Otoczka wypukła dla zbioru punktów w przestrzeni dwuwymiarowej

Opis problemu

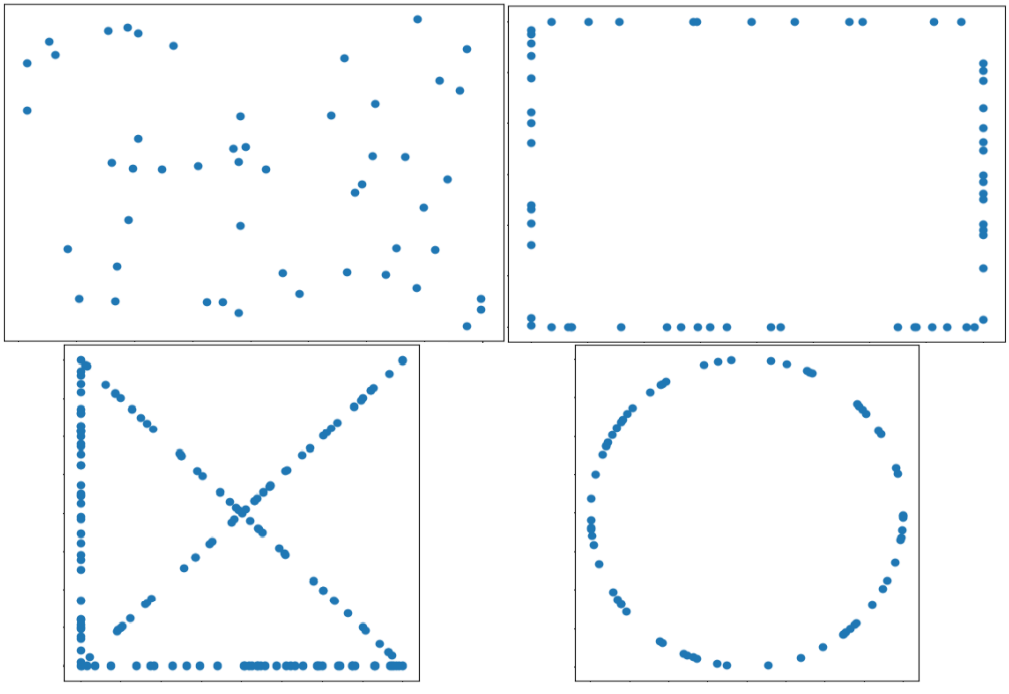
Zaimplementować następujące algorytmy wyznaczające otoczkę wypukłą zbioru punktów $2D$:

- Algorytm przyrostowy - $O(n \log n)$,
- Algorytm na górną i dolną otoczkę - $O(n \log n)$,
- Algorytm QuickHull - $O(n \log n)$,
- Algorytm dziel i zwyciężaj - $O(n \log n)$,
- Algorytm Chana - $O(n \log(h))$
- Algorytm Grahama - $O(n \log n)$
- Algorytm Jarvisa - $O(nh)$,

gdzie n to liczba wszystkich punktów, a h to liczba punktów na otoczce będącej poprawnym wynikiem. Następnie dobrać odpowiednio zbiory testowe i porównać efektywność powyższych algorytmów.

Zbiory testowe

Poniżej przedstawiono cztery rodzaje, wykorzystywanych w testach poprawności i wydajności, zbiorów (z wybranymi parametrami):



Wizualizacja 1: Zbiory testowe

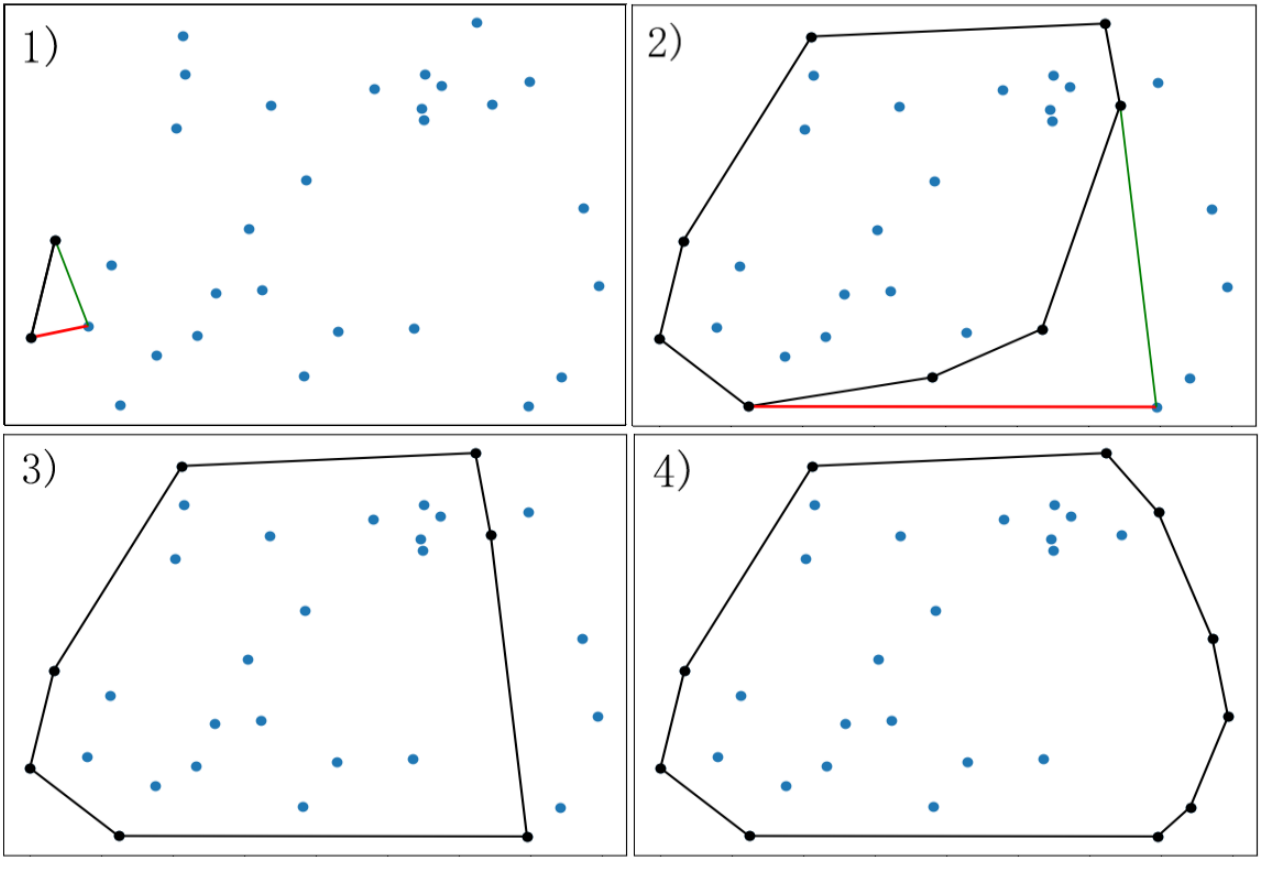
Algorytm przyrostowy

Opis działania:

Algorytm konstruuje kolejkę z posortowanymi malejąco wg współrzędnej x punktami ($points.sort() - O(n \log n)$). Potem tworzy listę reprezentującą otoczkę wyjmując 2 pierwsze punkty z kolejki (osobno rozpatrywany jest przypadek gdy pierwsze kilka punktów, ma taką samą współrzędną x - brane są wtedy dwa skrajne punkty z prostej, na której leżą). Następnie, w pętli, powtarzane są następujące czynności - aż do opróżnienia kolejki:

- wyjęcie pierwszego punktu z kolejki,
- znalezienie dla niego indeksów górnego i dolnego punktu styczności na otoczce binary searchem - $O(\log len(hull))$,
- modyfikacja listy otoczki, w taki sposób, aby w łańcuchu (dolny punkt styczności) - ... - (górny punkt styczności), przeciwnie do ruchu wskazówek zegara, zamienić jego środek na punkt wyjęty z kolejki.

Wizualizacja:
Poniżej przedstawiono wybrane klatki z wizualizacja działania algorytmu, od początku (rys. 1), przez ilustracje operacji wstawienia punktu za środek łańcucha (rys. 2 - rys. 3), do wyniku (rys. 4):



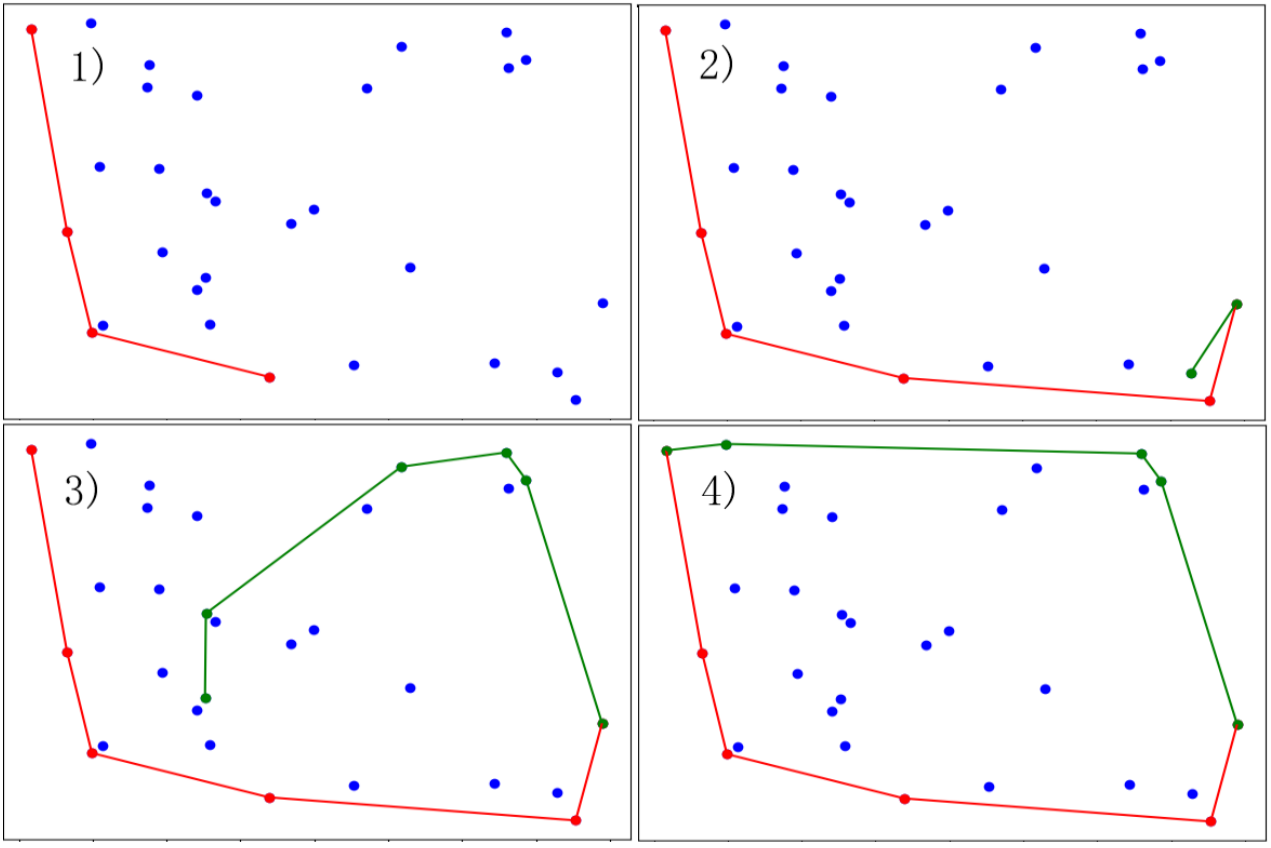
Wizualizacja 2: Algorytm przyrostowy

Algorytm na górną i dolną otoczkę

Opis działania:

Algorytm `upperlower()` opiera się na znalezieniu otoczki ograniczającej zbiór od góry, od doły oraz połączeniu ich. Na początku sortujemy punkty w pierwszej kolejności po współrzędnych x-owych, a w drugiej po y-owych. Dokonujemy tego za pomocą wbudowanej funkcji `points.sort()` w czasie $O(n \log n)$. Następnie przechodząc po wszystkich punktach, analogicznie do algorytmu Grahama, dodajemy punkty, które nie tworzą kątów wklęsłych patrząc od wewnątrz zbioru. To samo robimy przechodząc od tyłu po zbiorze punktów, co pozwala wyznaczyć górną otoczkę. Na koniec wystarczy połączyć obie otoczki. Jako, że początkowe sortowanie odbywa się w czasie $O(n \log n)$ oraz wykonujemy dwa liniowe przejścia, to finalny czas jest równy $O(n \log n)$.

Wizualizacja:
Poniżej przedstawiono wybrane klatki z wizualizacja działania algorytmu, od środkowej części procesu wyznaczania dolnej otoczki(rys. 1), przez początek i środek procesu wyznaczania górnej otoczki (rys. 2 - rys. 3), do wyniku (rys. 4):



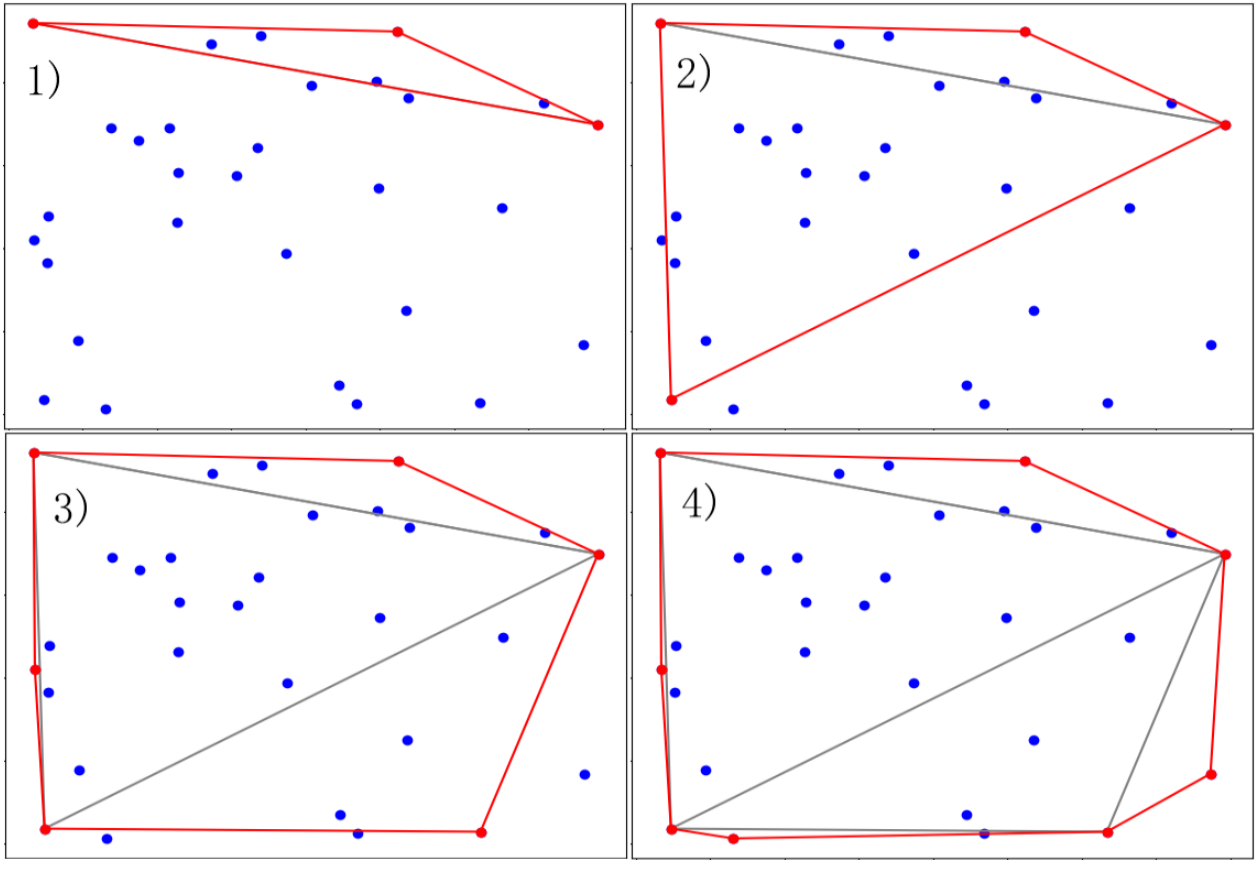
Wizualizacja 3: Algorytm na górną i dolną otoczkę

Algorytm Quickhull

Opis działania:

Zaimplementowaliśmy rekurencyjną wersje algorytmu `quickhull()`. Zaczynamy poprzez wybranie punktów skrajnych pod względem współrzędnej x-owej co jest dokonywane za pomocą `min`, `max` w czasie liniowym. Następnie pozostałe punkty dzielimy na dwa zbiory, zależnie od położenia względem prostej przechodzącej, przez znalezione wcześniej punkty. Zbiory wraz z punktami przekazujemy do rekurencyjnej funkcji. Znajduje ona punkt położony najdalej od prostej przechodzącej przez przekazane dwa punkty oraz wyznacza kolejne dwa zbiory znajdujące się na zewnątrz trójkąta łączącego trzy punkty. Do kolejnego wywołania przekazywane są odpowiednie zbiory wraz z punktami. Rekurencja wykonuje się, aż nie zostanie przekazany zbiór pusty, wtedy podział nie jest wykonywany. Jako, że zawsze dokonujemy dzielenia na 2 dopełniające się problemy, to złożoność tego algorytmu wynosi $\mathrm{O}(n \log n)$.

Wizualizacja:
Poniżej przedstawiono wybrane klatki z wizualizacja działania algorytmu:



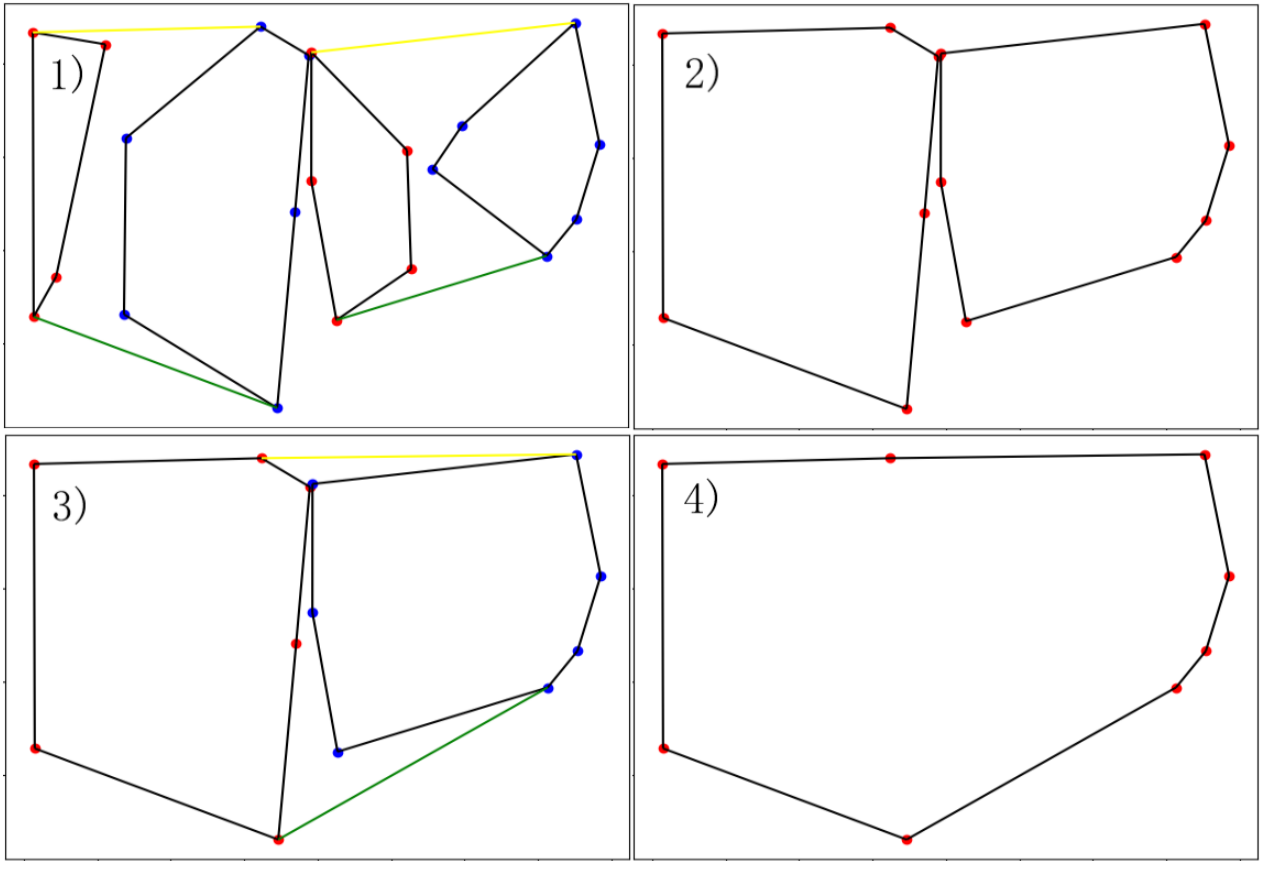
Wizualizacja 4: Algorytm QuickHull

Algorytm dziel i zwyciężaj

Opis działania:

Algorytm `divideconquer()` opiera się na rekurencyjnym dzieleniu zbioru punktów na, w przybliżeniu ± 1, dwie równe części (w naszej implementacji według współrzędnych x punktów - wymaga to ich wcześniejszego posortowania w $O(n \log n)$), aż uzyskamy zbiory o tak małej liczebności, że znalezienie ich otoczki wypukłej może się odbyć przez użycie algorytmu *brute force*, bez straty na złożoności. Następnie, idąc od końca, dokonuję się łączenia dwóch otoczek, powstałych z podzielenia tego samego zbioru. Dokonuje się tego poprzez znalezienie górnej i dolnej stycznej między dwoma otoczkami. W naszej implementacji, znalezienie stycznych polega na obraniu dwóch punktów (nazwijmy je l i r), po jednej z każdej otoczki, będącymi na starcie punktami leżącymi najbliżej długiej otoczki, i w pętli sprawdzanie czy zarówno (l, r) i (r, l) są odcinkami stycznymi do przeciwnych otoczek, odpowiednio zmieniając l lub r na sąsiednie jeśli tak nie jest. Po znalezieniu punktów styczności, tworzona jest nowa lista, reprezentująca otoczkę powstałą po złączeniu, poprzez scalenie fragmentów dwóch otoczek (pozbywając się fragmentów między punktami styczności od strony wewnętrznej układu). Ta metoda pozwala na scalenie dwóch otoczek o łącznej liczebności m w $O(m)$, zatem, na każdym poziomie rekurencji, wszystkie scalania działają łącznie w $O(n)$. Ponieważ zbiory dzielimy na 2 równe części, to wykonamy scalanie wszystkich obecnych par otoczek $\sim \log n$ razy, stąd złożoność algorytmu, wraz z posortowaniem, $O(n \log n)$.

Wizualizacja:
Poniżej przedstawiono wybrane klatki z wizualizacja działania algorytmu, ilustrujące scalanie otoczek (rys 1. - rys 2., rys 3. - rys 4.) i wynik działania algorytmu (rys 4.):



Wizualizacja 5: Algorytm dziel i zwyciężaj

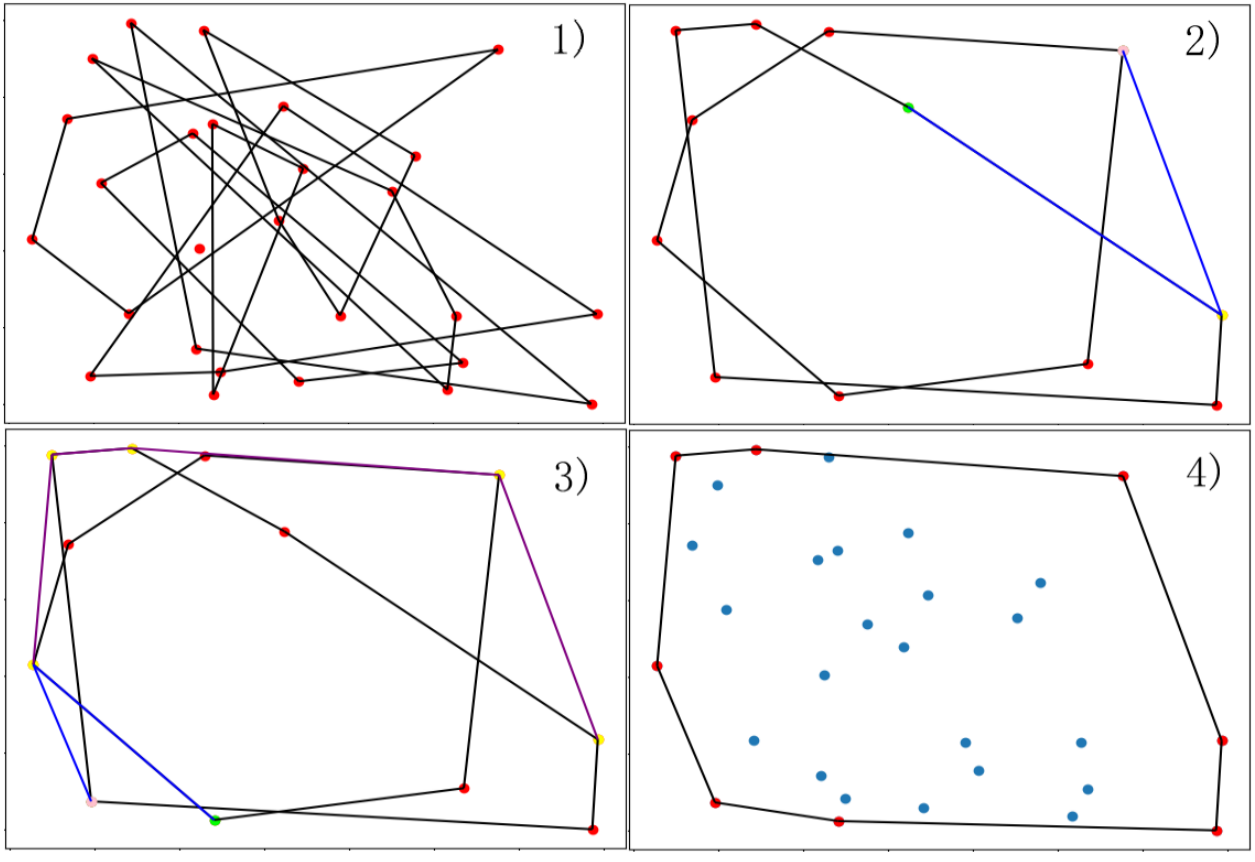
Algorytm Chana

Opis działania:

Algorytm zakłada znajomość liczby punktów h na otoczce. Punkty dzielone są na podzbiory o liczebności maksymalnie h. Dla każdego takiego podzbioru algorytmem Grahama wyznaczana jest jego otoczka. Na koniec zaczynając od punktu skrajnego stosujemy algorytm analogiczny do Jarvisa, z tym ze możemy rozpatrywać po jednym punkcie z każdej wyznaczonej otoczki (punkcie styczności), biorąc ten, tworzący największy kąt z 2 ostatnio dodanymi do otoczki punktami.
W praktyce raczej nie znamy h wiec algorytm wywołujemy dla 'zgadniętego' $h = m = 2^{2^t}$ gdzie t=1,2... Kluczowe dla algorytmu jest szybkie znajdowanie punktu styczności z ostatniego dodanego punktu 'marszu Jarvisa' do wyznaczonych przez algorytm Grahama otoczek. Używa się do tego algorytmu *binary search*, podobnie jak w Algorytmie przyrostowym, gdyż dla jednej otoczki osiąga on złożoność $O(\log(h))$, bardziej pożądane niż liniowe $O(h)$. Znalezienie nowego punktu do otoczki ma więc złożoność $O((n/h)\log h)$, a cały marsz Jarvisa $O(h{\cdot}(n/h)\log h) = O(n\log h)$. Wyznaczenie otoczek zbiorów, powstałych przez początkowy podział, algorytmem Grahama zajmuje również $O(n\log h)$, więc taka jest również złożoność całego algorytmu, zakładając, że znamy h. Przy wywołaniu dla mh złożoność wynosi natomiast $O(n(1+h/m)logm)$. Algorytm wywołujemy wiele razy, ale wybierając kolejne $m = 2^{2^t}$, końcowa złożoność nadal wynosi:

$$O(\sum_{t=1}^{\log\log h} n2t) = O(n2^{[\log\log h]+1}) = O(n\log h).$$

Wizualizacja:
Poniżej przedstawiono wybrane klatki z wizualizacja działania algorytmu: rys. 1. pokazujący podział na otoczki przy za małym m, rys. 2. wizualizujący rozpoczęcie działania marszu Jarvisa przy odpowiednim m ($m >= h$), rys 3. prezentujący dalszą część marszu Jarvisa i rys. 4. z wyznaczoną otoczką.



Wizualizacja 6: Algorytm Chana

Algorytm Grahama

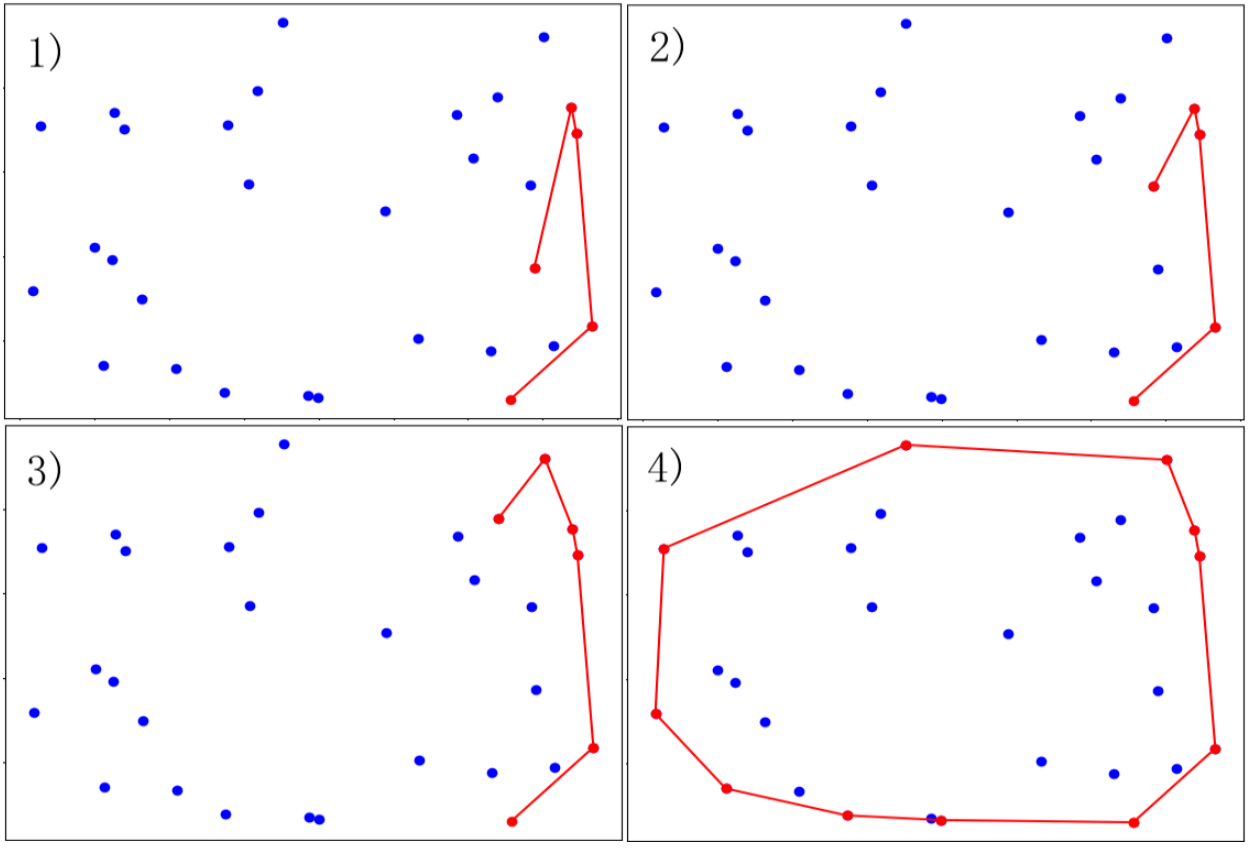
Opis działania:

Algorytm `grahams()` opiera się na wpierw posortowaniu punktów po kącie, który jest tworzony miedzy osią OX oraz prostą przechodzącą przez punkt startowy(taki który posiada najmniejszą współrzędną y-ową oraz w drugiej kolejności x-ową - znajdowalny w czasie $O(n)$) i rozpatrywany punkt. Dla punktów o tym samym kącie, sortowanie przebiega także, po odległości, od najbliższego do najdalszego. Możemy to osiągnąć za pomocą funkcji `cmp_to_key` z modułu `functools` oraz użyć funkcji `points.sort()` z opisanym komparatorem. Następnie dodajemy pierwsze dwa punkty do otoczki oraz przetwarzamy wszystkie pozostałe punkty zgodnie z:

- Dodajemy punkt do otoczki.
- Jeżeli utworzony został kąt wklęsły, patrząc od wewnątrz, to przedostatni punkt, aż do momentu pozbycia się kątów wklęsłych.

Jako, że liniowo przechodzimy po otoczce, a krok ten zajmuje tylko $O(1)$, co skutkuję złożonością całkowitą $O(n \log n) + n\ O(1) = O(n \log n)$.

Wizualizacja:
Poniżej przedstawiono wybrane klatki z wizualizacja działania algorytmu:



Wizualizacja 7: Algorytm Grahama

Algorytm Jarvisa

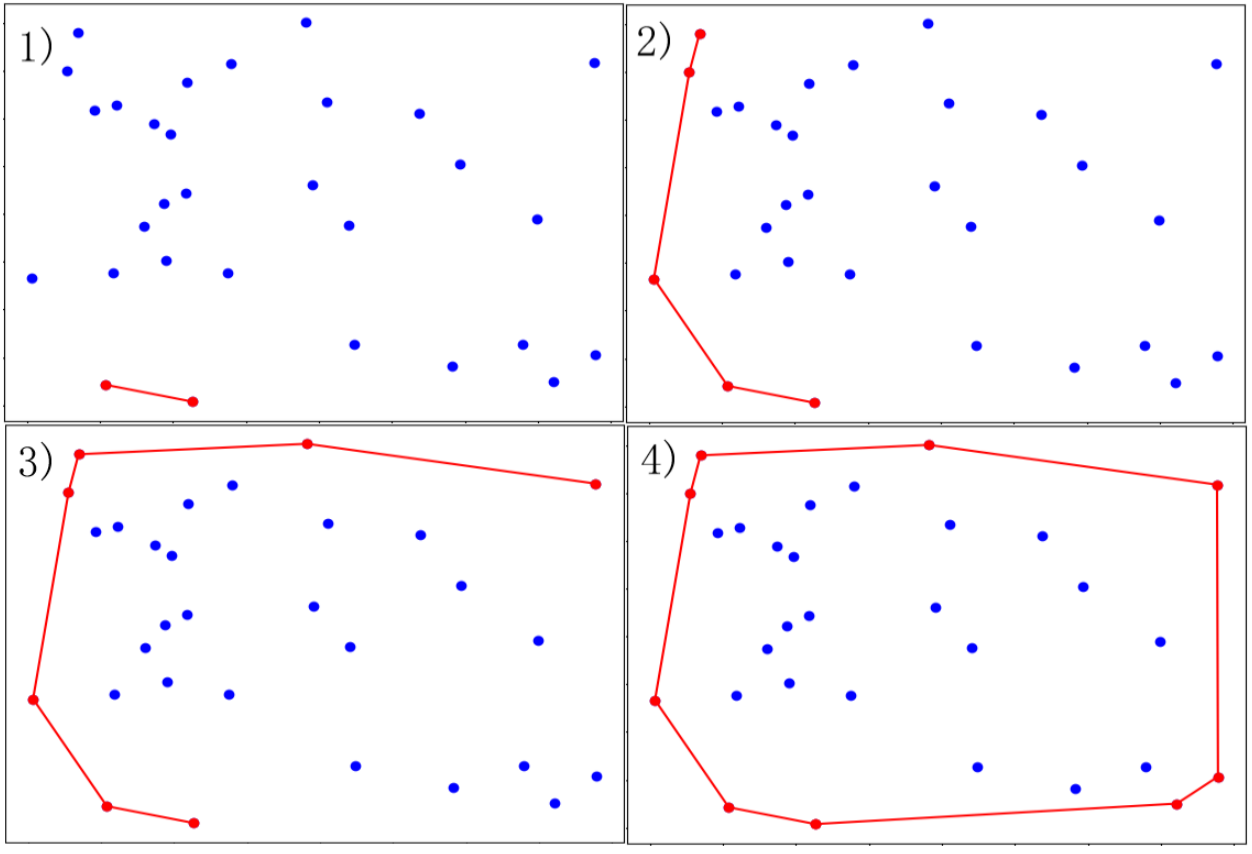
Opis działania:

Na początku w czasie liniowym znajdujemy punkt o najmniejszej współrzędnej x-owej (w drugiej kolejności y-owej). Potem przy użyciu `reduce` z modułu `functools` możemy znaleźć element minimalny z listy pod względem przekazanego komparatora, w tym przypadku będzie to kąt pomiędzy osią OX, a prostą przechodzącą przez punkt ostatnio dodany i punkt rozpatrywany. Algorytm trwa, aż punkt który powinien zostać dodany jako kolejny, jest pierwszym punktem otoczki. Kolejno wykonywać się będą następujące kroki:

- Znalezienie kolejnego punktu przy użyciu `reduce` - czas $O(n)$.
- Dodanie go do otoczki, jeżeli jest punktem pierwszym to przerwanie algorytmu.

Dla otoczek pesymistycznych (o liczebności n) algorytm posiada złożoność $O(n^2)$, natomiast w przypadku, gdzie punktów będzie k, $O(nk)$.

Wizualizacja:
Poniżej przedstawiono wybrane klatki z wizualizacja działania algorytmu:



Wizualizacja 8: Algorytm Jarvisa

Bibliografia

- wykład z przedmiotu Algorytmy geometryczne na kierunku Informatyka uczelni AGH,
- https://pl.wikipedia.org/wiki/Algorytm_Jarvisa,
- https://pl.wikipedia.org/wiki/Algorytm_Grahama,
- https://pl.wikipedia.org/wiki/Quickhull,
- https://en.wikipedia.org/wiki/Chans_algorithm,
- https://en.wikibooks.org/wiki/Algorithm_Implementation/Geometry/Convex_hull/Monotone_chain,
- 'Practical Geometry Algorithms with C++ Code' - Daniel Sunday PhD.