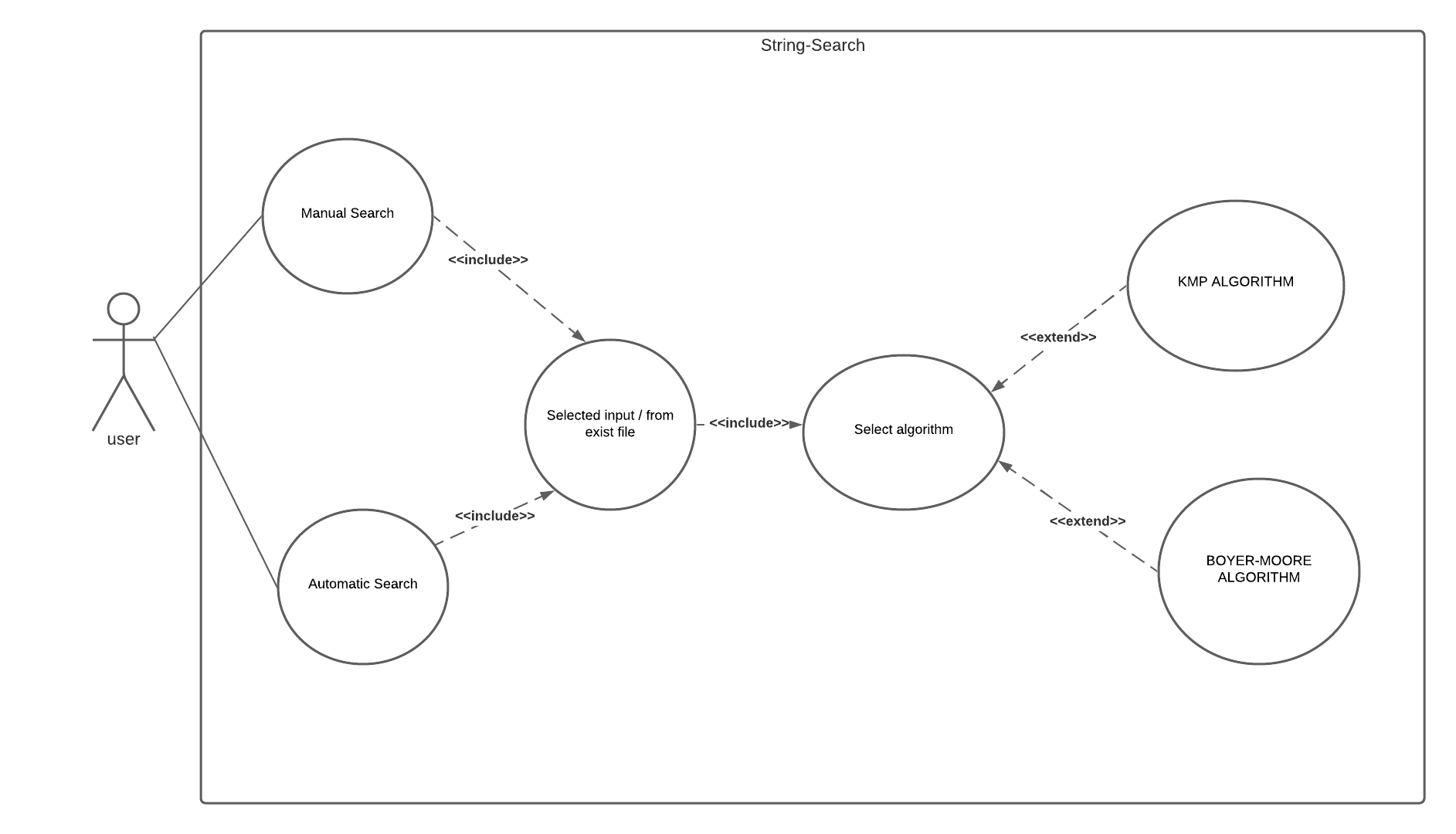
מסמך דרישות מפורט**:**

USE CASE DIAGRAM



**פונקציונאליות:**

בתחילת הרצת התוכנית, המשתמש מתבקש לבחור אחד משני האלגוריתמים.

לאחר בחירת האלגוריתם המשתמש יבחר באחת משתי האפשריות הקימיות :

1. הרצה ידנית צעד לאחר צעד על ידי המשתמש.
2. הרצה אוטומאטית.

\*הערה: נפרט על כל איטרציה של כל אלגוריתם בהתאם.

לאחר מכן המשתמש יבחר את הקלט: קלט מתוך קובץ טקסט קיים או קלט כלשהו לבחירתו, ואז יכניס את המחרוזת הרצויה לחיפוש.

לאחר סיום כל השלבים הללו התוכנית תרוץ לפי בחירות המשתמש בהתאם.

**אלגוריתם Knuth-Morris-Prat :**

מטרתו של האלגוריתם היא למצוא מופעים של מחרוזת Pבתוך מחרוזת T (טקסט) כאשר אורך של P הוא m והאורך של Tהוא n.

בכל פעם שהאלגוריתם יתקל בנקודת חוסר התאמה בטקסט, הוא יחזור להתאים את הרישא הארוכה ביותר של התבנית המתאימה לסיפא הארוכה ביותר של הטקסט בנקודה בו נתקלנו בחוסר התאמה.

**עיבוד מקדים:**

שאלה חשובה עולה מההסבר למעלה היא כמה תווים לקפוץ, כדי לדעת את זה מבצעים עיבוד מקדים למחרוזת בעזרת מערך של שלמים שנותנן לנו את כמות התווים שצריך לקפוץ.

אלגוריתם KMP מעבד מראש מחרוזת ובונה מערך עזר בגודל m (זהה לגודל התבנית) המשמש לדילוג על תווים תוך כדי התאמה. המערך מציין את הקידומת המתאימה הארוכה ביותר שהיא גם סיומת . המחרוזת כלה לא תחשב כקידומת מתאימה.

**איטראציה באלגוריתם זה:**

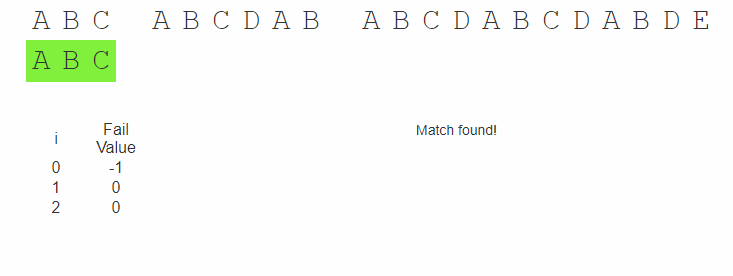
לצורך הדגמה נניח שאנחנו באיטראציה הראשונה, אז בודקים את התו הראשון בטקסט הנבחר (מיקום 0), אם התו הראשון במחרוזת החיפוש שווה לתו הראשון במחרוזת הנבחרת אז ממשיכים לבדוק את התו השני בשתי המחרוזות וכך הלאה עד ש: או שמוצאים כל התווים של המחרוזת (כל התווים שווים בשתי המחרוזות) או שמוצאים אי שוויון.  
במקרה הראשון: המחרוזת אותרה, אז האלגוריתם מודיע על הצלחה וממשיך לחפש את המופע הבא.

במקרה השני: נמצא תווים שונים, אז האלגוריתם מחשב את הקפיצה למיקום התו הבא שצריך ממנו להתחיל את הבדיקה שוב.

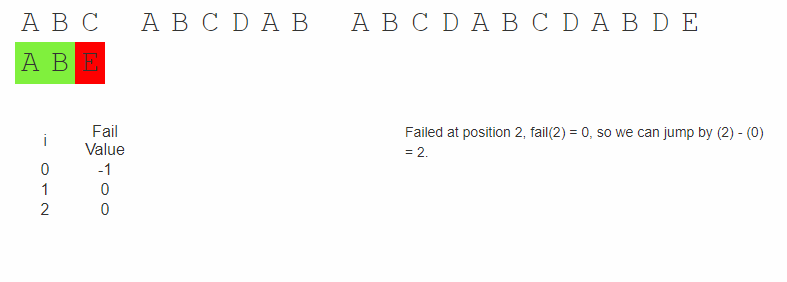
תהליך זה מבוצע עד שנאתר את המחרוזת הרצויה או נגיע לסוף הקלט.

**ויזואליזציה:**

**מקרה ראשון:**

המחרוזת נמצאת, כל תו נמצא נצבע בירוק, וממשיכים לחפש את המופע הבא.

**מקרה שני:**

****נמצא תו שונה אז נצבע באדום ומחושב גודל הקפיצה (התו הבא שממנו נתחיל לבדוק).

על סימון התו שבודקים בכל שלב נצבע אותו בצע מסויים למשל כחול ואז צבע ירוק אם משווים או אדום במקרה של אי שוויון.

הערה: תמונות להמחשה בלבד, יהייה כמה שינויים על העיצוב.

בזמן הויזואליזציה יוצג גם תהליך העיבוד וגם את מערך העזר שחישבנו בתהליך העיבוד שישמש אותנו לצורך חישוב הקפיצה כמו שמתואר על מנת שיהייה ברור למשתמש.

*Reference*: <http://whocouldthat.be/visualizing-string-matching/>

חישוב Fail Value :

לחישוב Fail Value נצטרך מערך שנקרא לו LPS שהגודל שלו יהיה שווה לגודל מחרוזת החיפוש LPS .

הערה: LPS במקום אפס שווה לאפס (לא כמו מה שמצוין בתמונה אנחנו מתחילים עם 0 ולא עם -1 )

אנו מחפשים LPS בתבניות משנה. באופן ברור יותר אנו מתמקדים במיתרי משנה של דפוסים שהם קידומת וסיומת. עבור כל תת-תבנית שמתחילה מאפס עד i כאשר i = 0 עד (m-1) LPS במקום m מאחסן אורך של הקידומת המתאימה המקסימלית שהיא גם סיומת של התבנית LPS .

למשל במקרה השני :

***מחרוזת החיפוש היא* :**

|  |  |  |
| --- | --- | --- |
| A | B | E |

אז המערך LPS :

1. המערך יהיה בגודל של מחרוזת החיפוש כלומר הוא שווה לשלוש
2. LPS[0] שווה לאפס
3. ניקח עכשיו את התת מחרוזת AB בדוק אם יש תת מחרוזת שהיא גם סיומת וגם קידומת מכיוון שיש רק שתי תווים שונים LPS[1] יהיה גם 0
4. ניקח עכשיו את התת מחרוזת ABE בדוק אם יש תת מחרוזת שהיא גם סיומת וגם קידומת מכיוון שיש רק שתי תווים שונים LPS[2] יהיה גם 0
5. אם היה במקום E את A אז היינו מקבלים LPS[2]=1 כי היא שווה לתו הראשון במחרוזת החיפוש

***LPS:***

|  |  |  |
| --- | --- | --- |
| 0 | 0 | 0 |

**אלגוריתם Boyer-Moore :**

האלגוריתם מבצע עיבוד מקדים למחרוזת החיפוש (התבנית), אך לא לטקסט שבו יבוצע החיפוש. תם בויאר-מור משתמש במידע הנאסף בשלב העיבוד המקדים על מנת לדלג על חלקים בטקסט, והודות לכך הוא משיג ביצועים טובים ביחס לאלגוריתמי חיפוש אחרים.

באופן כללי, האלגוריתם רץ מהר יותר ככל שמחרוזת החיפוש ארוכה יותר. התכונות החשובות של האלגוריתם הן שהוא מחפש מסוף מחרוזת החיפוש ולא תחילתה, והוא מדלג בטקסט על מספר תווים במקום לחפש תו-תו.

**עיבוד מקדים :**

לצורך אלגוריתם זה נצטרך, לבנות טבלת אי התאמה שתשמש אותנו לחשב את מרחק הקפיצה בכל פעם שנקבל אי שוויון בין שני תווים.

הטבלה מכילה מופע אחד בלבד של כל תוו שנמצא במחרוזת החיפוש ונותנת לו ערך לפי הנוסחה הבאה:

Max(1,length\_of\_pattern-index\_of\_current\_char-1)

ובסוף הטבלה מוסיפים תו למשל \* שערכו שווה לאורך מחרוזת החיפוש.

**איטראציה באלגוריתם זה:**

לצורך הדגמה נניח שאנחנו באיטראציה הראשונה, אז בודקים את התו האחרון בטקסט הנבחר (מיקום גודל המחרוזת פחות 1) , אם התו האחרון במחרוזת החיפוש שווה לתו האחרון במחרוזת במקום גודל המחרוזת פחות 1 אז ממשיכים לבדוק את התו לפני האחרון בשתי המחרוזות וכך הלאה עד ש: או שמוצאים כל התווים של המחרוזת (כל התווים שווים בשתי המחרוזות) או שמוצאים אי שוויון.  
במקרה הראשון: תת צעד: הצלחה

אם איתרנו מופע למחרוזת ועדיין לא הגענו לסוף המחרוזת שהתקבלה בקלט, אז נמשיך לחפש מופעים ועוברים לתו הבא ע"י קפיצה לפי אורך המחרוזת לחיפוש.

במקרה השני: נמצא תווים שונים, אז האלגוריתם מחשב את הקפיצה למיקום התו הבא שצריך ממנו להתחיל את הבדיקה שוב.

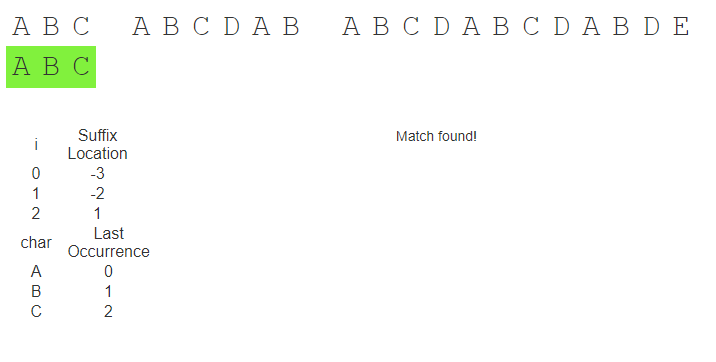
תת צעד: כישלון  
במקרה של אי התאמה כלומר קיבלנו שני תווים שונים, אם התו שייצר את האי שווין לא קיים בכלל במחרוזת החיפוש אז קופצים לפי הערך של \* צעדים שנמצא בטבלת האי התאמה.

אם התו נמצא במחרוזת החיפוש אז מוצאים את הערך שלו מטבלת האי התאמה וקופצים לפי ערך זה.

תהליך זה מבוצע עד שנאתר את המחרוזת הרצויה או נגיע לסוף הקלט.

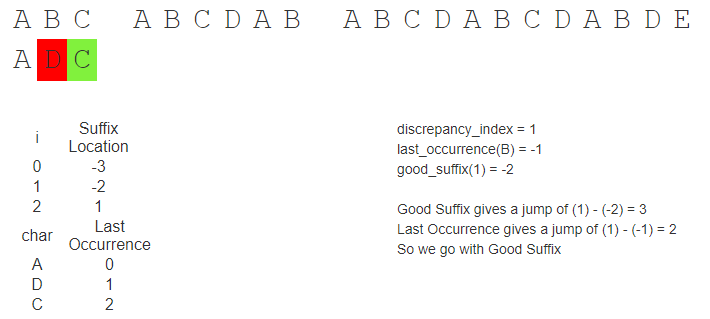
**ויזואליזציה:**

**מקרה ראשון:**

****

המחרוזת נמצאת, כל תו נמצא נצבע בירוק, בנוסף הודעת הצלחה מודפסת.

**מקרה שני:**

 נמצא תו שונה אז נצבע באדום, ומחושב גודל הקפיצה (התו הבא שממנו נתחיל לבדוק).

על סימון התו שבודקים בכל שלב נצבע אותו בצע מסויים למשל כחול ואז צבע ירוק אם משווים או אדום במקרה של אי שוויון.

הערה: תמונות להמחשה בלבד, ייהיה שינויים על העיצוב.

בויזואליזציה נוסיף את ה BAD MACH TABLE ואת תהליך העיבוד המקדים שחישבנו בשלב העיבוד על מנת לתאר איך מחושב אורך הקפיצה שיהייה ברור למשתמש.

*Reference*: <http://whocouldthat.be/visualizing-string-matching>

חישוב BAD MACH TABLE:

נעשה שימוש במערך עזר נקרא לו Values ניקח את האותיות של מחרוזת החיפוש נבדוק תו תו והערך של Values נחשב אותו לפי הנוסחה :

Max(1,length\_of\_pattern-index\_of\_current\_char-1)

אם תו מופיע יותר מפעם אחת אז צריך לעדכן אותו ונוסיף את תו \* שיהיה שווה לאורך מחרוזת החיפוש

למשל עבור המקרה השני :

מחרוזת החיפוש הייתה :

|  |  |  |
| --- | --- | --- |
| C | D | A |

המערך Values :

שיטת החישוב :

1. התו הראשון שמחשבים אותו הוא A צריך לבחור המקסימום בין 1 ו- (3-0-1=2) שזה שווה לשתיים
2. התו השני שמחשבים אותו הוא D צריך לבחור המקסימום בין 1 ו- (3-1-1=1) שזה שווה לאחד
3. התו השלישי שמחשבים אותו הוא E צריך לבחור המקסימום בין 1 ו- (3-2-1=0) שזה שווה לאחד
4. התו האחרון הוא כוכבית שהוא שווה לשלוש שזה אורך מחרוזת החיפוש נצטרך אותו כאשר אין תווים שלא זהים לתווים הנמצאים

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| \* | C | D | A | Letters |
| 3 | 1 | 1 | 2 | Values |

**Input File:**

First it will match the 0th index of pattern with the 0th index of the given text which is ‘m’ with ‘a’. Since they don’t match, we move to the next index of the text and we’ll compare ‘m’  with ‘c’, again it’s not a match. Then, again we will move to the next index and compare ‘m’ with the next index value which is ‘m’ now it’s a match then we move to the next index and search for more matches and compare ‘a’ with the next index value which is ‘a’ . So , again it’s a match and then we move to the next index and move to the next index and compare ‘l’ with ‘l’ .Finally, the whole substring matches and the and we would get 2 which is the index at which the substring exists. Time complexity in the worst case is O(m\*n). Where m and n are the length of S and P respectively.

Source: <https://www.educba.com/kmp-algorithm/>

יתכן שינויים בתוכן הקבצים במידת הצורך.