System rozwiązujący krzyżówki z podpowiedziami w języku naturalnym

(A solver for crosswords with hints in natural language)

Piotr Gdowski

Praca inżynierska

Promotor: dr Paweł Rychlikowski

Uniwersytet Wrocławski
Wydział Matematyki i Informatyki
Instytut Informatyki

28 czerwca 2019 r.

Streszczenie

W tej pracy opiszemy techniki rozwiązywania klasycznych krzyżówek, których podpowiedzi do haseł są w języku naturalnym. Skorzystamy z techniki zanurzeń słów do otrzymania wektorowej reprezentacji każdego wyrazu. Rozważymy różne sposoby obliczania wektora dla całego zdania, wspierając się częstotliwościami występowania wyrazów w języku. Pokażemy, w jaki sposób uzupełniać krzyżówkę, by szybko znaleźć jej rozwiązanie. Na koniec zaprezentujemy wyniki działania proponowanego algorytmu na przypadkach testowych. Wykonana analiza ukazuje przykład wykorzystania techniki zanurzeń słów do rozwiązywania łamigłówek opartych o język naturalny.

In this work, we describe how to solve a crossword, which has hints for words in a natural language. At first, we use a word embeddings technique to create a mapping from each word in the language to a multidimensional vector. We discuss some ideas for calculating sentence vector, using precomputed word frequencies. We present a method to find a crossword's solution quickly. At the end, we test proposed algorithm on real puzzles and comment the results. Perfomed analysis show an example of using word embeddings in natural language based puzzles.

Spis treści

Rozdział 1.

Wprowadzenie

Krzyżówka to łamigłówka słowno-literowa polegająca na wpisywaniu słów w odpowiednie kratki na podstawie dołączonych do diagramu podpowiedzi. Najczęściej pojawia się jako dodatek do publikowanych gazet i magazynów. Łatwość konstruowania diagramu oraz dobierania poziomu trudności sprawiły, że osoba w niemal dowolnym wieku mogła zostać odbiorcą tej łamigłówki.

1.1. Jak ludzie rozwiązują krzyżówkę?

Zastanówmy się, jak przeciętny amator łamigłówek będzie rozwiązywał dany diagram. Najczęstszą techniką jest rozpoczęcie od wpisywania haseł, których jest się pewnym, że znajdą się w rozwiązaniu. Alternatywnie, można również próbować wpisać na początku najdłuższe słowa, gdyż dają one najwięcej informacji dla kolejnych haseł, a ewentualna sprzeczność wyjdzie na jaw dość szybko. W ten sposób zazwyczaj daje się wypełnić około połowy diagramu. Gdy skończymy ten proces, przechodzimy do kolejnej metody, jaką jest zgadywanie słów w oparciu o częściową informację płynącą z liter wyrazów już wpisanych. Jeśli znamy ponad połowę liter w danym słowie, najczęściej jesteśmy w stanie je odtworzyć w całości, nawet niezależnie od treści podpowiedzi. W ten sposób znajdujemy hasła, które na początku były za trudne do odgadnięcia i w rezultacie stworzymy całe rozwiązanie.

1.2. Jak proponowany system będzie rozwiązywać krzyżówkę?

Spójrzmy na tę łamigłówkę z innej perspektywy. Zauważmy, że z jednej strony możemy ją traktować jako *problem spełnialności więzów*[1]. Druga część łamigłówki, polegająca na odnalezieniu pasujących słów na podstawie podpowiedzi, to problem z zakresu *przetwarzania języka naturalnego*[2].

1.2.1. Krzyżówka jako problem spełnialności więzów

Typowy problem więzowy składa się z trzech zbiorów - *zmiennych*, którym będziemy nadawać wartości, *dziedzin*, czyli zbiorów wartości dla każdej zmiennej oraz *więzów*, które określają, jakie konfiguracje wartości są niedozwolone w oczekiwanym rozwiązaniu. Wartościowanie spełniające wszystkie więzy to poprawne rozwiązanie. W naszym przypadku, zmiennymi będą puste luki do uzupełnienia, dziedziny to listy potencjalnie pasujących słów określonej długości, a więzy to wszystkie skrzyżowania wyrazów w diagramie - nie chcemy bowiem, by w kratkę była wpisana więcej niż jedna litera. Na tej części łamigłówki skupimy się w rozdziale 3.

1.2.2. Krzyżówka jako przetwarzanie języka naturalnego

W łamigłówce oprócz diagramu pojawia się aspekt językowy, jakim jest lista podpowiedzi do haseł. Zrozumienie podpowiedzi znacząco usprawni proces znajdowania pasujących słów oraz poprawi jakość otrzymywanych rozwiązań. Omówimy to w rozdziale 2.

1.2.3. Krzyżówka jako problem zabawkowy

Rozwiązywanie krzyżówek można traktować jako *toy problem*, czyli jako mało skomplikowany podproblem pewnego większego zadania, systemu. Pozwala również na łatwiejsze zrozumienie lub testowanie metodologii dla innych problemów.

Rozdział 2.

Przetwarzanie podpowiedzi do haseł

Najpierw chcielibyśmy stworzyć zestaw z danymi, na którym będziemy pracować. Wykorzystamy pomysł na reprezentację słów jako wektory o ważnej własności - wyrazy, które są w jakiś sposób do siebie zbliżone (na przykład mogą występować w tych samych kontekstach w zdaniach), będą posiadać w pewnym sensie podobne wektory. Dzięki temu, operację łączenia słów w jedno zdanie będzie można potraktować jak dodawanie wektorów w pewnej przestrzeni.

2.1. Tworzenie wektorów dla słów

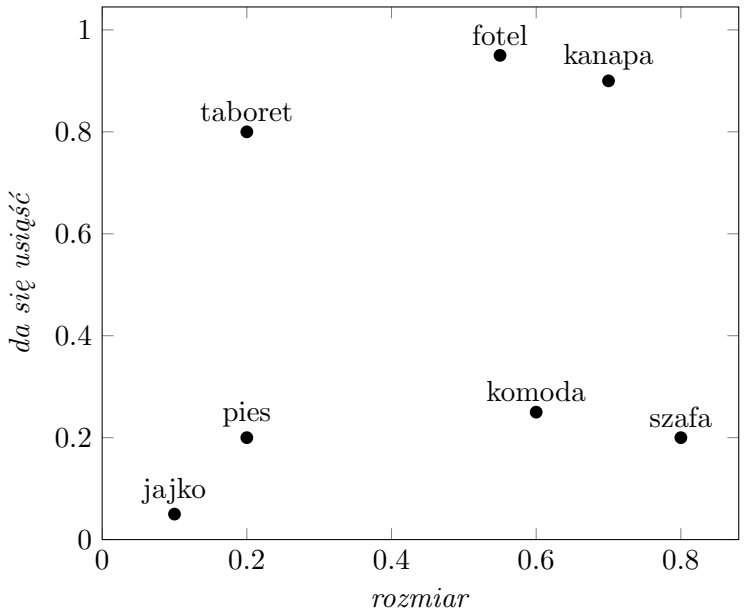
2.1.1. Wstępne intuicje

Najprostsza realizacja tego pomysłu jest następująca: niech każda współrzędna wektora zawiera informację o tym, jak bardzo posiada pewną cechę. Przykładowo, niech takimi kategoriami będą *rozmiar* oraz *da się usiąść*. Wówczas dla słów *kanapa*, *jajko*, *taboret*, *komoda*, *szafa*, *fotel* wartości współrzędnych mogą być takie, jak w tabeli 2.1:

Tablica 2.1: Przykładowe wektory pojedynczych słów

	rozmiar	*da się usiąść*
kanapa	0.7	0.9
szafa	0.8	0.2
jajko	0.1	0.05
taboret	0.2	0.8
komoda	0.6	0.25
fotel	0.55	0.95
pies	0.2	0.2

Dla słów z jednej grupy, w tym wypadku *mebli*, wybór takich kategorii jest nienajgorszy - można dostrzec, że słowa *kanapa* i *fotel* są sobie bliskie w interpretacji geometrycznej, podobnie jak para *komoda* i *szafa*. Natomiast dla innych słów, niekoniecznie związanych z *meblami*, możemy wyciągnąć błędne wnioski - *pies* i *jajko* nie mają za wiele wspólnego. W formie graficznej prezentujemy to zjawisko na rysunku 2.1. Aby to skorygować, należałoby dodać nową cechę (na przykład *czy da się zjeść*) oraz obliczyć jej wartości dla każdego słowa, co jest dość żmudne. Oczywiście dodawane współrzędne same w sobie nie muszą mieć wprost semantycznego sensu, lecz być częścią składową innych cech. Widzimy, że poprawne obliczanie wektorów dla słów nie jest prostym zadaniem.



Rysunek 2.1: Graficzna reprezentacja słów jako punkty w przestrzeni

2.1.2. Znajdowanie wektorów na podstawie korpusu językowego

Innym podejściem do problemu obliczania wektorów dla słów jest stworzenie modelu sieci neuronowej, bazującego na Word2Vec[3], o dwóch możliwych architekturach: *Continuous Bag of Words* oraz *Skip-gram*. W pierwszej z nich, model przewiduje słowo na podstawie kontekstów (okolicznych słów), w których się pojawia. W drugiej architekturze model użyje obecnego słowa do znalezienia potencjalnych kontekstów tego słowa. Jedną z implementacji modelu jest fastText [4], z której będziemy korzystać w dalszej części pracy.

2.2. Obliczanie wektorów zdań

Skupimy się teraz na informacjach dostarczonych przez instancję problemu, jaką jest krzyżówka. Oprócz samego diagramu i układu luk, otrzymujemy listę podpowiedzi, które są w postaci zdań lub równoważników zdań. Chcielibyśmy na podstawie takiej podpowiedzi znaleźć najbardziej pasujące słowo. Korzystając ze stworzonej już reprezentacji słów jako wektorów, moglibyśmy obliczyć średnią wektorów spośród słów występujących w zdaniu. Ilustruje to algorytm 2.2. Taka metoda nie jest do końca zła, lecz bardzo dużo szumu informacyjnego zostanie przeniesione przez słowa występujące w każdym kontekście np. zaimki czy spójniki. Oczywistym pomysłem jest zatem pomijanie słów z predefiniowanej przez nas listy słów. Tą ideę da się jednak usprawnić.

```
procedure AVERAGE(listOfWords)
    resultVector ← empty vector
    for word ∈ listOfWords do
        wordVector ← GETWORDVECTOR(word)
        resultVector ← resultVector + wordVector
    end for
    resultVector ← resultVector / SIZE(listOfWords)
    return resultVector
end procedure
```

Rysunek 2.2: Algorytm liczący średnią z wektorów słów tworzących zdanie

Dla większości języków nowożytnych znane są średnie częstotliwości wystąpienia w nich słów (na przykład dla języka angielskiego [5]). Z tą wiedzą możemy łatwo wyznaczyć listę najczęściej występujących wyrazów i w czasie obliczania wektora zdania po proste je ignorować. Metoda ignorująca 100 najpopularniejszych słów została zilustrowana na rysunku 2.3. Zauważmy, że w ten sposób, oprócz dyskryminowania niektórych wyrazów, możemy również promować te rzadko występujące, niosące spory ładunek informacyjny. W tym celu skorzystamy ze wzoru na *Inverted Document Frequency*[6] i przeskalujemy każdy wektor wyrazu ze zdania przez odwrotność logarytmu częstotliwości tego słowa. Ten pomysł realizuje procedura 2.4. Nic nie stoi na przeszkodzie, by połączyć oba pomysły w jednym algorytmie - zostało to zrealizowane na rysunku 2.5.

```
procedure Ban100(listOfWords)
    resultVector ←resultVector
    counter ← 0
    for word ∈ listOfWords do
        if word ∉ listOf100MostPopularWords then
            wordVector ← GetWordVector(word)
            resultVector ← resultVector + wordVector
            counter ← counter + 1
        end if
    end for
    resultVector ← resultVector / counter
    return resultVector
end procedure
```

Rysunek 2.3: Algorytm liczący średnią z wektorów słów tworzących zdanie pomijający wyrazy z listy stu najbardziej popularnych

```
procedure FreqLog(listOfWords)
    resultVector ← empty vector
    for word ∈ listOfWords do
        wordVector ← GetWordVector(word)
        wordVector ← wordVector / GetPopularity(word)
        resultVector ← resultVector + wordVector
    end for
    resultVector ← resultVector / Size(listOfWords)
    return resultVector
end procedure
```

Rysunek 2.4: Algorytm liczący średnią ważoną z wektorów słów tworzących zdanie, z wagą będącą odwrotnością logarytmu popularności

```
procedure Mixed(listOfWords)
    resultVector ← empty vector
    counter ← 0
    for word ∈ listOfWords do
        if word ∉ listOf100MostPopularWords then
            wordVector ← GetWordVector(word)
            wordVector ← wordVector / GetPopularity(word)
            resultVector ← resultVector + wordVector
            counter ← counter + 1
        end if
    end for
    resultVector ← resultVector / counter
    return resultVector
end procedure
```

Rysunek 2.5: Algorytm liczący wektora zdania korzystający z obu pomysłów

Uwaga. Może się zdarzyć, że dla niektórych danych wejściowych metody 2.3 i 2.5 zwrócą puste wektory. Będzie to oznaczać, że lista dyskryminowanych słów jest za duża, wobec czego aby uzyskać niezerowy wynik, należy ograniczyć rozmiar tej listy

2.3. Obliczanie odległości między wektorami

Chcielibyśmy jakoś określać, kiedy dwa wektory są do siebie podobne. Naturalną taką miarą jest cosinus kąta między nimi. Dla wektora słowa $\vec{u}$ i zapytania $\vec{q}$ wartość ta wyraża się wzorem:

$$rank_u = \cos(\vec{u}, \vec{q}) = \frac{\vec{u} \cdot \vec{q}}{||\vec{u}|| \cdot ||\vec{q}||} \tag{2.1}$$

Zauważmy, że jeśli wektory są *L2*-znormalizowane, wówczas długości tych wektorów są stałe i równe 1, a zatem wystarczy policzyć ich iloczyn skalarny. Dla większej liczby słów w zbiorze można użyć mnożenia macierzy przez wektor zapytania tj.

$$\begin{bmatrix} wektor_1 \\ wektor_2 \\ \vdots \\ wektor_n \end{bmatrix} \times \vec{q} = \begin{bmatrix} rank_1 \\ rank_2 \\ \vdots \\ rank_n \end{bmatrix} \tag{2.2}$$

Użycie wzoru 2.2 pozwala na skorzystanie z algorytmów szybkiego mnożenia macierzy, na przykład algorytmu Strassena[7], które są efektywniejsze od bezpośredniego obliczania podobieństwa ze wzoru 2.1.

2.3.1. Zestaw danych używany do testów

Na potrzeby testów przygotowaliśmy dane złożone z 600 000 pierwszych wektorów z [4] oraz listę częstotliwości występowania słów z [5].

2.3.2. Porównanie metod obliczających wektor zdania

Mamy zatem miarę oceny podobieństwa dwóch wektorów. W tabeli 2.2 prezentujemy wartości podobieństw pomiędzy wektorami zdań obliczonych przy pomocy metod 2.2, 2.3, 2.4 oraz 2.5 a wektorami słów, obliczonych przy pomocy wzoru 2.1.

Tablica 2.2: Tabela porównująca wyniki metod obliczających wektory zdań dla wybranych danych przy użyciu 2.3.1.

Podpowiedź do hasła	Hasło	Average	Ban100	FreqLog	Mixed
Unnatural beast	monster	**0.668**	**0.668**	0.6587	0.6587
Damaged by fire	burnt	0.6324	**0.6711**	0.6516	0.6704
Obtainable or accessible	available	0.7303	**0.7684**	0.762	0.768
It may have a list of dishes	menu	0.5006	0.6254	0.5355	**0.6274**
Home for pet fish	aquarium	0.5596	0.6097	0.5837	**0.6149**
Trip on an aircraft	flight	0.6555	0.6942	0.6958	**0.7005**
Meal in the middle of the day	lunch	0.5561	0.6948	0.6002	**0.7018**
Something with limbs and roots	tree	0.5469	0.5957	0.5776	**0.6037**
Platform enclosed by a railing or balustrade	balcony	0.6131	0.6895	0.6616	**0.6975**
Full of clouds	sky	0.5709	0.6238	0.6077	**0.6408**
Month before april	march	0.53	0.5326	**0.5612**	0.5532
Citrus fruit	grapefruit	0.7096	0.7096	**0.731**	**0.731**
Aviation complex	airport	0.5655	0.5655	**0.5905**	**0.5905**
Średni wynik		0.6030	0.6499	0.6320	**0.6583**

Widać, że choć w większości przypadków metody *Mixed* i *FreqLog* dają lepsze rezultaty, to *Average* mimo prostoty nie odstępuje znacząco od pozostałych. W dalszej części pracy będziemy używać tylko procedury *Mixed*.

2.4. Listy słów pasujących do podpowiedzi

Umiemy już stwierdzić, jak bardzo pojedyncze słowo pasuje do wybranej podpowiedzi. Teraz wszystko co trzeba zrobić, to dla każdego słowa obliczyć wzorem 2.2 podobieństwo, a następnie posortować malejąco po tej wartości. W tym miejscu warto zauważyć, że w krzyżówce mamy zawartą informację o długości hasła, zatem wystarczy wykonać obliczenia na właściwym podzbiorze słów. W tabeli poniżej prezentujemy przykładowe wyliczone listy słów z użyciem metody 2.5.

Tablica 2.3: Tabela zawierająca przykładowe listy słów dla podpowiedzi obliczone na danych 2.3.1.

trip on an aircraft	citrus fruit	something with limbs and roots	home for pet fish
planes 0.72	**grapefruit** 0.73	legs 0.64	wildlife 0.63
flight 0.7	**tangerines** 0.61	root 0.64	goldfish 0.63
flying 0.66	strawberry 0.61	ways 0.62	**aquarium** 0.61
pilots 0.63	watermelon 0.6	limb 0.61	reptiles 0.61
voyage 0.61	vegetables 0.58	feet 0.6	creature 0.61
hangar 0.61	eucalyptus 0.57	**tree** 0.6	business 0.59
cruise 0.6	**clementine** 0.57	ones 0.6	habitats 0.58
convoy 0.59	persimmons 0.55	soul 0.6	chickens 0.58
engine 0.58	sweet-tart 0.55	word 0.58	backyard 0.58
routes 0.58	**pineapples** 0.54	idea 0.58	critters 0.58

Słowa wytłuszczone to potencjalnie pasujące hasła w krzyżówce.

Rozdział 3.

Rozwiązywanie krzyżówki

W poprzednim rozdziale opisaliśmy metodę tworzenia list słów pasujących do odpowiednich luk wraz z wartościami podobieństwa do haseł. Innymi słowy - określiliśmy dziedziny zmiennych więzowych. Teraz zajmiemy się sposobem rozwiązywania więzów, czyli zaproponujemy kolejność zmiennych do ukonkretnienia w krzyżówce, metodę na znajdowania pełnego rozwiązania oraz ideę, kiedy powinniśmy przestać szukać dalszych rozwiązań.

3.1. Wybór kolejności luk do wypełnienia

W ogólnym podejściu w każdej iteracji algorytmu wybieralibyśmy lukę do wypełnienia oraz hasło, które się tam znajdzie. Załóżmy, że mamy rozłączne luki l_1 i l_2 oraz odpowiadające im pewne pasujące słowa v i u. Wpisując najpierw v do l_1, a potem u do l_2 otrzymamy taki sam stan jak w sytuacji, gdy wykonamy te dwa kroki na odwrót. To prowadzi do wniosku, że możemy na samym początku narzucić kolejność wybierania kolejnych luk do wypełnienia, mocno zawężając możliwości w pojedynczym stanie. Jak zatem dobrze wybrać tą kolejność? Inspiracją będzie technika rozwiązywania więzów zwana *First Fail*[1]. Chcielibyśmy skorzystać z faktu, że niektóre słowa przecinają się ze sobą, przez co wpisując jedno słowo, ograniczamy przestrzeń możliwych słów dla luk przecinających wpisane słowo. Z drugiej strony warto zacząć od słowa, które z pewnością znajdzie się w diagramie. Proponujemy następujący algorytm: pierwsza luka do wypełnienia to taka, która przecina się z największą liczbą innych luk, a dodatkowo suma wartości podobieństw pierwszych słów dla tej luki jest największa. Następne są dobierane tak, by przecinały się z największą liczbą wybranych już luk. W ten sposób każde kolejne wpisywane słowo będzie się krzyżowało z innym już wpisanym. Ilustruje to algorytm 3.1.

```
procedure CountCrossings(fixedHole, listOfHoles)
    counter ← 0
    for hole ∈ listOfHoles do
        if hole crosses fixedHole then
            counter ← counter + 1
        end if
    end for
    tieBreaker ← ∑_{i=1}^{5} GetWordsForHole(fixedHole)[i].rank
    return (counter, tieBreaker)
end procedure

procedure GetNextHole(candidates, listOfHoles)
    return ARGMAX_{cand∈candidates} CountCrossings(cand, listOfHoles)
end procedure

procedure OrderHoles(holes)
    firstHole ← GetNextHole(holes, holes)
    resultList ← [firstHole]
    holes ← holes \ firstHole
    while holes not empty do
        nextHole ← GetNextHole(holes, result)
        resultList ← resultList + [nextHole]
        holes ← holes \ nextHole
    end while
    return resultList
end procedure
```

Rysunek 3.1: Algorytm wyboru kolejności wypełniania haseł w krzyżówce

3.2. Szukanie rozwiązania

Chcąc znaleźć rozwiązanie, musimy narzucić jakiś sposób ukonkretniania kolejnych zmiennych. Można to rozumieć jako chodzenie po drzewie, w których wierzchołki to zbiory zmiennych z przypisanymi już wartościami, a krawędź będzie odpowiadać pojedynczemu nadaniu zmiennej pewnej wartości. Naszym celem byłoby wówczas dotarcie do liścia, w którym wszystkie zmienne są już ukonkretnione.

3.2.1. Backtracking

Najprostszym pomysłem jest technika znana jako *backtracking*[1], która odpowiada przeszukiwaniu drzewa w głąb. Oznacza to, że w każdej iteracji wybieramy dowolną zmienną, kolejno przypisujemy jej wszystkie możliwe wartości z dziedziny i próbujemy rozwiązać pozostałe zmienne. Taka metoda nie jest zbyt efektywna, jest jednak prosta w implementacji.

3.2.2. Przeszukiwanie z uwzględnieniem wartości podobieństw

Oprócz samego znalezienia rozwiązania, interesuje nas jego jakość. Miarą tej jakości będzie suma podobieństw wpisanych słów do podpowiedzi, określona w równaniu 2.1. Inspiracją dla proponowanego rozwiązania jest algorytm $A*$[1]. Na podstawie danych z częściowo wypełnionej krzyżówki będziemy chcieli stwierdzać, jak dobrze jest ona uzupełniona. Informacjami, jakie możemy przekazać takiej oceniającej funkcji są wartości podobieństw oraz liczba wpisanych już haseł. Chcielibyśmy, by funkcja promowała użycie słów z wysokimi wartościami podobieństw, a także by nie dopuszczała do dobierania pojedynczych słów o niskim rankingu, które będą miały niewielki związek z podpowiedzią, w celu szybkiego znalezienia rozwiązania, które zapewne będzie częściowo nieprawidłowe. Powyższe idee zostały zilustrowane w procedurze 3.2, która przyjmuje *listOfUsedWordRanks*, czyli listę wartości podobieństw słów, które już zostały wpisane, i zwraca liczbę.

```
procedure RankCrossword(listOfUsedWordRanks)
    if listOfUsedWorkRanks is empty then
        return ∞
    end if
    squareSum ← 0
    for ranking ∈ listOfUsedWordRanks do
        squareSum ← squareSum + ranking * ranking
    end for
    return 15 - squareSum - 10 * Min(listOfUsedWordRanks)
end procedure
```

Rysunek 3.2: Metoda oceniająca częściowo wypełnioną krzyżówkę

Posiadając taką funkcję, możemy na każdym kroku wybierać krzyżówkę, która jest *najlepiej* wypełniona, i do niej próbować wpisywać kolejne słowa. W ten sposób znajdziemy rozwiązanie, które jednocześnie będzie zadowalającej jakości.

3.3. Zakończenie przeszukania

Liczba potencjalnych rozwiązań krzyżówki rośnie wykładniczo wraz z liczbą luk do uzupełnienia, dlatego chcielibyśmy zakończyć wyszukiwanie w momencie, gdy uznamy, że nie poprawimy już rozwiązania. Jednym pomysłem na zaradzenie temu problemowi jest zaprzestania wyszukiwania w momencie, gdy od dłuższego czasu znajdowane rozwiązania nie są lepsze w sensie średniej podobieństw wpisanych słów od już znalezionego. Takie podejście zakłada jednak, że rozwiązania krzyżówek są rozłożone z rozkładem jednostajnym w liściach drzewa, co nie zawsze jest prawdą. Dlatego bezpieczniej byłoby zaprzestać dalszego przeszukania po określonym odstępie czasu bez poprawy rozwiązania niezależnie, czy jakiekolwiek znaleźliśmy. Takie podejście ilustruje następujący algorytm 3.3:

```
procedure FINISH(bestSolutionIteration, iteration, maxNoProgressIterations)
    if bestSolutionIteration < 0 then
        return false
    end if
    if bestSolutionIteration + maxNoProgressIterations ≤ iteration then
        return true
    end if
    return false
end procedure
```

Rysunek 3.3: Heurystyka zakończenia przeszukiwania

3.4. Podsumowanie algorytmu

Prezentujemy cały algorytm przeszukiwania wykorzystując metody 2.5, 3.1, 3.2 oraz 3.3. Przyjmowanymi arugmentami są *crossword* - pusta krzyżówka, *holes* - lista luk z pasującymi wyrazami oraz *maxNoProgressIterations* - stała określająca, po ilu iteracjach bez poprawy rozwiązania należy zakończyć działanie. Kolejka priorytetowa Q z porządkiem rosnącym będzie zawierać krotki, których składowymi będą: wartość funkcji 3.2 oceniającej diagram, listę wartości podobieństw słów już wpisanych, diagram oraz numer luki do uzupełnienia. Wartością priorytetu w kolejce będzie pierwsza współrzędna krotki. Metody *CanWriteInto* oraz *WriteInto* naturalnie oznaczają kolejno sprawdzenie, czy słowo da się wpisać do krzyżówki w lukę o zadanym numerze, oraz operacja wpisania wyrazu do diagramu.

```
procedure SolveCrossword(crossword, holes, maxNoProgressIterations)
    holes ← OrderHoles(holes)
    Q ← PriorityQueue()
    holeIndex ← 1
    rank ← RankCrossword([])
    Q.put((rank, [], crossword, holeIndex))
    iteration ← 0
    solutionIteration ← -1
    solution ← []
    bestSolutionValue ← rank
    while Q is not empty do
        iteration ← iteration + 1
        if Finish(solutionIteration, iteration, maxNoProgressIterations) then
            return solution
        end if
        (ranking, wordRanks, crossword, holeIndex) ← Q.pop()
        if holeIndex > Size(holes) then
            if ranking < bestSolutionValue then
                solutionIteration ← iteration
                solution ← crossword
                bestSolutionValue ← ranking
            end if
        end if
        for word ∈ Mixed(holes[holeIndex]) do
            if CanWriteInto(crossword, word, holeIndex) then
                newCrossword ← WriteInto(crossword, word, holeIndex)
                newWordRanks ← word.rank ∪ wordRanks
                newRank ← RankCrossword(newWordRanks)
                Q.put((newRank, newWordRanks, newCrossword, holeIndex + 1))
            end if
        end for
    end while
end procedure
```

Rysunek 3.4: Algorytm rozwiązujący krzyżówkę

Rozdział 4.

Wyniki

W tym rozdziale zaprezentujemy wyniki osiągnięte przy pomocy proponowanego w pracy algorytmu. Do testów wykorzystany został zestaw wektorów i stałe opisane w podrozdziale 4.1.

4.1. Zestaw danych i użyte parametry

Użyty zestaw danych jest taki sam, jak opisany w 2.3.1. Dodatkowo, w listach słów dla każdej podpowiedzi ignorowane były słowa, których podobieństwo było mniejsze niż 0.39. W metodzie sprawdzającej warunek końca przeszukiwania maksymalna liczba iteracji bez poprawy wartości rozwiązania to 50 000.

4.2. Testy

Porównamy wyniki działania na czterech krzyżówkach - trzy spośród nich zostały ułożone ręcznie i znajdują się na rysunkach A2, A3 i A4, czwarta została stworzona i opublikowana na łamach magazynu The Guardian[8], i znajduje się na rysunku A1.

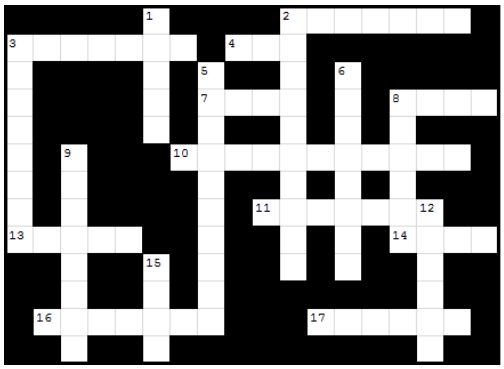
W tabeli 4.1 przedstawiamy krótką charakterystykę każdej z nich - rozmiar, liczba słów do wpisania czy suma pustych pól. Dodatkowo zawarte są informacje, jak trudna może być krzyżówka do rozwiązania. Znajdziemy tu dane o sumie słów w podpowiedziach, średniej pozycji pożądanych słów (z oryginalnego rozwiązania) na wygenerowanych listach pasujących wyrazów oraz średniej wartości ich podobieństw do odpowiednich wektorów podpowiedzi, a także średniej trafności słów, znanej jako *Mean Reciprocal Rank*[9], zadanej wzorem:

$$MRR = \frac{1}{|holes|} \cdot \sum_{word \in solution} \frac{1}{IndexOnList(word)} \tag{4.1}$$

Tablica 4.1: Tabela porównująca krzyżówki

	Łatwa	Średnia	Trudna	The Guardian[8]
Rozmiar	13x18	11x15	12x14	13x13
Liczba haseł	20	20	20	20
Liczba pustych pól	107	104	91	106
Liczba słów w podpowiedziach	49	54	65	45
MRR	0.517	0.441	0.589	0.388
Średnia pozycja słowa na liście	4.6	5.1	5.45	67.7
Średni ranking słów w rozwiązaniu	0.6283	0.6327	0.6245	0.5691

Diagramy są zbliżone rozmiarami, potencjalna trudność może być odwzorowana w długości podpowiedzi (im więcej słów, tym bardziej wynikowe słowa mogą być niezwiązane z podpowiedzią) oraz średnią pozycją słowa na liście. Przykładowa krzyżówka została pokazana na rysunku 4.1.



Poziomo
2 Unnatural beast
3 Cautious
4 Full of clouds
7 Plant base
8 Stupid person
10 Possibility
11 After this evening
13 Meal in the middle of the day
14 Something with limbs and roots
16 Aviation complex
17 Salt and …

Pionowo
1 Damaged by fire
2 Cryptic
3 Unlawful
5 Citrus fruit
6 Answer
8 Trip on an aircraft
9 Natural elevation
12 Take a journey
15 House top

Rysunek 4.1: Łatwa krzyżówka

4.3. Osiągnięte rezultaty

W tabeli 4.2 prezentujemy dane zwrócone przez program - ile wykonano iteracji (czyli ile częściowych rozwiązań zostało przeglądniętych), jak dobre znaleziono rozwiązanie (rozumiana jako średnia z wartości podobieństw słów do podpowiedzi) oraz ile słów zostało odgadniętych zgodnie z oczekiwaniami. Dodatkowo umieszczamy informacje o czasie działania algorytmu, rozbitego na czas ładowania wektorów do pamięci oraz samego szukania rozwiązania

Tablica 4.2: Tabela prezentująca wyniki algorytmu na testowych krzyżówkach, przy implementacji A.1.

	Łatwa	Średnia	Trudna	The Guardian[8]
Liczba poprawnie zgadniętych haseł	20	20	13	18
Liczba iteracji	50673	50032	50142	90690
Numer iteracji z najlepszym rozwiązaniem	672	29	141	40689
Wartość rozwiązania	0.6283	0.6327	0.6082	0.5753
Ładowanie wektorów 4.1.	05:45	07:15	06:17	05:49
Rozwiązywanie krzyżówki	01:19	01:24	01:13	02:26
Razem	07:04	08:39	07:30	08:15

Dwie kolejne tablice, 4.3 i 4.4 pokazują przykłady haseł kolejno poprawnie i niepoprawnie odgadniętych, wraz z ich wartościami podobieństw dla błędnych słów.

Tablica 4.3: Tabela z przykładami poprawnie odgadniętych słów

Podpowiedź	Zgadnięte słowo
something with limbs and roots	tree
having job security as a college professor	tenured
tiny marine organism	coral
platform enclosed by a railing or balustrade	balcony
slender mosque tower	minaret
obtainable or accesible	available

Tablica 4.4: Tabela z przykładami błędnie odgadniętych słów

Podpowiedź	Zgadnięte słowo	Rank	Prawidłowe słowo	Rank
home for pet fish	appetite	0.493	aquarium	0.615
manage to get a ladder	try	0.594	run	0.622
snooker ball worth five points	pins	0.405	blue	0.432
it may have a list of dishes	meat	0.521	menu	0.627
typeface with thick heavy lines	code	0.414	bold	0.424
stimulate into action	urge	0.509	prod	0.4

4.4. Wnioski

Pierwszą obserwacją jest fakt, że proponowane rozwiązanie zadziałało na przypadkach testowych. W przypadku łatwiej i średniej krzyżówki udało się znaleźć pełne rozwiązanie, a w pozostałych większość haseł. Co spowodowało, że

część słów nie została odgadnięta? Na brak sukcesu może składać się kilka czynników. Dużą rolę odgrywa tutaj zbiór wektorów, od którego zależy pozycjonowanie słów na listach - nawet drobne zmiany pozycji mogą skutkować zachowaniem funkcji heurystycznej. Ponadto, jeśli oczekiwane słowo ma dość niski ranking (0.5 i mniej), to jest spora szansa, że inne słowa zostaną włączone do rozwiązania zamiast niego. Widać to w rozwiązaniu krzyżówki [8] w tabeli 4.4 - słowo *urge* zostało wybrane zamiast *prod*, by zwiększyć całkowitą wartość rozwiązania. Jeśli mamy wiele takich haseł, wówczas możemy być zmuszeni do przeszukania sporej liczby węzłów, żeby znaleźć jakiekolwiek rozwiązanie - doskonale to prezentuje to przykład krzyżówki z The Guardian. Z drugiej strony, większość haseł jako synonimów podpowiedzi została odgadnięta (na przykład *arrange*), a także dla trudniejszych opisów słów, jak *tenured* czy *coral* algorytm nie był bezradny. Również sama metoda szukania rozwiązania mogła zawieść - w tabeli 4.4 widzimy, że mimo wysokich wartości podobieństw słów *menu*, *run* czy *aquarium* nie weszły one w skład najlepszego rozwiązania. Oznacza to, że algorytm wybrał złe poddrzewo do przeszukania, a potem zakończył przeszukiwanie.

Na koniec warto zauważyć, że wszystkie powyższe rozważania abstrahowały od konkretnego języka. Stąd wniosek, że mając zbiór wektorów oraz listę częstotliwości wystąpień słów możemy używać proponowanych metod do rozwiązywania krzyżówek w dowolnym języku.

Dodatek A

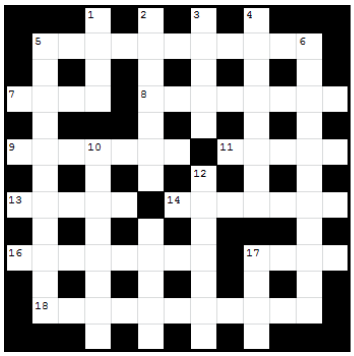
Dodatek

A.1. Implementacja i środowisko uruchomieniowe

Do testów została użyta własna implementacja, napisana w języku *Python 2.7*, dostępna pod adresem `https://github.com/horrorschau105/crossword-solver/tree/0b159ffe904fa7ae6b36ffdb48466ae4bc62d0c6`.
Program był uruchamiany na maszynie wirtualnej z system operacyjnym *Peppermint*, posiadającej 4 GB pamięci RAM.

A.2. Testowe krzyżówki

A.2.1. Diagramy



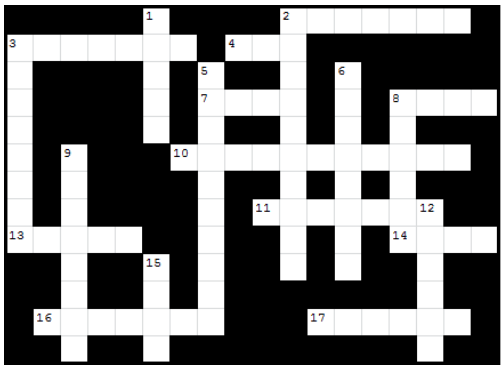
Poziomo

5 Venture owned by business associates
7 Typeface with thick heavy lines
8 Male descendant
9 Slender mosque tower
11 Unwind
13 Compassion for the unfortunate
14 Hold back
16 Exiled
17 Make dirty
18 Occurrence

Pionowo

1 Stimulate into action
2 Laugh scornfully
3 Suffering - litigation
4 Most disreputable
5 Multiply
6 Likelihood
10 Public records
12 Pleasent - soft drink
15 Gunfire
17 Large boat

Rysunek A1: Krzyżówka z The Guardian[1][8]



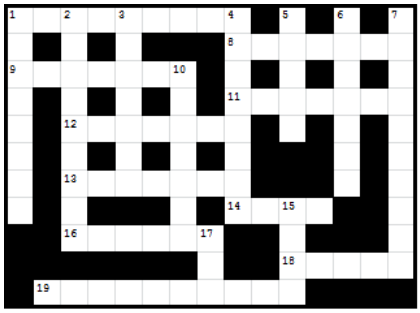
Poziomo

2 Unnatural beast
3 Cautious
4 Full of clouds
7 Plant base
8 Stupid person
10 Possibility
11 After this evening
13 Meal in the middle of the day
14 Something with limbs and roots
16 Aviation complex
17 Salt and …

Pionowo

1 Damaged by fire
2 Cryptic
3 Unlawful
5 Citrus fruit
6 Answer
8 Trip on an aircraft
9 Natural elevation
12 Take a journey
15 House top

Rysunek A2: Łatwa krzyżówka



Poziomo

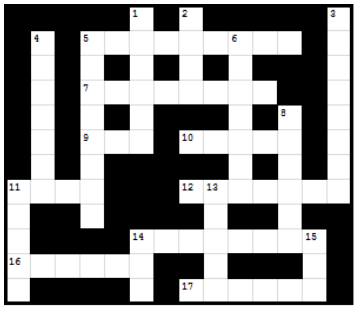
1 Not the same as
8 Obtain an item
9 Competition
11 Variant or revision
12 Place, distribute
13 Heading for, going
14 Make money
16 On the sky
18 Tiny marine organism
19 Shorten, contract

Pionowo

1 Christmas month
2 Marvellous
3 Above eye
4 Pass across
5 Gain points in the game
6 Restricted, confined
7 Important, significant
10 Having job security as a college professor
15 Car competition
17 Glide in the snow

Rysunek A3: Krzyżówka średniej trudności

[1]W oryginale podpowiedzi *14 poziomo* i *4 pionowo* zawierały dodatkowo frazę będącą anagramem hasła do zgadnięcia.



Poziomo

5 Obtainable or accesible
7 Parasol
9 Manage to get a ladder
10 Spirit, spectre
11 It may have a list of dishes
12 Mysterious, obscure
14 Assaulted
16 Oil painting cloth
17 String of onling posts

Pionowo

1 Coal, primarly
2 Snooker ball worth five points
3 Russian alphabet
4 Historical documents repository
5 Home for pet fish
6 Platform enclosed by a railing or balustrade
8 A sculpted figure
11 A month before April
13 Extent of stretch
14 … a stupid question
15 father

Rysunek A4: Trudna krzyżówka

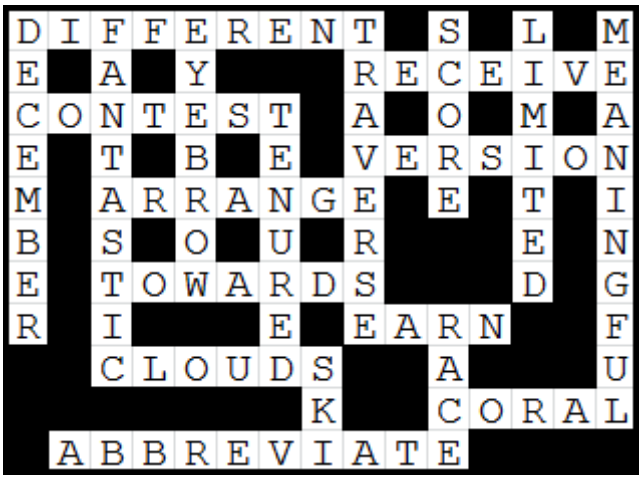
A.2.2. Rozwiązania

Rysunek A5: Rozwiązanie krzyżówki[8]

W rozwiązaniu krzyżówki A5 algorytm zastąpił *prod* oraz *bold* słowami *urge* oraz *code*.

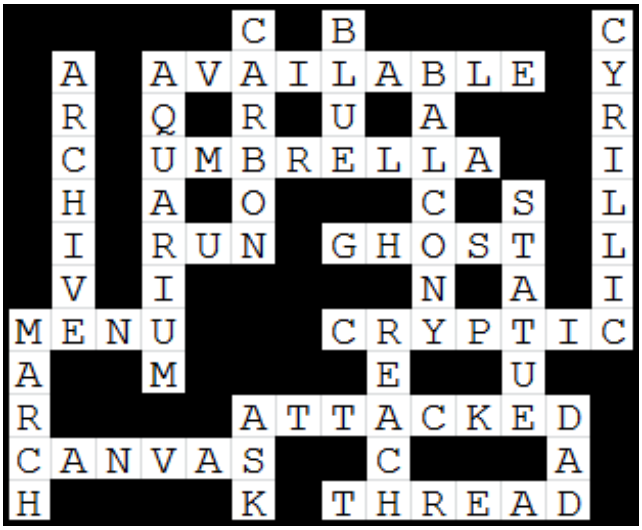
Rysunek A6: Rozwiązanie łatwej krzyżówki

Algorytm poprawnie odgadł wszystkie hasła krzyżówki A6.



Rysunek A7: Rozwiązanie średniej krzyżówki

Algorytm poprawnie odgadł wszystkie hasła krzyżówki A7.



Rysunek A8: Rozwiązanie trudnej krzyżówki

W rozwiązaniu krzyżówki A8 algorytm zastąpił *aqaurium*, *run*, *blue*, *menu*, *umbrella*, *available* oraz *carbon* słowami *appetite*, *try*, *pins*, *meat*, *parasols*, *avaliable* oraz *fairly*.

Bibliografia

[1] Stuart Russell and Peter Norvig. In *Artificial Intelligence: A Modern Approach*, 2010.

[2] Dan Jurafsky and James H. Martin. In *Speech and Language Processing, Third Edition*, 2018.

[3] Tomas Mikolov et al. Efficient estimation of word representations in vector space. 2013.

[4] Piotr Bojanowski, Edouard Grave, Armand Joulin, and Tomas Mikolov. Enriching word vectors with subword information. *arXiv preprint arXiv:1607.04606*, 2016.

[5] Wiktionary. Word Frequencies. `https://en.wiktionary.org/wiki/Wiktionary:Frequency_lists/PG/2006/04/1-10000`, 2006.

[6] Prabhakar Raghavan Christopher D. Manning and Hinrich Schütze. In *Introduction to Information Retrieval*, 2008.

[7] Ronald L. Rivest Thomas H. Cormen, Charles E. Leiserson and Clifford Stein. In *Introduction to Algorithms, Second Edition*, 2001.

[8] The Guardian. `https://www.theguardian.com/crosswords/quick/14256`, 2016.

[9] `https://en.wikipedia.org/wiki/Mean_reciprocal_rank`.