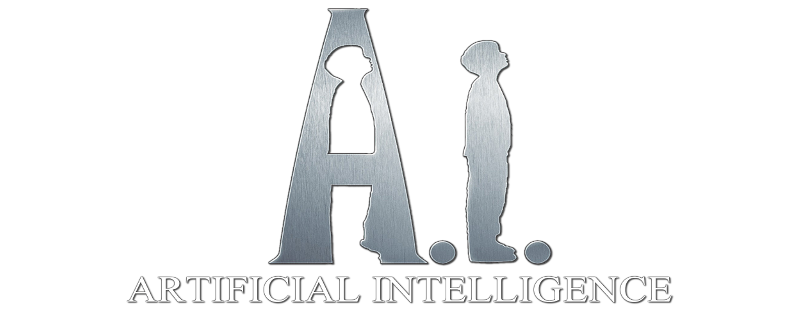
מעבדה בבינה מלאכותית, תשע"ח

פרויקט #1 – המכונית האדומה.

**מגישים:**

באסל סעד 307912402

עלי ספדי 205821713



**פונקציה היוריסטית #1: Distance**

מרחק X מהיציאה מוגדר כ- מספר הרכבים שחוסמים אותו.  
פונקציה זו היא אכן אדמסבילית כי היא תמיד נותנת ערך קטן או שווה לעלות האמתית.

**לוגיקה מאחורי הבחירה:** בעצם ככל שיש פחות רכבים שחוסמים את X, ככל שאנחנו מתקרבים לפתרון.

**הדגמות:**

**A**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  | X | X |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |

**B**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |
|  |  | O |  |  |  |
|  |  | O | X | X |  |
|  |  | O |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |

**עיקובים אפשריים במהלך הריצה:** נסתכל במצב שבו הערימה מכילה K מצבים מהם 2 מינימליים N1, N2:

**N1**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  |  | . | . | . |
|  |  | O |  |  | . |
|  |  | O | X | X | . |
|  |  | O |  |  |  |
|  |  |  |  |  |  |
|  |  | B | B |  |  |

**N2**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  |  | . | . | . |
|  |  | O |  |  | . |
|  |  | O | X | X | . |
|  |  | O |  |  |  |
|  |  |  |  |  |  |
|  |  |  | B | B |  |

במקרה זה בעצם האלגוריתם בוחר בצורה שרירותית מבין שני המצבים ולכן בתסריט הגרוע האלגוריתם יבחר את N1 יפתח אותו, ובכך זה יגדיל את כמות הצעדים עד להגיע למטרה (במקום לבחור את N2 ולהגיע לפתרון אחרי שני צעדים).

קוד:

**private** **void** calculate\_Fn(State node) {

**if**(node.isGoal()){

node.setFValue(-100.0);

}**else**{

**double** count = 0;

**char**[][] board = node.getBoard();

**int** CAR\_LENGTH = 2;

**for**(**int** i = node.cars.get("X").getPosition().y + CAR\_LENGTH ; i < 6;

i++){

**char** c = board[2][i];

**if**(c != 'X' && c >='A' && c <= 'Z'){

count += 10;

}

}

node.setFValue(count + node.g);

}

}

**פונקציה היוריסטית #2: Contra-Mobility**

פונקציה זו מחשבת עבור מצב כלשהו את מספר הצעדים האפשריים שאפשר לבצע על המצב, ונותנת את הערך הנגדי לו. פונקציה זו אדמסבילית (לפי הגדרה) היא תמיד נותנת ערך שלילי.

**לוגיקה מאחורי הבחירה:** ככל שיש יותר מצבים אפשריים שאפשר לעבור אליהם, ככל שעלתה דרגת החופש וככל שהתקרבנו לפתרון. על מנת לשמור על מונוטוניות (ככל שנוכל), לקחנו את המספר הנגדי כך שככל שיש יותר דרגת חופש (צעדים אפשריים) ככל שערך הפונקציה קטן. ערך הפונקציה נע

**הדגמות (עם המקרה הגרוע שתואר עבור הפונקציה הראשונה):**

**N1**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  |  | . | . | . |
|  |  | O |  |  | . |
|  |  | O | X | X | . |
|  |  | O |  |  |  |
|  |  |  |  |  |  |
|  |  | B | B |  |  |

**N2**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  |  | . | . | . |
|  |  | O |  |  | . |
|  |  | O | X | X | . |
|  |  | O |  |  |  |
|  |  |  |  |  |  |
|  |  |  | B | B |  |

אפשר לראות שבמקרה זה פונקציה זו מניבה מספר צעדים קטן יותר מהפונקציה הראשונה.

**עיקובים אפשריים במהלך הריצה:**

מהתוצאות שנציג למטה, מסתבר שה EBF של הפונקציה הזו גדול יותר מזה של הפונקציה הראשונה. (פירוט למטה).

קוד:

**private** **void** calculate\_Fn2(State node) {

**if**(node.isGoal()){

node.setFValue(-100.0);

}**else**{

**double** count = node.nextStates.size();

node.setFValue(-count + node.g);

}

}

**פונקציה היוריסטית #3: Contra-Mobility + Distance**

פונקציה זו מבצעת ממוצע משוקלל בין שתי הפונקציות האחרות. כמו כן היא פונקציה אדמסבילית מכיוון שהיא ממוצע משוקלל של ערכים שהם קטנים מהעלות האמתית.

**לוגיקה מאחורי הבחירה:** לצורך חישובים סטטיסטיים והשוואות.

**פונקציה היוריסטית #4: Dijkstra/The Null Function**

**המקרה הכי מעניין, פונקציה זו תמיד נתנה פתרון עם מספר צעדים זהה לזה שבקובץ הנתון.**

* החלטנו להריץ את הפונקציה רק מתוך התעניינות, והתפלאנו לדעת שהיא הניבה את התוצאות הכי טובות.

**תוצאות הניסויים שביצענו:**

**ניסוי #1.1 – השוואה בין מספר הצעדים שהאלגוריתם הניב עם היוריסטיקות שתיארנו לעיל:**

**FIGURE 1**

* אפשר לראות בגרף שהתוצאות הסופיות שהניב האלגוריתם בעזרת הפונקציה השנייה הן יותר טובות יחסית ברמות מעל 32.
* לעומת זאת, פונקציית המרחק (H1) הניבה תוצאות לא רעות ברמות פחות קשות וכן ברמות 23-28.
* פונקציית האפס: נתנה פתרון מעולה עם הכי פחות צעדים בכל הרמות.

**ניסוי #1.2 – השוואה בין תוצאות ה Branching Factor:**

**FIGURE 2**

* אפשר לראות מתוצאות החלק הזה שה-EBF של הפונקציה הראשונה (Distance) קטן יותר מזה של הפונקציה השנייה (Contra-Mobility). וזה בגלל העובדה שה- Contra-Mobility נותנת ערכים שליליים וזה גורם להימצאות המון צמתים עם ערך שווה בתוך הערימה של ה-Opened-List.
* כמו כן, אפשר לראות בשתי הרגרסיות שה- Branching factor הולך וקטן ככל שהרמה עולה וזה נובע מהעובדה ש- ככל שהרמה עולה -> ככל שיש מורכבות -> ככל שיש פחות מצבים שאפשר להיפתח אליהם ולכן ה- Branching קטן.

אז למה רואים שה-EBF הולך ונהיה שווה כמעט בשתי הפונקציה הראשונה ככל שהרמה עולה?

תשובה: זה בגלל שהפונקציה השנייה (Contra-Mobility) נותנת מספר h: . כלומר בבעיות שדורשות מספר צעדים קטן יחסית אז יהיו המון התנגשויות (בערכים) בתוך הערימה וזה גורר פיתוח מצבים שהם לא בהכרח הכי טובים. אבל ככל שעולים ברמה וככל שמספר הצעדים הקטן ביותר שדרוש לפתור רמה מסוימת עולה אז גם הפונקציה מתחילה להתייצב והערכים בתוך הערימה מתחילים להתרחק זה מזה.

**פירוט נוסף:**

**נזכור ש-** Contra-Mobility בעצם נותנת את המספר הנגדי של כמות הצעדים עבור מצב מסוים,

***– כמות הצעדים שעשינו מהשורש (ביחידות של 1).***

**ולכן בבעיות קלות שדורשות מספר קטן של צעדים כלומר כאשר ה קטן מ- 10 אז עלולים לקבל ערכים שווים בחישוב של . הערה: מספר 10 זה כמות צעדים טיפוסית מקסימלית (בערך) עבור מצב מסוים. יכול להיות מצב שיש בו יותר אבל אנחנו מסתכלים רק על מקרים מעניינים.**

* פונקציית האפס בעצם התחילה ברגל שמאל, וזה בגלל העובדה שברמות קלות, ישנם המון התנגשויות והבחירה בעצם תהיה שרירותית ולאו דווקא נכונה. בבעיות יותר קשות השונות בין ערכי ה-F גדלה ולכן זה מאפשר בחירה יותר נכונה.

**ניסוי #2.2 – שילוב בין שתי הפונקציות:**

בגלל התוצאות שקיבלנו לעיל, תהינו, מה אם נערבב בין שתי הפונקציות הראשונות ונבחן את התנהגות השילוב. ולכן חיברנו פונקציה חדשה שהיא בעצם ממוצע משוקלל של שתי הפונקציות הראשונות שהזכרנו לעיל.

וכמו שהזכרנו לעיל:

בעצם, עם הוספת פונקציה זו, אפשר לעשות אינספור קומבינציות. בחנו כמה מקרים ובסוף בחרנו שלושה מהם:

**FIGURE 3**

אפשר לראות בתוצאות שלהלן:

* בקומבינציה השניה נתנו ערך יותר גבוה לפונקציה השניה ומצאנו שבמקרים יותר קשים הביצועים יותר טובים (באפור).
* בסך הכל התוצאות היו יחסית דומות.

**FIGURE 4**

* רואים שהקומבינציה הראשונה (צהוב) בעצם נתנה ביצועים יותר טובים מהשתיים האחרות.
* אמנם מספר הצעדים (באיור 4) קצת יותר מהפתרון הנתון אבל ה- EBR כן ירד ביחס לתוצאות של הפונקציות , .

**נבחר את הפונקציה עם הקומבינציה השניה בתור , ולסיום – השוואת הביצועים של שלושת הפונקציות:**

**FIGURE 5**

* רואים שהבחירה שלנו עבור היתה מוצדקת! (כחול) שהיא הכי קרובה לביצועי פונקציית האפס (בצהוב) – ברמות קשות יותר וכן ברמות פחות קשות.

**FIGURE 6**

* גם פה בביצועי ה EBF רואים שהפונקציה שפיתחנו הניבה תוצאות לא רעות ויותר טובות משאר הפונקציות באופן כללי.

**לסיום – ברור כי כבר אין מה להשוות עם אלגוריתם החיפוש הלא מיודע שלא פתר אפילו בעיה אחת תחת הזמן שהקדשנו ~ 0.55 שניה.**

**FIGURE 7**

**לסיכום:**

**בעצם בחנו כמה מקרים וכמה אספקטים בעבודה זו. הגענו לפונקציית האפס (הכי טובה מבחינת מספר צעדים). כמו כן עשינו בדיקות חישוביות והשוואנו תוצאות בין מקרים\פרמטרים שונים והגענו לפונקציה שבאופן כללי הניבה ביצועים יותר טובים מבחינת זמנים ו EBF.**

**בסה"כ אפשר להתקדם ולפתח עוד המון כיוונים ולהגיע לתוצאות יותר טובות.**

**הזמן הכי מינימלי שהצלחנו להגיע אליו נע בין 0.255-0.27 שניות (חמישית שניה) – תלוי במחשב.**

**Implementation acknowledgements**

* This work was done using github.com and can be found @

<https://github.com/BaselSD/artificial-Intel-18>

* **We used Fibonacci heap data structure for our A\* implementation.**

**The source code we used is licensed and can be found at:**

[**https://github.com/jgrapht/jgrapht/blob/master/jgrapht-core/src/main/java/org/jgrapht/util/FibonacciHeap.java**](https://github.com/jgrapht/jgrapht/blob/master/jgrapht-core/src/main/java/org/jgrapht/util/FibonacciHeap.java)

* **We also used the HashMap implementation provided by Java utility library.**
* **For the OPEN-LIST of A\* we used the Fibonacci heap for fast   
  min-extracting in addition to HashMap for fast general-extracting.**
* **As for the CLOSE-LIST we used a HashMap for fast general-extracting.**

**Source-Code**