חיפוש:

מציאת פתרון לבעיה, כמו אפליקציית ניווט שמוצאת את המסלול הטוב ביותר מנקודת המוצא שלכם ליעד, או כמו משחק וגילוי הצעד הבא.

בעיות חיפוש

בעיות חיפוש כוללות סוכן שמקבל מצב התחלתי ומצב יעד, והוא מחזיר פתרון כיצד להגיע מהראשון לשני.

למשל: אפליקציית ניווט משתמשת בתהליך חיפוש טיפוסי, שבו הסוכן (החלק החושב של התוכנית) מקבל כקלט את המיקום הנוכחי שלך ואת היעד הרצוי, ובהתבסס על אלגוריתם חיפוש, מחזיר נתיב מוצע.

עם זאת, ישנן צורות רבות אחרות של בעיות חיפוש, כמו פאזלים או מבוכים.

מושגים

סוכן: ישות התופסת את סביבתה ופועלת על פיה. באפליקציית ניווט, לדוגמה, הסוכן יהיה ייצוג של מכונית שצריכה להחליט אילו פעולות לבצע כדי להגיע ליעד.

מצב: תצורה של סוכן בסביבתו. לדוגמה, בפאזל של 15 משבצות, מצב הוא כל דרך שבה כל המספרים מסודרים על הלוח.

מצב התחלתי: המצב שממנו מתחיל אלגוריתם החיפוש. באפליקציית ניווט, זה יהיה המיקום הנוכחי.

פעולות: בחירות שניתן לבצע במצב. ליתר דיוק, ניתן להגדיר פעולות כפונקציה. עם קבלת מצבים כקלט, פעולות מחזירות כפלט את קבוצת הפעולות שניתן לבצע במצבים. לדוגמה, בפאזל של 15 משבצות, הפעולות של מצב נתון הן הדרכים שבהן

ניתן להחליק ריבועים בתצורה הנוכחית (4 אם הריבוע הריק נמצא באמצע, 3 אם ליד צד, 2 אם בפינה).

מודל מעבר: תיאור של המצב הנובע מביצוע כל פעולה רלוונטית בכל מצב. ליתר דיוק, ניתן להגדיר את מודל המעבר כפונקציה. עם קבלת מצבים s ופעולה a כקלט ומחזירה את המצב הנובע מביצוע פעולה a במצב s. לדוגמה, בהינתן תצורה מסוימת של פאזל עם 15 דמויות (מצב s), הזזת ריבוע בכל כיוון (פעולה a) תביא לתצורה חדשה של הפאזל (המצב החדש).

מרחב המצבים: קבוצת כל המצבים שניתן להגיע אליהם מהמצב ההתחלתי על ידי כל רצף פעולות. לדוגמה, בפאזל עם 15 דמויות, מרחב המצבים מורכב מכל תצורות 16!/2 על הלוח שניתן להגיע אליהן מכל מצב התחלתי. ניתן לדמיין את מרחב המצבים כגרף מכוון עם מצבים, המיוצגים כצמתים, ופעולות, המיוצגות כחצים בין צמתים.

מבחן מטרה: התנאי הקובע האם מצב נתון הוא מצב מטרה. לדוגמה, באפליקציית ניווט, מבחן המטרה יהיה האם המיקום הנוכחי של הסוכן (ייצוג המכונית) נמצא ביעד. אם כן - הבעיה נפתרה. אם לא - נמשיך לחפש.

עלות נתיב: עלות מספרית הקשורה לנתיב נתון. לדוגמה, אפליקציית ניווט לא פשוט מביאה אותך ליעד שלך; היא עושה זאת תוך מזעור עלות הנתיב, ומציאת הדרך המהירה ביותר האפשרית עבורך להגיע למצב היעד שלך.

פתירת בעיות חיפוש:

פתרון

רצף פעולות המוביל מהמצב ההתחלתי למצב המטרה.

פתרון אופטימלי

פתרון בעל עלות הנתיב הנמוכה ביותר מבין כל הפתרונות.

בתהליך חיפוש, נתונים מאוחסנים לרוב ב**צומת**, מבנה נתונים המכיל את הנתונים הבאים:

- מצב
- צומת האב שלו, שדרכו נוצר הצומת הנוכחי
- הפעולה שהוחלה על מצב האב כדי להגיע לצומת הנוכחי
- עלות הנתיב מהמצב ההתחלתי לצומת זה

צמתים מכילים מידע שהופך אותם לשימושיים מאוד למטרות אלגוריתמי חיפוש.

הם מכילים מצב, שניתן לבדוק באמצעות מבחן המטרה כדי לראות אם זהו המצב הסופי.

אם כן, ניתן להשוות את עלות הנתיב של הצומת לעלויות הנתיב של צמתים אחרים, מה שמאפשר בחירת הפתרון האופטימלי.

לאחר בחירת הצומת, בזכות אחסון צומת האב והפעולה שהובילה מההורה לצומת הנוכחי, ניתן לעקוב אחר כל שלב בדרך מהמצב ההתחלתי לצומת זה, ורצף פעולות זה הוא הפתרון.

עם זאת, צמתים הם פשוט מבנה נתונים - הם לא מחפשים, הם מחזיקים מידע.

כדי לחפש בפועל, אנו משתמשים בגבול, המנגנון ש"מנהל" את הצמתים.

הגבול מתחיל בכך שהוא מכיל מצב התחלתי וקבוצה ריקה של פריטים שנחקרו, ולאחר מכן חוזר על הפעולות הבאות עד להגעה לפתרון:

חזור:

1. אם הגבול ריק:
   - עצור. אין פתרון לבעיה.
2. הסר צומת מהגבול. זהו הצומת שייבחן.
3. אם הצומת מכיל את מצב המטרה:
   - החזר את הפתרון. עצור.
4. אחרת:
   - * הרחב את הצומת (מצא את כל הצמתים החדשים שניתן להגיע אליהם מצומת זה), והוסף צמתים שנוצרו לגבול.
   - * הוסף את הצומת הנוכחי לקבוצה שנחקרה.

חיפוש עומק קודם:

חיפוש עומק קודם (DFS)

אלגוריתם חיפוש עומק קודם ממצה כל כיוון לפני שהוא מנסה כיוון אחר.

במקרים אלה, הגבול מנוהל כמבנה נתונים של stack.

הביטוי שצריך לזכור כאן הוא "אחרון נכנס ראשון יוצא".

לאחר הוספת צמתים לגבול, הצומת הראשון שמוסר ונלקח בחשבון הוא האחרון שנוסף.

התוצאה היא אלגוריתם חיפוש שהולך עמוק ככל האפשר בכיוון הראשון שמפריע לו, תוך השארת כל הכיוונים האחרים למועד מאוחר יותר.

לדוגמא: קחו סיטואציה שבה אתם מחפשים את המפתחות שלכם.

בגישת חיפוש עומק, אם תבחרו להתחיל בחיפוש במכנסיים, תחילה תעברו על כל כיס, תרוקנו כל כיס ותעברו על התכולה בזהירות.

תפסיקו לחפש במכנסיים ותתחילו לחפש במקום אחר רק לאחר שתסיימו לחלוטין את החיפוש בכל כיס במכנסיים.

יתרונות:

במקרה הטוב, אלגוריתם זה הוא המהיר ביותר.

אם הוא "מתמזל מזלו" ותמיד בוחר את הנתיב הנכון לפתרון (במקרה), אז חיפוש עומק לוקח את הזמן הקצר ביותר האפשרי כדי להגיע לפתרון.

חסרונות:

ייתכן שהפתרון שנמצא אינו אופטימלי.

במקרה הגרוע, אלגוריתם זה יחקור כל נתיב אפשרי לפני מציאת הפתרון, ובכך ייקח את הזמן הארוך ביותר האפשרי לפני שיגיע לפתרון.

```
# Define the function that removes a node from the frontier and
returns it.
    def remove(self):
        # Terminate the search if the frontier is empty, because
this means that there is no solution.
        if self.empty():
            raise Exception("empty frontier")
        else:
             # Save the last item in the list (which is the newest
node added)
            node = self.frontier[-1]
            # Save all the items on the list besides the last node
(i.e. removing the last node)
            self.frontier = self.frontier[:-1]
            return node
```

חיפוש רוחב קודם:

חיפוש רוחב-קודם (BFS)

אלגוריתם חיפוש רוחב-קודם יעקוב אחר מספר כיוונים בו זמנית, וייקח צעד אחד בכל כיוון אפשרי לפני שייקח את הצעד השני בכל כיוון.

במקרה זה, הגבול מנוהל כמבנה נתונים של תור.

הביטוי שצריך לזכור כאן הוא "נכנס ראשון, יוצא ראשון".

במקרה זה, כל הצמתים החדשים מצטברים בשורה, והצמתים נלקחים בחשבון על סמך איזה מהם נוסף ראשון (כל הקודם זוכה!).

התוצאה היא אלגוריתם חיפוש שלוקח צעד אחד בכל כיוון אפשרי לפני שייקח צעד שני בכל כיוון.

דוגמא:

נניח שאתם נמצאים במצב בו אתם מחפשים את המפתחות שלכם. במקרה זה, אם תתחילו עם המכנסיים שלכם, תחפשו בכיס הימני שלכם. לאחר מכן, במקום להסתכל בכיס השמאלי שלכם, תסתכלו במגירה אחת. אחר כך על השולחן. וכן הלאה, בכל מקום שאתם יכולים לחשוב עליו. רק לאחר שתמצו את כל המיקומים, תחזרו למכנסיים שלכם ותחפשו בכיס הבא.

יתרונות:

אלגוריתם זה מובטח שימצא את הפתרון האופטימלי.

חסרונות:

אלגוריתם זה כמעט מובטח שייקח יותר זמן מהזמן המינימלי לרוץ.

במקרה הגרוע ביותר, אלגוריתם זה לוקח את הזמן הארוך ביותר האפשרי לרוץ.

```
    # Define the function that removes a node from the frontier
and returns it.
    def remove(self):
        # Terminate the search if the frontier is empty, because
this means that there is no solution.
        if self.empty():
            raise Exception("empty frontier")
        else:
            # Save the oldest item on the list (which was the
first one to be added)
            node = self.frontier[0]
            # Save all the items on the list besides the first one
(i.e. removing the first node)
            self.frontier = self.frontier[1:]
            return node
```

חיפוש חמדן קודם (GFS)

אלגוריתם חיפוש "רוחב-ראשון" ו"עומק-ראשון" הם שניהם אלגוריתמי חיפוש לא-מושכלים.

כלומר, אלגוריתמים אלה אינם משתמשים בידע כלשהו על הבעיה שלא רכשו באמצעות חקירה עצמית.

עם זאת, לרוב, ידע מסוים על הבעיה זמין למעשה.

לדוגמה, כאשר פותר מבוך אנושי נכנס לצומת, האדם יכול לראות איזה כיוון הולך בכיוון הכללי של הפתרון ואיזה כיוון לא.

בינה מלאכותית יכולה לעשות את אותו הדבר.

סוג של אלגוריתם שלוקח בחשבון ידע נוסף כדי לנסות לשפר את ביצועיו נקרא אלגוריתם חיפוש מושכל.

חיפוש חמדני "הטוב ביותר קודם" מרחיב את הצומת הקרוב ביותר ליעד, כפי שנקבע על ידי פונקציה היוריסטית.

כפי ששמה מרמז, הפונקציה מעריכה כמה קרוב ליעד הצומת הבא, אך ניתן לטעות בה. י

עילותו של אלגוריתם "הטוב ביותר קודם" החמדני תלויה בטיב הפונקציה ההיוריסטית.

מרחק מנהטן

עם זאת, חשוב להדגיש שכמו בכל היוריסטיקה, היא יכולה להשתבש ולהוביל את האלגוריתם למסלול איטי יותר ממה שהיה הולך אחרת.

ייתכן שאלגוריתם חיפוש לא מושכל יספק פתרון טוב יותר ומהיר יותר, אך סביר פחות שיעשה זאת מאשר אלגוריתם מושכל.

חיפוש A*

חיפוש A*, פיתוח של אלגוריתם החיפוש החמדני הטוב ביותר, מתחשב לא רק בעלות המשוערת מהמיקום הנוכחי ליעד, אלא גם בעלות שנצברה עד למיקום הנוכחי.

על ידי שילוב שני ערכים אלה, לאלגוריתם יש

דרך מדויקת יותר לקבוע את עלות הפתרון ולמטב את בחירותיו תוך כדי תנועה. האלגוריתם עוקב אחר (עלות הנתיב עד עכשיו + עלות משוערת ליעד), וברגע שהוא עולה על העלות המשוערת של אפשרות קודמת כלשהי, האלגוריתם ינטוש את הנתיב הנוכחי ויחזור לאפשרות הקודמת, ובכך ימנע מעצמו ללכת בנתיב ארוך ולא יעיל ש-h(n) סימן בטעות כטוב ביותר.

שוב, מכיוון שגם אלגוריתם זה מסתמך על היוריסטיקה, הוא טוב כמו ההיוריסטיקה שהוא משתמש בה.

ייתכן שבמצבים מסוימים הוא יהיה פחות יעיל מחיפוש חמדני הטוב ביותר או אפילו מאלגוריתמים לא מושכלים. כדי שחיפוש A* יהיה אופטימלי, הפונקציה ההיוריסטית צריכה להיות:

1. קביל, או לעולם לא להעריך יתר על המידה את העלות האמיתית, ו-
2. עקבי, כלומר שעלות הנתיב המשוערת ליעד של צומת חדש בנוסף לעלות המעבר אליו מהצומת הקודם גדולה או שווה לעלות הנתיב המשוערת ליעד של הצומת הקודם..

חיפוש עוין

בעוד שבעבר דנו באלגוריתמים שצריכים למצוא תשובה לשאלה, בחיפוש עוין האלגוריתם מתמודד עם יריב שמנסה להשיג את המטרה ההפוכה.

לעתים קרובות, בינה מלאכותית המשתמשת בחיפוש עוין נתקלת במשחקים, כמו איקס עיגול.

מינימקס

סוג של אלגוריתם בחיפוש עוין, מינימקס מייצג תנאי ניצחון כ-(-1) לצד אחד ו-(+1) לצד השני.

פעולות נוספות יונעו על ידי תנאים אלה, כאשר הצד הממזער ינסה להשיג את הציון הנמוך ביותר, והצד הממקסם ינסה להשיג את הציון הגבוה ביותר.

ייצוג בינה מלאכותית של איקס עיגול:

- S_0: מצב התחלתי (במקרה שלנו, לוח ריק 3X3)
- שחקן/ים: פונקציה שבהינתן מצב s, מחזירה את תורו של השחקן (X או O).
- פעולות: פונקציה שבהינתן מצב s, מחזירה את כל המהלכים החוקיים במצב זה (אילו מקומות פנויים על הלוח).
- תוצאה/ות, a: פונקציה אשר, בהינתן מצב s ופעולה a, מחזירה מצב חדש. זהו הלוח שנוצר מביצוע הפעולה a במצב s (ביצוע מהלך במשחק).
- טרמינל/ים: פונקציה אשר, בהינתן מצב s, בודקת האם זהו השלב האחרון במשחק, כלומר אם מישהו ניצח או שיש תיקו. מחזירה True אם המשחק הסתיים, False אחרת.
- תועלת/ות: פונקציה אשר, בהינתן מצב סופי s, מחזירה את ערך התועלת של המצב: -1, 0 או 1.

כיצד האלגוריתם פועל:

באופן רקורסיבי, האלגוריתם מדמה את כל המשחקים האפשריים שיכולים להתקיים החל מהמצב הנוכחי ועד להגעה למצב סופי. כל מצב סופי מוערך כ-(-1), 0 או (+1).

אלגוריתם מינימקס במשחק איקס עיגול

בהתבסס על המצב בו נמצא תורו, האלגוריתם יכול לדעת האם השחקן הנוכחי, כאשר הוא משחק בצורה אופטימלית, יבחר בפעולה שמובילה למצב עם ערך נמוך או גבוה יותר.

בדרך זו, לסירוגין בין מזעור למקסום, האלגוריתם יוצר ערכים למצב שייווצר מכל פעולה אפשרית.

כדי לתת דוגמה קונקרטית יותר, נוכל לדמיין שהשחקן הממקסם שואל בכל תור: "אם אבצע את הפעולה הזו, ייווצר מצב חדש. אם השחקן הממזער משחק בצורה אופטימלית, איזו פעולה אותו שחקן יכול לבצע כדי להביא לערך הנמוך ביותר?".

עם זאת, כדי לענות על שאלה זו, השחקן הממקסם צריך לשאול: "כדי לדעת מה השחקן הממזער יעשה, אני צריך לדמות את אותו התהליך במוחו של הממזער: השחקן הממזער ינסה לשאול: 'אם אבצע את הפעולה הזו, איזו פעולה יכול השחקן הממקסם לבצע כדי להביא לערך הגבוה ביותר?'"

זהו תהליך רקורסיבי, וייתכן שיהיה קשה להבין אותו; בסופו של דבר, באמצעות תהליך חשיבה רקורסיבי זה, השחקן הממקסם מייצר ערכים עבור כל מצב שיכולים לנבוע מכל הפעולות האפשריות במצב הנוכחי.

לאחר שיהיו לו ערכים אלה, השחקן הממקסם בוחר את הגבוה ביותר.

the Minimax algorithm works the following way:

Given a state *s:*

The maximizing player picks action *a* in *Actions(s)* that produces the highest value of *Min-Value(Result(s, a))*.

The minimizing player picks action *a* in *Actions(s)* that produces the lowest value of *Max-Value(Result(s, a))*.

Function *Max-Value(state)*

```
v = -∞

if Terminal(state):
 return Utility(state)

for action in Actions(state):
 v = Max(v, Min-Value(Result(state, action)))
return v
```

Function *Min-Value(state)*:

```
v = ∞

if Terminal(state):
 return Utility(state)

for action in Actions(state):
 v = Min(v, Max-Value(Result(state, action)))
return v
```

גיזום אלפא-בטא

גיזום אלפא-בטא, דרך לייעל את המינימקס, מדלג על חלק מהחישובים הרקורסיביים שהם בעלי חשיבות שלילית.

לאחר קביעת הערך של פעולה אחת, אם ישנן ראיות ראשוניות לכך שהפעולה הבאה יכולה להביא את היריב לציון טוב יותר מהפעולה שכבר נקבעה, אין צורך לחקור פעולה זו עוד יותר מכיוון שהיא תהיה בהחלט פחות טובה מהפעולה שנקבעה קודם לכן.

ניתן להדגים זאת בקלות רבה ביותר באמצעות דוגמה:

שחקן הממקסם יודע שבשלב הבא, השחקן הממזער ינסה להשיג את הציון הנמוך ביותר. נניח שלשחקן הממקסם יש שלוש פעולות אפשריות, והראשונה מוערכת ב-4.

לאחר מכן השחקן מתחיל לייצר את הערך עבור הפעולה הבאה.

לשם כך, השחקן מייצר את הערכים של פעולות הממזער אם השחקן הנוכחי מבצע פעולה זו, בידיעה שהממזער יבחר את הנמוכה ביותר.

עם זאת, לפני סיום החישוב עבור כל הפעולות האפשריות של הממזער, השחקן רואה שאחת האפשרויות בעלת ערך של שלוש.

משמעות הדבר היא שאין סיבה להמשיך ולבחון את הפעולות האפשריות האחרות עבור השחקן הממזער. ערך הפעולה שטרם הוערכה לא משנה, בין אם הוא 10 או (-10). אם הערך הוא 10, הממזער יבחר באפשרות הנמוכה ביותר, 3, שכבר גרועה יותר מהפעולה שנקבעה מראש, 4. אם הפעולה שטרם הוערכה תתברר כ-(-10), הממזער יבחר באפשרות זו, (-10), שהיא אפילו פחות טובה עבור הממקסם. לכן, חישוב פעולות אפשריות נוספות עבור הממזער בשלב זה אינו רלוונטי עבור הממקסם, מכיוון שלשחקן הממקסם כבר יש בחירה טובה יותר באופן חד משמעי שערכה הוא 4.

מינימקס מוגבל עומק

ישנם סך של 255,168 משחקי איקס עיגול אפשריים, ו-10^{29000} משחקים אפשריים בשחמט. אלגוריתם המינימקס, כפי שהוצג עד כה, דורש יצירת כל המשחקים ההיפותטיים מנקודה מסוימת ועד למצב הסופי. בעוד שחישוב כל משחקי איקס עיגול אינו מהווה אתגר עבור מחשב מודרני, ביצוע פעולה זו בשחמט אינו אפשרי כיום.

מינימקס מוגבל עומק מתחשב רק במספר מוגדר מראש של מהלכים לפני שהוא נעצר, מבלי להגיע למצב סופי. עם זאת, זה לא מאפשר קבלת ערך מדויק עבור כל פעולה, מכיוון שסוף המשחקים ההיפותטיים טרם הושג. כדי להתמודד עם בעיה זו, מינימקס מוגבל עומק מסתמך על פונקציית הערכה שמעריכה את התועלת הצפויה של המשחק ממצב נתון, או, במילים אחרות, מקצה ערכים למצבים. לדוגמה, במשחק שחמט, פונקציית תועלת תקבל כקלט את התצורה הנוכחית של הלוח, תנסה להעריך את התועלת הצפויה שלו (בהתבסס על אילו כלי משחק יש לכל שחקן ומיקומם על הלוח), ולאחר מכן תחזיר ערך חיובי או שלילי המייצג עד כמה הלוח נוח לשחקן אחד לעומת שחקן אחר. ניתן להשתמש בערכים אלה כדי להחליט על הפעולה הנכונה, וככל שפונקציית ההערכה טובה יותר, כך אלגוריתם ה-Minimax שמסתמך עליה טוב יותר.