תרגול 1 – חלק ב'
מרחבי חיפוש

מבוא לבינה מלאכותית (236501)

מדעי המחשב, טכניון

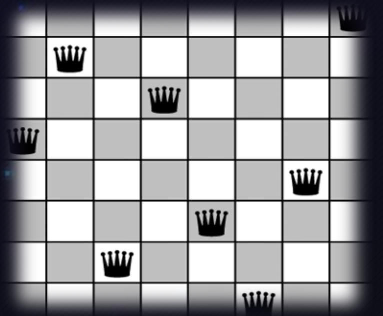
חורף 2022-3

15-Tile

River crossing puzzle

Urban Navigation

Eight Queens Puzzle



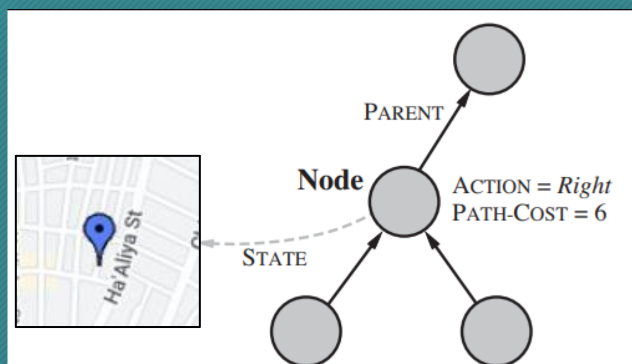
בעיות חיפוש

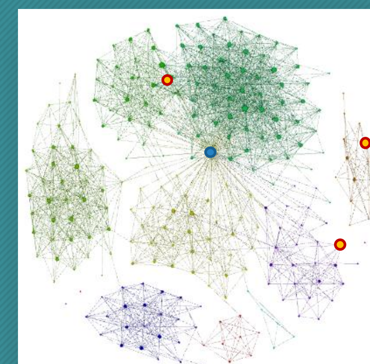
דוגמא לבעיית חיפוש שמוכרת לנו היטב:

מהי הדרך המהירה ביותר להגיע מתל אביב לחיפה?

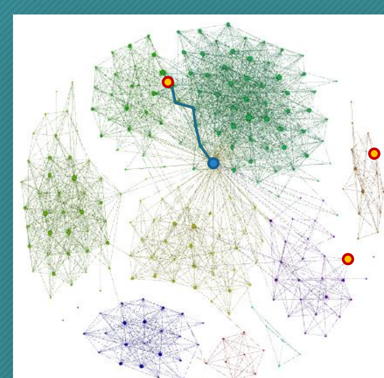
פתרון בעיות מורכבות באמצעות מחשב

- שלב ראשון – מייצגים את הבעיה באמצעות גרף מצבים





- שלב שני – מפעילים אלגוריתם למציאת מסלול מהמצב הנוכחי למצב המטרה



מרחב חיפוש - הגדרה

רביעייה (S,O,I,G)

- $\boldsymbol{S}$ קבוצת המצבים במרחב.
- $\boldsymbol{O}$ קבוצת אופרטורים/פעולות ממצב למצב עוקב,

$$O = \{o_1, \dots, o_k\} \quad o_i : S \to S \cup \{\phi\}$$

הפעלת $o_i \in O$ ממצב $s \in S$ אל הקבוצה הריקה $o_i(s) = \phi$
מסמלת כי לא ניתן להפעיל את האופרטור o_i על s.

- $S \ni \boldsymbol{I}$ מצב ההתחלתי.
- $S \supseteq \boldsymbol{G}$ קבוצת המצבים הסופיים.

יתכן ש- G יוגדר ע"י פרדיקט $P_G : S \to \{True, False\}$
ואז: $G \stackrel{\text{def}}{=} \{s \in S | P_G(s) = True\}$

מרחב חיפוש – הגדרות עזר נוספות

- פונקציית תחום המתאימה לכל אופרטור את $\boldsymbol{Domain}: \boldsymbol{O} \rightarrow \boldsymbol{P}(\boldsymbol{S})$ קבוצת המצבים שעליהם ניתן להפעיל אותו.

$$Domain(o) \stackrel{\text{def}}{=} \{s \in S \mid o(s) \neq \phi]\}$$

- פונקציית עוקב ממצב לקבוצת המצבים העוקבים $\boldsymbol{Succ}: \boldsymbol{S} \rightarrow \boldsymbol{P}(\boldsymbol{S})$

$$Succ(s) \stackrel{\text{def}}{=} \{s' \in S | \exists o \in O \; s.t. [s \in Domain(o) \wedge o(s) = s']\}$$

- קשתות גרף המצבים $\boldsymbol{E} \subseteq \boldsymbol{S}^2$

$$E \stackrel{\text{def}}{=} \{\langle s_1, s_2 \rangle | \exists o \in O \; s.t. [s_1 \in Domain(o) \wedge s_2 = o(s_1)]\}$$

כך מקבלים ש (S,E) הינו <u>גרף המצבים</u>.

מה מחפשים במרחב חיפוש?

הפתרון של חיפוש במרחב המצבים יכול להיות:

1. מצב סופי $s_g \in G$ (מצב העומד בקריטריון מסוים).
2. מסלול שלם ממצב ההתחלה למצב סופי:

$$\mathrm{s}_{\mathrm{i}_0} \rightarrow \mathrm{s}_{\mathrm{i}_1} \rightarrow s_{i_2} \rightarrow \cdots \rightarrow s_{i_n}$$

$$\mathrm{s}_{\mathrm{i}_0} = I, s_{i_n} \in G$$

$$\forall k\ \exists o\ s.t.\ s_{i_{k+1}} = o(s_{i_k})$$

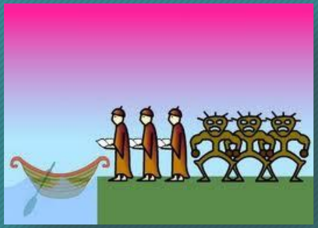
במקרה השני נוכל לייצג את התוצאה ב2 דרכים:

I. וקטור המצבים $< \mathrm{s}_{\mathrm{i}_0}, \mathrm{s}_{\mathrm{i}_1}, s_{i_2}, \ldots, s_{i_n} >$.

II. וקטור האופרטורים שהופעלו במעברים.

דוגמאות למרחבי חיפוש

- בעיית הקניבלים: http://en.wikipedia.org/wiki/Missionaries_and_cannibals_problem
- חידת 15-puzzle: http://en.wikipedia.org/wiki/Fifteen_puzzle
- חידת מסלול הפרש: http://en.wikipedia.org/wiki/Knight%27s_tour

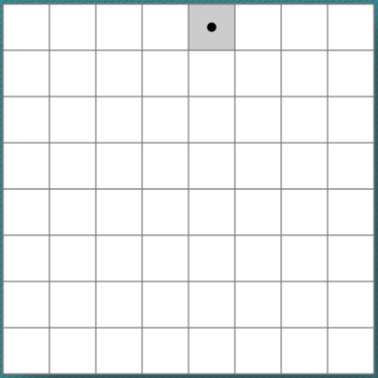


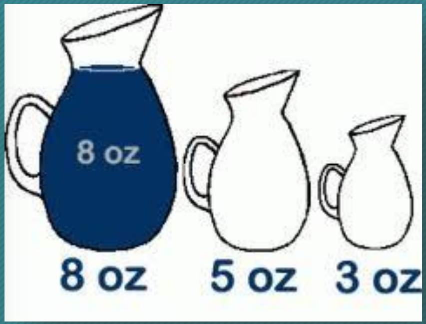
- חידת המכלים:

http://geekexplains.blogspot.co.il/2008/05/3-litres-and-5-litres-containers-puzzle.html

http://www.highiqpro.com/iq-brainteasers-puzzles-iq-tests/3-jugs-problem

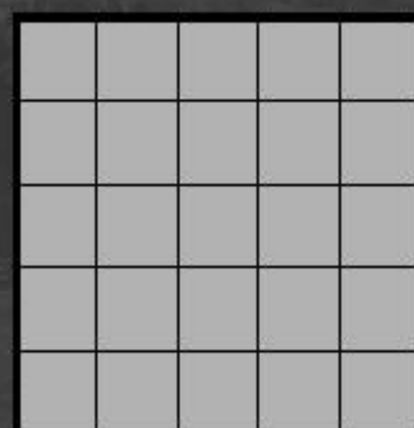
- חיפוש ברשת האינטרנט.
- ניווט עירוני עם GPS.





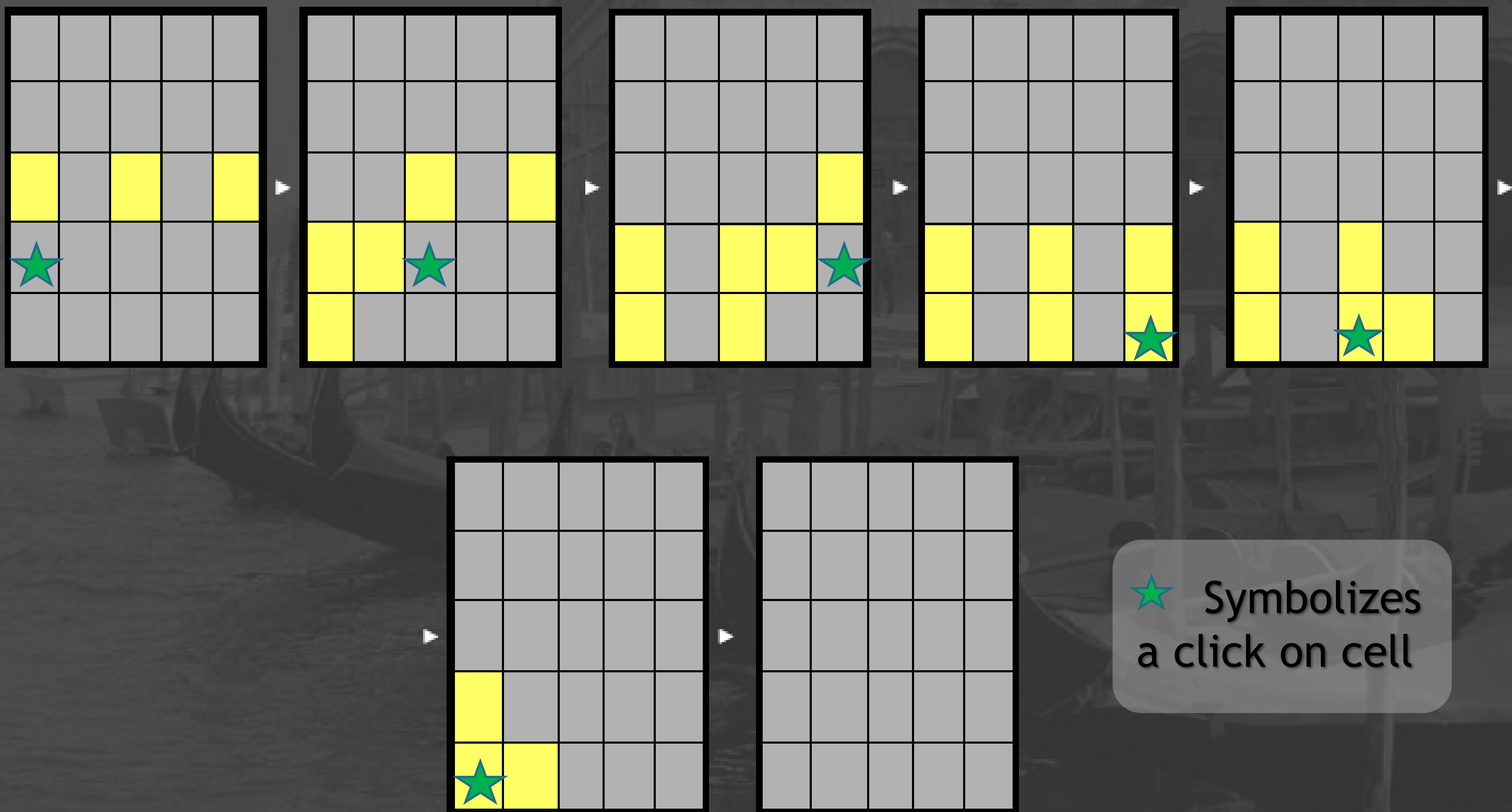
Example: LightzOut

Goal:



- 5X5 board
- Each cell can be "On" or "Off"
- Click on a cell inverts the states of its 4 neighbors

LightzOut – solution example



דוגמא: LightzOut

- איך נגדיר את קבוצת המצבים האפשריים (S)?

מצב המגדיר לכל משבצת בלוח האם האור בו דלוק.

$[5] \stackrel{\text{def}}{=} \{1,2,3,4,5\} \qquad S = \{0,1\}^{[5]\times[5]}$

- מהו המצב ההתחלתי?

מצב הלוח ההתחלתי תלוי בבעיה רוצים לפתור.

- מה הן הפעולות האפשריות (O)?

לחיצה על משבצת בלוח.

$O = [5] \times [5]$

דוגמא: LightzOut

- בהינתן אופרטור $O \in o$, למה שווה $Domain(o)$?

  ניתן להפעיל כל אופרטור על כל מצב.

  $\forall \mathrm{o} \in \mathrm{O}:\ \mathrm{Domain(o)} = S$

- מה קבוצת המצבים הסופיים (G)?

  מצב סופי יחיד - כל האורות כבויים.

  $\mathrm{G} = \{0\ matrix\ (5X5)\}$

- איזה סוג פתרון נחפש במרחב?

  מסלול שלם מהמצב ההתחלתי למצב הסופי.

דוגמא: LightzOut

תכונות מרחב החיפוש

- ייתכנו מצבים בלתי ישיגים.
- כל הפעולות הפיכות (לא תמיד זה כך!).
- הפעולה ההופכית לכל פעולה היא אותה פעולה עצמה.

ווריאציות להגדרת מרחבי חיפוש

- הוספת פונקציית עלות (בעיות תכנון)
$Cost: \{< s_1, s_2 > \mid s_1 \in S, s_2 \in Succ(s_1)\} \to \mathrm{R}$

- הגדרת קבוצת מצבים התחלתיים אפשריים מהם נבחר:
  - באקראי עפ"י התפלגות ידועה/לא ידועה.
  - ע"י אלגוריתם החיפוש.
  - ע"י יריב (adversary, rival).

ווריאציות נוספות

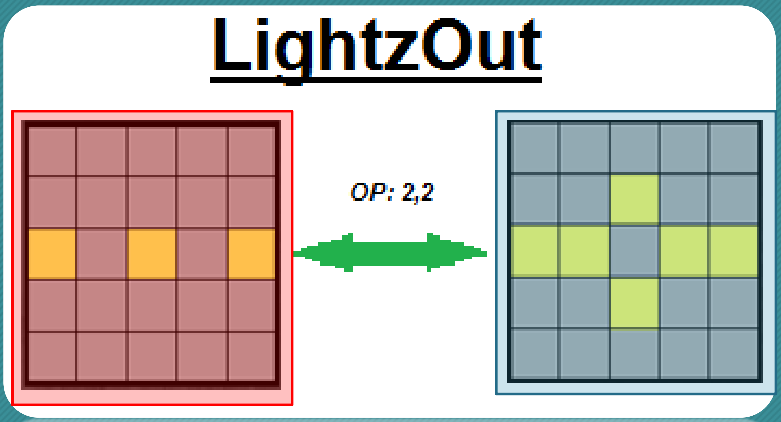
- במקום "מצבים סופיים", נגדיר פונקציית ערך ונחפש מצב (או מסלול למצב) בעל ערך אופטימלי. אלה הן בעיות אופטימיזציה, בהן ניתן:
  1. להגדיר ערך סף שכל ערך טוב ממנו ייחשב מצב סופי.
  2. להגדיר אלגוריתם חיפוש <u>anytime</u>, שמשפר את הפתרונות שהוא מוצא כאשר מאפשרים לו לרוץ יותר זמן.

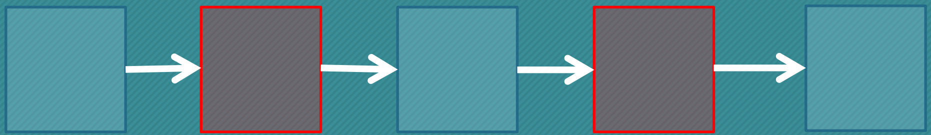
- בעיות בהן מחפשים מספר מצבים/מסלולים, לפי דרישות שונות/זהות, עם/ללא דרישה לסדר מסוים.

עץ חיפוש לעומת גרף מצבים

בהינתן גרף מצבים $G=(S,E)$ נציג מושג חדש: עץ חיפוש (V,E').

- מצב $s \in S$ הוא צומת בגרף המצבים, קונפיגורציה מסוימת אליה ניתן אולי להגיע בחיפוש כלשהו.
- צומת $v \in V$ בעץ חיפוש מסוים מייצג מצב בתוספת הקשר - המיקום ביחס לצמתים אחרים במהלך ריצת חיפוש ספציפית.
- דוגמה לגרף מצבים (חלקי):
- בעץ החיפוש יתכנו כמה מופעים של כל מצב:





מרחבי חיפוש (AI) לעומת גרפים (Algo 1)

בדרך כלל בבעיות אמיתיות:

1. מרחב המצבים עצום, אקספוננציאלי בקלט או יותר, אולי אפילו אינסופי.
2. לא נשמור את כל הגרף בזיכרון אלא נבנה חלקים ממנו בהדרגה במהלך החיפוש.
3. במימוש, נבנה מנגנון המחזיר מצבים עוקבים למצב מסוים לפי האופרטורים.
4. נעדיף לסייר במרחב בצורה חכמה, ע"י שימוש ביוריסטיקות (כללי אצבע).

- זהו חיפוש מיודע, בניגוד לחיפושים עיוורים, שאינם מיודעים. חיפוש מיודע הינו חיפוש שמשתמש בידע נוסף על העולם או ידע ספציפי על מרחב הבעיה.

מבוא לחיפוש מתקדם בגרפים

- **שלמות -** אלגוריתם חיפוש הינו **שלם** אם מובטח שיחזיר פתרון כאשר פתרון קיים.
- **קבילות -** אלגוריתם חיפוש הינו **קביל** כאשר מובטח שיחזיר פתרון בעל המחיר המינימלי כאשר קיים פתרון.

מה ניתן לומר על הקשר בין שלמות וקבילות?

- אם אלגוריתם הוא קביל אזי הוא שלם.

אלג' שלם: קיים פתרון ← מוחזר פתרון

אלג' קביל: קיים פתרון ← מוחזר פתרון <u>אופטימלי</u>

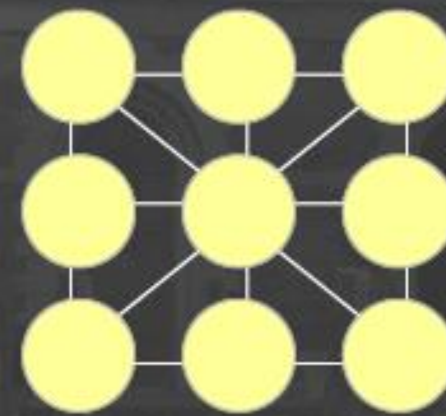
חיפוש עץ לעומת חיפוש גרף

- אלג' חיפוש עץ: אינו בודק האם מצב פותח בעבר. כתוצאה מכך, נוצרות כפילויות אשר באות לידי ביטוי ב"עץ החיפוש".
- אלג' חיפוש בגרף: מתחזק רשימה של מצבים שביקר בהם ונמנע מביקור חוזר במצבים אלה.

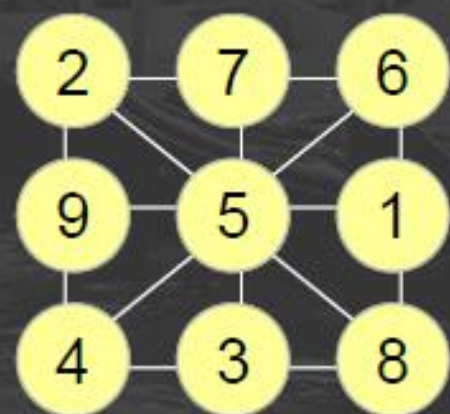
חיפוש עץ	חיפוש גרף
פשוט אלגוריתמית.	מורכב יותר - דורש זיכרון נוסף לתחזוקת קבוצת מצבים שפותחו.
חוסך בדיקת שייכות לקבוצת המצבים שפותחו (membership query).	חוסך פיתוח חוזר של מצבים ובכך חוסך זמן.
חוסך זיכרון (לא שומרים רשימה של צמתים שפותחו כבר)	

Example: 9-number puzzle

The numbers 1, 2, 3, 4, 5, 6, 7, 8, and 9 must be put in the depicted square, in such a way that the sums of the numbers in each row, column, and diagonal are equal.



Solution:



דוגמא: 9 numbers

- איך נגדיר את קבוצת המצבים האפשריים (S)?

  מצב ממפה לכל משבצת בלוח מספר, או מצב "ריק"

  $$S = ([9] \cup \{\phi\})^{[3]\times[3]}$$

- מהו המצב ההתחלתי?

  הלוח הריק: $I = \phi_{[3]\times[3]}\ matrix$

- מה הן הפעולות האפשריות (O)?

  הצבת ספרה חדשה/שונה במשבצת בלוח

  $$O = \{ put\langle row, col, val\rangle | row, col \in [3], val \in [9]\}$$

דוגמא: 9 numbers

- בהינתן אופרטור $O \in o$, למה שווה $Domain(o)$?

כל המצבים בהם המשבצת ריקה והספרה לא קיימת בלוח

$$Domain(put\langle r,c,v\rangle) = \{s \in S \mid s[r,c] = \phi \land not\ (v\ appears\ in\ s)\}$$

דוגמא: 9 numbers

- מה קבוצת המצבים הסופיים (G)?

לוח מלא בו כל ספרה שונה וסכום העמודות, השורות והאלכסונים שווה

$$
\begin{aligned}
&\boldsymbol{P_G(s) = \{s \in S \text{ such that:}}\\
&\qquad \forall(\boldsymbol{i},\boldsymbol{j}) \in [\mathbf{3}]^{\mathbf{2}}: \boldsymbol{s}[\boldsymbol{i},\boldsymbol{j}] \neq \boldsymbol{\phi} \qquad \wedge\\
&\qquad \forall(\boldsymbol{i_1},\boldsymbol{j_1}),(\boldsymbol{i_2},\boldsymbol{j_2}) \in [\mathbf{3}]^{\mathbf{2}}: (\boldsymbol{i_1},\boldsymbol{j_1}) \neq (\boldsymbol{i_2},\boldsymbol{j_2}) \rightarrow \boldsymbol{s}[\boldsymbol{i_1},\boldsymbol{j_1}] \neq \boldsymbol{s}[\boldsymbol{i_2},\boldsymbol{j_2}] \qquad \wedge\\
&\qquad \boldsymbol{\Sigma_{row\,1} = \Sigma_{row\,2} = \Sigma_{row\,3} = \Sigma_{col\,1} = \Sigma_{col\,2} = \Sigma_{col\,3} = \Sigma_{diag\,1} = \Sigma_{diag\,2}\}}
\end{aligned}
$$

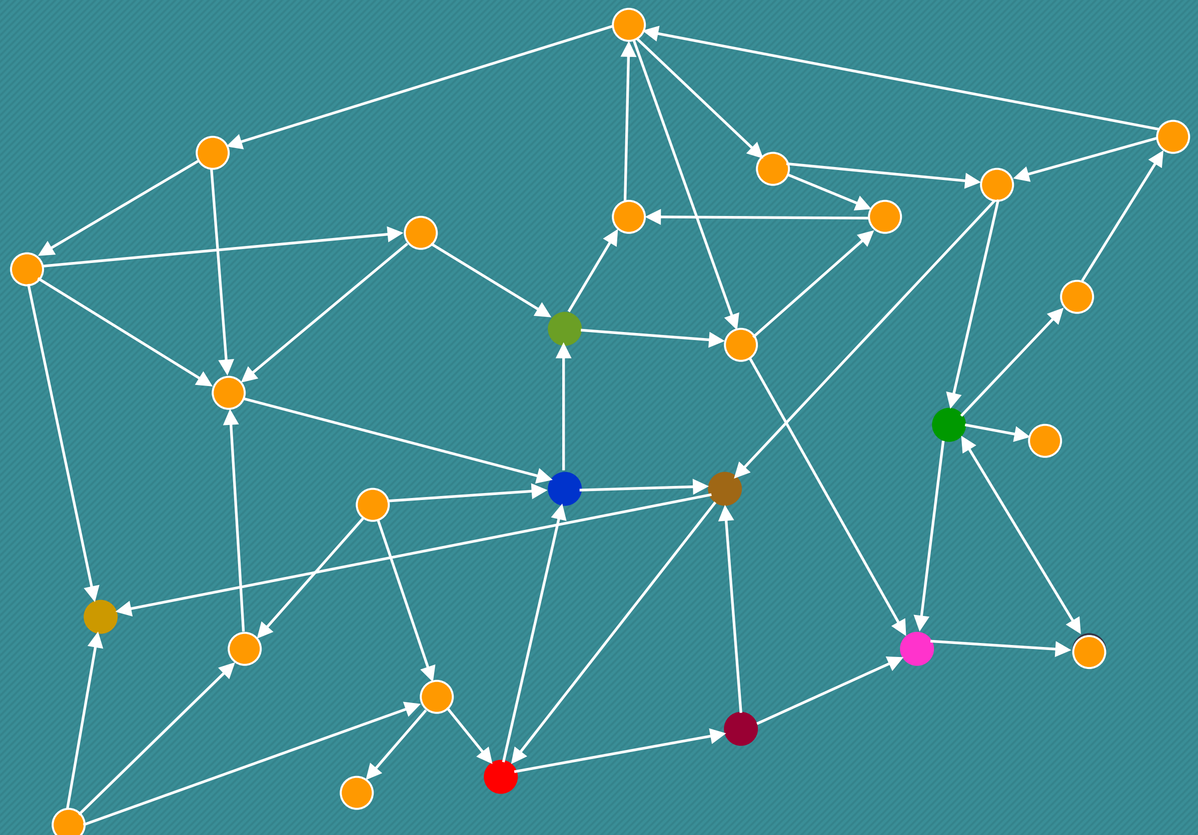
- איזה סוג פתרון נחפש במרחב?

מצב סופי בלבד.

דוגמא: 9-numbers

- האם יכולנו להגדיר את המרחב כקבוצת המצבים המלאים בלבד?
  - ✓כן, כאשר הפעולות הן החלפות בודדות.
- כיצד יכולנו להימנע מ"היתקעות" בחיפוש?
  - ✓ניתן היה להוסיף למרחב שהגדרנו פעולת הסרה ו/או החלפה.
- האם ניתן היה להגביל את האופרטורים כך שלא יאפשרו יצירת מצבים לא חוקיים?
  - ✓כן. למשל, יצירת שורה עם סכום גדול מהנדרש.
- מה לגבי הוספת אופרטור "undo"?
  - x לא! פעולה כזו מוגדרת לצומת בעץ החיפוש ולא למצב.

נושאים בסיסיים ועיקריים מאלגוריתמים 1

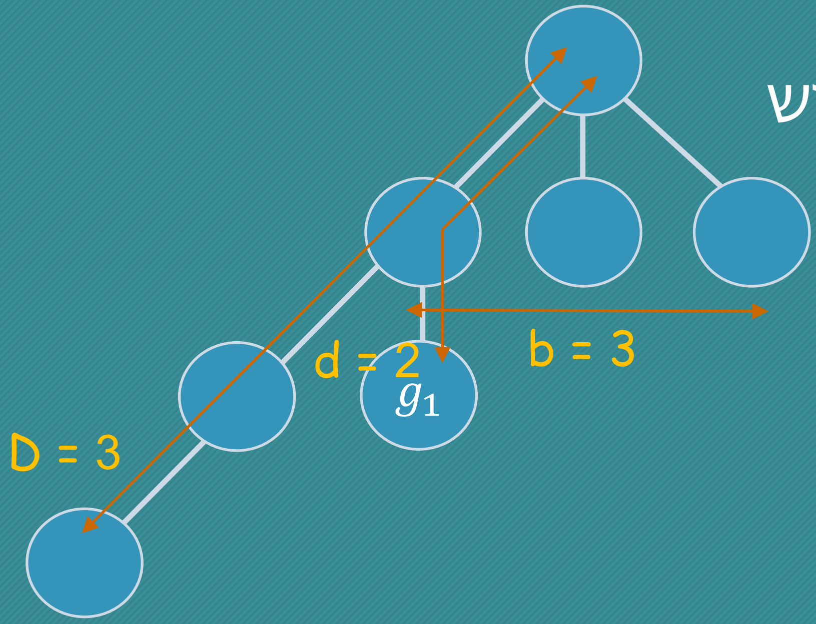


פרמטרים חשובים

b : מקדם הסיעוף (branching factor)

d : העומק של צומת המטרה הרדוד ביותר

D : העומק המקסימלי בעץ החיפוש



BFS – חיפוש לרוחב

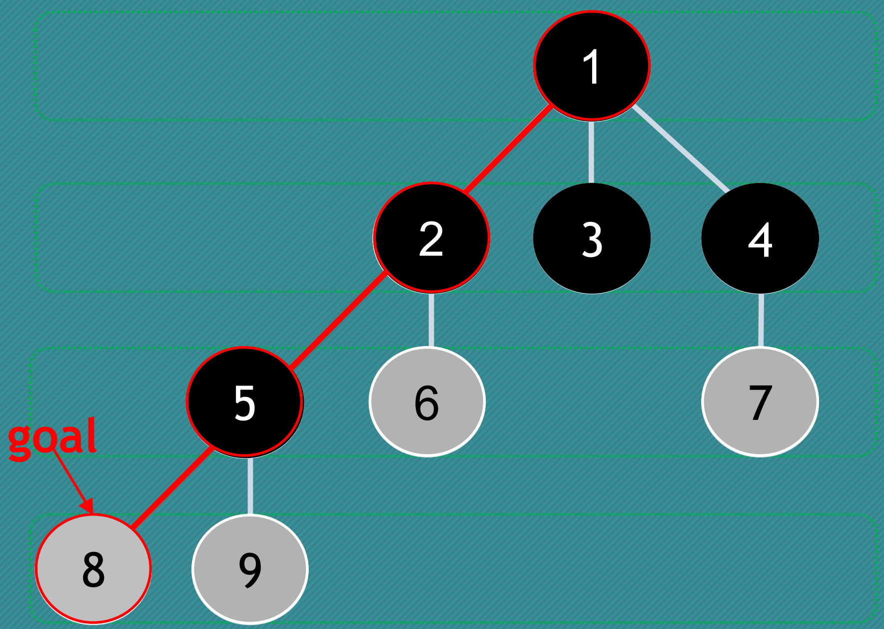
הצומת הבא לפיתוח: הצומת הרדוד ביותר.

Tree-Search

```
function Breadth-First-Search(problem):
        node ← make_node(problem.init_state, null)
        if problem.goal(node.state) then return solution(node)
        OPEN ← {node}  /* a FIFO queue with node as the only element */
        while OPEN is not empty do:
                node ← OPEN.pop()  /* chooses the shallowest node in OPEN */
        ➔      loop for s in expand(node.state):
                        child ← make_node(s, node)
                        if problem.goal(child.state) then return solution(child)
                        OPEN.insert(child)
        return failure
```

Graph-Search

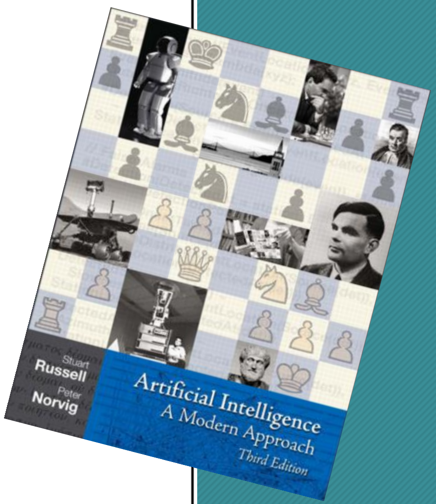
```
function Breadth-First-Search-Graph(problem):
        node ← make_node(problem.init_state, null)
        if problem.goal(node.state) then return solution(node)
        OPEN ← {node}  /* a FIFO queue with node as the only element */
        CLOSE ← {}  /* an empty set */
        while OPEN is not empty do:
                node ← OPEN.pop()  /* chooses the shallowest node in OPEN */
                CLOSE.add(node.state)
        ➔      loop for s in expand(node.state):
                        child ← make_node(s, node)
                        if child.state is not in CLOSE and child is not in OPEN:
                                if problem.goal(child.state) then return solution(child)
                                OPEN.insert(child)
        return failure
```



BFS

```
function BREADTH-FIRST-SEARCH(problem) returns a solution, or failure
  node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  frontier ← a FIFO queue with node as the only element
  explored ← an empty set
  loop do
     if EMPTY?(frontier) then return failure
     node ← POP(frontier)  /* chooses the shallowest node in frontier */
     add node.STATE to explored
     for each action in problem.ACTIONS(node.STATE) do
         child ← CHILD-NODE(problem, node, action)
         if child.STATE is not in explored or frontier then
             if problem.GOAL-TEST(child.STATE) then return SOLUTION(child)
             frontier ← INSERT(child, frontier)
```

Figure 3.11 Breadth-first search on a graph.



תכונות BFS

האם האלגוריתם שלם?

כן. בהנחה שמספר הפעולות / מקדם הסיעוף סופי (נניח זאת כברירת מחדל).

האם האלגוריתם קביל?

כן (תחת מחיר אחיד על הקשתות). המסלול המוחזר הוא תמיד הקצר ביותר.
כל המסלולים בעומק d נבדקים לפני המסלולים בעומק d+1.

סיבוכיות זמן?

$O(b^d)$

סיבוכיות מקום?

$O(b^d)$

b = מקדם הסיעוף
d = עומק הפתרון

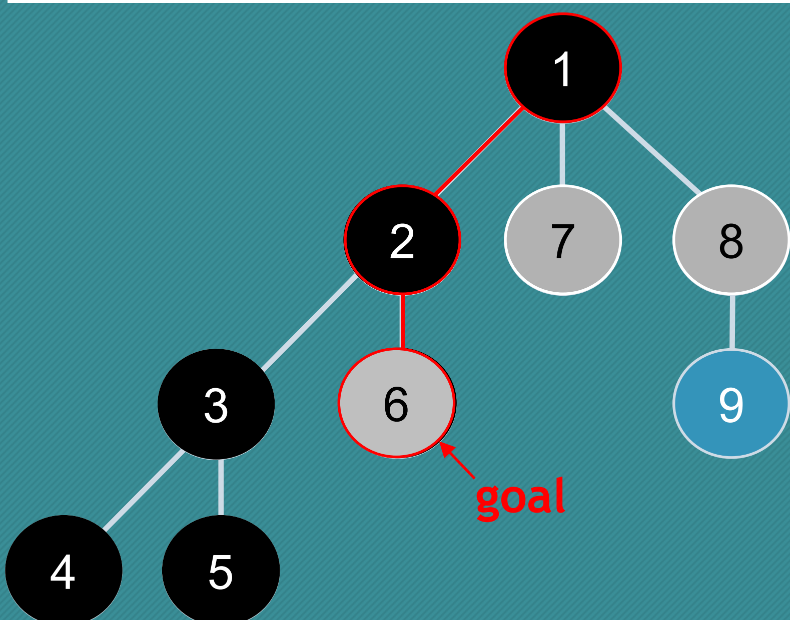
DFS – חיפוש לעומק

הצומת הבא לפיתוח: הצומת העמוק ביותר.

Tree-Search

```
function Depth-First-Search(problem):
        node ← make_node(problem.init_state, null)
        OPEN ← {node}  /* a LIFO queue with node as the only element */
        return Recursive-DFS(problem, OPEN)

function Recursive-DFS(problem, OPEN):
        node ← OPEN.pop()   /* chooses the deepest node in OPEN */
        if problem.goal(node.state) then return solution(node)
  →     loop for s in expand(node.state):
                child ← make_node(s, node)
                OPEN.insert(child)
                result ← Recursive-DFS(problem, OPEN)
                if result ≠ failure then return result
        return failure
```



Graph-Search

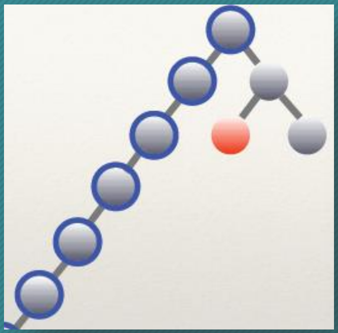
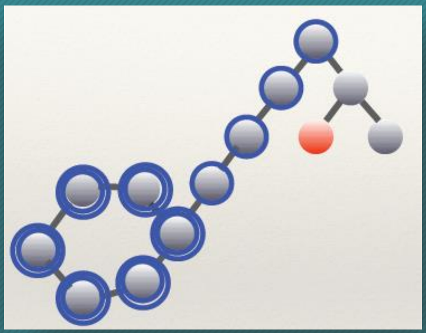
```
function Depth-First-Search-Graph(problem):
        node ← make_node(problem.init_state, null)
        OPEN ← {node}  /* a LIFO queue with node as the only element */
        CLOSE ← {}  /* an empty set */
        return Recursive-DFS(problem, OPEN, CLOSE)

function Recursive-DFS-G (problem, OPEN, CLOSE):
        node ← OPEN.pop()   /* chooses the deepest node in OPEN */
        CLOSE.add(node.state)
        if problem.goal(node.state) then return solution(node)
  →     loop for s in expand(node.state):
                child ← make_node(s, node)
                if child.state is not in CLOSE and child is not in OPEN:
                        OPEN.insert(child)
                        result ← Recursive-DFS-G(problem, OPEN, CLOSE)
                else: continue
                if result ≠ failure then return result
        return failure
```

n - צומת שנוצר

n - צומת שפותח

תכונות DFS

האם האלגוריתם שלם?
לא. יתכנו מעגלים, יתכן מסלול אינסופי.

האם האלגוריתם קביל? (תחת מחיר אחיד על הקשתות)
לא. שלמות היא תנאי הכרחי לקבילות.
בשלילה: אם היה קביל אזי היה גם שלם - סתירה.

סיבוכיות זמן?

$O(b^D)$

סיבוכיות מקום?

$O(bD)$

b = מקדם הסיעוף
D = העומק המקסימלי בעץ החיפוש

חיפוש לעומק – Backtracking (עצל)

Tree-Search

```
function Depth-First-Search(problem):
    node ← make_node(problem.init_state, null)
    OPEN ← {node}  /* a LIFO queue with node as the only element */
    return Recursive-DFS(problem, OPEN)
```

```
function Recursive-DFS(problem, OPEN):
    node ← OPEN.pop()   /* chooses the deepest node in OPEN */
    if problem.goal(node.state) then return solution(node)
    loop for s in lazy_expand(node.state):
        child ← make_node(s, node)
        OPEN.insert(child)
        result ← Recursive-DFS(problem, OPEN)
        if result ≠ failure then return result
    return failure
```

זהה ל-DFS, אך יוצר צמתים עוקבים רק **מיד** לפני פיתוח שלהם (יצירה עצלה) ולכן חוסך בזיכרון מקום.

כאן ההבדל לעומת DFS:
פונקציית העוקב מייצרת כל פעם את המצב העוקב הבא לפי דרישה.

```python
# a generator that yields items instead of returning a list
def firstn(n):
    num = 0
    while num < n:
        yield num
        num += 1
```

```python
sum = 0
for x in firstn(1000000):
    sum += x
```

לקריאה עצמית:
זו הזדמנות טובה להכיר generators בפייתון.
המילה השמורה yield בפונקציה
מאפשרת איטרציה עצלה.

תכונות Backtracking לעומת DFS

אילו מהתכונות משתנות ביחס ל-DFS?

האם האלגוריתם שלם?

לא. יתכנו מעגלים, יתכן מסלול אינסופי.

האם האלגוריתם קביל? (תחת מחיר אחיד על הקשתות)

לא. שלמות היא תנאי הכרחי לקבילות.

בשלילה: אם היה קביל אזי היה גם שלם - סתירה.

סיבוכיות זמן? $O(b^D)$

סיבוכיות מקום? $\cancel{O(bD)}$ $\boldsymbol{O(D)}$

b = מקדם הסיעוף

D = העומק המקסימלי בעץ החיפוש

DFS-L – חיפוש לעומק מוגבל

מה ההבדלים לעומת DFS?

Tree-Search

```
function DFS-L (problem, depth):
    node ← make_node(problem.init_state, null)
    OPEN ← {node}  /* a LIFO queue with node as the only element */
    return Recursive- DFS-L(problem, OPEN, depth)
```

```
function Recursive- DFS-L(problem, OPEN, depth):
    node ← OPEN.pop()   /* chooses the deepest node in OPEN */
    if problem.goal(node.state) then return solution(node)
    if depth == 0 then return failure
    loop for s in expand(node.state):
        child ← make_node(s, node)
        OPEN.insert(child)
        result ← Recursive- DFS-L(problem, OPEN, depth - 1)
        if result ≠ failure then return result
    return failure
```

תכונות DFS-L לעומת DFS

אילו מהתכונות משתנות ביחס ל-DFS?

האם האלגוריתם שלם?

~~לא. יתכנו מעגלים, יתכן מסלול אינסופי.~~

לא. יתכן שהפתרון בעומק גדול מהחסם.

האם האלגוריתם קביל? (תחת מחיר אחיד על הקשתות)

לא. שלמות היא תנאי הכרחי לקבילות.

סיבוכיות זמן? $\cancel{O(b^D)}$ $\boldsymbol{O(b^l)}$

סיבוכיות מקום? $\cancel{O(bD)}$ $\boldsymbol{O(bl)}$

b = מקדם הסיעוף

D = העומק המקסימלי בעץ החיפוש

l = חסם העומק

סיכום אלגוריתמי חיפוש לא-מיודעים

אלגוריתם	שלמות	קבילות	סיבוכיות זמן	סיבוכיות מקום
BFS	כן	כן	$O(b^d)$	$O(b^d)$
DFS	לא	לא	$O(b^D)$	$O(bD)$
Backtracking				$O(D)$
DFS-L	לא	לא	$O(b^l)$	$O(bl)$

b = מקדם הסיעוף המקסימלי d = עומק הפתרון

D = העומק המקסימלי בעץ חיפוש l = חסם העומק

- בכל הטבלה מניחים שמקדם הסיעוף חסום (b).
- קבילות BFS תלויה בכך שמחיר הקשתות אחיד.
- חסמי DFS , Backtracking מניחים גם שעומק החיפוש המקסימלי חסום (D).