363
```

Otrzymujemy tutaj bardzo dobry wynik (niewiele gorszy od symulowanego wyżarzania, którego wyniki zamieszczamy poniżej). Wynika to prawdopodobnie z tego, że znów mamy do czynienia z funkcją celu, w której nie wszystkie pokemony muszą mieć wpływ na wynik drużyny. Z wybranego składy drastycznie wyróżnia się Kartana, która jest najlepszym

pokemonem w drużynie (usage statistics) w aż 729 walkach - ten pokemon ma cepture_rate niewspółmierne do pozostałych statystyk (jest niewiele słabszy od najsilniejszych legendarnych pokemonów, a ma największe możliwe capture_rate).

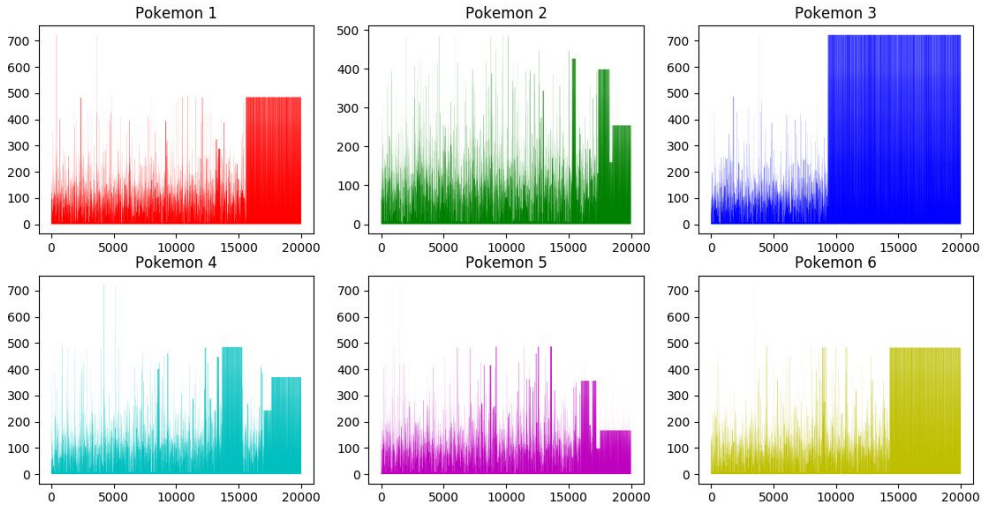


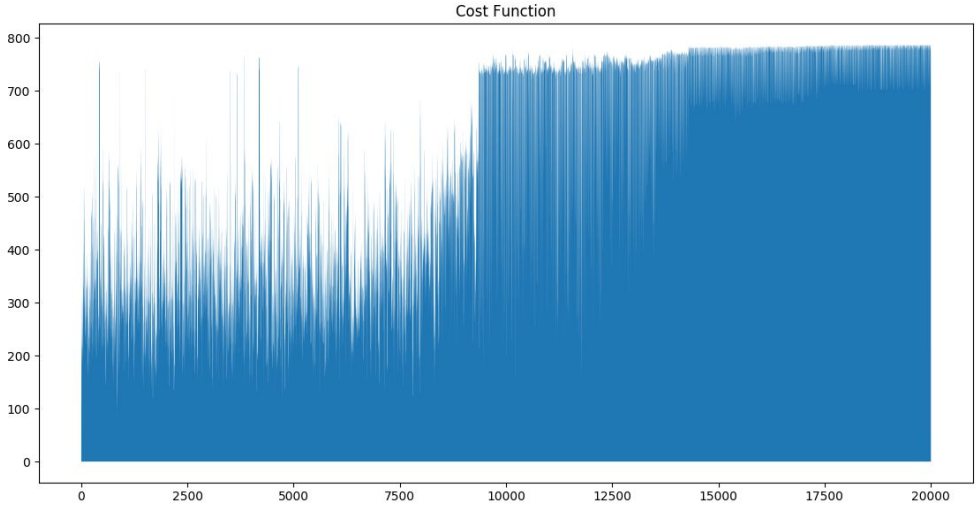


goal_function_max_fight_result_with_capture_rate, simulated_annealing:

```
best score: 787.86, best team: ['Minior', 'Togedemaru', 'Kartana',
'Crawdaunt', 'Shuppet', 'Hariyama']
best team normalized capture rates: [1.         0.70588235 1.
0.60784314 0.88235294 0.78431373]
pokemon usage statistics: [489, 8, 728, 11, 12, 24]
best team in terms of all goal functions:
goal_function_max_fight_result: 795.0
goal_function_mean_fight_result: 530.9166666666665
goal_function_max_fight_result_with_capture_rate:
787.8627450980391
```

Otrzymujemy tutaj rozwiązanie nieco lepsze od błądzenia przypadkowego. Znów sukces drużyny w ogromnym stopniu zależy od tego samego pokemona - Kartana (usage statistics 728), ale pojawia się również inny silny pokemon, który prawdopodobnie “nadrabia słabości” Kartany - Minior (usage statistics 489). Błądzenie przypadkowe “przypadkowo” znalazło skład z dobrym wynikiem - skład zawierający Kartanę, natomiast symulowane wyżarzanie dzięki poinformowaniu znalazło skład nieco lepszy wymieniając któregoś ze słabszych pokemonów na Miniora. Poza tym, ogólny przebieg funkcji celu w kolejnych iteracjach jest podobny jak przy innych funkcjach celu - decydujący wpływ na to ma temperatura.





Próbując znaleźć jeszcze lepsze rozwiązanie, uruchomiliśmy **goal_function_max_fight_result_with_capture_rate, simulated_annealing z większą liczbą iteracji** (100000 - już bez wykresów):

```
best score: 788.16, best team: ['Numel', 'Hariyama', 'Crawdaunt',
'Kartana', 'Minior', 'Tropius']
best team normalized capture rates: [1.         0.78431373
0.60784314 1.         1.         0.78431373]
pokemon usage statistics: [293, 21, 12, 729, 490, 17]
best team in terms of all goal functions:
goal_function_max_fight_result: 794.0
goal_function_mean_fight_result: 548.1666666666661
goal_function_max_fight_result_with_capture_rate:
788.1568627450978
```

Ostateczny wynik jest niewiele większy pomimo 5-krotnie większej liczby iteracji (788.16, wcześniej 787.86), natomiast widać, że algorytm oprócz Kartany i Miniora znalazł jeszcze

jednego pokemona mającego znaczący wpływ na wynik drużyny: Numel. Jak widać, “eksploatacja” wykonywana przez symulowane wyżarzanie przy małej temperaturze, może nieco poprawić wynik otrzymany w poprzednich iteracjach (w czasie “eksploracji”).

Wyniki goal_function_max_fight_result_with_capture_rate pokazują, że w naszym zestawie danych Kartana jest po prostu “za silna” w stosunku do capture_rate. Zgodnie z https://bulbapedia.bulbagarden.net/wiki/Kartana_(Pok%C3%A9mon) ma on capture rate równe 40, a nasz zbiór danych zapewne był oparty na innej wersji Pokemonów: *In Pokémon Sun and Moon, Kartana has a catch rate of 255*.

Dla wariantu goal_function_max_fight_result_with_capture_rate najlepsze okazywały się pokemony z największym możliwym capture_rate (co pokazują “usage statistics” powyżej), w związku z tym można sprawdzić wszystkie możliwe drużyny zawierające takie pokemony (jest 70 pokemonów z capture rate 255, 70 nad 6 to ok. 131 milionów - po odpowiednio długich obliczeniach można sprawdzić wszystkie drużyny) - nie podjęliśmy się tego z braku czasu.

Implementacja

Program został napisany w Pythonie 3.7. Oprócz biblioteki standardowej, skorzystaliśmy z:

- numpy
- matplotib
- simanneal (https://github.com/perrygeo/simanneal)

Uruchomienie python main.py --help zwróci listę parametrów, z którymi można wywołać program (dla uproszenia wszystkie parametry mają wartości domyślne). Obsługiwane parametry:
--file [nazwa] plik z danym wejściowymi, domyślnie data.csv
--goal [nazwa] funkcja celu, do wyboru: goal_function_max_fight_result, goal_function_mean_fight_result, goal_function_max_fight_result_with_capture_rate
--solver [nazwa] funkcja rozwiązująca problem przeszukiwania, do wyboru: random_search, simulated_annealing, greedy_search (to ostatnie zadziała poprawnie tylko dla funkcji celu goal_function_mean_fight_result
--iterations [liczba] liczba iteracji
--plot2d flaga określająca, czy rysować wykresy 2d (wykres z wynikami poszczególnych pokemonów i wykres z funkcją celu drużyny w kolejnych iteracjach)
--plot3d flaga określająca, czy rysować wykres 3d (wyniki pokemonów i funkcja celu na jednym wykresie)
--details wyświetli dodatkowe informacje o zwycięskiej drużynie (capture_rate, wyniki drużyny mierzone innymi funkcjami celu, w przypadku funkcji celu goal_function_max* udział poszczególnych pokemonów w zwycięskich walkach)
--outfile [nazwa] historia wyników w poszczególnych iteracjach zostanie zapisana do pliku o podanej nazwie

Możliwe rozszerzenia

1. Gdyby pokemonów było znacznie więcej i osiągnięcie dobrego wyniku byłoby trudniejsze, można by zmienić sposób przeszukiwania, np.:
   - w ramach szukania sąsiada drużyny wybierać pokemona, który ma być zastąpiony jakimś pokemonem spoza drużyny z prawdopodobieństwem zależnym od tego, w ilu walkach w danej iteracji dany pokemon okazał się najlepszy (im więcej, tym mniejsze prawdopodobieństwo usunięcia pokemona z drużyny)
   - wybór losowego pokemona spoza drużyny z prawdopodobieństwem zależnym od "indywidualnej funkcji celu" (im lepiej dany pokemon wypada w walkach indywidualnych, tym większe prawdopodobieństwo wybrania)
   - algorytm automatycznego dobierania parametrów algorytmu, takich jak temperatura w symulowanym wyżarzaniu
   - użycie innego algorytmu, np. algorytmu ewolucyjnego zaproponowanego w dokumentacji wstępnej
2. Interfejs programu nie jest "przyjazny" dla użytkownika. Można usprawnić interfejs na przykład poprzez:
   - uproszczenie parametrów wejściowych, np połączenie --details --outfile i --plot2d z automatycznym nadawaniem nazw plikom wyjściowym
   - wprowadzenie plików konfiguracyjnych umożliwiających "zlecanie" obliczeń wsadowych (kilka różnych wersji programu uruchamianych jedna po drugiej)
   - przerywanie obliczeń w dowolnym momencie, serializację wyników pośrednich i możliwość wznowienia obliczeń
   - stworzenie interfejsu graficznego