חיפוש לעומק
(Depth-first search)

- הצומת הראשון לפיתוח הינו הצומת המיצג את מצב ההתחלה
- הצומת הבא לפיתוח יהיה תמיד הצומת העמוק ביותר
- בין צמתים בעלי עומק שווה נבחר הצומת הבא בצורה שרירותית (למשל לפי סדר הופעתם)
- נשמרים רק צמתים שלא נחקרו לגמרי (כלומר שעדיין לא פותח תת העץ תחתם במלואו).

```
DFS (state path)
  if goalp(state) return(path)
  else
      for c in succ(state)
          DFS(c, path || state)
```


מימוש DFS ב-LISP

```
(defun dfs (state)
  (cond ((goalp state)(list state))
        (t (let ((childs (succ state)))
             (loop for c in childs
               for solution = (dfs c)
               do (when solution
                    (return-from dfs
                     (cons state solution)))))))
```


DFS עם הגבלת עומק

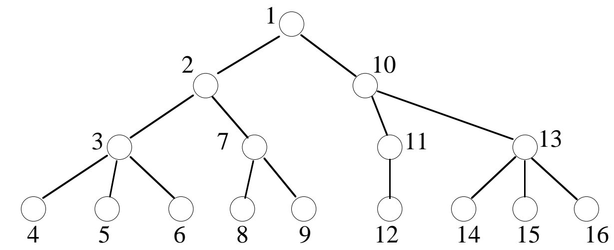
- האלגוריתם שלמעלה עלול להכנס לענף אינסופי או למעגל.
- כדי למנוע זאת משתמשים בד"כ ב-DFS עם הגבלת עומק.

```
DFS (state, depth, path)
    if goalp(state) return(path)
    else
        if depth=0 then return FAIL
        else
            for c in succ(state)
                DFS(c, depth-1, path || state)
```



```
(defun dfs-l (state depth)
  (cond ((goalp state)(list state))
        ((> depth 0)
         (let ((childs (succ state)))
             (loop for c in childs
               for solution =
                    (dfs-l c (1- depth))
               do (when solution
                    (return-from dfs-l
                        (cons state
                              solution)))))))
```

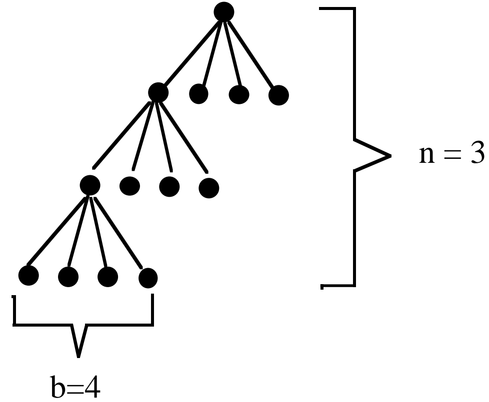
דוגמא לסדר פיתוח צמתים בחיפוש לעומק 3



שאול מרקוביץ©

חיפוש לעומק - דרישות זכרון

- מספר הצמתים שנשמרים בכל רגע הוא לכל היותר **b*n** כאשר b הוא המספר המקסימלי של ילדים לצומת ו n הוא עומק החיפוש הנוכחי.

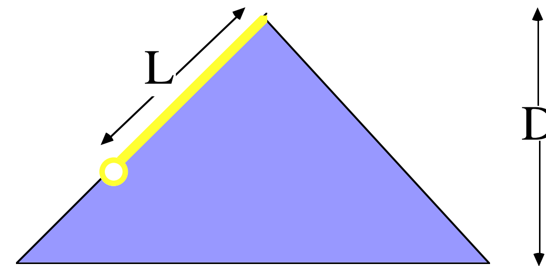


- לכן, במקרה הגרוע, ידרש זכרון של **b*D** כאשר D הינו הגבלת העומק.

שאול מרקוביץ©

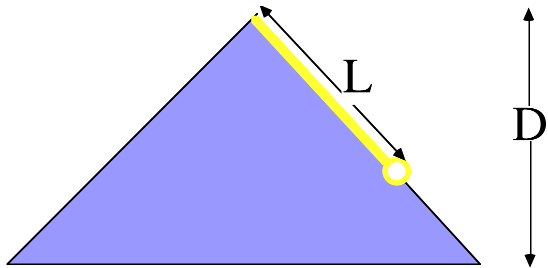
חיפוש לעומק - יעילות החיפוש:

- במקרה הטוב יגיע החיפוש ישירות למטרה ואז יפותחו L צמתים, כאשר L הינו אורך הפיתרון.



שאול מרקוביץ©

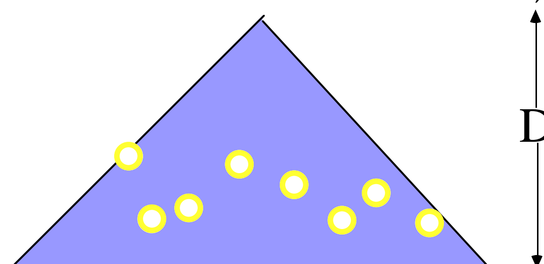
- במקרה הגרוע יפתח החיפוש את כל צמתי העץ עד לעומק D (כלומר $\frac{b^{D+1}-1}{b-1}$ פיתוחים)



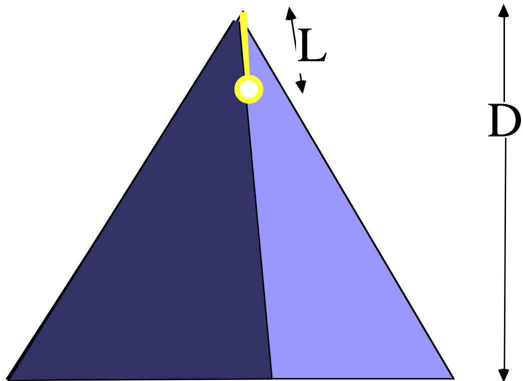
שאול מרקוביץ©

חיפוש לעומק - יעילות

- כאשר קבוצת צמתי המטרה גדולה, גדל הסיכוי לחיפוש יעיל

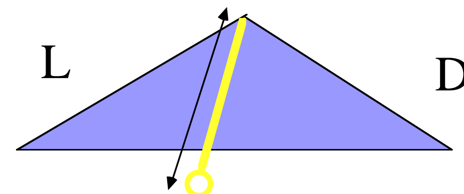


- קביעת הגבלת העומק D חשובה ובעייתית. D גדול מידי יגרום לבזבוז משאבים עצום (זכרו שיעילות החיפוש אקפוננציאלית ב-D)



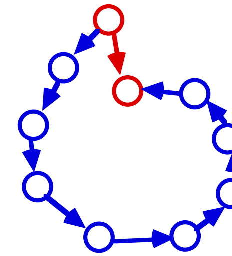
שלמות של DFS

- אלגוריתם חיפוש נקרא שלם אם מובטח שיעצור ויחזיר פתרון במידה וקיים פתרון.
- לגבי האלגוריתם המקורי (ללא הגבלת העומק) לא מובטח בכלל שיעצור.
- לגבי האלגוריתם עם הגבלת העומק מובטחת עצירה.
- כאשר $D \geq L$ האלגוריתם שלם.
- כאשר $D < L$ האלגוריתם אינו שלם: הוא יעצור ללא פתרון כאשר קיים פתרון.



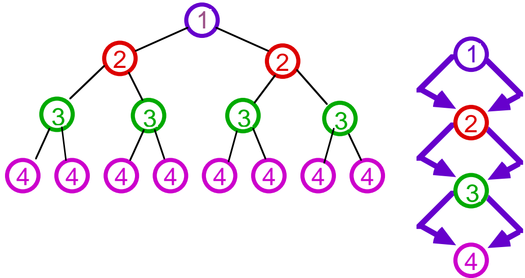
איכות הפתרון בחיפוש לעומק

- לא מובטח שנמצא את מסלול הפתרון הקצר ביותר (או בעל מחיר מינימאלי).



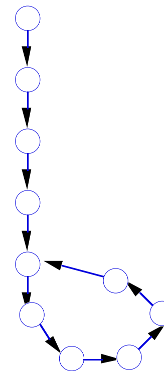
חיפוש לעומק בגרף

- האלגוריתמים שלמעלה מתיחסים למרחב החיפוש כאל עץ.
- DFS-L הינו שלם גם לגבי גרפים.
- אולם בגרפים יתכן שמצב ייוצג ע"י מספר רב של צמתים (כלומר חלקים של גרף המצבים יפותחו פעמים רבות).
- במקרה קיצוני יתכן סגודל עץ החיפוש יהיה אקספוננצאלי במספר המצבים:



פתרונות ל-DFS בגרפים

- ניתן לשמור על רשימת הצמתים שפותחו ולבדוק באם צומת חדש כבר פותח בעבר.
- תוספת כזו הופכת את דרישת הזכרון של האלגוריתם מלינארית לאקספוננציאלית ולכן אינה מקובלת.
- ניתן לבדוק באם צומת חדש נמצא במסלול מצומת ההתחלה. בדיקה כזו תמנע תנועה במעגל, אך תייקר את מחיר פיתוחו של כל צומת.



DFS עם בדיקת מסלול

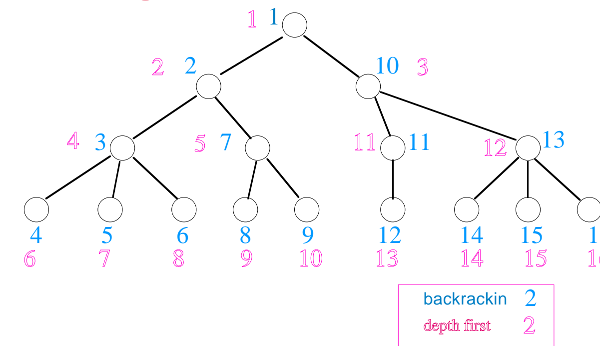
```
DFS (state, depth, path)
    if goalp(state) return(path)
    else
        if depth=0 then return FAIL
        else
            for c in succ(state)
                if c is not in path then
                    DFS(c, depth-1, path || state)
```

```lisp
(defun dfs-l-g (state depth parents)
  (cond ((goalp state)(list state))
        ((> depth 0)
         (let ((childs (succ state)))
             (loop for c in childs
               for solution =
                 (when (not (member c parents
                                 :test  #'equalp))
                   (dfs-l-g c (1- depth)
                              (cons state parents)))
               do (when solution
                   (return-from dfs-l-g
                       (cons state solution)))))))
```


חיפוש Backtracking

- וריאציה על חיפוש לעומק
- ניתן להשתמש בו כאשר פונקצית המעבר נתונה באמצעות קבוצת אופרטורים.
- במקום לפתח את כל הבנים של צומת, הפרוצדורה מייצרת בן אחד, חוקרת אותו לעומק, מייצרת בן שני, חוקרת אותו לעומק וכו'.
- פרוצדורה חסכונית עוד יותר בזכרון. מקסימום מספר הצמתים שנשמרים הוא עומק החיפוש D. (כמו כן נשמר בכל צומת מצביע לאופרטור האחרון שנוסה).

דוגמא לסדר יצור צמתים בחיפוש backtracking



יתרונות של חיפוש לעומק

- חסכוני בזכרון.
- אם קיימים מצבי מטרה רבים קיים סיכוי גדול למצא פתרון בצעדי חיפוש מועטים.
- באם אורך הפתרון קטן מהעומק המוגבל אזי החיפוש הינו שיטתי, כלומר, מציאת הפתרון מובטחת

חסרונות של חיפוש לעומק:

- בלי הגבלת עומק החיפוש עלול להקלע לענפים גדולים ואפילו אינסופיים
- אם אורך הפיתרון גדול מהגבלת העומק אזי לא ימצא פיתרון.
- פתרון שנמצא אינו בהכרח הקצר ביותר.

חיפוש לרוחב (Breadth-first search)

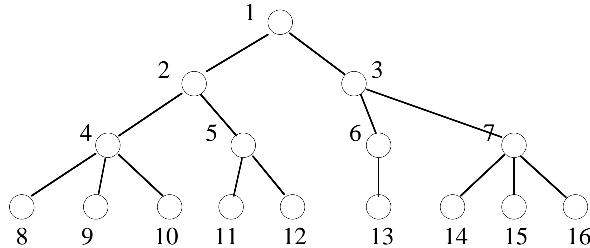
- הצומת הבא לפיתוח יהיה תמיד הצומת בעל העומק הקטן ביותר.
- בין צמתים בעלי עומק שווה נבחר הצומת הבא בצורה שרירותית (למשל לפי סדר הפיתוח)

```
BFS (init-state)
    OPEN ← (node(Si,NIL))
    While OPEN ≠ ( )
        next ← pop(OPEN)
        if goalp(next.state) then return(trace(next))
        OPEN ← OPEN || succ(next.state)
```



```lisp
(defun node-state (node)(first node))
(defun node-parent (node)(second node))
(defun bfs (state)
 (let ((open (list (list state nil))))
    (loop while open
      for next-node = (pop open)
      do
       (when (goalp (node-state next-node))
         (return-from bfs
          (reverse (trace-back next-node))))
       (setf open
         (nconc open
            (loop for s in
                  (succ (node-state next-node))
               collect (list s next-node))))))
```


דוגמא לסדר פיתוח צמתים בחיפוש לרוחב



חיפוש לרוחב מותאם לגרף

- הפרוצדורה שלמעלה תעבוד נכון גם בגרפים.
- גם כאן ניתן למנוע חיפוש חוזר בתתי גרפים ע"י שמירת הצמתים שפותחו.
- בניגוד לחיפוש לעומק, דרישות הזכרון כבר עכשיו הן אקספוננציאליות ולכן נהוג להוסיף את הבדיקה.

```
BFS-G (state)
    OPEN ← (node(state,NIL))
    While OPEN ≠ ( )
        next ← pop(OPEN)
        push (next, CLOSE )
        if goalp(next.state) then return(trace(next))
        for s in succ(next.state)
            if  s ∉ OPEN and s ∉ CLOSE then
                OPEN ← OPEN || s
```



```lisp
(defun find-state (state s-list)
  (find state s-list
     :test #'equalp :key #'node-state))

(defun bfs-g (state)
  (let ((open (list (list state nil)))
        (close nil))
    (loop while open
      for next-node = (pop open)
      do
       (when (goalp (node-state next-node))
         (return-from bfs-g
            (reverse (trace-back next-node))))
       (push next-node close)
       (setf open
         (nconc open
           (loop for s in
                    (succ (node-state next-node))
              when (not (or (find-state s open)
                             (find-state s close)))
              collect (list s next-node)))))))
```


חיפוש לרוחב - ביצועים

דרישות זכרון:

- מספר הצמתים שנשמרים בכל רגע הוא מספר הצמתים בעץ - לכל היותר $\frac{b^{L+1}-1}{b-1}$ - כאשר b הוא המספר המקסימלי של ילדים לצומת ו- L הוא אורך הפתרון הקצר ביותר.

יעילות החיפוש:

- במקרה הטוב יפותחו $\frac{b^{L-1}-1}{b-1}$ צמתים [הרמה האחרונה שמפותחת היא הרמה במרחק 2 מהעלים]. במקרה הגרוע יפותחו $\frac{b^{L}-1}{b-1}$ צמתים [יפותחו כל הצמתים במרחק 1 מהעלים].

איכות הפתרון:

- מובטחת מציאת מסלול הפתרון הקצר ביותר.

יתרונות וחסרונות של חיפוש לרוחב

יתרונות:

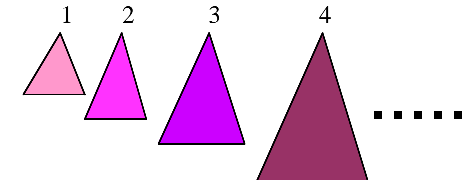
- מבטיח מציאת פתרון (אם אכן קיים פתרון)
- מוצא פתרון בעל מספר צעדים מינימלי

חסרונות:

- חיפוש מאד לא יעיל (מספר צעדי חיפוש גדול - אקספוננציאלי באורך הפתרון)
- דרישות זכרון גבוהות (אקספוננציאליות באורך הפתרון).

העמקה הדרגתית (iterative deepening)

- חיפוש לעומק מאפשר לנו לעבוד עם זכרון ליניארי
- חיפוש לרוחב מבטיח פתרון אופטימלי
- כדי לשלב את שני היתרונות פותח חיפוש ההעמקה ההדרגתית.
- האלגוריתם מפעיל חיפוש לעומק תחילה עם הגבלת עומק 1. באם פתרון אינו נמצא, מופעל חיפוש לעומק עם הגבלת עומק 2. וכך הלאה.
- האלגוריתם אינו זוכר דבר בין החיפושים (מלבד הגבלת העומק באיטרציה הקודמת)



 שאול מרקוביץ©

העמקה הדרגתית

ID (state)
    for d= 0 to max-depth
        solution ← DFS-L(state,d)
        if solution ≠ {} then return solution

```
(defparameter *max-depth* 100)
(defun id (state)
  (loop for d from 0 to *max-depth*
   for solution = (dfs-l state d)
   when solution do (return-from id solution)))
```

 שאול מרקוביץ©

העמקה הדרגתית - ביצועים

- דרישת הזכרון הינה לינארית באורך הפתרון.
- הפתרון שנמצא הינו הקצר ביותר.
- משלמים בזמן חיפוש - כל החיפושים של הרמות 1 עד L-1 למעשה מבוזבזים.
- כיוון שרוב הצמתים של עץ מתרכזים בעלים הבזבוז אינו רב כל כך.
- מספר הצמתים שיפותחו הינו

$$L\cdot 1+(L-1)b+(L-2)b^2+\cdots+1b^L$$

(אורך הפתרון הינו L. מפתחים L עצים. השורש מפותח L פעמים. כל צומת ברמה השניה מפותח L-1 פעמים.)

 שאול מרקוביץ©

דוגמא מספרית

- נניח L=5 ו-b=10
- חיפוש לרוחב:

- דרישת זכרון מקסימלית ומספר הצמתים שיפתח הינו 1+10+100+1000+10000+100000=111,111

חיפוש העמקה הדרגתית

דרישת זכרון מקסימלית 5*10=50

יפתח 6+50+400+3000+20000+100000=123,456

כלומר תמורת הגדלת זמן של 11% הקטנו דרישת זכרון פי 2000

 שאול מרקוביץ©

מציאת פתרון בעל מחיר מינימלי
Uniform-cost search

- וריאציה על חיפוש לרוחב
- משתמש בפונקצית מחיר המגדירה מחיר חיובי לכל מעבר בין צמתים.
- מטרת החיפוש למצוא פתרון בעל מחיר מינימלי (כאשר חיפוש לרוחב מחפש פתרון בעל מספר צעדים מינימלי).
- הצומת הבא לפיתוח הינו הצומת שמחיר המסלול אליו מינימלי.
- הפרוצדורה אינה מפסיקה כאשר נמצא פתרון (יתכן וישנו פתרון טוב יותר)
- הפרוצדורה מפסיקה כאשר מחיר כל המסלולים החלקיים גדול מהמחיר של הפתרון שנמצא.

שאול מרקוביץ©

מימוש

כדי להבטיח שימצא פתרון אופטימלי:

- רשימת הצמתים שיוצרו אך לא פותחו נשמרת ממוינת
- הצומת לפיתוח נלקח מתחילת הרשימה
- תנאי הסיום נבדק לגבי צומת לפני פיתוחו ולא בזמן יצורו.
- באם הפרוצדורה מיצרת צומת שכבר קיים וממתין לפיתוח, יש לעדכן את המצביע כך שיצביע להורה הזול יותר.

שאול מרקוביץ©

אלגוריתם Uniform-cost

קלט: מצב התחלתי, מצב סופי, פונקצית מעבר, פונקצית מחיר.

פלט: מסלול הפתרון הזול ביותר

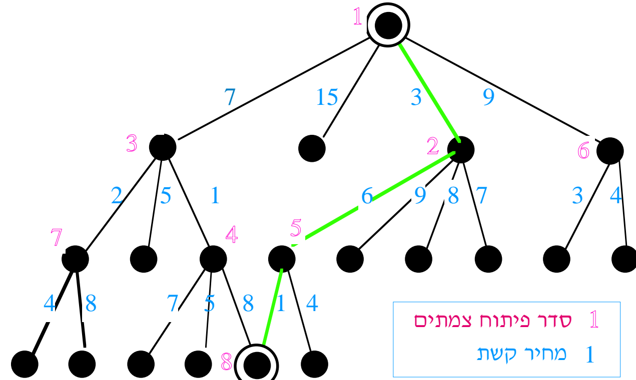
שאול מרקוביץ©

```
Uniform-cost-search(state)
    OPEN ← (node(state,NIL,0))      ; מצב, הורה, מחיר מסלול
    While OPEN ≠ ( )
        next ← pop(OPEN)
        push (next,CLOSE)
        if goalp(next.state) then return(trace(next))
        for s in succ(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(old-node,OPEN -{old-node})
                    end
                else
                    יצר צומת חדש והכנס למקום המתאים לפי המחיר
                    insert(node(s,next,new-cost),OPEN,g)
            end
        end
    end
    return FAIL
```


שאול מרקוביץ©

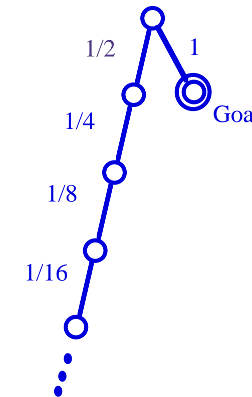
דוגמא לחיפוש uniform cost

המספרים המלאים הם מחירי המעברים. החלולים מיצגים את סדר פיתוח הצמתים.



שלמות Uniform-cost

- ללא תנאי נוסף Uniform-cost אינה שלמה.



-אם נוסיף את התנאי: קיים $\delta > 0$ המקיים
$\forall s \in S, \forall s' \in succ(s)[\cos t(s,s') \geq \delta]$ אזי האלגוריתם שלם.

Uniform-cost לעומת חיפוש לרוחב

- לחיפוש Unifrom-cost אותם יתרונות וחסרונות כמו לחיפוש לרוחב.
- נוסף היתרון של מציאת פתרון בעל מחיר מינימלי
- ניתן לראות חיפוש לרוחב כמקרה פרטי - מחיר מעבר הינו תמיד 1.