גרסה 1.01 – 24.03.2010

בינה מלאכותית

ניר אדר

Nir Adar

Email: nir@underwar.co.il

Home Page: http://www.underwar.co.il

אנא שלחו תיקונים והערות אל המחבר.

1. תוכן עניינים

2. מבוא

דרישות קדם להבנת המסמך:

- אלגוריתמים בתורת הגרפים – DFS, BFS.
- לוגיקה – תחשיב היחסים.
- מבני נתונים.

בינה מלאכותית זהו תחום מחקר במדעי המחשב.

המטרה: פיתוח אלגוריתמים לתפיסה, הסקת מסקנות ולמידה כדי לאפשר פתרון בעיות מורכבות.

מבחן טיורינג: מבחן שהוצע על ידי מתמטיקאי, שהציע שכאשר המחשב יעבור אותו, הוא ייחשב ליצור אינטליגנטי.

סוכן אינטיליגנטי הוא ישות התופסת את סביבתה ופועלת עליה כדי להשיג מטרות שהוגדרו על ידי אדוניה.

2.1. פתרון בעיות על ידי חיפוש במרחב מצבים

בהינתן בעיה אותה אנחנו רוצים לפתור, נפעל בצורה הבאה:

- נייצג את מצבי העולם האפשריים על ידי גרף מצבים.
- נייצג את הבעיה על ידי המצב הנוכחי ואת הפתרון הרצוי על ידי קבוצת מצבי מטרה.
- נפעיל אסטרטגית חיפוש למציאת מסלול בגרף המצבים מהמצב הנוכחי אל מצב מטרה.

<u>הגדרת תחום הבעיות על ידי גרף מצבים</u>

- הגדרת קבוצת המצבים האפשריים
- הגדרת האופרטורים המעבירים ממצב למצב

<u>ייצוג הקשתות בגרף המצבים על ידי אופרטורים</u>

אופרטור הינו פעולה שהסוכן יכול להפעיל כדי להעביר את העולם ממצב למצב.

אופרטור זוהי פונקציה המקבלת מצב ומחזירה מצב.

לכל אופרטור מוגדר תחום הגדרה. תהי O קבוצת האופרטורים ו- S קבוצת כל המצבים, אזי:

$$domain(o) \subseteq S, o \in O$$

הקשתות בגרף המצבים מוגדרות: $E = \{\langle s_1, s_2 \rangle \mid \exists o \in O [s_1 \in Domain(o) \& s_2 = o(s_1)]\}$.

לפעמים במקום להשתמש באופרטורים מציינים את רשימת המצבים הבאים מכל מצב:

$$Succ(s): S \to 2^S, s \in S$$

<u>הגדרת מחיר על הקשתות</u>

לפעמים נגדיר מחיר לא אחיד על כל הקשתות. המחיר יינתן על ידי פונקצית מחיר המגדירה מחיר מעבר בין שני מצבים עוקבים.

$$Cost: \{\langle s_1, s_2 \rangle \mid s_1 \in S, s_2 \in Succ(s_1)\} \to \mathbb{R}$$

2.2. אלגוריתמים לחיפוש במרחב מצבים

- אלגוריתמי החיפוש מתחילים מהמצב ההתחלתי.
- בתחילה הגרף מוגדר רק בצורה לא מפורשת.
- אלגוריתמי החיפוש חושפים בהדרגה חלקים מהגרף והופכים אותו לגרף מפורש.

<u>פתרון לבעיית חיפוש</u>

הפתרון לבעית חיפוש היא סדרת אופרטורים המובילים מהמצב ההתחלתי למצב סופי.

<u>פעולת פיתוח צומת</u>

הפעולה הבסיסית של אלגוריתמי החיפוש היא **פעולת פיתוח צומת**.

פעולה זו מקבלת צומת ומחזירה את קבוצת הצמתים העוקבים.

<u>אסטרטגיות חיפוש</u>

אסטרטגית חיפוש מגדירה כיצד יש לחפש במרחב. כל אסטרטגיות החיפוש מבצעות סדרה של **פיתוחי צמתים**. האסטרטגיות נבדלות **בבחירת** הצומת הבא לפיתוח ובהחלטה אילו צמתים ישמרו **בזיכרון**.

<u>מבנה הנתונים צומת</u>

צומת הוא מבנה נתונים המכיל: state – המצב במרחב המצבים אותו מייצג הצומת. parent – מצביע לצומת ממנו נוצר הצומת הנוכחי.

צומת מייצג שלב בתהליך החיפוש של הסוכן. כל צומת מייצג מצב אולם ייתכן שכמה צמתים ייצגו את אותו המצב.

<u>פונקציות עזר</u>

path – פרוצדורה המקבלת צומת, עוקבת אחרי מצביעי ההורים עד שמגיעה לצומת ההתחלה ומחזירה את המסלול.

```
path(N)
      if Null(parent[N]) then
            return [ ]
      else
            return (path(parent[N]) || state[N])
```

op-path - פרוצדורה המקבלת צומת, עוקבת אחרי מצביעי ההורים עד שמגיעה לצומת ההתחלה ומחזירה את האופרטורים שייצרו את המסלול.

```
op-path(N)
      if Null(parent[N]) then
            return []
      else
            return(op-path(parent[N]) || operator[N])
```

<u>הערכת ביצועים של אלגוריתמי חיפוש</u> נעשית על ידי שני מדדים עיקריים: משאבי החיפוש וטיב הפתרון.

בתחום **משאבי החיפוש** נדבר על:

- זמן הריצה – בד"כ מדד שאינו תלוי במחשב ספציפי, כגון מספר הצמתים שפותחו.
- צריכת הזכרון של האלגוריתם.

איכות הפתרון תימדד בד"כ על ידי אורך או מחיר המסלול. כאשר המסלול אינו דרוש – נמצא מדד אחר לאיכות מצבי המטרה. לדוגמא: דרגת הרלוונטיות של הדפים המוחזרים על ידי מנוע חיפוש.

תכונות של אלגוריתמי חיפוש

- **שלמות**: אלגוריתם חיפוש הינו שלם אם מובטח שיחזיר פתרון כאשר פתרון קיים.
- **אופטימליות**: אלגוריתם חיפוש הינו אופטימלי כאשר מובטח שיחזיר את הפתרון בעל המחיר המינימלי.
- **סיבוכיות זמן**
- **סיבוכיות זיכרון**

סוגי אלגוריתמי חיפוש

- אלגוריתמים **עיוורים** מחפשים פתרון בהינתן מרחב מצבים ובעיה ללא שוב מידע נוסף.
- אלגוריתמים **מיודעים** או **יוריסטיים** משתמשים בידע נוסף כדי לייעל את החיפוש.

3. חיפושים לא מיודעים במרחבי מצבים (חיפושים עוורים)

3.1. חיפוש לרוחב (Breadth-First-Search)

אסטרטגית חיפוש המפתחת את עץ החיפוש שכבה אחרי שכבה.

הצומת הבא לפיתוח: הצומת "הפתוח" הרדוד ביותר.

הצמתים שנשמרים בזיכרון: כל הצמתים הפתוחים.

```
Breadth-first-search(problem)

      OPEN ← ( make-node(problem[init-state], NIL) )
      while open ≠ ( ) do
            next-node ← pop(OPEN)
            loop for s in expand(next-node[state])
                  new ← make-node(s,next-node)
                  if problem[goal-test](s) then
                        return(path(new))
                  OPEN ← OPEN || (new)
```

problem זו הבעיה אותה אנו רוצים לפתור.

הפונקציה make-node מקבלת מצב + מצביע לאבא ויוצרת אוביקט מסוג צומת.

OPEN זוהי רשימת הצמתים הפתוחים שאפשר לפתח. כל צומת חדש אותו אנו מפתחים משורשר לסוף הרשימה, ובכך אנחנו יוצרים את החיפוש בשכבות.

3.1.1. חיפוש בגרף מצבים

האלגוריתם פורש במהלך החיפוש עץ.

האלגוריתם יעבוד נכון גם כאשר מרחב המצבים הוא גרף. במקרה כזה יתכן שצמתים רבים ייצגו את אותו מצב. במקרה זה יעילות האלגוריתם תיפגע.

3.1.2. התאמת BFS לחיפוש בגרפים

ניתן למנוע חיפוש חוזר על ידי שמירת הצמתים שפותחו. דרישות הזיכרון לא תגדלנה מכיוון שכל צומת שלא פותח החזיק את כל הצמתים במסלול לשורש לשם שחזור הפתרון.

האלגוריתם המשופר:

```
Breadth-first-search-G(problem)
  OPEN ← ( make-node(problem[init-state], NIL) )
  CLOSE ← NIL
  while open ≠ ( ) do
    next-node ← pop(OPEN)
    CLOSE ← CLOSE ⋃ {next-node}
    loop for s in expand(next-node[state])
      if not(find-state(s, OPEN) or find-state(s, CLOSE)) then
        new ← make-node(s, next-node)
        if problem[goal-test](s) then return(path(new))
        OPEN ← OPEN || (new)


find-state(state, node-list)
  for each node in node-list
    if node[state]=state then return TRUE
  return FALSE
```

3.1.3. תכונות אלגוריתם BFS

שלמות: האלגוריתם שלם, כלומר מובטח שיימצא פתרון לבעיה פתירה. נוכיח זאת על ידי כך שנניח שקיים פתרון באורך L. יהיה b מקדם הסיעוף, אז לאחר b^L צעדים בדק האלגוריתם את כל המסלולים האפשריים באורך L ולכן מצא בהכרח את הפתרון.

אופטימליות: האלגוריתם לא בהכרח את המסלול הקצר ביותר אם לכל קשת מחיר שונה. אם לקשתות מחיר זהה, האלגוריתם אופטימלי.

סיבוכיות הזמן והזכרון: יהיה b מקדם הסיעוף ו- L אורכו של המסלול אל הפתרון הקצר ביותר, אזי סיבוכיות הזמן והזיכרון הן $O\left(b^L\right)$.

3.1.4. בעית הזכרון

עבור מקדם סיעוף גדול (למשל 10), אפילו מסלולים באורך קצר תופסים כמות זכרון עצומה. במקרים כאלו, אם לא קיים מסלול קצר מאוד אל הפתרון, BFS אינו מסוגל לספק תשובה בזמן סביר. לפיכך, נציג אלגוריתמים נוספים לחיפוש בגרף המצבים.

3.2. חיפוש לעומק (Depth-First Search)

- הצומת הראשון לפיתוח הינו הצומת המייצג את מצב ההתחלה.
- האלגוריתם עוצר כאשר הוא מגיע למצב המטרה.
- הצומת הבא לפיתוח הינו הצומת העמוק ביותר.
- בין צמתים בעלי עומק שווה נבחר הצומת הבא שרירותית.
- נשמרים בזיכרון רק צמתים שלא נחקרו לגמרי.

```
DFS (state, goal-test)
  if goal-test(state) then return ((state))
  else
    for c in succ(state)
      solution ← DFS(c, goal-test)
      if solution ≠ FAIL then
        return(state || solution)
      return(FAIL)


DFS-search (problem)
  DFS(problem[init-state], problem[goal-test])
```

3.2.1. חיפוש לעומק עם הגבלת עומק

נשים לב כי האלגוריתם עלול להיתקע במקרה שבגרף שלנו קיים ענף אינסופי או מעגל.

מכאן ניתן להסיק: אלגוריתם DFS אינו שלם.

בגלל הבעיות הנ"ל משתמשים בד"כ בהגבלה על עומק החיפוש.

<u>שלמות</u>: בגלל הגבלת העומק מובטח לנו שהאלגוריתם יעצור.

נסמן: D - הגבלת העומק, L - עומק הפתרון. האלגוריתם שלם כאשר מתקיים $D \geq L$. האלגוריתם אינו שלם כאשר $D < L$.

סיבוכיות זיכרון: נסמן: b מקדם הסיעוף, d העומק הנוכחי, D הגבלת העומק, $b \cdot d$ מספר הצמתים שנשמר בזיכרון בכל רגע.

מתקיים: סיבוכיות הזיכרון הינה: $O(b \cdot D)$. במקרה הגרוע ביותר.

סיבוכיות זמן: במקרה הגרוע ביותר האלגוריתם מייצר עץ מלא בעומק D ולכן סיבוכיות הזמן היא $O(b^D)$.

יעילות: האלגוריתם יעיל כאשר קיימים מצבי מטרה רבים המפוזרים אחיד על פני המרחב.

חיפוש לעומק בגרף: ניתן להוסיף לאלגוריתם רשימה של צמתים שפותחו בצורה דומה לזו של חיפוש לרוחב. עם זאת, הוספה זו תבטל את יתרונו המרכזי של אלגוריתם זה – זיכרון ליניארי לעומת מערכי.

```
DFS-L (state, goal-test,depth)
  if goal-test(state) then return ((state))
  else
    if depth=0 then return FAIL
    for c in succ(state)
      solution ← DFS(c, goal-test,depth-1)
      if solution ≠ FAIL then
        return(state || solution)
    return(FAIL)


DFS-L-search (problem)
DFS(problem[init-state], problem[goal-test], L)
```

3.2.2. חיפוש backtracking

דומה לחיפוש לעומק. במקום לפתח את כל הבנים של כל צומת, אנחנו מייצרים בן אחד, חוקרים אותו לעומק, ורק אז מפתחים את הבן הבא.

יתרונות: חסכוני בזכרון – לא כל הצמתים מפותחים בזמן המעבר. נשתמש בחיפוש זה כאשר מקדם הסיעוף גדול.

3.2.3. חיפוש לרוחב מול חיפוש לעומק

- חיפוש לעומק מאפשר לעבוד עם זכרון ליניארי אך תלוי בהגבלת העומק והמסלול שהוא מוצא אינו אופטימלי.
- חיפוש לרוחב אינו תלוי בהגבלת עומק ומוצא מסלול קצר ביותר, אך דורש זיכרון מערכי.

3.3. חיפוש העמקה הדרגתית (Iterative Deepening)

חיפוש העמקה הדרגתית משלב את היתרונות של חיפוש לרוחב וחיפוש לעומק.

האלגוריתם מפעיל חיפוש לעומק עם הגבלת עומק 1. אם אינו מוצא פתרון הוא מפעיל חיפוש לעומק עם גבלת עומק 2 וכך הלאה.

שלמות: אלגוריתם החיפוש על ידי העמקה הדרגתית הינו שלם.

אופטימליות: אלגוריתם ההעמקה ההדרגתית מוצא את הפתרון הקצר ביותר.

סיבוכיות זיכרון: ליניארית כמו חיפוש לעומק. חיפוש העמקה הדרגתית לינארי באורך הפתרון הקצר ביותר.

סיבוכיות זמן: $t=\sum_{i=0}^{L}(L+1-i)\cdot b^{i}$

3.4. חיפוש מחיר אחיד (Uniform Cost Search)

האלגוריתם שומר כמו חיפוש לרוחב רשימה של צמתים פתוחים.

הצומת הבא לפיתוח הוא הצומת שמחיר המסלול אליו מינימלי.

האלגוריתם עוצר כאשר צומת בעל מחיר מסלול מינימלי הינו צומת מטרה.

שלמות: ללא תנאי נוסף האלגוריתם אינו שלם. אם פונקצית המחיר חסומה מלמטה בחסם חיובי אזי חיפוש מחיר-אחיד הינו שלם.

אופטימליות: האלגוריתם מחזיר את מסלול הפתרון בעל המחיר המינימלי.

```
Uniform-cost-search(problem)

  OPEN ← ( make-node(problem[init-state], NIL, 0) )
  CLOSE ← NIL
  while open ≠ ( ) do
    next ← pop(OPEN)
    CLOSE ← CLOSE ∪ {next}
    if problem[goal-test](next) then
      return(path(new))
    loop for s in expand(next[state])
      if not(find-state(s, CLOSE) then
        new-cost ← next[g]+cost(next[state], s)
        old-node ← find-state(s, OPEN)
        if old-node then
          if old-node[g] > new-cost then
            old-node[g] ← new-cost
            old-node[parent] ← next
            ; Insert an item into a sorted list according to field g
            insert(old - node, OPEN - {old-node}, g)

        else ; no node with same state in open
          new ← make-node(s, next, new-cost)
          insert(new, OPEN, g)
    return(FAIL) ; OPEN is empty - no solution
```

סיבוכיות זמן: נסמן ב- b את מקדם הסיעוף. נניח שקיים פתרון שמחירו C. במקרה הגרוע ביותר משקל כל הקשתות זהה, ונסמנו δ. הפתרון יימצא במרחק $\frac{C}{\delta}$, ולכן יפותחו $\frac{b^{\left\lceil \frac{C}{\delta} \right\rceil}-1}{b-1}$ צמתים.

סיבוכיות זיכרון: מספר הצמתים שנשמרים בכל רגע הוא מספר הצמתים בעץ, ולכן לכל היותר ישמרו $\frac{b^{\left\lceil \frac{C}{\delta} \right\rceil}-1}{b-1}$ צמתים, לפי אותו חישוב שנעשה בסיבוכיות הזמן.

3.5. טבלת סיכום חיפושים לא מיודעים

אלגוריתם חיפוש	שלמות	אופטימליות	סיבוכיות זמן ומקום
BFS	שלם	אם לקשתות מחיר זהה, האלגוריתם אופטימלי.	b מקדם הסיעוף, L אורכו של המסלול אל הפתרון הקצר ביותר. **סיבוכיות הזמן והזיכרון** הן: $O\left(b^L\right)$
DFS עם הגבלת עומק	שלם כאשר הפתרון נמצא במגבלת העומק.	אינו מוצא בהכרח את המסלול הקצר ביותר.	b מקדם הסיעוף, D הגבלת העומק. **סיבוכיות זיכרון**: סיבוכיות הזיכרון הינה: $O(b \cdot D)$. **סיבוכיות זמן**: $O\left(b^D\right)$.
Iterative Deepening	שלם	אופטימלי	**זיכרון**: ליניארית כמו חיפוש לעומק. **סיבוכיות זמן**: $t=\sum_{i=0}^{L}(L+1-i)\cdot b^i$
חיפוש מחיר אחיד	אינו שלם במקרה הכללי. שלם אם פונקציית המחיר חסומה מלמטה.	אופטימלי	**סיבוכיות הזמן והזיכרון** הן: $O\left(b^L\right)$.

4. חיפושים מיודעים

4.1. מבוא

החיפושים בגרפים שראינו עד כה השתמשו רק בהגדרת הבעיה כדי לנסות להגיע אל הפתרון.
אסטרטגיות חיפוש "מיודעות" משתמשות בידע נוסף כדי לזרז את החיפוש.
הידע הנוסף מקודד בדרך כלל בפונקציה להערכת מצבים הנקראת פונקציה יוריסטית.
הפונקציה לרוב מנסה להעריך את המרחק אל המטרה, ובמקרים כאלה נעדיף לפתח ראשית מצבים בעלי ערך יוריסטי נמוך שיותר סביר שיקרבו אותנו אל פתרון הבעיה.

דוגמא:
בעיית הניווט – אנו רוצים להגיע מנקודה A במרחב אל נקודה B.
יוריסטיקה אפשרית אחת: מרחק אווירי. בשטחים ללא מכשול זוהי היוריסטיקה הטובה ביותר. בשטחים עם מכשולים היא עלולה להיות מטעה. (למשל: הדרך הישירה חסומה, אולם יש דרך שעוקפת את המכשול, שבתחילה תראה כאילו היא מאריכה את המסלול).
יוריסטיקת מרחק מנהטן: יוריסטיקה לשימוש במרחבי סריג. המרחק של מצב מהמטרה מוערך על ידי $\Delta X + \Delta Y$.

4.2. Greedy Best-First Search

הרעיון: בכל שלב בחיפוש נפתח את הצומת המבטיח ביותר על פי הפונקציה היוריסטית.
נשמור את כל הצמתים שלא פותחו. נשמור גם את הצמתים שכבר פותחו כדי למנוע ביקורים חוזרים.

תכונות:

- האלגוריתם שלם במרחבים סופיים ואינו שלם במרחבים אין סופיים.
- הפתרונות אינם בהכרח אופטימליים.
- דרישות הזמן והזיכרון תלויות בפונקציה היוריסטית.

```
Best-first(init)
  OPEN ← { node(init, h(init), NIL)} ; CLOSE ← {};
  Loop while OPEN is not empty
  N ← POP(OPEN) ; CLOSE ← CLOSE ∪ m{N};
  Loop for s in SUCC(N.state)
    If NOT(member-state(s, OPEN ∪ CLOSE) then
      If GOAL(s) then return(s || trace(N))
      OPEN ← Insert(node(s, h(s), N), OPEN, h)
```

init זהו המצב ההתחלתי.

הפונקציה h זוהי הפונקציה היוריסטית המתאימה ערך לכל צומת.

כל צומת מורכבת מ: מצב, ערך יוריסטי ומצביע אל האב.

הפונקציה Insert מכניסה את הצומת החדש ממוין, לפי הערך שלו h. בצורה כזו כל פעם הערך הבא שנבחר לפתח יהיה הצומת שערכו היוריסטי הוא הקטן ביותר.

5. אלגוריתמים לחיפוש יוריסטי מקומי

5.1 מבוא

אלגוריתמי חיפוש מקומי שומרים מצב אחד בזיכרון (או מספר מצבים קטן).
בכל צעד עוברים לאחד השכנים של המצב הנוכחי. בד"כ בחיפושים מסוג זה אנחנו לא שומרים את המסלול בו אנו עוברים.

אלגוריתמי חיפוש מקומי מתאימים לבעיות בהן המסלול אינו מעניין, או בעיות במרחבים בעלי מקדמי סיעוף גדולים.
אלגוריתמי חיפוש מקומי הם חיפושים מיודעים המשתמשים בפונקציה יוריסטית על מנת לבחור את הצומת איתו הם ימשיכו.

hill-climbing .5.2

5.2.1. חיפוש hill-climbing

החיפוש מתחיל במצב ההתחלתי הנתון. בכל שלב מפתחים את כל השכנים ובוחרים את השכן הטוב ביותר. האלגוריתם עוצר אם הגיע למצב מטרה או אם השכן הטוב ביותר אינו טוב יותר מהמצב הנוכחי.

```
SAHC(state) ;Steepest ascent hill-climbing
  current ← state
  loop until resources exhausted
    best-val ← infinity; best-states ← nil
    loop for op in legal-operators(current)
      new ← op(current); new-val ← h(new)
      if goalp(new) then return(new)
      if new-val < best-val then
        best-val ← new-val; best-states ← {new}
      else if new-val = best-val then
        best-states ← best-states ∪ {new}
      if best-val ≥ h(current) then return(current)
      current ← select-random(best-states)
  return(current)
```

5.2.2. חיפוש Stochastic hill-climbing

אלגוריתם הזהה ל-hill-climbing אולם מוכן לצעוד בכיווני שיפור שאינם התלולים ביותר.

האלגוריתם בוחר אקראית מבין הצעדים המשפרים עם הסתברות פרופורציונית לעוצמת השיפור.

```
Stochastic-hill-climbing(state)
  current ← state; current-val ← h(state)
  loop until resources exhausted
    improving-moves ← {}
    loop for op in legal-operators(current)
      new ← op(current); new-val ← h(new)
      if goalp(new) then return(new)
      improvement ← current-val - new-val
      if improvement > 0 then
        improving-moves = improving-moves ∪ {(op , improvement)}
    if improving-moves={} then return(current)
    current ←
      Select randomly from improving-moves with probability
      proportional to improvement
    current-val = h(current)
  return(current)
```

First-choise hill-climbing .5.2.3

ישנם תחומים בהם מקדם הסיעוף הינו גדול מאוד. במקרים כאלו יידרשו אלפי צעדים כדי לחשב את כל הבנים של המצב הנוכחי. במקרים כאלו נעדיף אלגוריתם המגריל אופרטורים ומנסה אותם עד אשר מושגת התקדמות.

```
First-choice-stochastic-hill-climbing(state)
  current ← state
  loop until resources exhausted
    improved ← false
    loop for op in random-mix(legal-operators(current))
                                                until improved
      new ← op(current)
      if goalp(new) then return(new)
      if h(new) < h(current) then
        improved ← true; current ← new
    if not improved then return(current)
  return(current)
```

Simulated-annealing .5.3

האלגוריתמים הקודמים התעקשו לבחור רק צעד משפר או לא מזיק. אסטרטגיה כזו תביא להתקעות במינימום מקומי. Simulated-annealing מתירה צעדים למעלה. ההסתברות לבחירת צעדים אלו יורדת ככל שמתקדם החיפוש.

Random-restart hill climbing .5.3.1

מפעילים את hill climbing באופן איטרטיבי. כאשר החיפוש נתקע, החיפוש מתחיל ממצב התחלתי חדש המיוצר רנדומלית.

5.4. אלגוריתם *A

אלגוריתם *A הינו למעשה best-first המתחשב בדרך שכבר **עשה** וגם בדרך **הצפויה** כדי להעריך צומת. האלגוריתם מעריך צומת n באמצעות הפונקציה $f(n)=g(n)+h(n)$ כאשר $g(n)$ הינו מחיר המסלול הקצר ביותר לצומת n שמצאנו עד כה ו- $h(n)$ הינו המרחק המוערך למטרה.

אם $h(n)$ תמיד אופטימית, ניתן להוכיח כי *A מחזירה פתרון בעל מחיר מינימלי.

הסכם: כאשר מדברים על best-first מניחים שהפונקציה משתמשת ביוריסטיקה כדי להגיע למטרה מהר ככל האפשר ללא התחשבות באורך הפתרון.

בכל פעם אנו בוחרים לפיתוח את הצומת שהפונקציה היוריסטית נותנת לו את הערך הנמוך ביותר. אנחנו מבצעים זאת על ידי שמירת הצמתים הפתוחים ברשימה ממוינת.

5.4.1. תיאור האלגוריתם

נתונים:

1. מצב התחלתי S_i.
2. פרדיקט לבדיקת מצב סופי: $G(s)=TRUE \Leftrightarrow s\in S_g$.
3. פונקצית מעבר $succ:S\to 2^S$ המקבלת מצב ומחזירה קבוצת מצבים.
4. פונקצית מחיר $Cost:\{\langle s_1,s_2\rangle \mid s_1\in S, s_2\in succ(s_1)\}\to\mathbb{R}^+$ החסומה מלמטה.
5. פונקציה יוריסטית $h:S\to\mathbb{R}^+$ המקבלת מצב ומחזירה הערכה של מחיר המסלול המינימלי למצב כלשהו בקבוצת מצבים המטרה.

כמו כן, נגדיר כל צומת על ידי החמישיה Node(state, parent, g, h, f).

```
ASTAR(state)
  OPEN ← (node(state, NIL, 0, h(state), h(state)))
  While OPEN ≠ ( )
    ; Take the next node to process, mark it is closed.
    next ← pop(OPEN)
    push (next, CLOSE)

    ; If we reached the target, print the way to it and
    ; finish. We knows that it is the shortest way we can find.
    if goalp(next.state) then return(trace(next))

    ; For each son s of the selected node
    for s in succ(next.state)

      ; Calculate the cost of s
      new-cost ← next.g+cost(next.state, s)

      ; Check if we already found it earlier,
      ; and also we didn't pass it yet (Not in close list)
      old-node ← find-state(s, OPEN)

      if old-node ≠ {} then

        ; OK, we found it earlier. Let check if we found
        ; better path to that node. If yes, we need to reenter
        ; it to the open list, according to its new cost
        if old-node.g > new-cost then
          old-node.g ← new-cost; old-node.f ← old-node.g+old-node.
          old-node.parent ← next; delete(old-node, OPEN);
          ; Reinsert old node according to new f
          insert(old-node, OPEN, f)
        end

      ; If the node is already in the closed list, and also
      ; we find better cost for it, we should take it
      ; back to the OPEN list
      else old-node ← find-state(s, CLOSE)
      if old-node ≠ {} then
      if old-node.g > new-cost then
        old-node.g ← new-cost; old-node.f ← old-node.g+old-node.
        old-node.parent ← next; delete(old-node, CLOSE);
        insert(old-node, OPEN, f)
      end

      ; If we are here, it means that we just reached that node
      ; in the first time. insert it to the OPEN list as new node
      else
        insert(node(s, next, new-cost, h(s), h(s)+new-cost), OPEN, f)
      end;
  end;
  return FAIL
```

5.4.2. תכונות פורמליות של A*

- האלגוריתם עוצר על גרפיים סופיים.
- האלגוריתם שלם הן על גרפים סופיים והן על גרפים אינסופיים.
- האלגוריתם קביל (מחזיר פתרון בעל מחיר מינימלי) בתנאי ש- h קבילה.

הוכחת אי עצירה: האפשרות היחידה לאי עצירה היא מעגל, אולם זה לא יכול לקרות כי נגיע שוב לצומת n וניתן לה לכאורה מחיר גבוה יותר ממה שכבר היה לה.

5.4.3. קבילות של A*

הגדרה: אלגוריתם חיפוש הוא **קביל** אם מובטח שהוא ימצא פתרון אופטימלי (בעל מחיר מינימלי) כאשר קיים פתרון.

סימונים:

$g^*(n)$ - המחיר הנמוך ביותר של מסלולים מהמצב ההתחלתי S_i למצב n.

$h^*(n)$ - המחיר הנמוך ביותר של מסלולים ממצב n למצב כלשהו ב- S_g

$C^* = h^*(S_i)$ - מחיר אופטימלי לפתרון

$f^*(n) = g^*(n) + h^*(n)$ - מחיר מינימלי של מסלול מ- S_i למצב כלשהו ב- S_g.

הגדרה: פונקציה יוריסטית נקראת **פונקציה קבילה** אם לכל n מתקיים: $0 \le h(n) \le h^*(n)$ (h תמיד אופטימית בהערכת המחיר למטרה).

מתקיים כי עבור כל פונקציה קבילה, $\forall s, s \in S_g$ מתקיים בהכרח $h(s) = 0$.

דוגמאות ליוריסטיקות קבילות: עבור בעיות מפה – מרחק אווירי, עבור בעיות סריג – מרחקי מנהטן.

למה 1: בכל שלב לפני ש-A* עוצרת, קיים צומת n ברשימת ה-OPEN השיך למסלול אופטימלי המקיים $f(n) \le C^*$.

הגדרה: נאמר שפונקציה יוריסטית h_2 **יותר מיודעת** מפונקציה יוריסטית h_1 אם שתיהן קבילות, ולכל n שאינו מטרה מתקיים $h_2(n) > h_1(n)$.

משפט: A* המשתמש ב- h_1 יפתח כל צומת שיפתח A* המשתמש ב- h_2 (כלומר שימוש ביוריסטיקה יותר מיודעת מביא לחיפוש יעיל יותר).

הגדרה: פונקציה יוריסטית h תקרא **פונקציה מונוטונית** אם $\forall s \in S, \forall s' \in SUCC(s)$ מתקיים כי $\left[h(s) - h(s') \le COST(s, s')\right]$.

כלומר, איננו מסתפקים בכך שהפונקציה היוריסטית תהיה אופטימית באופן גלובלי אלא דורשים שהיא תהיה אופטימית ביחס לכל צעד בדרך אל המטרה.

(הפונקציה יכולה לעלות ולרדת, ועדיין תמיד להיות אופטימית ביחס למרחק אל המטרה. פונקציה כזו אינה מונוטונית).

טענה: שימוש ביוריסטיקה מונוטונית מבטיח שהפונקציה f תהיה מונוטונית עולה.

משפט: עבור A^* המשתמש ב- h מונוטונית מתקיים: $\forall n \in CLOSED\left[g(n) = g^*(n)\right]$ (כלומר מובטח שכל צומת שפותח מצביע למסלול האופטימלי אל מצב ההתחלה).

מסקנה: אין צורך להעביר צמתים מ-CLOSE ל-OPEN ופיתוחם מחדש נחסך.

משפט: עבור יוריסטיקה מונוטונית, A^* היא גם אופטימלית במשאבי חיפוש.

5.4.4. השוואה בין A^* עם פונקציה קבילה ל-Uniform Cost Search:

מדד	Uniform Cost Search	A^* עם פונקציה יוריסטית קבילה
טיב הפתרון	מחזיר פתרון אופטימלי	מחזיר פתרון אופטימלי
סיבוכיות זמן	$O(b^L)$	$O(b^L)$
סיבוכיות זיכרון	$O(b^L)$	$O(b^L)$

במקרה הגרוע ביותר של UCS – לכל הקשתות מחיר זהה.

במקרה הגרוע ביותר של A^*, הפונקציה היוריסטית תחזיר ערך זהה לכל הקשתות. במקרה זה האלגוריתמים יתנהגו בצורה זהה.

5.4.5. החלשת דרישת האופטימליות

נוותר על האופטימליות כדי להשיג זמן חיפוש קצר יותר.

פונקציה הערכה עם משקולות: $f_w(n)=(1-w)\cdot g(n)+w\cdot h(n), 0\le w\le 1$.

$w=0$ יוצר חיפוש Uniform-cost. $w=0.5$ פירושו חיפוש A* רגיל. $w=1$ הינו Best-first.

f_w קבילה עבור h קביעה ו- $0\le w\le 0.5$. עבור $w>0.5$ מתקיים כי f_w אינה בהכרח קבילה גם אם h קבילה.

הגדלת w תשיג לנו פתרון בזמן קצר יותר, אולם איכות הפתרון הממוצעת תרד (מסלול ארוך יותר).

<u>הגדרה</u>: יהי C^* המחיר האופטימלי ויהי $\varepsilon\ge 0$. אלגוריתם המבטיח מציאת פתרון בעל מחיר של כל היותר $(1+\varepsilon)C^*$ נקרא **ε-קביל**.

5.4.6. משקלות דינמיים

יהי N חסם עליון על עומק צמתי הפתרון (עומק במובן של מספר קשתות בגרף). נגדיר $d(n)$ כעומק הצומת n (מספר הקשתות במסלול מהשורש). נגדיר:

$$f_{dw}(n)=g(n)+h(n)+\varepsilon\left(1-\frac{d(n)}{N}\right)h(n)$$

<u>משפט</u>: אם h קבילה אז A* שמשתמש ב- f_{wd} הינו ε-קביל.

האלגוריתם מתחיל עם משקל גדול יותר ל- h: $(1+\varepsilon)$.

כאשר עומק החיפוש גדל דואג האלגוריתם למנוע הסתבכויות בענפים עמוקים מדי על ידי הקטנת משקלו היחסי של h.

5.4.7. אלגוריתם Iterative-deepening A*

אחד מחסרונותיו של A* הוא דרישת הזיכרון שלו.

האלגוריתם הבא הינו שיפור של אלגוריתם ההתקדמות ההדרגתית. הוא מאפשר שימוש ביוריסטיקה ומבטיח פתרונות קבילים. דרישות הזיכרון הינן ליניאריות במחיר הפיתרון האופטימלי.

האלגוריתם:

1. קבע את $h(s_i)$ כעומק המקסימלי.
2. ערוך חיפוש לעומק. עצור בכל ענף עבורה f גדולה מהסף. אם מצאת פתרון, החזר אותו.
3. אם בחיפוש לעומק לא נמצא פתרון, הגדל את הסף בחריגה המינימלית שנמצאה בשלב 2, וחזור לשלב 2.

```
DFS-C (state, g, path)
  f ← g+h(state)
  if f > max-f then new-max-f ← min(new-max-f,f)
  else
    if goalp(state) return(path)
    else
      for c in succ(state)
        DFS-C(c, g+cost(state,c), path state)

IDA* (state)
  max-f ← h(state)
  new-max-f ← infinity
  loop while resources are available
    solution ← DFS-C(state,0,())
    if solution <> () then return (solution)
    max-f ← new-max-f
```

5.5. חיפוש דו כיווני

ישנן בעיות עבורן ניתן למצוא את הפונקציה ההופכית לפונקצית המעבר. במקרים כאלו ניתן לחפש בשני כיוונים במקביל: ממצב ההתחלה לכיוון המצב הסופי ומהמצב הסופי לכיוון מצב ההתחלה.

6. משחקים

6.1. מבוא

גורמים להתעניינות במשחקים:

- דרך פשוטה לחקור מצב של תחרות (מטרות סותרות) בין סוכנים אינטליגנטיים.
- סימן לאינטליגנציה אצל בני אדם.
- חוקים מעוטים וברורים ומדדי הצלחה אובייקטיביים (למשל דירוג בשחמט).
- מרחבי חיפוש מאוד גדולים.

6.2. Perfect Information Games

משחקים כמו שח, דמקה נקראים-**Two-player, sum-zero, perfect information games**.
במשחקים אלו משחקים 2 שחקנים יריבים. סכום אפס פירושו שכל שחקן רואה כישלון של היריב כהצלחה שלו. המונח **perfect information** נועד להבדיל משחקים כמו שח ודמקה ממשחקים כמו פוקר; שם אין בידי השחקן את כל האינפורמציה הנחוצה לו.

6.3. עץ משחק

עץ המשחק מתאר את כל אפשרויות התפתחות המשחק.
"שורש" העץ הינו המצב הנוכחי, ממנו מתפצל ענף עבור כל צעד חוקי של המחשב.
מכל ענף כזה מתפצלים ענפים עבור על התגובות החוקיות של היריב.
ההתפצלות נפסקת במצבי סיום המסומנים ב"נצחון" (למחשב), "תיקו", או "הפסד".

בעוד שבמערכת חד סוכנית ההחלטות נעשות כולן בידי הסוכן היחיד, במערכת רב סוכנית ההחלטות נעשות ע"י כמה סוכנים. במשחקים כמו שחמט ההחלטות נעשות לסרוגין: פעם ע"י השחקן ופעם ע"י החשקן (הסוכן) היריב. בכל מקום שתור השחקן לשחק הוא יבחר כמובן בצעד הטוב ביותר עבורו.
בכל מקום שתור היריב לשחק קיימת אי ודאות לגבי ההחלטה הצפויה.

MINIMAX .6.4

האסטרטגיה המקובלת ביותר: כיון שאין ודאות לגבי בחירת היריב אנו מניחים את הגרוע ביותר - כלומר, שהיריב בוחר את הצעד הגרוע ביותר עבור השחקן.

לאסטרטגיה כזו קוראים מינימקס: השחקן בוחר בצעד הטוב ביותר ("המקסימלי") והיריב את הרע ביותר עבור השחקן ("המינימלי").

משחקים כמו שחמט, דמקה וכו' הינם **משחקי סכום אפס**: ניצחון של שחקן אחד הינו הפסד של השני. במשחקים כאלה ניתן לבטא את האסטרטגיה גם: מניחים שהיריב בוחר את הצעד הטוב ביותר עבורו.

6.4.1. פרוצדורת מינימקס

- התחל מרמת העלים של עץ המשחק.
- בכל פיצול, אם תור השחקן לשחק סמן את נקודת הפיצול בסימון הטוב ביותר של הילדים.
- אם תור היריב לשחק סמן את נקודת הפיצול בסימן הגרוע ביותר של הילדים.
- בסיום התהליך מסומן שורש העץ בסימון. זהו הערך הטוב ביותר שיכול השחקן להשיג במשחק אם היריב משחק אופטימלית.
- ערך זה מובטח. אין שום סיכוי שנקבל ערך גרוע יותר. לכל היותר, אם ישחק היריב באופן לא אופטימלי נקבל תוצאה טובה יותר.

<u>בעיה</u>: מספר המצבים האפשריים עד לנצחון המשחק הינו לרוב אדיר ולא ניתן מעשית לחישוב.

<u>פתרון</u>: במקום לפתח את העץ עד שמגיעים למצבי סיום, מפתחים אותו עד לעומק מסוים.

העומק נקבע לפי הזמן שמוקצב לביצוע מהלך.

<u>בעייה</u>: כיצד נסמן את עלי העץ?

<u>פתרון</u>: שימוש בפונקצית הערכה. פונקציה המחזירה לכל לוח ציון מספרי.

הפונקציה מחזירה ערכים גבוהים עבור מצבים שמוערכים כטובים לשחקן ומחזירה ערכים נמוכים עבור מצבים שמוערכים כרעים לשחקן (או טובים ליריב).

<u>המשפט המרכזי של MINIMAX</u>: אם M הוא ערך המינימקס המוחזר על ידי חיפוש בעומק D, אזי קיימת אסטרטגיה <u>המבטיחה</u> כי אחרי D צעדים נגיע למצב שבו הפונקציה היוריסטית היא לפחות M, ויותר מכך, זוהי היוריסטיקה המקסימלית שניתן להבטיח בתום D צעדים.

6.4.2. פרוצדורת מינימקס מעודכנת

- התחל מרמת העלים של עץ המשחק.
- בכל פיצול, אם תור השחקן לשחק סמן את נקודת הפיצול בסימון המקסימלי ביותר של הילדים.
- אם תור היריב לשחק סמן את נקודת הפיצול בסימן המינימלי ביותר של הילדים.
- בסיום התהליך מסומן שורש העץ בסימון. זהו הערך הטוב ביותר שיכול השחקן להשיג במשחק אם היריב משחק אופטימלית.
- ערך זה מובטח. אין שום סיכוי שנקבל ערך גרוע יותר. לכל היותר, אם ישחק היריב באופן לא אופטימלי נקבל תוצאה טובה יותר.
- המחשב יבצע את הצעד המוביל לבן בעל ציון המינימקס הגבוה ביותר.

```
Minimax(board, depth, type)
  ; If we reached the depth limit, use the heuristic function
  ; to rank the board.
  if depth=0 then
    return(evaluate(board))
  else

    ; If it is our turn now, make the choice that will
    ; give us the maximum result
    if type=max then
      cur-max ← -infinity
      loop for b in succ(board)
        b-val ← minimax(b, depth-1, min)
        cur-max ← max(b-val, cur-max)
      end loop
      return cur-max

    ; If it is the enemy turn, choose the best board for him.
    else (type=min)
      cur-min ← infinity
      loop for b in succ(board)
        b-val ← minimax(b, depth-1, max)
        cur-min ← min(b-val,cur-min)
      end loop
      return cur-min
```

6.4.3. גיזום - (α-β pruning)

$\alpha-\beta$ זוהי שיטה ל"גיזום" ענפים בעץ המשחק, המקיימת שעץ רגיל ועץ גזום יחזירו את אותו הערך. האלגוריתם מוותר על פיתוח ענפים שאינם יכולים לשנות את ערך המינימקס של השורש.

עקרון פעולה: אם תוך כדי החיפוש בענף מתברר שהיריב יכול לתת תשובה גרועה יותר מהאלטרנטיבה שיש ביד עד כה, אל תטרח לבדוק האם יש לו תשובה עוד יותר גרועה.

למה בעצם שנרצה לגזום את העץ? התשובה היא שעל ידי התעלמות מחלקי העץ הלא רלוונטיים (גיזומם), אנחנו מסוגלים באותו זמן להכנס עמוק יותר אל חלקי העץ שכן מעניינים אותנו, ולהסתכל יותר צעדים קדימה במשחק.

חסמי החיתוך

חסם α: חסם החיתוך לצומת j מסוג Min הוא חסם תחתון הנקרא α. זהו הערך הגבוה ביותר שקים לעת עתה לכל אבות ה-Max של j. ניתן להפסיק את פיתוח j ברגע שהערך שלו שווה ל- או קטן מ-α.

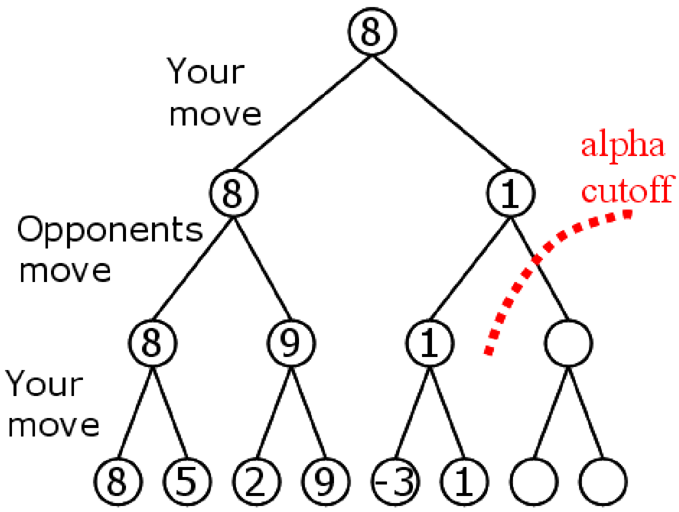
חסם β: חסם החיתוך לצומת j מסוג Max הוא חסם עליון הנקרא β. זהו הערך הנמוך ביותר שקים לעת עתה לכל אבות ה-Min של j. ניתן להפסיק את פיתוח j ברגע שהערך שלו שווה ל- או גדול מ-β.

```
Minimax-ab(board, depth, type, alpha, beta)
  if depth=0 then return(evaluate(board))
  else
    if type=max then
      cur-max ← -infinity
      loop for b in succ(board)
        b-val ← minimax-ab(b,depth-1,min,alpha,beta)
        cur-max ← max(b-val, cur-max)
        alpha ← max(cur-max, alpha)
        if cur-max ≥ beta then finish loop
      end loop
      return cur-max
    else (type=min)
      cur-min ← infinity
      loop for b in succ(board)
        b-val ← minimax-ab(b,depth-1,max,alpha,beta)
        cur-min ← min(b-val,cur-min)
        beta ← min(cur-min,beta)
        if cur-min ≤ alpha then finish loop
      end loop
      return cur-min
```

הקריאה לפונקציה Minimax תעשה על ידי:

Minimax-ab(board, D, max, $-\infty$, $+\infty$)

דוגמא לגיזום:



הערך של התנועה שאנו מבצעים הוא 8 (עד לנקודה שבה מתבצע החיתוך). אם נלך בשורש ימינה, לפי מה שאנחנו יודעים היריב מסוגל להגיע לערך 1. היריב לא יעשה פעולה שתגרום לערך זה לעלות, ולכן הערך של תת העץ הימני יהיה קטן תמיד מ-8. לפיכך ניתן להתעלם מהצמתים הנותרים.

6.4.4. סידור יוריסטי

ניתן לחשב שבמקרה האופטימלי גיזום $\alpha-\beta$ נותן חסכון בפקטור של שורש => אפשרות לחיפוש בעצים בעומק כפול תחת משאבי זמן זהים. מכאן, כדאי להשקיע בסידור הילדים.

הסדר האידיאלי אינו ידוע (אחרת העץ היה פתור - היינו יודעים מהו הצעד הטוב ביותר).

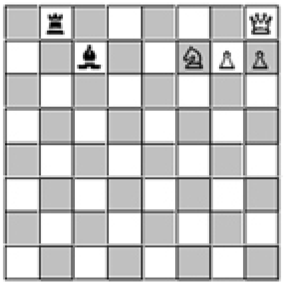
ניתן לסדר את הילדים ע"י שימוש בפונקצית ההערכה. כיוון שהפעלת פונקצית ההערכה יקרה, משתמשים ביוריסטיקות גסות יותר. למשל: מנסים קודם כל לקיחות, אח"כ איומים, אח"כ צעדים קדימה ולבסוף צעדים אחורה.

6.4.5. אפקט האופק (The Horizon Effect)

תופעת בת היענה - "אם אסתיר את עיני - הרע יעלם".

נניח עץ אחד עם עומק קבוע, למשל 2. נביט בדוגמא הבאה: לכידת המלכה ע"י הצריח של היריב אינה ניתנת למניעה. הצבת הסוס לחסימת המלכה לא תציל אותה, אבל תדחה את הלכידה לעומק 4.

עומק 4 הוא מעבר ל"אופק" לכן הצבת הסוס נראה כמהלך טוב.



6.4.6. חיפוש עד רגיעה (Waiting for Quiescence)

בעיה: בחיפוש לעומק קבוע יתכן והחיפוש יפסק באמצע סדרה של החלפת כלים.

הפתרון: החיפוש ממשיך כל עוד ישנן תנודות חזקות במצב (למשל לקיחת כלי או התקרבות של חיל לשורת ההכתרה).

כלומר עץ החיפוש אינו בעל עומק אחיד. מעבר לאופק האחיד ישנם ענפים עמוקים יותר.

7. שימוש בלוגיקה לייצוג ידע

המטרה שלנו: לאפשר למחשב להוכיח משפטים.

הבעיה איננה כריעה לחלוטין. המחשב מסוגל להוכיח משפט כאשר הוא נכון, אבל אם המשפט אינו נכון, הוא עלול להיכלא ללולאה אינסופית בניסיון להוכיחו.

תחשיב היחסים, המוכר מלוגיקה, מאפשר לנו לייצג ידע בצורת נוסחאות.

כשנרצה להוכיח נוסחה P נבנה מאגר נוסחאות אותו נקבע כאקסיומות, וננסה להוכיח את הנוסחה.

<u>פרוצדורת ההוכחה של נוסחה</u>:

1. הכנס את קבוצת האקסיומות ל-S
2. אם הנוסחה המבוקשת נמצאת ב-S, החזר "כן".
3. אחרת הפעל את כללי ההיסק על נוסחאות ב-S.
4. הוסף את הנוסחאות החדשות ל-S.
5. חזור לשלב 2.

7.1. רזולוציה

תהליך ההוכחה שתואר לעיל אינו יעיל. דרך יעילה יותר להוכיח משפטים היא הרזולוציה.

לפני שנתאר את התהליך, נתאר מספר כלים שיעזרו לנו.

7.1.1. העברת הפסוקים לצורת CNF - הרעיון

בגדול: ביטוי מצורת CNF זהו ביטוי מהצורה:

$$\left(L_{11} \vee L_{12} \vee \ldots L_{1n_1}\right) \wedge$$
$$\ldots$$
$$\left(L_{m1} \vee L_{m2} \vee \ldots L_{mn_m}\right)$$

כאשר L_{ij} הוא פסוק אטומי או שלילת פסוק אטומי.

כל אחת מהשורות הנ"ל נקראת פסוקית (clause).

לנוסחה הנ"ל מתייחסים כקבוצת פסוקיות. לכל פסוקית מתייחסים כקבוצת ליטרלים:

$$\left\{\left\{L_{11},L_{12},...,L_{1n_1}\right\},...,\left\{L_{m1},L_{m2},...,L_{mn_m}\right\}\right\}$$

קיים אלגוריתם להמרת כל נוסחה לנוסחה מצורת CNF.

7.1.2. כלל הרזולוציה

ההנחה השנייה שאלגוריתם הרזולוציה מסתמך עליה היא ההנחה כי מ- $\left(\left(A\vee L\right)\wedge\left(B\vee\neg L\right)\right)$ נובע $\left(A\vee B\right)$.

הוכחת ההנחה:

נניח כי $\left(\left(A\vee L\right)\wedge\left(B\vee\neg L\right)\right)$ נכון. אזי $\left(A\wedge L\right)$ נכון וגם $\left(B\wedge\neg L\right)$ נכון.

אם A נכון, אזי בפרט $\left(A\vee B\right)$ וגמרנו.

אם A אינו נכון, אז L חייב להיות נכון, ולכן $\neg L$ אינו נכון, ומכאן ש- B נכון.

אם B נכון, אזי בפרט $\left(A\vee B\right)$ וגמרנו.

סוף ההוכחה.

נשתמש בהנחה זו בהמשך כדי לצמצם פסוקיות ולקבל ביטויים פשוטים יותר.

7.1.3. אלגוריתם הרזולוציה: (ללא משתנים)

נתונה קבוצה אקסיומות A, נוסחה P אותה אנחנו רוצים להוכיח.

הרעיון: נוסיף את שלילת P אל קבוצת האקסיומות, וננסה להגיע לפסוקית ריקה (סתירה). אם הגענו, הרי שהוכחנו על דרך השלילה כי הטענה נכונה. (אם הגענו לפסוקית ריקה סימן שהקבוצה שלנו כללה טענה והיפוכה. אנחנו יודעים שהאקסיומות נכונות, ולכן נובע ששלילת הנוסחה אינה נכונה, כלומר הנוסחה נכונה).

האלגוריתם:

1. הפוך את $A \wedge \neg P$ לקבוצת פסוקיות E.
2. אתחל: $D \leftarrow E$.
3. המשך עד אשר לפחות מהתנאים הבאים מתקיים:
   - נמצאה סתירה.
   - לא ניתן להמשיך.
   - המשאבים שהוקצו להוכחה נצרכו.

   a. בחר שתי פסוקיות ב- D מהצורה: $A = \{A_1, ..., A_n, L\}, B = \{B_1, ..., B_m, \neg L\}$.
   b. $D \leftarrow D \bigcup \{\{A_1, ..., A_n, B_1, ..., B_m\}\}$.
4. בסיום:
   a. אם הפרוצדורה עצרה בגלל סתירה, החזר "כן". (P נובעת מ-A).
   b. אם הפרוצדורה עצרה מחוסר אפשרות להמשיך, החזר "לא".
   c. אם הפרוצדורה עצרה מחוסר משאבי חיפוש, החזר "לא ידוע".

<u>תכונות תהליך הרזולוציה:</u>

- התהליך הוא נאות – כלומר כל פסוקית שנגזרה מקבוצת פסוקיות D על ידי תהליך הרזולוציה נובעת לוגית מ-D.
- אם קבוצת פסוקיות D אינה ספיקה, ניתן לגזור מ-D את הפסוקית הריקה.

7.1.4. רזולוציה עם משתנים

במקרה הכללי יכילו הנוסחאות גם משתנים. במקרה זה הצעד בו אנו בוחרים את הפסוקיות מסובך יותר. במקום לחפש שני ליטרים זהים, כאשר אחד מהם מופיע עם סימן שלילה, אנו נחפש שני ליטרלים שניתן להפוך אותך לזהם על ידי הצבה מתאימה.

לדוגמא, ניתן להפוך את הליטרלים father(avraham, yzhak) ו-father(X, Y) לזהים על ידי ההצבה X/avraham, Y/yzhak, אולם אין הצבה שתהפוך את הליטרלים father(avraham, yzhak) ואת father(X, X) לזהים.

לתהליך המוצא הצבה שהופכת שני פסוקים לזהים קוראים **האחדה (unification)**.

אלגוריתם הרזולוציה:

נתונות אוסף הנחות (נוסחאות בנויות כהלכה, אותן נכנה אקסיומות) $A_1,...,A_n$ ונתון משפט T אותו נרצה להוכיח.

1. הפוך את הנוסחה $A_1 \wedge ... \wedge A_n \wedge \neg T$ לאוסף של *פסוקיות* $\{C_1,...,C_m\}$ (בצורת CNF).
2. התחל עם אוסף הפסוקיות $P = \{C_1,...,C_m\}$.
3. בצע את הלולאה הבאה:
   a. בחר 2 פסוקיות $C_h = \{L_1,...,L_h\}, C_l = \{D_1,...,D_k\}$ כך שקיימים $D_j = \varphi, L_i = \neg\psi$ כאשר ψ, φ ניתנים להאחדה עם הצבה Θ.
   b. תהי $C_{new} = \left[C_h \cup C_k - \{L_i, D_j\}\right]\Theta$.
   c. החלף את שמות המשתנים ב- C_{new} לשמות חדשים שלא מופיעים ב- P.
   d. $P \leftarrow P \cup \{C_{new}\}$.
4. כל עוד לא מתקיים אחד מהשלושה:
   a. הפרוצדורה עצרה בגלל סתירה, החזר "המשפט הוכח"
   b. הפרוצדורה עצרה מחוסר אפשרות להמשיך, החזר "המשפט לא ניתן להוכחה".
   c. הפרוצדורה עצרה מחוסר משאבי חיפוש, החזר "המשאבים שהוקצו אזלו".

7.1.5. פרוצדורות בתהליך הרזולוציה

על מנת לממש את תהליך הרזולוציה צריך לפרט 3 פרוצדורות הנחוצות לתהליך:

1. הפיכת נוסחה כלשהי לקבוצת פסוקיות מצורת CNF.
2. מציאת הצבה מאחדת (יוניפיקציה).
3. הגדרת אסטרטגית חיפוש – איזה זוג ייבחר בשלב 3.a.

העברת נוסחאות לצורת CNF

שלב 1: הסרת סימני הגרירה $\rightarrow$.

כדי לבצע זאת, נשתמש בשקילות: $\boxed{A \rightarrow B \equiv \neg A \vee B}$

דוגמא: $\forall s \forall t\left[\forall x\left[\in(x,s) \rightarrow \in(x,t)\right] \rightarrow \subseteq(s,t)\right]$

יהפוך ל- $\forall s \forall t\left[\neg \forall x\left[\neg \in(x,s) \vee \in(x,t)\right] \vee \subseteq(s,t)\right]$.

שלב 2: הקטן את טווח השלילות לפסוקים אטומיים

כדי לעשות זאת, נשתמש בשקילויות הבאות:

$\neg(\neg A) \equiv A$	$\neg(A \wedge B) = \neg A \vee \neg B$	$\neg(A \vee B) = \neg A \wedge \neg B$
$\neg \exists X\left[P(X)\right] \equiv \forall X\left[\neg P(X)\right]$	$\neg \forall X\left[P(X)\right] \equiv \exists X\left[\neg P(X)\right]$	

דוגמא:

$$\forall s \forall t\left[\neg \forall x\left[\neg \in(x,s) \vee \in(x,t)\right] \vee \subseteq(s,t)\right] \Rightarrow$$
$$\forall s \forall t\left[\exists x \neg\left[\neg \in(x,s) \vee \in(x,t)\right] \vee \subseteq(s,t)\right] \Rightarrow$$
$$\forall s \forall t\left[\exists x\left[\neg\neg \in(x,s) \wedge \neg \in(x,t)\right] \vee \subseteq(s,t)\right] \Rightarrow$$
$$\forall s \forall t\left[\exists x\left[\in(x,s) \wedge \neg \in(x,t)\right] \vee \subseteq(s,t)\right]$$

שלב 3: שנה את שמות המשתנים כך שלא יופיע אותו שם בשני כמתים

לדוגמא, הנוסחה $\forall x P(x) \vee \forall x Q(x)$ תהפוך ל- $\forall x P(x) \vee \forall y Q(y)$.

שלב 4: העבר את כל הכמתים לתחילת הנוסחה תוך שמירה על סדר הופעתם

$$\forall s \forall t\left[\exists x\left[\in(x,s) \wedge \neg \in(x,t)\right] \vee \subseteq(s,t)\right] \Rightarrow$$
$$\forall s \forall t \exists x\left[\left[\in(x,s) \wedge \neg \in(x,t)\right] \vee \subseteq(s,t)\right]$$

שלב 5: הסר את הכמתים הישיים בתהליך סקולומיזציה:

כמת ישי = $\exists$.

1. יהי n מספר הכמתים הכוללים משמאל לכמת הישי. הנוסחה עם הכמת הישי נמצאת בטווח n הכמתים. יהיו $x_1,...,x_n$ המשתנים של הכמתים הללו.
2. החלף את כל ההופעות של המשתנה של הכמת הישי בפונקציה חדשה בעלת שם כלשהו חדש (שאינו מופיע במקום אחר) בעלת n ארגומנטים $x_1,...,x_n$. הפונקציה הזו נקראת **פונקצית סקולם**. לדוגמא, $\forall X_1...\forall X_n \exists Y\left[p\left(Y,X_1,...,X_n\right)\right]$ תהפוך ל- $\forall X_1...\forall X_n\left[p\left(f_{20}\left(X_1,...,X_n\right),X_1,...,X_n\right)\right]$ כאשר f_{20} הינו שם פונקציה חדש.
3. אם לא קיימים כמתים כוליים משמאל, הפונקציה תהיה בעלת 0 ארגומנטים ונקראת **קבוע סקולם**.

$$\forall s\forall t\exists x\left[\left[\in\left(x,s\right)\wedge\neg\in\left(x,t\right)\right]\vee\subseteq\left(s,t\right)\right]\Rightarrow$$
$$\forall s\forall t\left[\left[\in\left(F(s,t),s\right)\wedge\neg\in\left(F(s,t),t\right)\right]\vee\subseteq\left(s,t\right)\right]$$

שלב 6: הסר את כל הכמתים הכוללים

בשלב זה קיימים רק כמתים כוללים. מסירים אותם וזוכרים שכל המשתנים הינם של כמתים כוללים.

$$\forall s\forall t\left[\left[\in\left(F(s,t),s\right)\wedge\neg\in\left(F(s,t),t\right)\right]\vee\subseteq\left(s,t\right)\right]\Rightarrow$$
$$\left[\left[\in\left(F(s,t),s\right)\wedge\neg\in\left(F(s,t),t\right)\right]\vee\subseteq\left(s,t\right)\right]$$

שלב 7: הפוך את הנוסחה לקוניונקציה של דיסיונקטים

נשתמש בשקילות: $\left(A\wedge B\right)\vee C\equiv\left(A\vee C\right)\wedge\left(B\vee C\right)$

$$\left[\left[\in\left(F(s,t),s\right)\wedge\neg\in\left(F(s,t),t\right)\right]\vee\subseteq\left(s,t\right)\right]\Rightarrow$$
$$\left[\in\left(F(s,t),s\right)\vee\subseteq\left(s,t\right)\right]\wedge\left[\neg\in\left(F(s,t),t\right)\vee\subseteq\left(s,t\right)\right]$$

שלב 8: נקרא לכל דיסיונקציה בשם clause. שנה שמות משתנים כך שבכל clause יהיו שמות אחרים.

נשתמש בשקילות הבאה: $\forall X\left[P(X)\wedge Q(X)\right]\equiv\forall X\left[P(X)\right]\wedge\forall X\left[Q(X)\right]$

$$\left[\in\left(F(s,t),s\right)\vee\subseteq(s,t)\right]\wedge\left[\neg\in\left(F(s,t),t\right)\vee\subseteq(s,t)\right]$$
$$\Downarrow$$
$$\left[\in\left(F(x_1,x_2),x_1\right)\vee\subseteq(x_1,x_2)\right]\wedge$$
$$\left[\neg\in\left(F(x_3,x_4),x_3\right)\vee\subseteq(x_3,x_4)\right]$$

האחדה (יוניפיקציה)

הצבה היא אוסף של זוגות $\{x_1/t_1,\ldots,x_n/t_n\}$ כאשר t_i הם ביטויים ו- x_i הם משתנים ומתקיימים התנאים:

1. $x_i\neq x_j$ לכל $i\neq j$.
2. x_i אינו מופיע באף אחד מהביטויים $t_1,\ldots,t_n$.

ניתן להפעיל הצבה σ על נוסחה φ ולקבל נוסחה חדשה על ידי החלפת כל המשתנים בנוסחה המופיעים בהצבה בביטויים המתאימים. מסמנים את הביטוי החדש ב- $\varphi\sigma$.

דוגמא: $P(x,x,y,v)\{x/A,y/F(B),z/w\}=P(A,A,F(B),v)$.

הרכבה של הצבות: תהי σ הצבה. אם בהצבה τ לא מופיעים משתנים המקבלים ערך בהצבה σ ניתן להגדיר את ההרכבה של τ על σ בצורה $\sigma\tau$.

תהי $\sigma=\{x_1/t_1,\ldots,x_n/t_n\}$. תהי τ הצבה כנ"ל, אזי: $\sigma\tau=\{x_1/t_1\tau,\ldots,x_n/t_n\tau\}\cup\tau$.

כלומר, מפעילים את τ על הביטויים של σ ומוסיפים את הזוגות החדשים לזוגות של τ. דוגמא:

$$\{w/G(x,y)\}\{x/A,y/B,z/C\}=\{w/G(A,B),x/A,y/B,z/C\}$$

האחדה (unification) של שני ביטויים היא הצבה ההופכת אותם לזהים.

ייתכן יותר ממאחד (הצבה) אחד לשני ביטויים. מאחד γ של פסוקים φ ו- ψ נקרא **המאחד הכללי ביותר** אם לכל מאחד σ של שני הביטויים קיימת הצבה δ כך ש- $\psi\sigma = \psi\gamma\delta = \varphi\sigma = \varphi\gamma\delta$.

אלגוריתם ליוניפיקציה: $UNIFY(L_1, L_2)$

1. אם $L_1 = L_2$ החזר NIL.
2. אם L_1 הוא משתנה: אם L_1 מופיע ב- L_2 החזר FAIL, אחרת החזר (L_1/L_2)
3. אם L_2 הוא משתנה: אם L_2 מופיע ב- L_1 החזר FAIL, אחרת החזר (L_2/L_1)
4. אם L_1 או L_2 קבועים: אם $L_1 = L_2$ החזר NIL, אחרת החזר FAIL.
5. אם ל- L_1 ול- L_2 מספר ארגומנטים שונה, החזר FAIL.
6. אתחל את SUBST לערך NIL.
7. לכל i החל מ-1 ועד מספר הארגומנטים:
   a. $S \leftarrow UNIFY(\arg(L_1, i), \arg(L_2, i))$.
   b. אם S הוא FAIL החזר FAIL.
   c. הפעל את S על שארית L_1 ו- L_2
   d. $\text{SUBST} \leftarrow APPEND(S, \text{SUBST})$.
8. החזר את SUBST

נשים לב מה האלגוריתם מקבל ומה הוא מחזיר: האלגוריתם מקבל שני ביטויים שלכל אחד מספר פרמטרים כלשהו. אחרי מספר בדיקות, בסעיף 7 העיקרי, אנו כבר יודעים שמדובר בשני ביטויים עם מספר ארגומנטים זהה. ננסה למצוא הצבה עבור כל אחד מהפרמטרים. אם נצליח למצוא הצבה עבור כולם, נחזיר אותה. אחרת נחזיר FAIL. האלגוריתם מחזיר את המאחד הכללי ביותר.

אסטרטגיות רזולוציה

אלגוריתם הרזולוציה לוקח הרבה זמן וזכרון כאשר אנו מחפשים סתירות. אנחנו מנסים לחסוך במשאבים על ידי אסטרטגיות שונות.
נתייחס אל אלגוריתם הרזולוציה כאל חיפוש. מצב הוא אוסף פסוקיות. המצב ההתחלתי הוא האקסיומות ושלילת המשפט, פונקצית המעבר היא כלל הרזולוציה. מרחב המצבים גדול מאוד, ולכן נשתמש באסטרטגית חיפוש.

גישות אפשריות:

a. BFS – ביצוע רזולוציה על כל הפסוקים האפשריים – יוצר רמה שניה של פסוקים. אז נבצע רזולוציה על כל הרמות הקודמות, וכו'.

b. אסטרטגית הסרה – פסוקית שאין לה משלים תיזרק. למשל, אם מופיע $P(x)$ בפסוקית ואין אף פסוקית אחרת המכילה $P(x)$, אז אפשר לזרוק את הפסוקית ולהקטין את מרחב החיפוש.

נכנה בשם **ליטרל טהור** ליטרל שאין לו משלים. בזמן הרזולוציה לא נוצרים ליטרלים טהורים. לכן רק בתחילת התהליך נזרוק פסוקיות בעלות ליטרלים טהורים.

c. הסרת טאוטולוגיות – מכיוון שטאוטולוגיה לעולם לא תביא לסתירה, הרי שהיא מיותרת.

d. רזולוציית יחידה – תמיד עדיף לבצע רזולוציה כאשר אחד מהביטויים באורך 1.

למשל, רזולוציה של פסוקית מאורך 4 ופסוקית מאורך 5 ייתן פסוקית באורך 7. במקרה זה הארכנו את גודל הפסוקית. לעומת זאת אם נבצע רזולוציה בין פסוקית באורך k לפסוקית באורך 1 נקבל פסוקית באורך $(k-1)$.

דגש: רזולוציית יחידה אינה שלמה – יש פעמים שהיא לא מוצאת הוכחה.

רזולוציית יחידה שלמה עבור קבוצת פסוקיות הורן – שבהן יש ליטרל חיובי אחד לכל היותר (ליטרל חיובי = ליטרל ללא סימן $\neg$).

e. אסטרטגית קבוצת תמיכה (Set of support) –

הרעיון של האסטרטגיה: בתהליך הרזולוציה אנחנו מנסים להראות שהנחת השלילה גורמת לסתירה (ולכן אינה ספיקה). סתירה חייבת לערב את שלילת המשפט, מכיוון שאנחנו מניחים שהאקסיומות עקביות. לפיכך בכל שלב נרצה להשתמש בפסוקית של המשפט, או צאצאית של פסוקית כזו, בנוסף לפסוקית מחוץ למשפט.

הגדרה: קבוצת פסוקיות A מהווה **קבוצת תמיכה** עבור קבוצת פסוקיות B אם B-A ספיקה.

אסטרטגית קבוצת התמיכה מחייבת שלפחות אחת משתי הפסוקיות עליהן מופעל כלל הרזולוציה תהיה מתוך קבוצת התמיכה, או צאצאית של קבוצת התמיכה. נבחר בהתחלה את קבוצת התמיכה להיות שלילת המשפט אותו אנו רוצים להוכיח.

מציאת עץ הוכחה – נעשית על ידי טיפוס מעלה מהפסוקית הריקה אל ההורים שלה וכו' עד שמגיעים לאקסיומות, או לשלילת המשפט שהוכחנו. אחת משתי הפסוקיות שיוצרות פסוקית ריקה חייבת להיות צאצאית של שלילת המשפט.

7.1.6. תרגיל

תהי T קבוצת כל הפסוקיות שנוצרו משלילת המשפט. תהי A קבוצת כל הפסוקיות שנוצרו מהאקסיומות. תהי P קבוצת כל הפסוקיות המשתתפות בהוכחה.

1. האם יתכן כי T ו-P הן זרות?
2. האם יתכן כי P ו-A הן זרות?

תשובה:

1. יתכן ש-P ו-T הן זרות, במקרה הקיצוני בו קבוצת האקסיומות אינה עקבית (ואז נגיע לסתירה בלי צורך בשום קבוצת תמיכה).
אם האקסיומות עקביות אז לא ייתכן ש-P ו-T זרות, משום שהדרך להגיע לסתירה היא על ידי שימוש באחת ההנחות החדשות שבקבוצה T. כיוון שבתהליך ההוכחה אנו מגיעים לסתירה, מתחייב שהשתמשנו במהלך ההוכחה באחת מהפסוקיות שב-P, כי A בפני עצמה היא עקבית.

2. יתכן ש-P ו-A הן זרות. זה יקרה במקרה בו המשפט אותו אנו רוצים להוכיח הוא טאוטולוגיה. לדוגמא: $(X \vee \neg X)$.
שלילת הפסוק הינה: $(X \wedge \neg X)$. במעבר לפסוקיות, נוסף את הפסוקיות $\{X\},\{\neg X\}$. איחוד של שתיהן ייתן את הסתירה המבוקשת, ללא שימוש בפסוקיות מ-A.

8. למידה

למידה היא תהליך. הקלט של התהליך הוא התנסויות (דוגמאות). בעקבות הדוגמאות חל שינוי בלומד.
ייתכן שינוי לטובה או לרעה. נמדוד את טיב השינוי על פי היכולת לבצע קבוצת משימות.
טיב השינוי תלוי גם במדדים – ייתכן שלפי מדד אחד (למשל, מהירות הפתרון) הלומד ישתפר, ובתחום אחר (איכות הפתרון, למשל) יהפך גרוע יותר.
הגדרה: למידה הינה תהליך המקבלת התנסויות כקלט, ומבצע שינויים בבסיס הידע במטרה לשפר, על פי מדד נתון, את היכולת הפוטנציאלית של פותר הבעיות, המשתמש בבסיס הידע, לפתור קבוצת בעיות.

תהי X קבוצת אובייקטים. תהא $C \subseteq X$ תת קבוצה שלה הנקראת **מושג המטרה**.

מסווג הינו פונקציה בוליאנית $f : X \to \{0,1\}$.

נגדיר בשם **מסווג המטרה** את המסווג הבא: $f_c(x) = \begin{cases} 1 & x \in C \\ 0 & x \notin C \end{cases}$

בהינתן אובייקט $x \in X$ נאמר שהמסווג שוגה כאשר $f(x) \neq f_c(x)$.

בהינתן קבוצת אובייקטים $X_T \subseteq X$ נגדיר את דיוק המסווג כמספר: $\dfrac{\left|\{x \in X_T \mid f(x) = f_c(x)\}\right|}{|X_T|}$.

נשאף למצוא מסווגים בעלי דיוק גבוה על קבוצת הבעיות העתידיות.

8.1. בעית הלמידה

נתונה קבוצת דוגמאות מסומנות מתוך קבוצת המטרה $E = \{\langle x_1, f_c(x_1)\rangle, \ldots, \langle x_m, f_c(x_m)\rangle\}$.

אלגוריתם ללמידת מסווגים מקבלת כקלט קבוצת דוגמאות מסומנות E ומוציא כפלט מסווג.
אם E מכיל את כל האובייקטים ב- X אזי ניתן פשוט לשמור אותם בטבלה. מכיוון שבד"כ נתונה תת קבוצה של X צריך אלגוריתם הלמידה להכליל: להסיק מתוך דוגמאות שראה לגבי דוגמאות שהוא לא ראה. כדי שיהיה ניתן להכליל מגדירים קבוצות **תכונות**: אוסף של פונקציות הממפות איברים ב- X לתחום סופי. דוגמאות הלמידה הינן זוגות $\langle x_i, f_c(x_i)\rangle$ כאשר x_i מיוצג על ידי ווקטור של ערכי התכונות.

נכנה דוגמאות בשם **דוגמאות חיוביות** אם הן שייכות לקבוצת המטרה. נכנה דוגמאות בשם **דוגמאות שליליות** אם הן אינן שייכות לקבוצת המטרה.

בהינתן קבוצת דוגמאות, נוכל לבנות בעזרתן **עץ החלטה**. עץ החלטה נבנה על פי הרעיון הבא:
כל צומת (כולל השורש) מייצגת תכונה. מהצומת יוצאות קשתות לפי מספר הערכים האפשריים לתכונה.
אנו מתחילים עם קבוצת כל הדוגמאות ומתפצלים לפי התכונות. עוצרים את הבניה כאשר כל הדוגמאות תחת ענף מסוים שייכות לקבוצת המטרה או שכל הדוגמאות אינן שייכות לקבוצת המטרה.

8.2. ID3

ID3 היא התוכנית הידועה ביותר ללמידת מסווגים. התוכנית מקבלת כקלט קבוצת דוגמאות ומוציאה עץ החלטה המסווג נכון את כל הדוגמאות שניצפו. התוכנית מתחילה משורש העץ ומפתחת תתי עצים.
ID3 תפתח צומת אם הוא מכיל דוגמאות חיוביות ושליליות, ותעצור אם כל הדוגמאות הן מאותו סוג.
כל צומת מתפצל על תכונה מסוימת. התוכנית יוצרת תת עץ לכל ערך אפשרי של התכונה. קבוצת הדוגמאות מתחלקת לפי ערכי התכונה ותת הקבוצות מועברות לתת העצים.
ההחלטה על איזו תכונה לפצל מתבססת על יוריסטיקה האומרת: נבחרת התכונה המביאה לתוספת מקסימלית של אינפורמציה.
נכנה תכונה בשם **תכונה אינפורמטיבית** אם היא מחלקת את קבוצת הדוגמאות בצורה טובה.
תוספת האינפורמציה (gain) – לכל תכונה נמדדת תוספת האינפורמציה. התכונה המביאה לתוספת האינפורמציה הגדולה ביותר נבחרת לפיצול.
נסמן ב-p את מספר הדוגמאות החיוביות, וב-n את מספר הדוגמאות השליליות, אז ההסתברות ליפול בקבוצת החיוביים הינה $\frac{p}{p+n}$ וההסתברות ליפול בקבוצת השליליים הינה $\frac{n}{p+n}$.

אי הוודאות בצומת מוגדרת על ידי נוסחת שנון:

$$I(p,n)=\frac{p}{p+n}\cdot\log_2\left(\frac{p}{p+n}\right)-\frac{n}{p+n}\cdot\log_2\left(\frac{n}{p+n}\right)$$

כאשר אנחנו משנים מעט את log ומגדירים כי $\lim_{x\to 0}(\log_2 x)=0$ וכן $\log_2 0=0$.

אי הוודאות בבנים של צומת נקבע על ידי הממוצע המשוקלל של אי הוודאות בכל אחד מהם.

תוספת האינפורמציה היא אי וודאות הצומת פחות אי הוודאות בבנים:

$$GAIN = I\left(p,n\right) - \sum_{i=0}^{n} \frac{E_i}{|Examples|} \cdot I\left(p_i,n_i\right)$$

E_i הוא מספר הדוגמאות שעברו לבן ה- i, p_i זהו מספר הדוגמאות החיוביות בבן ה- i ו- n_i הוא מספר הדוגמאות השליליות בבן ה- i.

$GAIN$ נע בין 0 ל-1, כאשר 1 יסמל שיפור מקסימלי ו-0 פירושו שאין כלל שיפור.

מה יקרה במידה ועברנו על כל הדוגמאות ועדיין לא הגענו לחלוקה מוחלטת לפי ערכים חיוביים וערכים שליליים? שתי אפשרויות:

- זריקת הדוגמאות הבעיות בהנחה שהן רעש.
- קביעת ערך העלה לפי הערך של רוב הדוגמאות שבו.

<u>האלגוריתם של ID3</u>:

נגדיר: I היא קבוצת הדוגמאות. $A = A_1,...,A_n$ היא קבוצת כל התכונות האפשריות, ו- $V_1,...,V_n$ אלו התחומים של התכונות השונות, כאשר $V_i = \{V_{i1},...,V_{ik}\}$

```
ID3 (examples, Att)
      if examples = {} then return leaf(null)
      P ← positive examples
      N ← negative examples
      if N = { } then return leaf(P)
      if P = { } then return leaf(N)
      For each Ai in Att compute gain(Ai, examples)
      Select A' such that gain(A', examples) is maximal
      for each Vi in V' compute
            Ei = { e in examples | A'(e) = Vi }
            Si = ID3(Ei, Att - {A'})
      N ← new-node()
      test(N) ← A'
      children(N) ← {<Vi,Si> | i=1...n}
      Return N
```

8.3. בעיות לתוכניות ללימוד מסווגים

- הדוגמאות אינן מייצגות את העולם האמיתי.
- קיים רעש בדוגמאות (דוגמאות לא נכונות)
- אין מספיק דוגמאות
- תחומים רציפים של תכונות
- מושגים קשים ללימוד
- מחירים שונים למבחנים שונים
- ערכים חסרים

8.3.1. רעש בדוגמאות

נגדיר דוגמאות לרעש כאלה נגיע לעלים שלא ניתן לפצלם, אך הדוגמאות בהן אינן בעלות סיווג יחיד. במקרה זה הערך של רוב הדוגמאות יקבע את הערך הנכון.

8.3.2. אין מספיק דוגמאות

הבעיה נוצרת כאשר מופיעים עלים ריקים (אין דוגמאות עבור ערכי התכונות שהם מייצגים).
הפתרון – קביעת סיווג ברירת מחדל עבור הבנים. לרוב זה יהיה הסיווג הדומיננטי של צומת האב.

8.3.3. תחומים רציפים של תכונות

ID3, כפי שהוצג, יכול לטפל בערכים דיסקרטיים בלבד ולא בערכים רציפים. הפתרון לבעיה הוא דיסקרטיזציה של התחום. הדיסקרטיזציה יכולה להיות סטאטית, מוגדרת מראש, או דינמית, בהתאם להתפלגות הדוגמאות.

8.3.4. מחירים שונים למבחנים שונים

ייתכן שמחיר חישוב התכונות הינו שונה, ולכן נעדיף להפעיל את התכונות בסדר מסוים, גם אם תכונה מסויימת היא יעילה יותר מאחרות.

הפתרון – בהינתן הגדרת פונקצית מחיר על התכונות, נשקלל את המחיר עם חישוב ה-gain.

8.4. סיווג מבוסס דוגמאות *(Instance Based Learning)*

הרעיון: במקום לפעול על כל הדוגמאות ולבנות מסווג, אנו בונים את המסווג בשלבים, על ידי הכנסת הדוגמאות אחת אחרי השניה. המסווג יוצר את ההכללה לפי הדוגמאות שעד כה ברשותו.

8.4.1. סיווג השכן הקרוב ביותר (Nearest neighbor classification)

<u>למידה</u>: שמירת דוגמאות האימון.

<u>סיווג</u>: בהינתן דוגמא לא מסווגת, סווג אותה על פי השכן המסווג הקרוב ביותר.

<u>מדד הקירבה</u>: מרחק במרכב התכונות. בד"כ משתמשים בפונקצית המרחק האוקלידי:

$d(x,y)=\sqrt{\sum_{i=1}^{n} h\left(a_i(x),a_i(y)\right)^2}$ כאשר $a_1(x),...,a_n(x)$ הינן ערכי תכונות הדוגמה.

כאשר התכונות מספריות, $h\left(a_i(x),a_i(y)\right)=a_i(x)-a_i(y)$, אחרת נהוג להשתמש ב:

$$h\left(a_i(x),a_i(y)\right)=\begin{cases}0 & a_i(x)=a_i(y)\\ 1 & a_i(x)\neq a_i(y)\end{cases}$$

8.4.2. אלגוריתם IB3

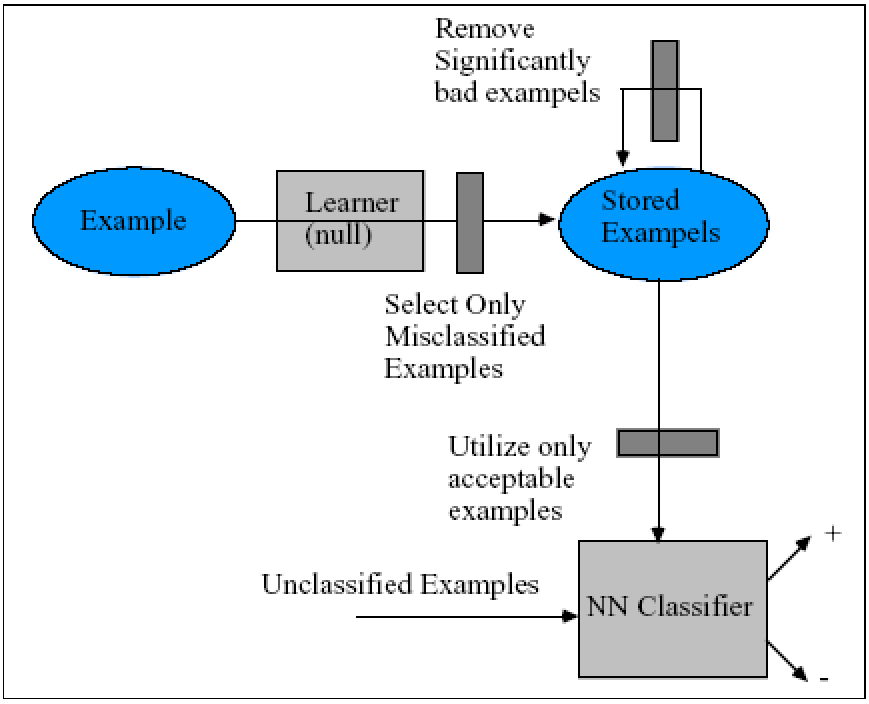
אלגוריתם הדרגתי ללמידת NN.

האלגוריתם מפעיל מסנן למידה: רק דוגמאות עליהן טועה המסווג הנוכחי מאוחסנות. לכל דוגמא בקבוצת האימון נשמרת היסטורית ההצלחה שלה (כמה דוגמאות היא סיווגה נכון).

לכל דוגמא בודקים אם רמת הדיוק שלה גבוהה באופן מובהק מתדירות הסיווג שלה בכלל הדוגמאות.

אם הדיוק גבוה יותר באופן מובהק הדוגמא מסומנת כקבילה ומשתתפת במסווג. אם הדירוג נמוך יותר באופן מובהק הדוגמא מסומנת כבלתי קבילה ונמחקת מהמאגר.

אחרת, הדוגמא נשמרת "על תנאי": היא איננה משתתפת בסיווג אולם אם היא קרובה יותר לדוגמא חדשה מאשר השכן הקרוב ביותר מבין הקבילים, בוחנים אותה על הדוגמא החדשה ומעדכנים את ההיסטוריה שלה.



האיור לקוח משקפי הקורס בטכניון "מבוא לבינה מלאכותית"

IB3 לרוב מראה ביצועים טובים יותר מ-NN ומשתמש בהרבה פחות זכרון.