פרויקט באימות פורמלי

מגישים:

אסף בן אור 209381599

דניאל יופה 324061878

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

https://github.com/AsafBenor/Formal_Varification_Final.git

חלק 1

.1

נגדיר את ה FDS של לוח הסוקובן:

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

-V

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

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

```
symbol_map = {'@': '_', '+': '_', '$': 'b', '*': 'b', '#': 'x', '.':
'.', '_': '_'}
```

ממנו נגדיר שני לוחות בוליאניים:

- לוח board שמכיל true אם בתא יש קופסה, אחרת false.
- לוח walls שמכיל true אם בתא יש קיר, אחרת false.

את 2 הלוחות אנו שומרים כמערך דו ממדי בגודל n*m .

בנוסף יש לנו את המשתנים הבאים:

- משתנה שמייצג את תזוזת השומר $movment \in \{u, d, r, l, 0\}$
- שני משתנים שמחזיקים את ערכי השורה והעמודה של השומר: worker_row, worker_col.

-θ

מצב התחלתי של המערכת- 2 לוחות בוליאניים שמציינים מיקומי ארגזים ומיקומי קירות.

בנוסף 2 משתנים שמחזיקים קורדינטות מיקום התחלתי של השומר.

movement=0 במצב ההתחלתי.

$-\rho$ -

ראשית, הגדרנו 8 משתנים אשר משתנים בכל איטרציה:

```
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. 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.

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

המעברים האפשריים של המערכת יהיו נתונים לפי המצבים הבאים:

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, ולכן 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.

פונקציית המעברים של הלוח הבוליאני של הארגזים:

```
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"
```

בפונקציה זו ניתן לראות שבהתאמה לארבעת הכיוונים:

- אם נרצה להזיז הקופסא ימינה למשבצת i,j אז בודקים אם השחקן נמצא בעמודה j-2ובשורה i והתנאי rightpush מתקיים.
- אם נרצה להזיז את הקופסא שמאלה למשבצת i,j אז בודקים אם השחקן נמצא בעמודה j+2 ובשורה i והתנאי leftpush מתקיים.
- אם נרצה להזיז הקופסא למעלה למשבצת i,j אז בודקים אם השחקן נמצא בעמודה jובשורה i+2 והתנאי uppush מתקיים.
- אם נרצה להזיז הקופסא למטה למשבצת i,j אז בודקים אם השחקן נמצא בעמודה j ובשורה i-2 והתנאי downpush מתקיים.

ולבסוף פונקציית המעברים של ה:movement

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

פונקציה זו מגדירה את התנועה הבאה שהיא משתנה בין.u,d,l,r

```
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"
```

J-

ה justiceמבטיח שלעולם לא יהיה ארגז במיקום של קיר וגם שלא יהיה ארגז במיקום שלא ניתן להוציא אותו ממנו, מה שחוסך מקרים רבים ובכך מצמצם ריצות אפשריות שלא יקדמו אותנו לפתרון.

C- לא היה צורך ב-Compassion במימוש שלנו.

.2

```
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 (שהארגזים אמורים 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 הארגזים נמצאים 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:

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).

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 .

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:

הלוח-

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

-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\[illegible] [illegible]\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\[illegible] [illegible]\sokoban>
```

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

לוח 2:

הלוח-

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

-BDD

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

C:\Users\asafb\OneDrive - Bar-Ilan University - Students\[redacted] [redacted]\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:
lulDulldRRDrdLuuurDlllddRRl

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

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

ניתן לראות שזמן הריצה בעזרת BDD קטן מזמן ריצה בעזרת SAT. עם זאת, שמנו לב שעבור לוחות גדולים יותר, הזיכרון של המחשב גדל במידה רבה בהרצה עם BDD, מה שקורה במידה פחותה ב-SAT. כלומר, ה-trade-off בין שני ה-solvers הוא זכרון-זמן.

למשל, עבור הלוח הבא:

חלק 4

בחלק זה הגדרנו הרצה איטרטיבית שמתבצעת כך שבכל פעם אנו מגדירים רק נקודה (goal) אחד בלבד שאליו ניתן להביא את הקופסאות, ובכל פעם נבדוק איך ניתן להביא קופסה אחת לנקודה זו.

נתחיל בנקודה הכי רחוקה מעמדת השחקן בתחילת המשחק ונסיים בזו הקרובה ביותר (מכיוון שנקודות קרובות עלולות לחסום את המעבר לנקודות הרחוקות יותר).

בכל פעם שנצליח לביא קופסא לנקודה נסמן את התא אליו הבאנו את הקופסה בתור x (קיר) וכך באיטרציה הבאה נדע שאנו פועלים עם קופסה אחת פחות והשחקן והקופסאות האחרות אינם יכולים לדרוך בתא זה.

בתיקיית הGitHub צירפנו גם דוגמאות לכמה זמן לוקח לכל לוח לרוץ בהרצה רגילה שאינה איטרטיבית לצורך השוואה.

דוגמא 1:

```
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:
lldd1UU1uR
solving for box number 3 :
Execution time:  1.925093173980713 seconds
The board is solveable. Solution:
uu1D
solving for box number 4 :
Execution time:  0.27600717544555664 seconds
The board is solveable. Solution:
rdd11dRurrdd1UUUU
Total Time : 34.45418429374695 seconds


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

דוגמא 2:

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 בהחלט 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).

דוגמא 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.

לסיכום, ניתן לראות שישנו tradeoff משמעותי בין BDD Solver לבין SAT Solver, כאשר האחד מביא לנו תוצאה מהירה אך זקוק לזיכרון רב, ואילו האחר צורך פחות זיכרון אך ריצתו ארוכה יותר.

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.