פרויקט באימות פורמלי

מגישים:

אסף בן אור 209381599

דניאל יופה 324061878

תאריך הגשה: 25.05.2024

כל קבצי הפייתון, צילומי המסך, קבצי הפלט וקובץ README נמצאים ב-GitHub:

https://github.com/daniel1407y/FormalVerificationProject

חלק 1

.1

נגדיר את ה FDS של לוח הסוקובאן:

$$D = \{V, \theta, \rho, J, C\}$$

כאשר:

- **V:**

אנחנו מקבלים כקלט לוח מגודל n*m משתנים שהם תאים בלוחות וכל תא מהווה משתנה יחיד (בקובץ txt.).

מכיוון שה-nuXmv לא מקבל חלק מהתווי ה-XSB ביצענו את התרגום הבא:

a	@
o	+
b	$
v	*
x	#
g	.
_	_

כלומר, כל תא board[i][j] בלוח המשחק יכול לקבל את ערך הערכים הבאים:

$$\{a, o, b, v, x, g, \_\}$$

על מנת לפשט את הבעיה והקטין את הסיבוכיות, שמנו לב כי ניתן להפטר ממספר "מצבים" אפשריים של כל תא ולהגדיר את הלוח בצורה שונה (נסביר לפי הסימנים המקובלים בפורמט XSB לצורכי נוחות):

- במקום שתא יוכל להכיל את הערך '@' נשמור את מיקום השומר ב-2 משתנים worker_row, worker_col.
- מכיוון שאין צורך ב-'@', הרי שגם אין צורך ב-'+'.
- תאים שהם goal נשארים קבועים במהלך הריצה. לכן, אם נשמור את אותם תאים שמסמנים goal במשתנה מסוים, הרי שאין צורך בסימבול '.' .
- מכיוון ששמרנו את תאי ה-goal במשתנה, הרי שהסימבול '*' לא באמת משמעותי- מספיק שנשנה את התא ל-'$'.

הגענו למצב שכל תא board[i][j] בלוח המשחק יכול לקבל את ערך הערכים הבאים:

$$\{b, x, \_\}$$

ובנוסף יש לנו 2 משתנים עבור מיקום השחקן.

כמו כן, התאים המסמנים קירות לעולם לא ישתנו, ולכן נוכל ליצור לוח **קבוע** חדש שישמור את מיקומי הקירות בלבד. על מנת לחסוך בזיכרון, ניתן להגדירו כלוח בוליאני שמכיל true אם בתא יש קיר, אחרת false.
כעת לוח המשחק מכיל שני סימבולים בלבד: $\{b, \_\}$. נוכל להגדיר גם את לוח המשחק בצורה בוליאנית, כך שהלוח מכיל true אם בתא יש קופסה, אחרת false.

כמו כן יש לנו משתנה המייצג את תזוזת השומר: $movement \in \{u, d, r, l, 0\}$

סה"כ, המשתנים של ה-FDS הם:

- לוח בוליאני board שמכיל true אם בתא יש קופסה, אחרת false.
- לוח בוליאני walls שמכיל true אם בתא יש קיר, אחרת false.
- שני משתנים שמחזיקים את ערכי השורה והעמודה של השומר: worker_row, worker_col.
- משתנה שמייצג את תזוזת השומר $movement \in \{u, d, r, l, 0\}$

- ***θ:***

המצב התחלתי של המערכת הוא 2 לוחות בוליאניים שמציינים מיקומי ארגזים ומיקומי קירות.

בנוסף 2 משתנים שמחזיקים את קואורדינטות מיקום התחלתי של השומר.

movement=0 במצב ההתחלתי.

- ***ρ:***

ראשית, הגדרנו 8 משתנים אשר משתנים בכל איטרציה. המשנים האלו יקבעו את ה-state הבא של מיקום השומר ושל הלוח board.

```
DEFINE
    down_step := worker_row<{rows-1} & !walls[worker_row+1][worker_col] & !board[worker_row+1][worker_col];
    down_push := worker_row<{rows-2} & board[worker_row+1][worker_col] & !walls[worker_row+2][worker_col] & !board[worker_row+2][worker_col];

    right_step := worker_col<{columns-1} & !walls[worker_row][worker_col+1] & !board[worker_row][worker_col+1];
    right_push := worker_col<{columns-2} & board[worker_row][worker_col+1] & !walls[worker_row][worker_col+2] & !board[worker_row][worker_col+2];

    left_step := worker_col>0 & !walls[worker_row][worker_col - 1] & !board[worker_row][worker_col - 1];
    left_push := worker_col>1 & board[worker_row][worker_col - 1] & !walls[worker_row][worker_col - 2] & !board[worker_row][worker_col - 2];

    up_step := worker_row>0 & !walls[worker_row - 1][worker_col] & !board[worker_row - 1][worker_col];
    up_push := worker_row>1 & board[worker_row - 1][worker_col] & !walls[worker_row - 2][worker_col] & !board[worker_row - 2][worker_col];
```

המשתנים הבוליאניים הללו קובעים:

- אם התזוזה של השומר לכיוון מסוים אפשרית: step_ בודק אם אין קופסה ואין קיר במרחק 1 בכיוון אליו השומר רוצה לזוז (לפי שני הלוחות הבוליאניים המוגדרים ו-movement). אם התזוזה לכיוון אפשרית, אז המשתנה יהיה true אחרת false.
- אם דחיפת קופסה לכיוון מסוים אפשרית: push_ בודק אם יש קופסה במרחק 1 מהשומר, ואין קופסה או קיר במרחק 2 מהשומר בכיוון אליו השומר רוצה לזוז. אם הדחיפה אפשרית, אז המשתנה יהיה true אחרת false.

<u>המעברים האפשריים של המערכת יהיו נתונים לפי המצבים הבאים:</u>

- מטריצת הקירות מוגדרת בתור מטריצה קבועה ולכן לא רלוונטי לה פונקציית מעברים.

- פונקציית המעברים של תנועת השומר נתונה על ידי:

```
# update worker_row, worker_col according to movement
def worker_location_change(rows, columns):
    model_content = f'''
next(worker_row):=
case
    movement = u & (up_step | up_push): worker_row - 1;
    movement = d & (down_step | down_push): worker_row + 1;
    TRUE: worker_row;
esac;

next(worker_col):=
case
    movement = r & (right_step | right_push): worker_col + 1;
    movement = l & (left_step | left_push): worker_col - 1;
    TRUE: worker_col;
esac;
    '''
    return model_content
```

ניתן לראות ששורת ועמודת השומר מתעדכנים בהתאם לכיוון התזוזה וכן בתנאי שהתזוזה אפשרית (_step או _push).

כך אם התזוזה היא למעלה/למטה ואפשרית אז השורה תגדל/תקטן ואם התזוזה ימינה/שמאלה אפשרית אז העמודה תגדל/תקטן.

- <u>פונקציית המעברים של הלוח הבוליאני של הקופסאות:</u>

```python
def moves(i,j, rows, columns, board):
    model_content=""
    # right
    if i>=0 and j-2>=0 and board[i][j-2]!="x":
        model_content+=f"\tmovement=r & right_push & worker_row={i} & worker_col={j-2} : TRUE;\n"
    if i>=0 and j-1>=0 and board[i][j-1]!="x":
        model_content+=f"\tmovement=r & right_push & worker_row={i} & worker_col={j-1} : FALSE;\n"

    # down
    if i-2>=0 and j>=0 and board[i-2][j]!="x":
        model_content+=f"\tmovement=d & down_push & worker_row={i-2} & worker_col={j} : TRUE;\n"
    if i-1>=0 and j>=0 and board[i-1][j]!="x":
        model_content+=f"\tmovement=d & down_push & worker_row={i-1} & worker_col={j} : FALSE;\n"

    # left
    if i>=0 and j+2<columns and board[i][j+2]!="x":
        model_content+=f"\tmovement=l & left_push & worker_row={i} & worker_col={j+2} : TRUE;\n"
    if i>=0 and j+1<columns and board[i][j+1]!="x":
        model_content+=f"\tmovement=l & left_push & worker_row={i} & worker_col={j+1} : FALSE;\n"

    # up
    if i+2<rows and j>=0 and board[i+2][j]!="x":
        model_content+=f"\tmovement=u & up_push & worker_row={i+2} & worker_col={j} : TRUE;\n"
    if i+1<rows and j>=0 and board[i+1][j]!="x":
        model_content+=f"\tmovement=u & up_push & worker_row={i+1} & worker_col={j} : FALSE;\n"
    model_content+=f"\n"
```

על מנת לצמצם את הסיבוכיות, אנחנו מבצעים בדיקות האם התזוזה אפשרית בהתאם לאינדקס התא: למשל, אם הקופסה נמצאת משמאל לקיר, אז לעולם לא נוכל להזיז את הקופסה ימינה. לכן, אין צורך בלרשום case כזה (מתבצע בעזרת בדיקות ה-if בפייתון).

בפונקציה זו ניתן לראות שינוי תא (הזזת קופסה) לארבעת הכיוונים:

- <u>ימינה:</u>
  - אם התזוזה היא ימינה, ודחיפה ימינה אפשרית, והשומר נמצא באותה שורה של הקופסה ו-2 עמודות משמאלה, אז התא יהפוך להיות true (הייתה קופסה עמודה אחת משמאל לתא ולכן right_push=true, והשחקן דחף אותה לתא הנוכחי. כלומר, יש כעת קופסה בתא).
  - אם התזוזה היא ימינה, ודחיפה ימינה אפשרית, והשומר נמצא באותה שורה של הקופסה ו-1 עמודות משמאלה, אז התא יהפוך להיות false (יש קופסה בתא הנוכחי, השחקן נמצא משמאל לקופסה, right_push=true ולכן השחקן דוחף את הקופסה מהתא הנוכחי. כלומר, אין יותר קופסה בתא).

- <u>למטה:</u>
  - אם התזוזה היא למטה, ודחיפה למטה אפשרית, והשומר נמצא 2 שורות מעל לתא ובאותה עמודה, אז התא יהפוך להיות true.
  - אם התזוזה היא למטה, ודחיפה למטה אפשרית, והשומר נמצא שורה אחת מעל לתא ובאותה עמודה, אז התא יהפוך להיות false.

- <u>שמאלה:</u>
  - אם התזוזה היא שמאלה, ודחיפה שמאלה אפשרית, והשומר נמצא באותה שורה של הקופסה ו-2 עמודות מימינה, אז התא יהפוך להיות true.
  - אם התזוזה היא שמאלה, ודחיפה שמאלה אפשרית, והשומר נמצא באותה שורה של הקופסה ו-1 עמודות מימינה, אז התא יהפוך להיות false.
- <u>למעלה:</u>
  - אם התזוזה היא למעלה, ודחיפה למעלה אפשרית, והשומר נמצא 2 שורות מתחת לתא ובאותה עמודה, אז התא יהפוך להיות true.
  - אם התזוזה היא למעלה, ודחיפה למעלה אפשרית, והשומר נמצא שורה אחת מתחת לתא ובאותה עמודה, אז התא יהפוך להיות false.

- <u>פונקציית המעברים של ה:movement</u>

```
model_content+=f"next(movement):={{u, d ,l ,r}};"
```

פונקציה זו מגדירה את התנועה הבאה שהיא משתנה בין $\{u, d, l, r\}$.

- <u>***J:***</u>

```
model_content+=f"\nJUSTICE"
model_content+=f"\n\t"
for i in range(rows):
    for j in range(columns):
        if board[i][j]!='x' and board[i][j]!='.':
            border = ((board[i][j+1]=='x' or board[i][j-1]=='x') and (board[i-1][j]=='x' or board[i+1][j]=='x'))
            if border:
                model_content += f"!board[{i}][{j}] & "
        if board[i][j]=='x':
            model_content += f"!board[{i}][{j}] & "
model_content = model_content[:-3]  # remove the last " & "
model_content+=f";\n"
```

ה-justice מבטיח שלעולם לא יהיה קופסה במיקום של קיר וגם שלא תהיה קופסה במיקום שלא ניתן להוציא אותו ממנו (לדוגמה, מצב בו יש קיר מימין ומעל לקופסה), מה שחוסך מקרים רבים ובכך מצמצם ריצות אפשריות שלא יקדמו אותנו לפתרון.

- <u>***C:***</u>

לא היה צורך ב-Compassion במימוש שלנו.

2. נגדיר את ה-temporal logic:

```
model_content += f"DEFINE\n"
model_content +=f"\treach:= "
for i in range(rows):
    for j in range(columns):
        if board[i][j]=='.':
                model_content += f"board[{i}][{j}] & "
model_content = model_content[:-3]  # remove the last " & "
model_content+=";"

model_content += f"\n\tLTLSPEC G(!reach)"
```

$$LTLSPEC : G(!reach)$$

משתנה reach מוגדר כנקודות בהן יש נקודה בלוח הקלט (שהקופסאות אמורות להיות עליהם בשביל ניצחון).

אנחנו בחרנו להשתמש בדרך השלילה- זאת אומרת שהתוכנה שלנו מוגדרת לבדוק מקרה שבו לעולם, בשום פיצול באפשרויות לא נגיע למצב שבו כל הקופסאות נמצאות על נקודת reach.

כך, אם reach מתקיים משמע שלא ניתן לנצח ולכן ה output של LTL יהיה true כי לא ניתן לנצח.

אם דווקא כן ניתן לנצח, הרי שה-output של ה-LTL יהיה false כי כן ישנו מצב שבו כל הארגזים יהיו על נקודות הניצחון (reach) ולכן הריצה תיתן לנו את הדרך שבה זה "לא מתקיים" משמע את דרך הניצחון.

חלק 2

על מנת להריץ את הקוד, יש לנווט לתיקייה המכילה את קבצי הקוד, ולהריץ את הפקודה הבאה דרך ה-cmd:

py sokoban.py input.txt BDD

כאשר input הוא שם קובץ הטקסט המורץ, ו-BDD הוא ה-solver engine (יכול להיות BDD או SAT, כאשר אם רוצים להוסיף הגבלה על מספר הצעדים עבור SAT צריך להוסיף רווח ולכתוב את מספר הצעדים, אחרת מספר הצעדים הוא הדיפולטיבי 10).

הרצנו מספר לוחות פתירים ולא פתירים. נציג את הלוחות ואת תוצאת הריצה:

- לוח פתיר 1:

```
########
####__##
##___$##
#_$.@_##
#___.__#
#_$_.__#
###__###
########
```

```
Execution time:  3.2756712436676025 seconds
The board is solveable. Solution:
uurDDlddrruLulldlluRlddRRR
```

- לוח פתיר 2:

```
#########
####__###
####$__##
####__$_#
#_$___._#
#__..___#
##._@$_##
##__#####
#########
```

```
Execution time:  38.75794339179993 seconds
The board is solveable. Solution:
urrdLLLulluRRRurDrdLLuururDluulDDDrR
```

- לוח לא פתיר 1:

```
#######
###$_.#
#@____#
#####$#
#####.#
#######
```

```
Execution time:  0.05510520935058594 seconds
The board is not solveable
```

- <u>לוח לא פתיר 2:</u>

```
#########
#_______#
#__@____#
#_______#
#####___#
#_$_#####
#___#####
###.#####
#########
```

```
Execution time:  0.10806465148925781 seconds
The board is not solveable
```

את קבצי ה-smv. וה-out ניתן למצוא בתיקיה המתאימה לשם הקובץ שהורץ בתור ./outputs .

<u>הסבר הקוד</u>

הקוד מורכב ממספר קבצים:

- <u>Sokoban.py:</u> הפונקציה הנ"ל מקבלת את שורת הפקודה שהמשתמש מכניס ב-cmd ומבצעת שני דברים:
  - Extraction ללוח המשחק שבקובץ ה-txt. ל-String.
  - Extraction ל-solver engine שהוכנס ומספר הצעדים עבור ה-SAT (אופציונאלי).
  - העברת כל הפרטים הנ"ל לפונקציה המתאימה- solver.py או solver_iterative.py (נפרט בשאלה 4).

- <u>Solver.py:</u> הפונקציה מקבלת את כל הפרטים מ-sokoban.py ומבצעת:
  - יוצרת תיקייה ב-outputs/. עם שם הקובץ שנקלט וה-solver_engine שלו.
  - קוראת לפונקציה assign_board שב- board.assignment.py, ומקבלת חזרה את לוח המשחק בצורה של מערך דו מימדי, ללא השחקן.
  - קוראת ל-generate_nusmv_model שב-model_generation.py, ומקבלת חזרה מחזורת ארוכה המכילה את כל תוכן קובץ ה-nuxmv.
  - שומרת את תוכן המחזורת בקובץ smv.
  - מריצה את run_nuxmv שב-run_nuxmv.py ומקבלת כפלט את תוכן קובץ ה-out. של קובץ ה-smv. שהורץ בה.
  - קוראת ל-extract_LURD שב-LURD_format_creator.py ומקבלת כפלט את הפורמט LURD.
  - יוצרת קובץ LURD.out המכיל את פתרון הריצה וזמן הריצה.

- Board_assignment.py: מקבלת כקלט מחזורת המייצגת את לוח המשחק, ומחזירה כפלט את לוח המשחק בצורה של מערך דו מימדי ללא השחקן, ו-list שמחזיק את מיקום השומר.

- Model_generation.py: מקבל כקלט את לוח המשחק ואת מקום השומר ויוצרת את תוכן קובץ ה-smv. בהתאם ל-FDS שהוגדר בסעיף הקודם.

- Run_nuxmv.py: מריץ את קובץ ה-smv. בהתאם ל-solver_engine ולמספר הצעדים שהוכנסו כקלט על ידי המשתמש. מוציא כפלט קובץ out.

- LURD_format_creator.py: מטרת הקובץ היא ליצור קובץ בפורמט LURD תקין, כלומר שיכיל אות גדולה עבור דחיפה, ואות קטנה עבור תזוזה.
  - מקבל כקלט את תוכן קובץ ה-out., כלומר את תוצאת הריצה.
  - אם reach=true הרי שלא ניתן לפתור את הלוח ולכן קובץ ה-LURD שיוצר יכתוב שאין פתרון.
  - ניצור list שמכיל 2 lists: אחד עבור התזוזה והאחר עבור pushes. כל תא ברשימת התזוזות תכיל את התזוזה שהתקיימה בצעד הנוכחי, וכל תא ברשימת הדחיפות יכיל את הדחיפות האפשריות בשלב זה (כלומר שערכן true), כלומר את האות הראשונה של כיוון הדחיפה האפשרית (u,d,l,r).
  - נעבור על הרשימות ואם עבור אותו אינדקס יש תזוזה לכיוון מסוים, וגם push לאותו כיוון, אז נהפוך את ה-push לאות גדולה. כך ניצור פורמט LURD תקין.

חלק 3

בחלק זה השוונו בקוד בין ריצה של BDD לבין ריצה של SAT ומדדנו את הזמן שלקח לבצע.

לאחר כל הרצה נפתחה תיקייה עם השם של קובץ הטקסט בצירוף המילה BDD או SAT שבה יש name.out,name.smv ו name_LURD.out .

כל התיקיות מצורפות בקבצי ההגשה בתיקייה בGIT .

לוח 1:

```
#######
#######
#____.##
#_$_$.##
####__##
##._@$##
####__##
#######
```

BDD:

```
Output saved to bad_BDD\bad.out
Execution time:  0.7230000495910645 seconds
The board is solveable. Solution:
drUUlUlulldRRuRDDrdLL

C:\Users\asafb\OneDrive - Bar-Ilan University - Students\[redacted]\sokoban>
```

SAT:

הגבלנו את ה sat ל 35 צעדים וקיבלנו:

```
Output saved to bad_SAT\bad.out
Execution time:  4.592982292175293 seconds
The board is solveable. Solution:
drUUlUlulldRRuRDDrdLL

C:\Users\asafb\OneDrive - Bar-Ilan University - Students\[redacted]\sokoban>
```

כאן ניתן לראות שזמן הריצה של הBDD קצר משמעותית מזמן הריצה של SAT .

לוח 2:

```
########
#_____##
#_$$_@##
#_#_.###
#_._ ###
#_$_._##
##__####
########
```

BDD:

```
Output saved to example2_Q3_BDD\example2_Q3.out
Execution time:  1.9489822387695312 seconds
The board is solveable. Solution:
ullDulldRRDrdLuuurDllldddRRd

C:\Users\asafb\OneDrive - Bar-Ilan University - Students\□□□□□ □□□□□□\sokoban>
```

SAT:

הגבלנו את ה sat ל 35 צעדים וקיבלנו:

```
Output saved to example2_Q3_SAT\example2_Q3.out
Execution time:  21.01600217819214 seconds
The board is solveable. Solution:
lulDulldRRDrdLuuurDllldddRRl

C:\Users\asafb\OneDrive - Bar-Ilan University - Students\□□□□□ □□□□□□\sokoban>
```

גם כאן ניתן לראות שזמן הריצה של הBDD קצר משמעותית מזמן הריצה של SAT. ניתן לראות שהפתרון שונה אך נכון גם כן.

עם זאת, לא בכל המקרים BDD מהיר מ-SAT:

לוח 3:

```
##########
#_@#__._#
#__#__$__#
#__#___$_#
#__#__#._#
#__#__#__#
#__#__#__#
#__#__####
#_____$.#
####.$___#
##########
```

BDD:

```
INPUT BOARD :
##########
#_@#__._#
#__#__$__#
#__#___$_#
#__#__#._#
#__#__#__#
#__#__#__#
#__#__####
#_____$.#
####.$___#
##########
Execution time:  46.60945439338684 seconds
The board is solveable. Solution:
ddddddddrrrrRdlLuuuuuurUrD
```

SAT:

```
INPUT BOARD :
##########
#_@#__._#
#__#__$__#
#__#___$_#
#__#__#._#
#__#__#__#
#__#__#__#
#__#__####
#_____$.#
####.$___#
##########
Execution time:  14.762916803359985 seconds
The board is solveable. Solution:
ddddddddrrrrRldLuuuuuurUrD
```

לוח 4:

```
###########
####_____.#
#_$__$__###
#____.__###
##____#_###
#_________#
#___$___##
#_#_@__#__#
#_##____.##
#_____#####
###########
```

BDD:

```
INPUT BOARD :
###########
####_____.#
#_$__$__###
#____.__###
##____#_###
#_________#
#___$___##
#_#_@__#__#
#_##____.##
#_____#####
###########
Execution time:  349.95627069473267 seconds
The board is solveable. Solution:
uRRRurDDuuluullUdllllluRRRuRDuRRR
```

SAT:

```
C:\Users\User\Desktop\final\Formal_Varification_Final>py sokoban.py Q3_ex4.txt SAT 45
INPUT BOARD :
###########
####_____.#
#_$__$__###
#____.__###
##____#_###
#_________#
#___$___##
#_#_@__#__#
#_##____.##
#_____#####
###########
Execution time:  149.45771622657776 seconds
The board is solveable. Solution:
uRRRurDDullulluuUlldllluRRRuRDuRRR
```

ניתן לראות כי מספר הצעדים בלוחות הדוגמה פחות או יותר זהה. ההבדל העיקרי בין לוחות 1 ו-2 לבין לוחות 3 ו-4 הוא גודל ומורכבות הלוח: לוחות 1 ו-2 קטנים וקלים מלוחות בעוד שלוחות 3-4 גדולים וקשים יותר. ככל שהלוח קטן ופשוט יותר, כך ה-BDD solver יפתור מהר יותר, אך ככל שהלוח גדול ומסובך יותר- ה-SAT יפתור מהר יותר

חלק 4

בחלק זה הגדרנו הרצה איטרטיבית שמתבצעת באופן הבא:

נחלץ מהלוח את ה-goals כפי שעשינו בריצה הרגילה, אך כעת במקום לרוץ על כולן יחד, נרוץ בלולאה. בכל איטרציה של הלולאה, ניקח את אחת הנקודות, ואת השאר נסמן בתור '_' ונריץ באופן זהה להרצה הרגילה. בשלב זה הבאנו קופסה לאחת המטרות. לאחר מכן ניצור לוח חדש: הלוח החדש יכיל קיר '#' במקום אליו הגענו, שמטרתו לסמן שלא ניתן לנוע או לדחוף קופסה לתא זה. בהתאם לתזוזה האחרונה שבוצעה נוכל לדעת באיזה מיקום השומר נמצא, ולשנות את ערך המיקום בהתאם, כך שיהיה "צמוד" לקופסה שהוזזה למטרה. לאחר מכן נבצע את האיטרציה הבאה של הלולאה, בהתאם למטרה הבאה.

נשים לב שהמטרה אליה נרצה להגיע תחילה היא המטרה הרחוקה ביותר ממיקום השחקן בתחילת המשחק, והמטרה האחרונה עליה הלולאה תרוץ היא המטרה הקרובה ביותר. הסיבה לכך היא שנקודות קרובות עלולות לחסום את המעבר לנקודות הרחוקות יותר. שיטה זו עלולה להגדיל את זמן הריצה בחלק מהמקרים (למשל, מקרים שהקופסאות אינן מפריעות אחת לשנייה, ואחת מהן צמודה לשומר והאחרת רחוקה ממנו. הוא יפתור תחילה את הרחוקה, ויאלץ לחזור לקרובה), אולם היא מונעת מקרים רבים מאוד בהם התוצאה תהיה שאין פתרון למרות שפתרון כן קיים (למשל Sokoban Classic 1).

הסבר הקוד

הקוד למקרה האיטרטיבי נמצא בקובץ iterative_solver.py בלבד ומשתמש בשאר הקבצים שתוארו בסעיף 2. לכן נסביר רק קובץ זה.

- py נקרא ע"י sokoban.py ומקבל כקלט את הלוח כמחרוזת.
- נוצרת תיקייה ב-outputs/. בשם filename_iterative כאשר filename הוא שם הקובץ txt. שהתקבל כקלט ע"י המשתמש.
- הלוח ומיקום השומר מתקבלים מהקריאה ל-board_assignment.py.
- המרחקים מהשחקן ל-goals מחושבים ונשמרים ברשימה.
- מתבצעת לולאה הרצה על אותה רשימה בסדר הפוך (מהרחוק ביותר לקרוב ביותר).
- בכל איטרציה הלוח משתנה כך שיכיל את הנקודה הנוכחית בלבד, ושאר הנקודות הופכות לסימבול '_' . אם כבר הגענו למטרה בנקודה זו, הופכים אותה לקיר 'x'.
- מבצעת קריאה ל-model_generation.py בו נוצר נוצרת המחרוזת עבור קובץ ה-smv. עבור הנקודה הנוכחית בלבד.
- מתבצעת קריאה ל-run_and_file_creation. בפונקציה זו:
  - נוצר הקובץ smv.
  - מתבצעת קריאה ל-run_nuxmv.py אשר מריצה את הקובץ smv ומחזירה כפלט את הקובץ out. .
  - מתבצעת קריאה ל-LURD_format_creator.py כדי ליצור את הפורמט LURD התקין. לאחר מכן ה-LURD וזמן הריצה נשמרים בקובץ.
- הלוח מוחזר לצורתו המקורית.
- מתבצעת קריאה ל-extract_new_board_formation בה הלוח החדש נוצר בהתאם ל-LURD:
  - יכיל x במיקום שאליו הבאנו את הקופסה, כך שלא יהיה ניתן לזוז או לדחוף לתא זה.
  - יכיל _ במיקום המקורי של הקופסה.
- מתבצעת בדיקה של האות האחרונה ב-LURD ולפיה נקבע המיקום של השחקן לאיטרציה הבאה (צמוד לקופסה שהובאה למטרה).
- בסיום כל האיטרציות מודפס זמן הריצה הכולל ללוח.

בתיקיית הGitHub צירפנו דוגמות לכמה זמן לוקח לכל לוח לרוץ בהרצה רגילה שאינה איטרטיבית לצורך השוואה (דוגמאות 1-3 בעזרת BDD ודוגמה 4 בעזרת SAT).

<u>דוגמה 1:</u>

```
C:\Users\asafb\OneDrive - Bar-Ilan University - Students\□□□□□ □□□□□□\sokoban>py sokoban.py Q4_ex1.txt BDD
INPUT BOARD :
#########
####___##
#__._#$##
#_$@_#_##
##._.#__#
#_______#
#_$$__.##
###__####
#########
solving for box number 1 :
Execution time:  26.714139461517334 seconds
The board is solveable. Solution:
ruurrDDDD
solving for box number 2 :
Execution time:  5.525997877120972 seconds
The board is solveable. Solution:
lldlUUluR
solving for box number 3 :
Execution time:  1.925093173980713 seconds
The board is solveable. Solution:
uulD
solving for box number 4 :
Execution time:  0.27600717544555664 seconds
The board is solveable. Solution:
rddlldRurrddlUUUU
Total Time : 34.45418429374695 seconds


C:\Users\asafb\OneDrive - Bar-Ilan University - Students\□□□□□ □□□□□□\sokoban>
```

<u>דוגמה 2:</u>

```
שורת הפקודה

C:\Users\asafb\OneDrive - Bar-Ilan University - Students\□□□□□ □□□□□□\sokoban>py sokoban.py Q4_ex2.txt BDD
INPUT BOARD :
##########
####__####
###__$.__#
###.__$__#
#_$_$___##
#__.__@_##
####_#####
####.#####
##########
##########
solving for box number 1 :
Execution time:  26.38899803161621 seconds
The board is solveable. Solution:
llUruL
solving for box number 2 :
Execution time:  6.303004741668701 seconds
The board is solveable. Solution:
rRuLulDDDDD
solving for box number 3 :
Execution time:  2.058997392654419 seconds
The board is solveable. Solution:
ullluRRRurDrdLL
solving for box number 4 :
Execution time:  0.3199472427368164 seconds
The board is solveable. Solution:
rrruUruL
Total Time : 35.07591462135315 seconds
```

ניתן לראות ששיטת ההרצה האיטרטיבית שיפרה משמעותית את זמן הריצה ולוחות שעבורם היה לוקח באופן רגיל זמן רב לפתור אותם נפתרו בשיטה זו בשניות בודדות (ניתן לראות בGitHub את ההשוואות).

דוגמה 3:

```
C:\Users\User\Desktop\formal\Formal_Varification_Final>py sokoban.py Q4_ex3.txt iterative
INPUT BOARD :
#########
#__#___##
#__##__##
#_@___###
##$____##
#____#.##
#__$_#.##
#_______#
###____##
#########

solving for box number 1 :
Execution time:  4.249801874160767 seconds
The board is solveable. Solution:
rddDldRRRdrU
solving for box number 2 :
Execution time:  0.27526211738586426 seconds
The board is not solveable
Total Time : 4.537034749984741 seconds
```

בדוגמה זו ניתן לראות לוח שהרצתו בצורה איטרטיבית נכשלה ולא נמצא פתרון. הסיבה לכך תמונה לכך שפתרון איטרטיבי הוא מאין פתרון חמדני, כלומר- מוצא את הפתרון הטוב ביותר לנקודה הספציפית הנוכחית, ללא התחשבות בעתיד. דבר זה גורם לכך שניתן למשל לפתור קופסה אחת, אך השנייה תתקע ולא ימצא פתרון ללוח, כפי שניתן לראות בדוגמה זו.

דוגמה 4:

בדוגמה זו נריץ על לוח גדול יותר, בגודל 11x12 (132 תאים).

```
C:\Users\User\Desktop\final\Formal_Varification_Final>py sokoban.py Q4_ex4.txt iterative
INPUT BOARD :
############
####_____._#
#_$__$____#
#____.__##$#
##____#_##.#
####_____###
#____$___###
#_#_@__#__##
#_##____.###
#.__$_##___#
############
solving for box number 1 :
Execution time:  541.5128638744354 seconds
The board is solveable. Solution:
uuuurUluRRRR
solving for box number 2 :
Execution time:  37.080116748809814 seconds
The board is solveable. Solution:
drrD
solving for box number 3 :
Execution time:  18.68704390525818 seconds
The board is solveable. Solution:
ullldddldLdddLLL
solving for box number 4 :
Execution time:  23.092314958572388 seconds
The board is solveable. Solution:
rruuluRRRRurDD
solving for box number 5 :
Execution time:  3.7401440143585205 seconds
The board is solveable. Solution:
uuluullllllluRRRurD
Total Time : 624.1220102310181 seconds
```

ניתן לראות שהפתרון האיטרטיבי מצליח לפתור גם לוחות גדולים ומסובכים הרבה יותר בזמן קצר מאוד (10 דקות עבור לוח עם 5 קופסאות).

לסיכום, ניתן לראות ש-BDD Solver מצליח למצוא פתרון ללוחות קטנים באופן מהיר מאוד ביחס ל-SAT, ועבור לוחות גדולים ומסובכים ה-SAT מצליח להגיע לפתרון הרבה יותר מהר מה-BDD.

כמו כן, פתרון איטרטיבי יכול להביא לנו פתרון מהיר משני ה-solvers הקודמים, אך יתכן שלא יוכל למצוא ללוח פתרון גם אם קיים שכן הוא פותר באופן חמדני.