Optymalizacja Kombinatoryczna – laboratoria

Zadanie pierwsze

Gliwiński Jarosław Marek
Kruczyński Konrad Marek
Grupa dziekańska I5

23 stycznia 2009

1 Wstęp

Przypomnimy na początek treść zadania ze strony WWW.

> Mamy dany zbiór punktów na płaszczyźnie, o współrzędnych będących liczbami naturalnymi. Należy znaleźć taki cykl prosty obejmujący wszystkie punkty, że wielokąt zdefiniowany przez ten cykl jest prosty i posiada minimalną możliwą powierzchnię.
>
> Wielokąt jest prosty jeżeli jedynymi punktami wspólnymi jego dwóch boków są wierzchołki boków sąsiednich. Cykl jest prosty jeśli nie odwiedza żadnego wierzchołka więcej niż raz. Dopuszcza się zastosowanie pewnego przybliżenia przy obliczaniu powierzchni wielokąta.

Rozważania wstępne obejmowały odpowiednie zdefiniowanie problemu. Zauważmy, że cykl jest jednoznacznie wyznaczony pewną permutacją wierzchołków (należy przy tym zauważyć, ze nie dla każdej permutacji będziemy w stanie wyznaczyć koszt (powierzchnię), ponieważ niektóre z nich nie tworzą wielokątów). Uzyskujemy dzięki temu natychmiastowe wyobrażenie idei algorytmu typu „brute force” oraz jego złożoności.

2 Zagadnienia pomocnicze

Zanim przejdziemy do opisu „sedna” projektu, tj. algorytmów znajdowania wielokąta o polu najmniejszym, należy omówić niezbędne przy ich implementacji kwestie pomocnicze. Czas poświęcony na ich implementację (również opracowanie w niektórych przypadkach) stanowił istotną część poświęconego ogólnie projektowi.

2.1 Generacja instancji

Instancje używane do testów metod rozwiązania są tworzone przy pomocy prostego generatora sparametryzowanego wg rozmiaru instancji i rozrzutu punktów (maksymalnych

występujacych w instancji liczb). Do generowania sekwencji pseudolosowych wykorzystywana jest funkcja generujaca liczby o orzkładzie równomiernym z przedziału ¡from, to) `UniformFromTo(double from, double to)` z napisanej przez nas biblioteki pomocniczej (zawierającej także inne popularne rozkłady, takie jak normalny czy wykładniczy). Wyniki są rzutowane na liczby całkowite i zabezpieczane przed powtarzaniem lokalizacji punktów. Z kolei rozkład normalny jest wykorzystywany w drugiej metodzie generowania instancji, polegającej na losowaniu współrzędnych biegunowych dla punktu. W tym przypadku kąt ma rozkład Gaussa. Proporcje między punktami generowanymi jednym i drugim generatorem są regulowane ostatnim z parametrów generatora.

2.2 Badanie bycia wielokątem

Zauważmy, że gdy choć konkretna permutacja punktów nie musi tworzyć wielokąta (zakładam, że łączymy kolejne punkty, a następnie pierwszy z ostatnim), to nie pozostawia żadnego punktu niepołączonego, każdy stanowi też początek dwóch odcinków (lub początek i koniec zależnie od konwencji). Otrzymujemy zatem pętlę, a jedynym zagrożeniem jest możliwość przecinania się odcinków. W przeciwnym wypadku tworzą zamkniętą krzywą regularną, co jest przez nas pożądane.

Każdy odcinek reprezentowany jest przez klasę `odcinek`, której konstruktor otrzymuje jako parametry dwa obiekty klasy `punkt`. Wspomniane uczynione są klasycznie za wyjątkiem dodatkowego, publicznego pola `nr` zawierającego unikalny numer punktu wyprodukowanego przez generator. Potrzeba stworzenia takiego pola omówiona zostanie później. Przy pomocy przeciążonego operatora

```
bool odcinek::operator==(odcinek & odc);
```

dokonuje się sprawdzenia faktu przecinania się dwóch odcinków (wartość `true` w przypadku odpowiedzi pozytywnej). Algorytm działa w następujący sposób:

1. Obliczenie współczynnika kierunkowego i wyrazu wolnego prostej, której dany odcinek jest fragmentem. Przy zadaniu tym wykorzystywany jest punkt początkowy i końcowy odcinka oraz układ równań liniowych. Jeżeli prosta zawierająca dany odcinek jest pionowa, informacja o tym również zapisywana jest w klasie odcinek.

2. Operator za pomocą kolejnego układu równań wyznacza punkt przecięcia się prostych zawierających odcinki, oddzielnie obsługując przypadek równoległych. Należy przy tym uwzględnić możliwość, iż jedna z prostych (lub obie) są pionowe.

3. Sprawdzane jest, czy punkt przecięcia zawiera się w obu odcinkach. Dozwolonym wyjątkiem od tego jest ewentualność, iż punktem wspólnym jest koniec jednego i początek innego odcinka. Jeżeli odcinki należą do tej samej prostej, mamy nieskończenie wiele punktów do wyboru. W tym przypadku rzutujemy na oś X punkty początkowe i końcowe odcinków, a po ich posortowaniu stwierdzamy ewentualne nachodzenie.

Ponadto należy wspomnieć, iż wszelkie powyższe obliczenia wykonywane są na liczbach wymiernych w celu uzyskania odpowiedniej dokładności (przeciwdziała to możliwościom uznania prostych o jedynie podobnych współczynnikach kierunkowych za równoległe). Klasa zawiera zaimplementowane wszystkie potrzebne operatory. Ponieważ przy operacjach związanych z liczbami wymiernymi dosyć często dochodzi do ich rozszerzenia, po każdej operacji wykonywane jest podzielenie licznika i mianownika przez (największy wspólny) dzielnik wyznaczany przy pomocy algortymu Euklidesa. Zastosowanie wyżej wymienionej klasy spowalnia wprawdzie wszelkie obliczenia dotyczące odcinków, zapewnia jednak odpowiednią dokładność.

Wielokąty wygodnie jest przechowywać w stworzonej do tego celu klasie `wielokat`. Z istotniejszych metod warto wymienić

```
bool czyWielokat();
bool czyWielokat2(odcinek & op1, odcinek & op2);
```

, które za pomocą wyżej wymienionego operatora testują w danym wielokącie odcinki w układzie „każdy z każdym". Ponadto druga z nich podaje parę odcinków, na których wykryła pierwsze przecięcie (dalsze nie są w tej sytuacji poszukiwane).

2.3 Pomiar pola

Pomiar pola jest również istotną składową projektu. W naszym wykonaniu pomiar został wykoanany metodą rekurencyjnej triangulacji otrzymanego wielokąta, realizowanej przez podział wielokąta przekątnymi. Powrót z rekurencji następowałe oczywiście po odkryciu wielokąta składowego, który nie ma przekątnej, czyli trójkąta. Suma takich składowych pozwala obliczyć pole.

Warto wspomnieć o ciekawej własności badanych wielokątów, która pozwoliła na dokładne (bez zaokrągleń) liczenie pola na liczbach całkowitych. Otóż podwojone pole wielokąta o współrzędnych naturalnych zawsze będzie liczbą naturalną. Wynika to wprost z triangulacji: pole wielokąta jest sumą trójkątów składowych, zaś pole każdego z tych trójkątów jest połową pola prostokąta opartego o podstawę i wysokość trójkąta. Pole takiego prostokąta, jak można zauważyć, zawsze bedzie liczbą całkowitą, a zatem suma ich bedzie także całkowita. Suma ta równa jest właśnie podwojonemu polu badanego wielokąta.

Metoda nie gwarantuje dokładnego pomiaru w przypadku wielokątów, które nie są proste, jednak badania wskazują, iż nieraz się to udaje. Kwestia ta tak czy inaczej nie ma znaczenia, bowiem każdy generowany wielokąt jest weryfikowany pod kątem bycia prawidłowym wielokątem prostym.

2.4 Pomiar czasu

Na koniec warto wspomnieć o sposobie pomiaru czasu. Systemem operacyjnym, w jakim testowaliśmy algorytmy, był Windows Vista. Zgodnie z informacjami, jakie udało się nam uzyskać, najdokładniejszą metodą pomiaru czasu jest użycie funkcji pochodzących z API Windows: QueryPerformanceFrequency oraz QueryPerformanceCounter. Aby zmi-

nimalizować fluktuacje spowodowane innymi wątkami, jakie mógł wykonywać w tym czasie procesor, przydzieliliśmy naszemu programowi priorytet czasu rzeczywistego.

3 Rozwiązanie problemu

W ogólności problem rozwiązywany jest następująco:

1. Punkty wejściowe są sortowane według współrzędnych biegunowych, wprost tworząc startowy wielokąt.
2. Krótkie, a następnie stopniowo coraz dłuższe wycinki cyklu są poddawane modyfikacji kolejności łączenia punktów
3. Przeszukiwanie wycinków płynnie przechodzi w permutację wszystkich wierzchołków

3.1 Metoda biegunowa

Najpierw wyliczany jest środek ciężkości układu punktów, który następnie przyjmowany jest jako środek biegunowego układu współrzędnych. Następnie współrzędne kartezjańskie punktów przeliczane są na ten nowy układ, po czym sortowane według współrzędnej kątowej, a w drugiej kolejności (w przypadku arbitrażu) rosnąco według odległości od środka ciężkości. W dalszej kolejności punkty są łączone w kolejności uzyskanej przez sortowanie. To pierwsze przybliżenie, jakkolwiek jest to jedynie metoda wstępna, a jakość uzyskanych rozwiązań ma charakter losowy przy zastosowanych różnorodnych instancjach.

3.2 Metoda progresywna

Główna metoda rozwiązania opiera się na następującej obserwacji: skoro możliwe jest uzyskanie optymalnego rozwiązania w czasie $O(n!)$ poprzez sprawdzenie wszystkich możliwych poprawnych wielokątów, możliwe jest przybliżenie tego rozwiązania poprzez sprawdzenie wszystkich możliwych przebiegów pewnego odpowiedniej wielkości (wyrażonej w liczbie punktów $m < n$) wycinka dowolnego istniejącego poprawnego wielokąta. Jednocześnie, im więcej wycinków w ten sposób „permutujemy" , tym większe są szanse dobrego przybliżenia optymalnego rozwiązania. Jednocześnie wraz ze spadkiem wartości m maleje złożoność obliczeniowa. Wniosek z powyższego jest taki, że stosując tę metodę „progresywnie" od bezwzględnie małych m do bliskich wielkości instancji można uzyskać stopniowe przybliżanie optymalnego rozwiązania, kolejno wyczerpując na każdym etapie wszystkie możliwe wycinki o danej długości. Summa summarum pozwala to przyspieszyć uzyskiwanie „dobrych" rozwiązań, jednakże kosztem wydłużenia całkowitego czasu działania metody. Czas z założenia musi być dłuższy, ponieważ tak czy inaczej na końcu rozwiązywany jest pełen „bruteforce", który ponownie przegląda także rozwiązania wcześniej już uzyskane.

3.3 Przeszukiwanie wyczerpujące

Jak wspomniano, metoda progresywna kończy się wywołaniem dla wycinka o długości równej wielkości instacji, zarazem wywołując pełne wyczerpujące poszukiwanie rozwiązania. Dosyć oczywistą implementacją metody przeszukiwania dokładnego, jaka nasuwa się po zauważeniu faktu, iż wszystkie wielokąty są określone jednoznacznie przez pewne permutacje ciągu punktów, jest przetestowanie wszystkich takich obiektów kombinatorycznych. Z uwagi na łatwość samej implementacji (kolejne permutacje generowaliśmy przy pomocy biblioteki STL) na początku zdecydowaliśmy się na jej zastosowanie.

Zgodnie z przewidywaniami metoda okazała się niezwykle powolna. Dość zauważyć, że choć z trzech punktów utworzyć można tylko jeden trójkąt, otrzymamy dlań aż 6 różnych permutacji. Dla większych wielokątów liczba ta odpowiednio rośnie. Byłoby ponadto wskazanym, aby przy stwierdzeniu, iż dane dwa odcinki przecinają się, odrzucić wszystkich kandydatów na wielokątów te odcinki zawierających.

W celu pozbycia się wyżej wymienionych wad zdecydowaliśmy się na zmianę metody na rekurencyjną. Ta buduje wielokąt dołączając kolejne punkty, a ściślej:

1. Dołącza do wielokąta nowy punkt (podany w parametrze jej wywołania), co tworzy w nim nowy odcinek.

2. Nowo dołączony odcinek badany jest na możliwość przecinania ze wszystkimi już w danym wielokącie istniejącymi. W przypadku wykrycia przecięcia metoda kończy pracę, a przez to odrzuca wszystkie wielokąty zawierające już utworzony zestaw odcinków.

3. Jeżeli dołączony punkt jest ostatnim, jaki można było dodać, jego pojawienie się powoduje utworzenie dwóch odcinków. W tym miejscu drugi z nich badany jest na przecięcia z pozostałymi.

4. W przeciwnym wypadku metoda oznacza dany punkt jako wykorzystany i wywołuje się rekurencyjnie dla jeszcze niewykorzystanych punktów.

5. W przypadku, gdy uda się pomyślnie zamknąć wielokąt, mierzone jest jego pole, a gdy okaże się mniejsze od najmniejszego dotychczas odnotowanego, wielokąt i jego pole jest zapisywany.

Ponadto pierwsze wywołanie metody jako kandydata wybiera zawsze jeden z punktów, nie testując pozostałych. W naszej implementacji jest to zawsze pierwszy punkt stworzony przez generator. Dzięki temu n-krotnie zmniejszamy ilość przeszukiwań, nie tracimy natomiast żadnych pożądanych instancji, bowiem na kształt (i pole) wielokąta nie ma wpływu wybór punktu początkowego.

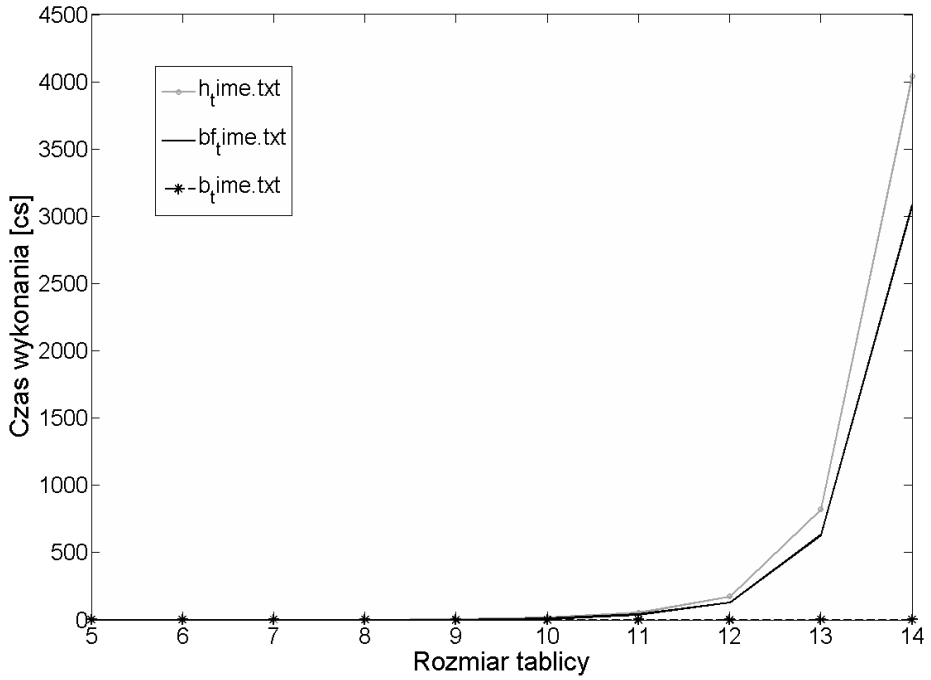
Dzięki wyżej wymienionym operacjom uzyskaliśmy znaczne przyspieszenie działania programu, ostatecznie zaimplementowaliśmy jeszcze jedną optymalizację w algorytmie dokładnym. Otóż warto zauważyć, iż skoro w zagadnieniu mamy n punktów, odcinek jest jednoznacznie wyznaczony przez 2 z nich, natomiast ich para przez cztery, to liczba wszystkich możliwych testów przecięć może rosnąć co najwyżej jak $O(n^4)$. Uznaliśmy,

że wobec tego warto wprowadzić buforowanie operacji badania przecięć, tj. wyniki testu zapamiętywać i korzystać z nich w późniejszych testach. W obliczu faktów opisanych już wyżej (wiele przypadków, wykorzystanie liczb wymiernych) dało to kolejne znaczne zwiększenie wydajności algorytmu dokładnego. Wymaga to wprawdzie tablicy o rozmiarze n^4 słów, co jednak przy ilości pamięci, w jaką wyposażone są współczesne komputery i rozmiarach instancji, dla jakich użycie tego algorytmu w ogóle ma sens, nie wydaje się problemem. Godnym uwagi jest fakt poprawienia wydajności przy użyciu tej właśnie drogi, gdzie na ogół większość wysiłków skupia się na lepszym wykorzystaniu czasu procesora.

Na koniec warto wspomnieć, iż ostateczna wersja algorytmu dla trzynastokątów okazała się o około 5 rzędów wielkości szybsza od pierwotnej, co jest znakomitym wynikiem.

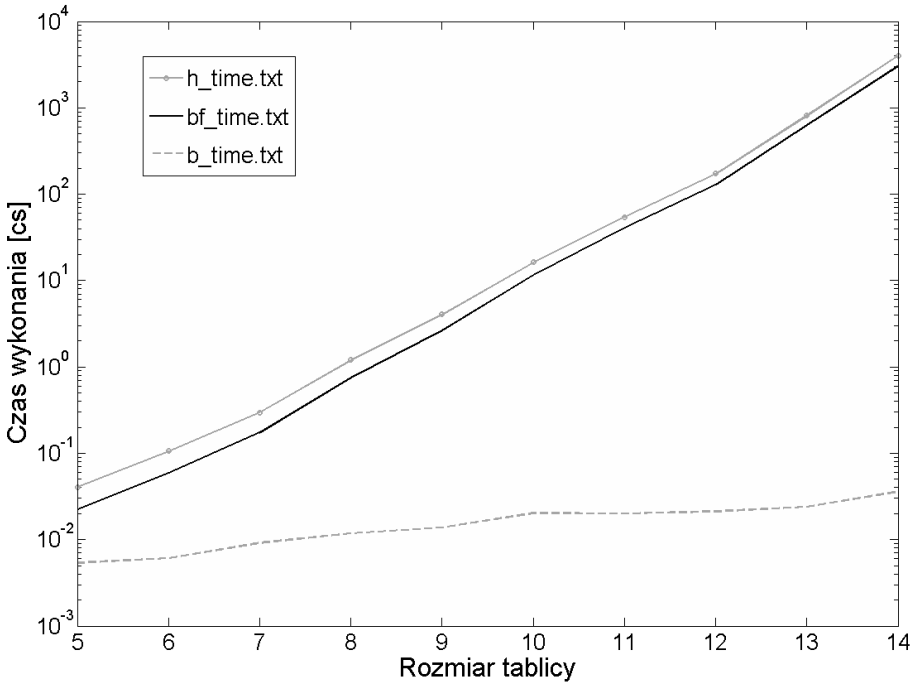
4 Pomiary

Pomiarów dokonano w zakresie rozmiaru instancji od 5 do 14. Górną granicę wybrano ze względu na granicę uciążliwości sumarycznego czasu testowania. Dla każdego rozmiaru instancji wykonano 30 iteracji każdej z metod (wstępna – „biegunowa", wyczerpująca, wyczerpująca z progresywnym startem). Zgodnie z intuicyjnymi przewidywaniami okazało się, że dla poszukiwania wyczerpującego metoda z dołączoną progresywną heurystyką okazała się wolniejsza od czystej wyczerpującej.

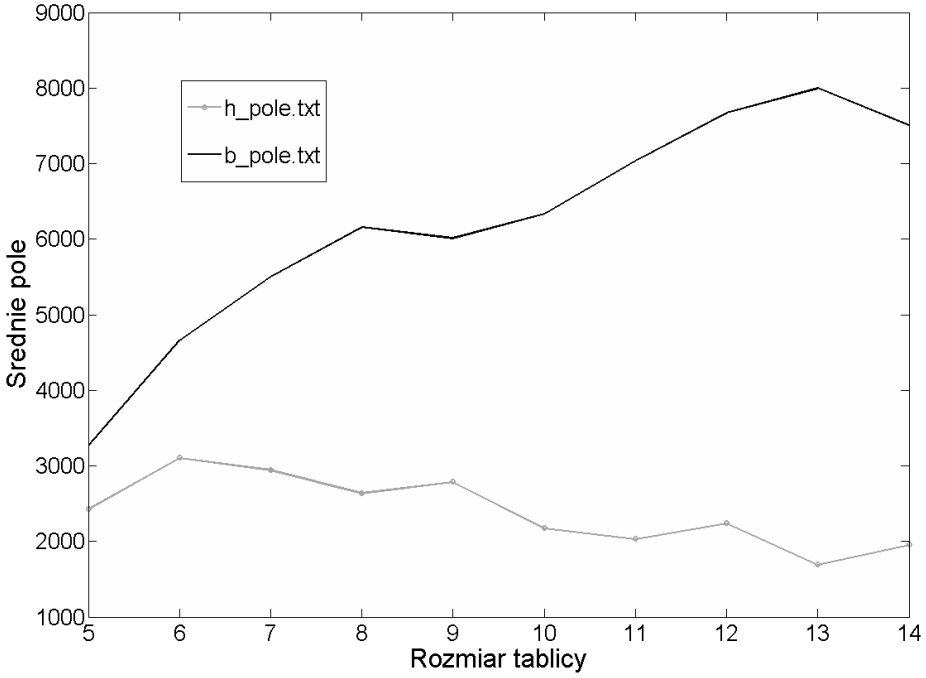


Wstępne przetwarzanie, ze względu na swoją liniową złożoność, okazało się oczywiście wirtualnym liderem pomiaru czasu. By lepiej uwidocznić różnice pomiędzy pomiarami,

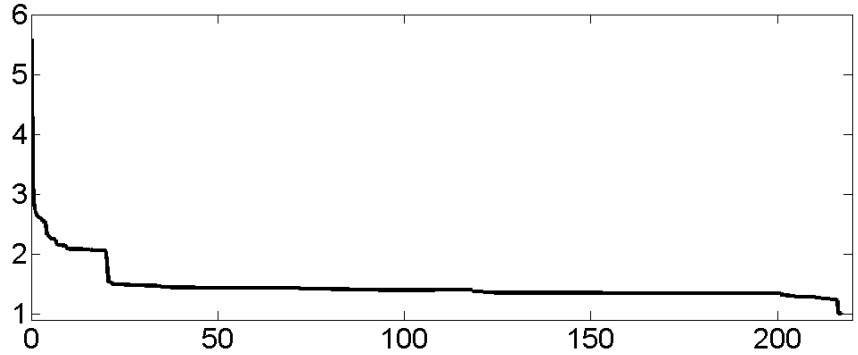
wykonano wykres w skali logarytmicznej.



Oczywiście, jak już wspominano, kosztem skróconego czasu przetwarzania jest dokładność. O ile dla trywialnych dla dzisiejszych systemów rozmiarów instancji metoda biegunowa daje efekty, które można uznać za akceptowalne, to wraz ze wzrostem ilości okrążanych punktów skuteczność znacznie maleje.



Nie sposób zbadać pod tym kątem algorytmu heurystyczno-wyczerpującego, ponieważ tak czy inaczej kończy się on znalezieniem optymalnego pola. W tym przypadku konieczna była analiza danych pochodzących z pośrednich wyników. Uśredniony z 15 iteracji przebieg docierania do optymalnego rozwiązania (liczby na osi pionowej oznaczają jego wielokrotności) prezentuje się następująco:



Jak widać, zgodnie z założeniami w początkowej fazie działania algorytmu pole szybko dąży do optymalnego, zaś wraz z upływem czasu zmiany stają się coraz wolniejsze.

5 Podsumowanie

Zadanie należało do jednych z najciekawszych problemów, z jakimi spotkaliśmy się podczas naszej akademickiej kariery. Główną tego przyczyną był fakt, iż analizowano problem niesztampowy, należało również opracować własne rozwiązanie. Po rozważaniach nad ideami algorytmów i długiej pracy implementacyjnej (oraz równie ważnej i niekrótszej testowej) powstało ponad 3000 linii kodu. Ów zorganizowany jest w sposób obiektowy, co znacząco zwiększa jego czytelność i umożliwia łatwe wykorzystanie fragmentów rozwiązania przy innych problemach. Łatwo bowiem zauważyć, ze zagadnienia takie jak liczby wymierne z dużą dozą prawdopodobieństwa znajdą zastosowanie w wielu projektach informatycznych – ostatecznie tego typu uniwersalne klasy przeniesione zostały do biblioteki `szprotka`.

Należy także nadmienić, iż dla algorytmu rozwiązujący tę klasę problemów niezwykle ciężko znaleźć praktyczne zastosowanie. W praktyce powstające wielokąty mają tak wymyślne kształty, że ciężko określić jakiekolwiek ich zastosowanie. Ewentualnie dla bardzo specyficznych instancji algorytm może znaleźć zastosowanie przy minimalizacji kosztu związanego na przykład z zakupem ziemi ze względu na pewne określone lokalizacje rozrzucone po okolicy bądź prowadzeniem robót ziemnych (minimalizacja pracy wykonanej przez koparkę). W ostateczności wynik programu można wykorzystać jako sympatyczną tapetę na pulpit.