

**האוניברסיטה הפתוחה**  
**המחלקה למתמטיקה ולמדעי המחשב**

## **פתרון פזלי הרכבה כלליים**

עבודת תזה זו הוגשה כחלק מהדרישות לקבלת תואר

"מוסמך למדעים" M.Sc. במדעי המחשב

באוניברסיטה הפתוחה

החטיבה למדעי המחשב

על-ידי

**שחר טל**

העבודה הוכנה בהדרכתם של

פרופ' גיל ברקת, הטכניון

ד"ר ג'ק וינשטין, האוניברסיטה הפתוחה

**ספטמבר 2010**



## תוכן

7	עמוד	תקציר
9		מבוא
11		תיאור הבעיה
15		שלמות ב-NP
17		האלגוריתמים - ייצוג
19		אוריינטציות
21		החלק הייחודי
23		הפתרונות הייחודיים
25		פתרון הפזל – נסיגה לאחור
29		Matrix Cover
33		היוריסטיקות: size, stranded
35		שיפורים
37		מפתחות ומנעולים
39		תוצאות
45		סיכום
47		מקורות
49		נספח א' שריגים ומקרים מיוחדים
53		נספח ב' מדריך הפעלה
67		נספח ג' מראי מקום לקוד
82		נספח ד' פירסום מכנס FAW 2010



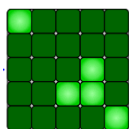
## רשימת איורים

עמוד 14	איור 1 - שריג אורתוגנולי תלת-מימדי
14	איור 2 - פזל וחלקים לדוגמה בשריג אור' דו-מימדי
15	איור 3 - רדוקציה ל- Bin Packing
19	איור 4 - השוואת אוריינטציות לצורך יצירתן
21	איור 5 - אוריינטציות הפזל
24	איור 6 - החלק הייחודי בפתרון הסימטרי
25	איור 7 - עוגן הפזל מספיק לצורך שיכון
29	איור 8 - שיכון ברדוקציה לכיסוי מטריצה
30	איור 9 - ייצוג עבור Dancing Links
37	איור 10 - ייצוג מפתחות ומנעולים
39	איור 11 - דוגמה לפזלים מטבלת התוצאות
42	איור 12 - דוגמה לפזלים נוספים
51	איור 13 - שריג משושי (ראה גם בנספח ד')
52	איור 14 - שריג כדורי
52	איור 15 - שריג טנגרם
53	איור 16 - מקרה פרטי של ייצוג על-פני שריג אור' תלת-מימדי





## תקציר



בעבודה זו יתוארו שיטות כלליות לפתרון פזלים בעלי שריג (lattice) מוגדר היטב (באופן מופשט, קיים מכנה משותף בין החלקים) מסוג הרכבה (put together), להבדיל מפזל קומבינטורי, כמו הקובייה ההונגרית, או Lights Out בו פעולה על 'חלק' בפזל מביאה אותו לתמורה חדשה.

הפזל יוגדר כגרף המושרה מהשריג, ועליו תופעל נסיגה לאחור למציאת כל הפתרונות. חלקי הפזל יתוארו כתת-גרפים של גרף השריג, והפתרון יהיה כיסוי (ללא חפיפות) של גרף זה.

בדרך הטריויאלית לפתרון, הייצוג יהיה straight forward (ייצוג גרפים). מאידך, ניתן לבצע רדוקציה לבעיית בחירת תת מטריצות (Exact Cover), והפעלת טכניקת Dancing Links in Back-Track כמתואר במאמר של Knuth [Kn00].

לבסוף, יוצגו שריגי מבחן ומסקנות.



לדוגמה, הנה שריג אורתוגוני דו-מימדי: פזל המטרה בגודל  $10 \times 6$  ריבועים, והחלקים הם 12 ה-pentominoes. החלק הגרעיני (תא) הוא משבצת בודדת, כאשר ניתן ע"י טרנספורמציות לסובבה ולהופכה (שיקוף). באנלוגיה המדוברת לגרף, משבצת זו הינה צומת, ולה קשתות בהתאם לשכנים האפשריים לאותה משבצת.

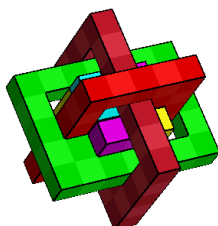


דוגמה מסובכת יותר, הינה ה-tangram. להלן פזל המטרה הוא ריבוע. ניתן לראות כי השריג מורכב ממשולשים קטנים, ב-8 סיבובים אפשריים. החלק הגרעיני הקטן (להלן תא) מסומן בשחור על גבי שריג הפזל. חלק גרעיני זה אינו קיים במציאות ולמעשה כל חלק של הפזל מורכב מאוסף חלקים גרעיניים כאלה (אך באוריינטציות שונות).



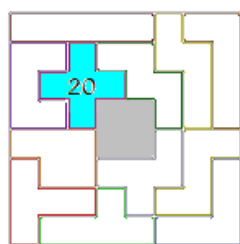


פזלים מסוג הרכבה הינם קלסיים. הנפוצים ביותר הינם הפזלים הדו-מימדיים, אותם יש לכסות/ למלא ללא חפיפות בחלקים הנתונים, כאשר מותר כמובן לסובב חלקים אלו, ואולי גם להופכם (אם אין מגבלת צבעים לדוגמה). בימינו קל למצוא גם פזלים תלת-מימדיים, ואף פירמידות כדורים מורכבות.

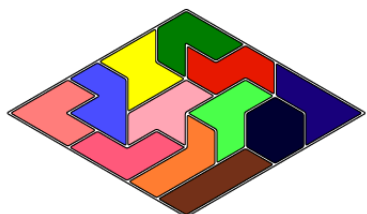


בפזלי הרכבה יש "לארוז" את החלקים, כך שלא תהייה חפיפות ולא יישאר קטע פזל "פנוי". להבדיל, פזלים אחרים בהם איננו עוסקים הם פזלים מסוג Burr - החלקים שזורים זה בזה כך שיש חשיבות לסדר ההרכבה, וכן פזלים קומבינטוריים – כמו פזלים הדורשים מניפולציה חוזרת עד להשגת תבנית המטרה (כמו מגדלי האנוי), או להבדיל - מציאת מיקום החלקים כך שאילווצים מסוימים יסופקו (8 מלכות).

קבוצת חלקים מפורסמת אשר מרכיבה פזלי הרכבה רבים ונחקרה רבות היא ה-pentominoes, זהו מקרה פרטי של polyominoes – קבוצת ריבועים המחוברים בפאותיהם – בו מספר הריבועים הוא 5. הקורא מופנה להרחבה במבוא אשר בנספח ד'. בהקשר לקבוצה זו, נחקרו שיטות לריצוף המישור תוך



שימוש בתכונות ייחודיות של הקבוצה ובפרט עבור פזל ספציפי. [Sc58] Scott מצא את כל 65 הפתרונות לפזל  $8 \times 8$  ללא המרכז  $2 \times 2$  רק בעזרת פיצול מרחב הפתרונות: הוא הבחין כי לפנטומינו  $X$  רק 3 מקומות אפשריים (19, 20, 33). גישתו הראשונה היתה להציב את הפנטומינוס אחד-אחד ע"פ סדר אקראי. בגישה זו ע"פ שיטת מונטה-קרלו<sup>1</sup> [Kn75]  $2 \times 10^{12} \sim$  אפשרויות (!), לעומת זאת הגישה הפשוטה של התקדמות ע"פ סדר לקסיקוגרפי מצמצמת את מספר האפשרויות ל- $9 \times 10^6$  (!). אופטימיזציות נוספות שננקטו הן: שימוש ברמת מאפייני המכונה הספציפית (לדוגמה, גודל הרגיסטר); שיבוץ החלקים לפי מספר האפשרויות לשיבוץ כל חלק [Sc58, GB65, Br71]; או "הפוך": שיבוץ בפזל ע"פ מספר האפשרויות לשיבוץ כל משבצת. Fletcher [Fl65] השתמש בתכונות השריג – מבנה גאומטרי פשוט יחסית - כדי לחסוך בדיקות מיותרות בניסיון השיכון.<sup>2</sup>

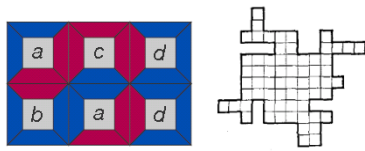


בדומה ל-polyominoes, חלקי ה-hexamond הם מקרה פרטי של Polyamonds אלא שכאן החלק הגרעיני הוא משולש שווה-צלעות,

<sup>1</sup> לקיחת מדגם מהעץ לצורך מתן קירוב למספר הענפים הכולל.

<sup>2</sup> ראה מימוש ב-C (לעומת שפת FAP משנות ה-60) ושגרת findLoc ב-<http://eklhad.net/polyomino/hexsol.c>

ובכל חלק 6 משולשים. Torbijn [To69] מצא את 156 הפתרונות לשריג המשושים בגודל 6x6 ללא מחשב.



ידוע כי פתרון הפזלים הוא בעייה שלמה ב-NP. הדבר הוכח בעזרת רדוקציה מבעיית הריצוף של Wang [Le78]. כל חלק ניתן לייצוג בעזרת פולינום, כאשר הצבעים מקודדים בעזרת "בליטות"/"שקעים". הוכחה אחרת ב-DD07 מבצעת את הרדוקציה מפזל רגיל (jigsaw).

יש להניח אם כן כי אין דרך יעילה יותר לפתרון מאשר brute force search. הפתרון הפשוט - ניסוי וטעייה (generate and test) – הרכבת חלק אחר חלק. שימוש במתודולוגיית הנסיגה לאחור back track מאפשר מעבר שיטתי על כל הפתרונות, והיא טובה יותר מאשר יצירת כל הפתרונות האפשריים ובדיקתם אחד אחד – שכן היא מאפשרת לבנות בשלבים את הפתרון המועמד לבדיקה – generate, test and expand – the children - ולבצע תיקונים במהירות – לעומת בדיקת סדר גודל של מספר המשבצות בחזקת מספר החלקים.<sup>3</sup> בכל שלב, נבדק האם ניתן לשכן את החלק הנכחי בפתרון, ואם לאו, מבוצע שחזור מצב אחרון לאחר, ונסיון עם חלק אחר, כך נחסך תת עץ שלם לחיפוש. למעשה, נעשה כאן Depth First Search (רקורסיה מאפשרת מימוש "טבעי" של חיפוש זה). זו היא נסיגה לאחור קלאסית – back-track – chronological.

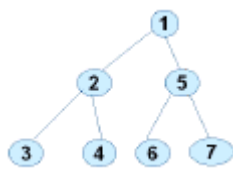
Knuth [Kn00] הציע טכניקה אותה הוא מכנה בשם dancing links. לביצוע מימוש הנסיגה לאחור באופן מהיר בהרבה. שילוב שיטה זו עם ייצוג בעיית הפזל כבחירת תת מטריצות (יש לבחור קבוצת שורות מתוך מטריצת 1 ו-0, כך שבכל עמודה יהיה 1 יחיד) מאפשר מציאת פתרון בזמן סביר.

בעבודה זו סקירה מפורטת של ייצוג הפזל כך שיתאים לפתרון בשיטות הנ"ל: הנסיגה לאחור הטרינויאלית, ומאידך, שימוש בטכניקת ה-Dancing Links. יוסברו היורסיטיקות שונות ליעול החיפוש, ותיבחן התאמת הייצוג על שריגים מסוגים מגוונים: דו ותלת מימד, כדורים, משושים, טנגרם, וצירופי מנעולים. כן תיסקר הדרך להתמודד עם פתרונות כפולים (סימטריים), או אף חלקים כפולים (זהים).

קיימות שלוש קטגוריות כלליות [Ko98] לבעיות חיפוש: מציאת רצף צעדים ממצב התחלתי לסופי (פזלים קומבינטוריים, הסוכן הנוסע); משחקים עם ידע מושלם (דמקה, רברסי ואלמנט מזל – שש בש); וסיפוק אילוצים (8 מלכות, שיבוץ משמרות, הפזלים שלנו) – כאן הרצף לא חשוב, אלא הסיפוק עצמו. כמו שהגבול אינו מוחלט, ולעתים בעיה מתאימה ליותר מקטגוריה יחידה, כך גם השיטות לקטגוריה מסוימת, טובות לאחרת. השיטה הפשוטה הינה Brute Force, זו שיטה סיסטמטית, כזו שתמצא בוודאות פתרון

<sup>3</sup> למעשה זוהי מכפלת מספר האוריינטציות של n החלקים במספר האפשרויות לסדר את החלקים, לדוגמה  $n! * 4^n$

אם קיים, לעומת חיפוש גנטי לדוגמה. כמובן שאין צורך להשתמש ב-Breadth First Search אשר מוצא את הפתרונות על פי "טיבם" – מספר צעדים למטרה קטן ביותר, כיוון שאין כאלו, ובנוסף אנו מעוניינים



בכל הפתרונות – כך שחבל על כמות הזכרון והזמן שהחיפוש צורך – מעריכי בעומק הפתרון. שימוש ב-Depth First Search יצרוך זכרון לינארי בעומק הפתרון (כמספר החלקים) וזמן מעריכי – עדיין מיוצרים כל הפתרונות האפשריים – אלא שכיוון שהבעיה היא סיפוק אילוצים אין סיכון לתת-עץ אינסופי.<sup>4</sup>

ראוי לציין, כי ניתן לשפר את שיטת ה-Back Track הנ"ל, ע"י שימוש בטכניקות מגוונות (intelligent backtracking). להבדיל מהפתרון הנאיבי, בו השמת החלקים היא ע"פ סדר מסוים (מלמעלה למטה, ומשמאל לימין), ניתן לחשוב על השמת חלק במשבצת פנויה, בה מספר החלקים הניתנים לשיבוץ הינו מינימלי. כך נבחר רק הצעד עם מספר התת ענפים המינימלי להמשך, ומרחב החיפוש קטן. זה אינו Best First Search, עדיין מתבצעת סריקה של כל האפשרויות לפתרון, אלא שבסדר שונה ותוך חסכון בחיפושים "מיותרים". Best First Search מתחשב במרחק הפתרון ו"אורך" הצעד<sup>5</sup> – Breadth הוא מקרה פרטי כאשר "אורך" הצעד הוא תמיד 1. כאן, אופי הבעיה מאפשר גיזום הענפים מידי ללא צורך בסריקתם. טכניקה זו (Size Heuristics בהמשך) קרויה "סידור משתנים" – בחירת המשתנה לשיבוץ (משבצת בפזל) עם מספר אפשרויות ההשמה המינימלי. כאמור, מפאת אופי הבעיה אין משמעות כאן ל"סידור".

טכניקה נוספת הינה Back-Jumping – לא חוזרים צעד אחד לאחור ומבצעים את ההשמה שוב (עם חלק אחר), אלא חוזרים מספר צעדים לאחור, עד להשמה שהיא המקור לצורך לחזור לאחור. לדוגמה, אם בוצעו ההשמות  $x, y$  וכעת אף  $z$  אינו מסתדר עם  $x$ , אין צורך לשכן את  $y$  מחדש, שכן עדיין  $z$  לא יסתדר – ניתן **לקפוץ** לאחור ולשכן את  $x$  מחדש. ניתן גם לבצע "בדיקה קדימה" כדי לא לבצע השמה  $x$  שלא משאירה אף  $z$  אפשרי.

Back-Jumping לא נבדק, אך Forward Checking מומש חלקית בעזרת ה-Stranded Heuristic, אשר בודק אם לא נותרת משבצת ללא סיכוי לשיבוץ (אם היא חלק מ"א"י קטן מדי).

<sup>4</sup> ע"י Depth First Iterative Search ניתן להימנע ממקרה כזה, זהו ביצוע DFS לעומק 1, לאחר מכן לעומק 2 וכן הלאה – מעין שילוב של DFS ו-BFS.

<sup>5</sup> לכן BFS דורש הערכה לגבי כדאיות הצעד הבא, בפזל 15 לדוגמה, הערכה כזו יכולה להיות סכום מרחקי מנהטן מהפתרון.



## תיאור הבעיה

יהי  $L$  שריג הנוצר מתבנית החוזרת על עצמה. לחלק בתבנית החוזר על עצמו נקרא מעתה תא ולמקבילו בגרף הדואלי לשריג, צמת.

התבנית הפשוטה מורכבת כאמור מצומת  $v$  יחיד, ומהקשתות המכוונות היוצאות ממנו. מספר הקשתות הוא  $\text{degree}(v)$ , ונסמן אותן בתוויות  $0..d(v)-1$  לשם נוחיות.

$L$  מוגדר ע"י אופן החיבור של  $v$  אחד למשנהו. אם התבנית מורכבת מצומת יחיד, אז הגדרת אופן החיבור שקולה להגדרת השידוך בין הקשתות.

לדוגמה,  $L$  ישר מוגדר ע"י צומת  $v$  המתחבר לשכנו בקשת ימין ושמאל. השידוך הוא בין קשת ימין של  $v$  לקשת שמאל של שכנו של  $v$ . בנוסף, מוגדרת טרנספורמציה על צומת  $v$ , דהיינו תמורה על תוויות הקשתות. לדוגמה, בישר הנ"ל, היפוך קשת ימין לקשת שמאל, או  $t(0|1) = (1|0)$ .

דוגמאות מורכבות של כל השריגים שנבחנו, להלן בנספח שריגים.

פזל שריג  $P$  (להלן פזל) הוא תת-גרף סופי של השריג  $L$ . בגבולותיו, קשתות ללא שידוך, נקרא להן קשתות "מתנדדות".

לדוגמה, חלק משריג קו ישר וגבולותיו:



חלק בפזל הוא גם תת-גרף סופי של  $L$ .

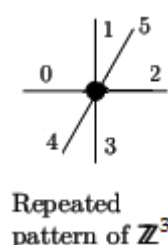
פתרון לפזל הוא כיסוי  $P$  באוסף חלקיו, כך ש-

- כל הצמתים ב- $P$  מכוסים ע"י צמתי החלקים,
- תוויות הקשתות של החלקים תואמות לתוויות ב- $P$  (מלבד הקשתות בגבולות).

לפתרון כזה נקרא מעתה שיכון.

גרסאות של הפזל מוגדרות מקיום חלקים זהים, אוסף הטרנספורמציות המותרות על החלקים וכו'. לדוגמה, סיבוב חלק, והיפוכו (גם בתלת-מימד, על אף שלא ניתן לבצע פעולה זו בעולם הפיזי שלנו<sup>6</sup>).

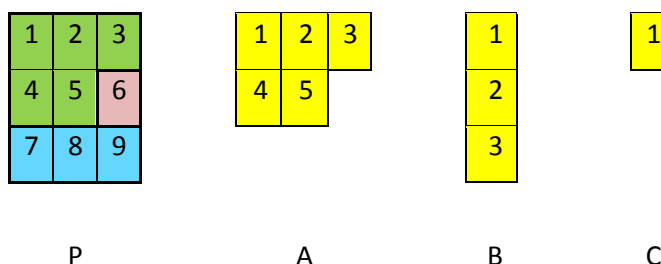
בחיפוש פתרון, נתעניין בהכרעה, האם קיים פתרון או לא (מציאת שיכון אקראי) וכן, במספר הפתרונות הקיימים.



לדוגמה: שריג אורתוגוני תלת-מימדי. התא- צמת בעל 4 שכנים-קשתות: נסמן West-East, 0-West, 1-North, 2-East, 3-South, 4-Front, 5-Back RC = Rotate סיבוב North-South וכן Back-Front. טרנספורמציות אפשריות הן סיבוב  $(RC_x(0|1|2|3|4|5) = (1|2|3|0|4|5))$  Clockwise ב-3 הצירים (לדוגמה  $(RC_x(0|1|2|3|4|5) = (1|2|3|0|4|5))$ ), וכן היפוך מראה  $t(0|1|2|3|4|5) = (2|1|0|3|4|5)$  flip East-West כל אוריינטציה אפשרית של חלק, מתקבלת ע"י הרכבות RC ו-flip חוזרות ונשנות על כל התאים בחלק.

- איור 1 -

פזל המטרה P הינו ריבוע  $3 \times 3$ , החלקים A, B, C. ופתרון אפשרי:



- איור 2 -

שיכון הצמתיים בפתרון זה: חלק A שוכן על תאים 1-5 של P, חלק B על 7-9, חלק C על 6.

חלק משיכון הקשתות (בפזל P 12 קשתות, או ליתר דיוק, 24 חצאי קשתות): בחלק B, הקשת North/South בין תא 1 ל-2 שוכנת על קשת East/West ב-P, בין תא 7 ל-8.

הפתרון ניתן לייצוג (בלתי תלוי בסוג השריג) בהתאם לסדר התאים בפזל:  $A1A2A3A4A5C1B1B2B3$ , או, ע"פ סדר החלקים והאוריינטציות שלהם בפתרון:  $A1B2C1$  כאשר כאן ה-1 מייצג את האוריינטציה הראשונה מתוך כל הקיימות (לחלק B קיימות 2 אוריינטציות, זו המצויינת, וסיבוב RC יחיד; סיבוב RC נוסף יביא לחלק השקול ל-B1 שכן לסדר התאים המתקבל אין למעשה משמעות).

<sup>6</sup> בדו-מימד, ניתן להפוך (למעשה לשקף) את החלק Z ולקבל את ולקבל את ניתן לדמות את הפעולה להיפוך פיזי של ממש. לעומת זאת, בתלת-מימד, אין דרך לבצע היפוך כזה, שכן הדבר דורש "יציאה" אל המרחב ה-4 מימדי.

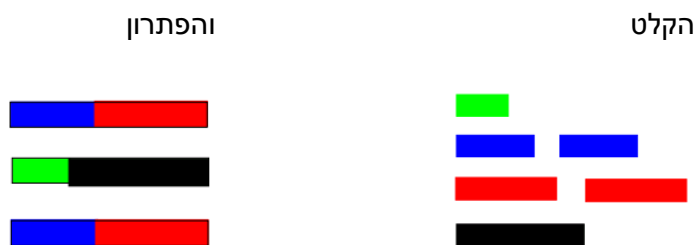
## שלמות ב-NP

מחלקת הבעיות P כוללת את הבעיות שניתנות להכרעה בזמן (P) פולינומי יחסית לגודל הקלט. אלו בעיות "קלות"; המחלקה NP כוללת בעיות שפתרון ניתן לבדיקה בזמן פולינומי, אלו בעיות "קשות". הן ניתנות לפתרון במחשב לא דטרמיניסטי (ומכאן השם Non-deterministic Polynomial) כיוון שמחשב כזה יכול ל"נחש" את הפתרון ואימות הפתרון הוא כאמור פולינומי. כיוון שהבעיה הינה שיכון, או Exact Cover, ברור שהיא בעיה קשה (שייכת ל-NP). שכן בכל אלגוריתם יש לנסות את כל האפשרויות ומספרן אינו פולינומי. השיפורים יהיו ברמת צמצום מספר ה-trial and error של נסיונות השיכון.

נוכיח כי המקרה הפשוט של שריג אורתוגונלי דו-מימדי שייך למחלקת הבעיות השלמות ב-NP. זו תת-קבוצה של בעיות ב-NP אשר ניתן לבצע רדוקציה אליהן בזמן פולינומי מכל בעיה ב-NP. אלו בעיות קשות "מאוד". אם ניתן לפתור בעיה שכזו בזמן פולינומי, הרי שניתן לפתור את כל בעיות NP בזמן כזה. עבור ההוכחה, נותר למצוא רדוקציה מבעיה השייכת למחלקת ה-NP-complete. לשם כך, נתבסס על רדוקציה מבעיית אריזת התרמילים (bin packing) [GJ79].<sup>7</sup>

בבעיה זו, בקבוצה E קיימים n איברים, בגדלים  $s_1, s_2, \dots, s_n$ . יש להכריע האם ניתן לחלק את E ל-k תת-קבוצות לכל היותר, כך שסכום האיברים המקסימלי בכל תת-קבוצה הוא m לכל היותר.

לדוגמה: האלמנטים  $E = \{2, 3, 3, 4, 4, 5\}$ ,  $k = 3$ ,  $m = 7$ :

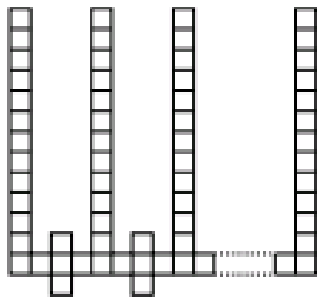


מתיאור הבעיה נובע כי  $\sum s_i \leq km$ , אחרת התשובה שלילית.

נציג את הבעיה כרדוקציה לפזל מטרה כמתואר באיור. החלקים יהיו:

- בפזל n "מקלות" עם אורכים  $s_1, s_2, \dots, s_n$  ורוחב 1.

<sup>7</sup> שרשרת הרדוקציות עבור בעייה זו היא Bin packing  $\rightarrow$  Partition  $\rightarrow$  Vertex Cover  $\rightarrow$  Subset Sum.



- איור 3 -

- בין ה"מקלות" מחברים  $k-1$  עותקים של הפנטומינ "X" (ליצירת פזל קשיר).



-  $\sum s_i$  -  $km$  ריבועים בגודל  $1 \times 1$  (לכיסוי מושלם של הפזל)

גודל הפזל הוא  $km + 5(k-1)$ . סדר גודל  $\Theta(km)$

מספר החלקים הוא  $\sum s_i + k - 1 + n$ . ביטוי זה חסום מלמעלה ע"י  $\Theta(km + n) = k - 1 + (m+1)n$  ושווה למספר החלקים בפזל. בבעיית התרמילים המקורית גודל הזכרון הנדרש הוא  $\Theta(n + \log(km))$ . נראה כי גודל הפזל והזמן לבנייתו הינם פולינומיים ב- $n, m, k$ , אך למעשה גודל הזכרון וזמן הרדוקציה לבניית הפזל הם  $\Theta(n + \log(km))$  תודות לצורה החוזרת – הפנטומינ "X"; את הפזל ניתן לאחסן "מכווץ" בגרסת RLE (Run Length Encoding).

אם כן, בזמן לינארי בגודל הפזל, ניתן לאמת מועמד לפתרון. וכן, אימות כי פתרון לבעיית האריזה אכן פותר גם את הפזל המתואר ולהיפך. חלקי הפנטומינ "X" יוצרים קשירות, והריבועים  $1 \times 1$  מבטיחים כי כל הפזל מכוסה.

ב-[KPS08] סקירה מרתקת של פזלים נוספים (לאו דווקא ריצוף) וסיווגם למשפחת הסיבוכיות שלהן. לדוגמה, טטריס, שולה מוקשים ומחשבת – אף הם שייכים למחלקת NP-complete.



## האלגוריתמים - ייצוג

כמתואר למעלה, פזל המטרה והחלקים מיוצגים ע"י רשימת צמתים וחצאי קשתות המחוברות לכל צמת. לדוגמה: החלק B הנ"ל מורכב מ-3 צמתים. צמת 1 עם קשת South אל צמת 2. צמת 2 עם קשת North אל צמת 1, ועם קשת South אל צמת 3, וכן הלאה.

בנספח מדריך למשתמש דוגמה לייצוג בתוכנה (כתצורה הניתנת לשימוש מחוץ לקוד).

אחד מהצמתים בכל חלק יסומן כ"מוצא" החלק. לדוגמה, בשריג אורתוגונלי דו-מימדי, המוצא יכול להיות החלק השמאלי-עליון ביותר. באופן דומה, יסומן גם "מוצא" פזל המטרה.

כיוון שלכל קשת סימון, הרי שהקישור בין הצמתים משרה עליהם יחס סדר, בהינתן כי לכל קשת משקל כלשהו. נחליט שרירותית כי המוצא הינו הצומת ה"קטן ביותר" ע"פ סדר זה.

לדוגמה, בשריג הריבועי הדו-מימדי, נחליט כי המשקל הינו אינדקס הסדר הלקסיקוגרפי, לדוגמה משקל מירבי עבור ציר  $x$  ומינימלי עבור ציר  $y$  (משווים את קואורדינטת  $x$  ואם יש שוויון, את קואורדינטת  $y$ ).<sup>8</sup>

חישוב נקודות ה"מוצא" מתבצע ע"י סריקה לעומק (או לרוחב), בכל טיול על קשת מעודכן הסימון על צומת היעד בהתאם למשקל הקשת. בסיום הסריקה נבחר הצומת עם הסימון המינימלי.

---

<sup>8</sup> לנוחיות המימוש, המשקל לא היה כמתואר, אלא בדומה למיקום הספרות: לקשתות הסימטריות North|South משקל 10, ולקשתות East|West משקל 1. אזי, בחלק A הנ"ל, הסימון לצמתים 1..5 יהא בהתאמה 1,2,3,11,12 (שים לב, שדוגמה זו אינה תקפה עבור פזל מטרה עם רוחב או גובה גדול מ-10, שכן אז הסימון אינו חד-חד-ערכי).

1	2	3	4	5	6	7	8	9	10	11
11										



## אוריינטציות

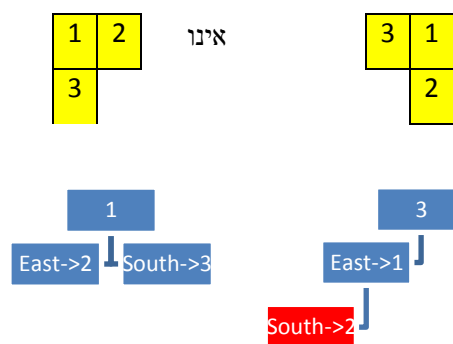
האוריינטציות האפשריות מתקבלות ע"י הרכבת הטרנספורמציות האפשריות ובכל סדר אפשרי. בשלב האתחולי, על כל חלק, החל מצומת המוצא, מחושבות האוריינטציות האפשריות, ובסריקה לעומק.

ראשית, מחושב מספר האוריינטציות הנגזרות מכל סוג טרנספורמציה. לדוגמה, בשריג שלנו, יש 4 טרנספורמציות עבור הסוג Rotate Clockwise (בכל השריגים שנבחנו, מתקיים שביצוע חוזר ונשנה של טרנספורמציה הביא לחזרה אל האוריינטציה ההתחלתית).<sup>9</sup>

לאחר מכן, מחושבות 4 אוריינטציות אלו, כפול 2 האוריינטציות האפשריות עבור סוג הטרנספורמציה השני האפשרי: היפוך East-West.

למעשה, זו דרך נאיבית ללא שימוש בידע על החלק (לדוגמה, בשריג אורתוגונלי דו-מימדי ( $d=2$ ) מספר האוריינטציות המקסימלי אינו  $4^d$  אלא רק  $d*2=4$ , ואם נאפשר שיקוף או הפיכת החלק, כלומר סיבוב במימד השלישי, כפליים; ולא  $4 \times 4 = 16$ ).<sup>10</sup>

כיוון שקיימים חלקים עם סימטריות, השלב המסכם הינו הורדת הכפילויות. באמצעות השוואת סריקה לעומק של האוריינטציה הנבחנת, מול אלו שחושבו כבר לפניה. סריקה לעומק נקראת "זהה" אם סדר ההליכה בסריקה על האוריינטציה הנבחנת זהה לסדר על האוריינטציה השנייה.



- איור 4 -

<sup>9</sup> למעשה, חשוב גם סדר הפעלת הטרנספורמציות על סוגיהן (לדוגמה, האם קודם  $RC^k$  ואח"כ  $Flip^j$  כאשר  $j, k$  בהתאם לאוריינטציות הדרושות בכל סוג טרנספורמציה) כדי לחזור לאוריינטציה המקורית (ב- $d=2$ ,  $j=2$ ,  $k=4$ ), כאמור, לא נמצא שריג בו סדר זה חשוב.

<sup>10</sup> וכן עבור  $d=3$  מספר האוריינטציות המקסימלי אינו  $4^d$  אלא רק  $d*2^d=24$ , ואם נאפשר שיקוף (הפעם סיבוב במימד הרביעי) נכפיל המספר ל-48; ולא  $4 \times 4 \times 4 = 64$ .

בפועל, צומת הכניסה (להבדיל מנקודת המוצא) יכול להיות כל צומת בחלק, ולא רק צומת המוצא (דהיינו עבור כל צומת בחלק הנבחן, בודקים זהות פלט סריקה לעומק, מול כל צומת כניסה בכל אוריינטציה נבחנת). לדוגמה, חלק B לאחר סיבוב אחד  $BxRC^1$  זהה בצורתו ל- $BxRC^3$  ( $\begin{bmatrix} 1 & 2 & 3 \\ 3 & 2 & 1 \end{bmatrix}$ ). כדי לגלות זאת, יש להשוות סריקה לעומק החל מצומת 1, מול סריקה לעומת החל מצומת 3.

כיוון שכל האוריינטציות שנמצאו ישתתפו בחיפוש הפתרונות, יש לחשב גם עבורן את צומת המוצא שלהן.

## החלק הייחודי

חלק מהפתרונות הינם כפולים, ומהווים סימטריות של פתרונות ייחודיים. ניתן להימנע מבדיקת הכפילויות (כפי שתואר בסעיף "אוריינטציות" להלן) בשלב הריצה, ולהוסיף שלב נוסף לאתחול: מציאת החלק הייחודי.

יהא  $S$  פתרון של הפזל, אם לפזל עצמו קיימות אוריינטציות זהות (נקרא להן סימטריות) המתקבלות ע"י הרכבה כלשהי  $T$  של טרנספורמציות, אזי גם  $T(S)$  פתרון לפזל. הדבר שקול לביצוע  $T$  על כל חלק ב- $S$ . כלומר, מציאת כל הסימטריות של הפזל, והורדתן מחלק מסוים, תבטיח שלא ימצא גם  $T(S)$  תוך כדי החיפוש. חלק מסוים זה יהיה בעל מספר האוריינטציות הרב ביותר (מינימום סימטריות) – נסמנו כ"חלק הייחודי".

לדוגמה, לפזל  $P$  באיור 2 (עמוד 14) יש 8 סימטריות, לחלק  $A$  גם 8 אוריינטציות,  $B$  רק עם 2 ו- $C$  ללא אוריינטציות. כדי להימנע ממציאת פתרונות סימטריים, נשתמש רק באוריינטציה הנתונה של  $A$  (באיור 2) ונמחק את האוריינטציות המתקבלות מ- $T$  כלשהו המביא את  $P$  לסימטריה שלו.

1	2
3	4

1	2
3	4

3	1
4	2

4	3
2	1

2	4
1	3

|  | | | | | | | | | |

1	2
3	4

1	3
2	4

3	4
1	2

4	2
3	1

הנה 8 האוריינטציות האפשריות (מימין) עבור פזל הדומה ל- $P$  (רק קטן יותר). בשורה הראשונה 4 סיבובים בכיוון השעון, ובנוסף, בשורה השנייה, על כל אחד מהסיבובים, ניתן לבצע שיקוף אנכי.

### - איור 5 -

<table><tr><td>1</td><td>2</td><td>3</td></tr><tr><td>4</td><td>5</td><td>6</td></tr></table>	1	2	3	4	5	6	<table><tr><td>1</td><td>2</td><td>3</td></tr><tr><td>4</td><td>5</td><td>6</td></tr></table>	1	2	3	4	5	6	<table><tr><td>6</td><td>5</td><td>4</td></tr><tr><td>3</td><td>2</td><td>1</td></tr></table>	6	5	4	3	2	1
1	2	3																		
4	5	6																		
1	2	3																		
4	5	6																		
6	5	4																		
3	2	1																		
	<table><tr><td>3</td><td>2</td><td>1</td></tr><tr><td>6</td><td>5</td><td>4</td></tr></table>	3	2	1	6	5	4	<table><tr><td>4</td><td>5</td><td>6</td></tr><tr><td>1</td><td>2</td><td>3</td></tr></table>	4	5	6	1	2	3						
3	2	1																		
6	5	4																		
4	5	6																		
1	2	3																		

בפזל זה לעומת זאת, כיוון שאין סימטריה בציר  $x$  וציר  $y$ , יהיו רק 4 אוריינטציות.

כמות האוריינטציות של הפזל משפיעה על כמות הפתרונות הנוספים שיש, על כל פתרון ייחודי. בדוגמה השנייה להלן, מספר הפתרונות הכללי הינו מקסימום פי 4 (תלוי בחלקים) מכמות הפתרונות שהאלגוריתם מוצא (אלא אם ביקשנו מראש למצוא את כולם ללא הבחנה בכפילויות).

לדוגמה, בפזל  $2 \times 2$ , אם החלקים היו תא בודד ■ וחלק דמוי האות L ■■■, היינו מקבלים פתרון אחד ייחודי, ו-3 נוספים, סה"כ 4. למרות שלשריג 8 אוריינטציות, לחלק L "רק" 4 ולפיו נקבעים מספר הפתרונות הכולל. זהו גורם הסימטריה.

## הפתרונות הייחודיים

כאשר נמצא פתרון, יש לסווגו: האם מדובר בפתרון סימטרי או ייחודי. דהיינו הופיע בעבר או בסימטריה, או כתמורה זהה (רלוונטי אם יש M חלקים זהים, ואז יהיו M! פתרונות זהים, פזל Slothouber–Graatsma לדוגמה<sup>11</sup>). אם כן – מדלגים על מנייה זו כפתרון ייחודי.

פתרון סימטרי אפשרי כאשר החלק הייחודי מופיע בפתרון מושרה (= סימטריה של פתרון ייחודי) במיקום מסוים, ובפתרון אחר, הוא מופיע שוב במיקום זה (וכמובן באותה אוריינטציה). במצב זה, הפתרון שנמצא נשמר ברשימה חדשה – רשימה של פתרונות עם פוטנציאל לכפילות. דהיינו, כאשר החלק הנוכחי לשיכון הינו הייחודי, בודקים היכן יהיה מיקומו בפתרון המושרה (implied בשילוב עם unique In Implied – ראה להלן) ומאפשרים רק לאחד מפתרונות אלו להיחשב כייחודי<sup>12</sup> (למעשה מבצעים גיזום של עץ החיפוש). כפועל יוצא, חיפוש הכפילות אינו בכל רשימת הפתרונות בכל פעם, אלא בחלק קטן מאוד ממנה (כמובן שניתן לחפש גם ייצוג hash של הפתרון) בהתאם למיקום החלק הייחודי (יש כאן מיפוי מהמיקום המדובר לרשימת הפתרונות המתאימה).

דוגמה: בשריג אורתוגונלי תלת-מימדי  $5 \times 4 \times 3$  וחלקי ה-pentominoes (אחד מהמקרים ה"קשים" ביותר שנבחן – 3,940 פתרונות ייחודיים), גודל הרשימה הינו 398, ומתוך 60 התאים בפזל ו-6 של החלק הייחודי F, רק 16 מקרים הכילו כפילויות, דהיינו גודל הרשימה בממוצע היה 24.

דרך חלופית אפשרית לבדיקת כפילות הפתרון הינה חסימת המיקום המושרה של החלק הייחודי, כך שהבדיקה תהיה כבר בשלב הבנייה (ברגע שמשכנים את החלק הייחודי) ולא במהלכו (נמצא שיכון חוקי בפוטנציה), ולא תהיה השוואת פתרון שלם אלא מיקום בלבד. בפועל התברר שיש פתרונות שונים על אף שהחלק הייחודי באחד מהם, נמצא במיקום מושרה בפתרון השני.

מבני הנתונים המאפשרים ביצוע הבדיקה הינם:

○ Implied – כנזכר בסעיף אוריינטציות, הסריקה לעומק מסמנת כל צומת בסריקה. הסימון הינו מזהה התא המקביל מהחלק שאיתו מתבצעת ההשוואה. באתחול, מזהה זה הינו אינדקס התא ברשימת התאים = מיקומו הסידורי (ע"פ הסדר השרירותי שניתן בהגדרת החלק), רק לאחר מכן הופך המזהה

<sup>11</sup> בפזל ריבועי תלת-מימדי זה בגודל  $3 \times 3 \times 3$  קיימים 6 חלקים בגודל  $1 \times 2 \times 2$  ו-3 קוביות. לפזל פתרון ייחודי אחד בלבד. ניתן במקום לבדוק זהות התמורות, לחסום השמה של עותק זהה של חלק במיקום בו כבר בוצעה השמה של חלק זה.

<sup>12</sup> שרירותית, ע"י דרישה שהמיקום בנסיון הפתרון הנוכחי, יהיה לפני המיקום בפתרון המושרה. כאמור, אם המיקום זהה מחפשים ברשימה כפילות.

להיות בהתאם לסדר הגדרת הקשתות. על-כן, כאשר נמצאה כפילות בשלב הקודם, תיווצר כניסה בטבלה בשם implied, שתכיל את הסימונים הנ"ל ע"פ הסדר בו בוצע הביקור (כמובן, רק אם לא נמצאה כפילות זו קודם).

לדוגמה, אם נפרוש את פזל השריג האורתוגונלי הדו-מימדי  $2 \times 2$  באיור 5 לרשימה, נקבל 1 2 3 4. ה- implied יהיו פרישות האוריינטציות, 4 3 2 1, 3 1 4 2, וכן הלאה. כך בהינתן פתרון ייחודי ניתן מידית לספק גם את כל הפתרונות הסימטריים שלו.

○ Unique In Implied – עבור כל אוריינטציה של החלק הייחודי, מציאה באיזה פתרון כפול הוא יופיע באוריינטציה זו, כפי שהיא בפתרון הייחודי שנמצא.

I X F F			לדוגמה, באיור אחד מהפתרונות
I F F U	X X X L		הייחודיים בפזל שריג אורתוגונלי תלת-
I W F U	Y P P U	Y X N L	מימדי $5 \times 4 \times 3$ עם חלקי ה-pentomino:
I W W U	T P P P	Y N N L	קומה אחר קומה.
I Z W W	T T T U	Y N V L	
	T Z Z Z	Y N V L	
		V V V Z	

### - איור 6 -

Y X N L			בפתרון הכפול השישי (מתוך 8
Y N N L	X X X L		הסימטריות) שנגזר ממנו (ע"י שיקוף
Y N V L	Y P P U	I X F F	"קומה" ראשונה ואחרונה – הקומות
Y N V L	T P P P	I F F U	מתחלפות) יופיע החלק הייחודי F באותה
V V V Z	T T T U	I W F U	אוריינטציה בדיוק (רק בקומה שונה)
	T Z Z Z	I W W U	
		I Z W W	

וכך יהיה בכל פתרון שבו החלק הייחודי יהיה באוריינטציה זו.

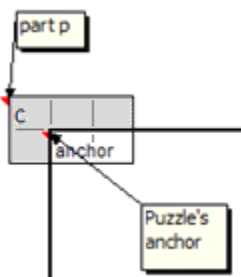


## פתרון הפזל

### נסיגה לאחור

השיטה הראשונה והטריויאלית לפתרון, היא נסיגה לאחור עם שימוש בייצוג הנ"ל במדויק. האלגוריתם מנסה לבצע שיכון (או התאמה) בכל שלב, בין גרף של חלק (וכל האוריינטציות שלו) מהחלקים ה"פנויים", לגרף הפזל. בנסיון זה, מבצעים מיפוי בין הצמתים של החלק הנוכחי, לצמתי הפזל, כאשר נקודת המוצא של הפזל, הינה הצומת עם הסימון הנמוך ביותר, או הצומת השמאלי-עליון שעדיין לא בוצע לו התאמה (צומת "ריק", להלן פנוי). במיפוי זה, על כל (חצאי) הקשתות של החלק להתאים לתת-גרף בפזל שהתמלא.

בסעיף "תיאור הבעיה" דוגמה לשיכון.



- איור 7 -

שיכון החלקים נעשה ע"פ סדר מסוים. בתחילה, כל החלקים פנויים, וכן כל הצמתים בפזל פנויים. עוגן הפזל הוא המוצא ההתחלתי שלו (כאמור, התא השמאלי בשורה העליונה). הצומת המתאים למוצא זה, יתמלא ע"י תא  $c$  כלשהו של חלק שרירותי  $p$  (אוריינטציה כלשהי אפשרית שלו). קל לראות כי  $c$  מהווה מוצא של  $p$ , אחרת המוצא של  $p$  יתפס ע"י תא מהפזל שאינו המוצא שלו, בסתירה למינימליות נקודת המוצא. על כן שיכון חלק בפזל מתבצע ע"י זיהוי העוגן כנקודת המוצא של החלק, וסריקת גרף החלק, בד בבד עם סריקת גרף הפזל כולו. הסריקה כאמור (לעומק, אפשרי גם לרוחב) היא בהתאם לתוויות הקשתות. תוצאת הסריקה היא סימון תת-גרף בפזל הנתפס ע"י חלק זה.

✓ ניתן גם לבצע שיכון של  $p$  החל מכל תא שלו, אך כאמור אין לכך טעם. התחלה החל מעוגן החלק משפרת בפועל כ-45% מזמן החישוב.<sup>13</sup>

אם במהלך הסריקה, תא בגרף הפזל מסומן כבר, או שנדרש מעבר בלתי אפשרי על הגרף (החלק יוצא מגבולות הפזל) הסריקה נכשלת והסימונים נמחקים.

<sup>13</sup> תאורטית 40%. לדוגמה, ב-pentominoes (5 תאים), בממוצע בחצי מהמקרים נתחיל בעוגן, לעומת חישוב מראש שלו.  $40\% = 1 / (5/2) -$  דהיינו היחס בין בחירה "מראש" בעוגן, לעומת מעבר על 2.5 חלקים אקראיים בממוצע.

✓ ניתן לשמור את היתכנות המעבר באופן סטטי, לכל חלק + אוריינטציה + תא עוגן בפזל, נבדוק אם השמתו בכלל אפשרית (בהנחה שהתת-גרף המתאים בפזל פנוי כולו) כדי להימנע מהשמה שתיכשל בוודאות. שימוש בטבלה זו מוביל לחסכון של עד 50% בזמן החישוב. לדוגמה, בדר"כ אין אפשרות שיבוץ חלק על גבולות הפזל. לכן, ככל שהפזל גדול, השיפור פוחת שכן התאים בגבול מהווים אחוז קטן יותר מסך התאים בפזל). בנוסף, מחיקת אוריינטציות אשר אין להן שום אפשרות שיכון (עקב גודל השריג).

כאשר הסריקה מצליחה, החלק  $p$  עובר לרשימת החלקים התפוסים, ועוגן הפזל "מתקדם" לתא הבא הפנוי עם המזהה שערכו מינימלי.

תהליך זה חוזר על עצמו (נבנה עץ חיפוש לעומק) עד שכל חלקי הפזל מסומנים (שקול לכך שלעוגן הפזל אין לאן להתקדם, או שכל החלקים תפוסים אם אין חלק מיותר – ראה דוגמת פזל stairs) כלומר נמצא פתרון, או שאף חלק פנוי (על כל האוריינטציות שלהם) אינו יכול להתאים למצב הפזל הנוכחי.

לדוגמה: באיור 2 בסעיף תיאור הבעיה, לחלק A יש 8 אוריינטציות (מתוכן נשאיר רק אחת כיוון שגם לפזל 8 אוריינטציות), ל-B 2 (ולא 4, כי החלק סימטרי אופקית ואנכית) ול-C רק 1. מספר התמורות האפשרי הינו  $1 * 2 * 1$  כפול מספר האפשרויות לסדר את החלקים  $= 3!$ . בכל אחת מ-12 האפשרויות, מפסיקים מיד כאשר מגלים שהיא שיכון לא אפשרי; לדוגמה, באוריינטציות הנ"ל, שיכון B, ואחריו C ואז A אינו אפשרי.

בכל מקרה, מבצעים Back Track: הסרה מהפזל של החלק האחרון שעבר התאמה (שחזור סימון תאים כפנויים בפזל, ושחזור החלק להיות פנוי) על-מנת להמשיך לחפש אחר פתרונות נוספים (אם מעוניינים בכך). החיפוש ממשיך באוריינטציה הבאה של החלק  $p$ , או אם לא נשארו אוריינטציות שלו, לחלק הפנוי הבא.

ראוי לציין כי הקשתות של החלקים אינן תורמות לסריקה על-גבי גרף הפזל: בשיכון של חלק, מתקדמים לפי הקשתות שלו גם על הפזל, ולאחר השיכון המוצלח, מתקדמים רק על גבי גרף הפזל, לעוגן הבא, ללא אפשרות התקדמות אל עבר החלק הבא (הוא טרם ידוע).

להלן פסידו קוד לסיכום התהליך:

## שיכון

אם לא נותרו חלקים לשיכון (או הפזל מלא, והפתרון חוקי וכו'):

פלוט את הפזל ואת וקטור החלקים בפתרון;

עבור כל חלק שנותר לשיכון,

ועבור כל אוריינטציה של החלק,

אם החלק ניתן לשיכון בעוגן הפזל:

בצע את שיכון החלק וקדם את עוגן הפזל;

הוצא את החלק מרשימת החלקים שנותרו לשיכון;

הוסף את החלק (ואת אינדקס האוריינטציה שלו) לסוף וקטור החלקים בפתרון;

**קרא ל-שיכון;**

בצע Back Track:

הסרה של החלק מהפזל, וקידום לאחור של עוגן הפזל;

החזרת החלק לרשימת החלקים שנותרו לשיכון;

הורדת החלק מסוף וקטור החלקים בפתרון.



## Matrix Cover

המנוע השני – טכניקת ה-DLX - Dancing Links - לפתרון בעיית כיסוי מטריצה - מתבסס על מימוש open source מבית Apache של Hadoop. מעל המנוע הולבשה מעטפת הייצוג שהוסברה לעיל. מנוע זה משפר מאוד את זמני הריצה ומבוסס על מאמר של Knuth [Kn00].

בקצרה, פותרים כאן בעיית NP שלמה - Matrix Cover. בקלט נתונה מטריצה בינרית (עם ערכי 0/1) ומחפשים את אוסף השורות כך שכל עמודה מכילה בדיוק פעם אחת 1. זהו השיכון. הקלט מתקבל ע"י התייחסות לעמודות כאל תא בפזל (תא "תפוס"=1). לקלט מוסיפים עמודות כמספר החלקים, בהן יהיה 1 ב-id (הפעם מיקום סידורי) של החלק שביצע את השיכון, כך מובטח שכל חלק ייכלל בשיכון פעם אחת בלבד. השורות הינן כל השיכונים האפשריים של כל חלק באוריינטציות השונות שלו.

לדוגמה, השיכון לעיל הוא אוסף השורות להלן, מתוך כל האפשרויות:

1 1 1 1 1 0 0 0 0 A 1 0 0

0 0 0 0 0 0 1 1 1 B 0 1 0

0 0 0 0 0 1 0 0 0 C 0 0 1

### - איור 8 -

באוסף השורות האפשריות, יופיע החלק C 9 פעמים (כמספר התאים בפזל). B יכול להיות אופקי או אנכי, על כן הוא יופיע 6 פעמים (הפזל בגודל 3x3). וכדומה עם A.

האלגוריתם דומה לאלגוריתם הטריטוריאלי לפתרון הבעיה, אלא שבמקום להשתמש במצביעים לתיאור רשימות ה-1-ים במטריצה, משתמשים במערך (ואז אין מחיקה אמיתית של ה-data אלא רק shift עם האינדקסים של התאים העוקב והקודם ברשימה). בנסיגה לאחור, מורידים את החזרה לאחור ב-state, ע"י  $next[prev[x]] \leftarrow next[x]$  וכן  $prev[next[x]] \leftarrow prev[x]$ . ניתן כעת לבצע undo מידית ע"י  $next[prev[x]] \leftarrow x$  וכן  $prev[next[x]] \leftarrow x$ . שימוש בצעד זה משפר מאוד את הביצועים (לדוגמה, "נחסכת" הסרת השיכון האחרון שבוצע).

לדוגמה, אם האיבר השני ברשימה המעגלית ( $x = 20$ ) נמחק, הרי שביצוע השלבים לעיל "יחזיר" אותו.

Previous index	0	1	2 -> to be 1
X	10	20 -> to be deleted	30
Next index	2 -> to be 3	3	0
index	1	2	3

להלן פירוט הפסידו קוד העיקרי:

#### פתרון Matrix M Cover

אם  $M$  ריקה – החיפוש הושלם, הדפס את הפתרון, אחרת:

בחר עמודה  $c$

בחר שורה  $r$  שרירותית כך ש- $M[r,c] = 1$

הוסף את  $r$  לפתרון החלקי

לכל  $j$  כך ש- $M[r,j] = 1$ : (1) מחק עמודה  $j$  מ- $M$ ;

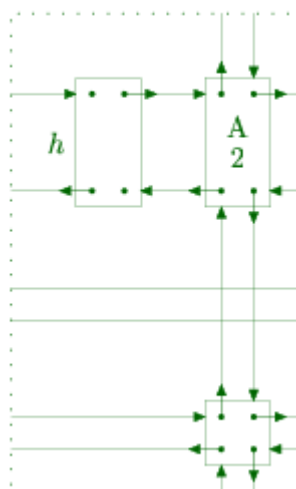
(2) לכל  $i$  כך ש- $M[i,j] = 1$ , מחק שורה  $i$ .

ושוב ברקורסיה על  $M$  הנוכחית

ולפירטי מימוש כלליים:

כל 1 במטריצה יוגדר כרשומה עם 5 שדות: קישורים לארבעת השכנים,

ושדה כותרת (מכיל לדוגמה id ומספר ה-1ים). בנוסף,  $h$  יהיה עוגן רשימת הכותרות.



- איור 9 -

solveDLX(level)

אם  $Right[h] = h$  (ריקה  $M$ ) החיפוש הושלם, הדפס את הפתרון, אחרת:

בחר עמודה  $c$  ע"י חוקיות מסוימת, לדוגמה, בדומה לנסיגה לאחור הטריויאלית, בחירת  $Right[h]$  (החלק השמאלי ביותר שאינו מכוסה)

קרא  $cover$  לעמודה  $c$  (סימון השיכון)

לכל  $r$  ברשימה  $r \leftarrow Down[c], Down[Down[c]]...$  (ועד ש- $c = r$ )

$r \leftarrow O[level]$  (הרשימה  $O$  תסייע בהדפסת הפתרון)

לכל  $j$  ברשימת  $Right$  של  $r$  (ועד ש- $j = r$ )

$cover$  לעמודת הכותרת של  $j$

$solveDLX(level+1)$

(ובחזרה,  $undo$  לשינויים)

$r \leftarrow O[level]$

$c \leftarrow C[r]$

לכל  $j$  ברשימת  $Left$  של  $r$  (ועד ש- $j = r$ )

$UnCover$  לעמודה הכותרת של  $j$

$UnCover$  לעמודה  $c$ .

לא ניכנס כאן לפרטים נוספים כמו אופן הדפסת הפתרון (בעזרת מערך  $O$ ).

לב האלגוריתם הוא השימוש בפעולת cover וביטולה ב-unCover:

Cover(c) (כיסוי, אין כאן מחיקה של ממש)

$$L[R[c]] \leftarrow L[c]$$

לכל  $i$  ברשימת ה-Down של  $c$  (ועד ש- $c = i$ )

לכל  $j$  ברשימת ה-Right של  $i$  (ועד ש- $j = i$ )

$$D[U[j]] \leftarrow D[j]; U[D[j]] \leftarrow U[j]$$

UnCover(c)

לכל  $i$  ברשימת Up עד ש- $c = i$

לכל  $j$  ברשימת Left של  $i$  ועד ש- $j = i$

בצע צעד "הריקוד":  $j \leftarrow U[D[j]]$  וכן  $j \leftarrow D[U[j]]$

$$R[L[c]] \leftarrow c \text{ וכן } L[R[c]] \leftarrow C$$

הקורא מופנה לתיאור שלם של השיטה אשר מופיע ב-[Kn00].



## אי בודד (Stranded)

ניתן לחתוך ענפים בעץ החיפוש, אם נוצר מצב בו החלק הקשיר הפנוי בגרף הפזל, ה"מתחיל" מהעוגן שלו, קטן מכל החלקים שעדיין פנויים, או, בהרחבה של הרעיון, חלק קשיר פנוי, שגודלו אינו מתאים לגודלי כל החלקים הפנויים (לדוגמה, אי עם 11 חלקים, כאשר נותרו 3 חלקים בגודל 5 תאים כל אחד).

שימוש בהיורסטיקה זו חוסך 35% מזמן החישוב. ההיורסטיקה המורחבת למעשה מאריכה את זמן החישוב (כיוון שמתבצעות סריקות איים גדולים עד כדי גודל הפזל).

חישוב גודל האי מתבצע באמצעות סריקה החל מהעוגן הנוכחי של הפזל, אך עם התקדמות רק לעבר תאים פנויים.

היוריסטיקה זו היא מימוש חלקי של טכניקת forward checking. מימוש מלא היה בודק את כל המשבצות – האם קיימת בכל אחת אפשרות שיבוץ – ולא רק את המשבצות בתוך ה"אי".

## תת-עץ מינימלי (Size)

Golomb and Baumert [GB65] הציעו לבחור בכל שלב של צמצום מרחב החיפוש את הבעיה ה"פחות גדולה" – כלומר שמספר הענפים בה הוא מינימלי. בשיטת כיסוי המטריצה, מדובר בבחירת העמודה לשיכון עם מספר ה-1-ים הקטן ביותר. מרחב הפתרונות כולו יעבור סריקה בסדר חכם יותר.

המימוש הוא די ישיר (יחסית לנסיגה לאחור הטריויאלית): בחירת העמודה c באלגוריתם solveDLX תהיה באמצעות שדה חדש Size ברשימת הכותרות, אשר יכיל את מספר ה-1-ים בעמודה. נותר לבחור בכל פעם את העמודה עם ה-Size המינימלי, ולעדכן כמובן בכל cover/ uncover את השדה

$$S[C[j]] \leftarrow S[C[j]] \pm 1$$

היוריסטיקה זו מבוססת על טכניקת variable ordering – בחירה בתת-ענף ה"מבטיח".



לאחר מקצה שיפורים נרחב, ובהם שימוש יעיל יותר במאפייני השפה (כמו רשימה מקושרת – גישה סדרתית, מול מערך - גישה ישירה לאלמנט), צומצם זמן החישוב של פתרון פזל ריבועי תלת-מימדי  $5 \times 4 \times 3$  עם 12 ה-pentominoes לכשעה במנוע הנסיגה לאחר (מ-3 יממות!<sup>14</sup>) ולכדקה במנוע Matrix cover by Dancing Links במחשב 2.2 GHz.

להלן שיפור משמעותי של המרכיב המשמעותי ביותר בזמן הריצה: נסיון השיכון בכלל, ובפרט יוקר ה-DFS:

- חישוב trips – שמירת מסלולי DFS. סריקה זו הינה רקורסיבית, ובנוסף בודקת בכל תא האם כבר ביקרנו בו. כדי לחסוך בתקורה הכרוכה בכך, נשמר סדר הסריקה בכל חלק, דהיינו מאיזה תא בחלק מתקדמים, ועל איזו קשת.

לדוגמה, בחלק A הנ"ל, בהנחה שסדר הקשתות מזרח, מערב, דרום וצפון, תוצאת הסריקה לעומק תהיה: מתא 1 מזרחה ל-2, מ-2 מזרחה ל-3, מ-2 דרומה ל-4, ומ-4 מערבה ל-5.

- שמירה מדויקת של ההשמה. בתוך המערך התלת-מימדי: חלק  $k$  אוריינטציה  $r$  תא בפזל  $c$ , נשמרת רשימה של התאים המשוכנים (אם השיכון אפשרי כמובן). באופן כזה אין טעם לבצע שיכון בכל התקדמות בעץ החיפוש אלא אם הוא אפשרי (על פזל פנוי לחלוטין), רק לאחר מכן מבוצע נסיון שיכון על פזל הנוכחי.

שיפורים אחרים מסומנים ב- ✓ בסעיף "נסיגה לאחור": חישוב עוגן החלק, ובדיקת שהשיכון לא נידון מראש לכשלון.

<sup>14</sup> בשל כך עבור אלגוריתם ה-Back Track (Dancing Links "מספיק" מהיר) מומשה גם גרסה לא רקורסיבית, המאפשרת שמירה אוטומטית או יזומה של מצב המחסנית ובכך לעצור את החיפוש הארוך ולהמשיכו מאוחר יותר. הדבר בוצע ע"י חיקוי הרקורסיה: בכל קריאה רקורסיבית, נדחף למחסנית המצב הנוכחי (פרמטרי כניסה, משתנים מקומיים וכתובת חזרה), וביציאה מהקריאה נשלף ושוחזר מצב זה. כך עד שאין יותר כתובות חזרה במחסנית.

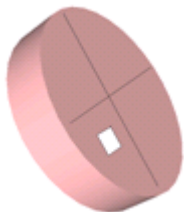


## מפתחות ומנעולים

לתוכנה נוספה אפשרות להלביש מסכת התאמות בין החלקים, בדומה לפזל ילדים רגיל. לכל (חצי) קשת של תא, תיתכן התאמה אחת או יותר של תאים. קיימות קבוצות מפתחות וקבוצות מנעולים, וגרף דו-צדדי ביניהם (בפזל ילדים קלאסי כדוגמה, לכל מנעול מפתח אחד בלבד). כמובן שיתכן מפתח "ג'וקר" שמתאים לכל מנעול, ושכלל אין חובה לקיום מנעול או מפתח בכל תא.

עבור שיטת הנסיגה לאחור, בזמן השיכון, נבדקת ההתאמה. כיוון שהחיפוש מתאים את החלקים בשכנות, מצטמצמים הנסיונות לשיכון. לעומת זאת, ב-Matrix Cover השיכון אינו סידרתי, ולמעשה מתקבלים כל הפתרונות כאילו לא היה אילוץ ההתאמות, ורק לאחר מציאת כל מועמד לפתרון, מתבצעת בדיקה המאשרת התאמתו לסכימת ההתאמות הנתונה<sup>15</sup>.

יתרה מזאת, שיטת Matrix Cover בתוספת היוריסטיקת גודל התת-עץ המינימלי לא מתאימה כלל כיוון שלא נבדקת ההשמה בזמן אמת, ובגלל צמצום התת-עצים לחיפוש רק ע"פ מספר הבנים שלהם, "מתפספסים" פתרונות.



הפזל שנבחר להדגמת שימוש במפתחות ומנעולים הינו גליל דיסקיות. כל דיסקית יכולה לכלול, בכל רבע מעגל שלה, מפתח באורך כלשהו או מנעול (חור). בדוגמה שלנו, היו תריסר דיסקיות, ואורך מפתח היה 1 או 2.



כיוון שהמנעול/ מפתח הוא תכונה של הקשת הקונקרטית של התא, על-מנת לייצג פזל זה, בוצע חיתוך של כל דיסקית לרבעים: התא אינו דיסקית, אלא רבע; וחלק תמיד מכיל 4 רביעיות משלימות. קיימות 4 קשתות לחיבור הרבעים (ו-4 קשתות סימטריות) וכן קשת "קדימה" ו"אחורה". כאמור, לכל קשת (כאן, רק קשת מסוג קדימה או אחורה) נוספה תכונה של מנעול (ערך 0) או מפתח (ערכים 1 או 2). בבדיקת ההתאמה נבדק אם קשת 0 מתאימה לקשת 1/2, במידה וכן, בוצע סימון כי רבע זה אינו פנוי יותר. יש לתת את הדעת כי מפתח באורך 2 "נעל" 2 רבעים סמוכים מאחד צדדיו.

עוד על דרכי ייצוג של פזלים מורכבים – בנספח שריגים ומקרים מיוחדים.

- איור 10 -

<sup>15</sup> כאמור, מספר הפתרונות חסום מלמעלה ע"י מכפלת מספר האוריינטציות של החלקים במספר האפשרויות לסדר את החלקים (בפזל תלת-מימד  $2 \times 5 \times 6$  עם חלקי ה-pentominoes – 536 מיליון אפשרויות). כיוון שמספר הפתרונות קטן מאוד – התקורה בבדיקת ההתאמה אינה מורגשת.



השיטות שתוארו מומשו בשפת Java על מחשב מרובה-ליבות ונבחנו במערכות ההפעלה חלונות ולינוקס.

מימוש המקביליות בין הליבות הוא סטטי, דהיינו התייחסות לפיצול מרחב החיפוש רק בהתחלה; להבדיל ממימוש דינמי בו מתבצע זיהוי של core במצב idle ורתימתו מחדש למאמץ החיפוש. ניתן היה להשתמש לדוגמה ב-Fork/Join framework (אשר ייכנס ל-Java 7) כדי שמימת מרחב החיפוש תתבצע באופן מאוזן בכל הליבות לאורך כל החיפוש, אולם מטעמי הפשטות הפתרון שנבחר הינו פיצול "ידוע מראש"<sup>16</sup>: מרחב החיפוש פשוט נחצה לשניים (או יותר) כבר בענף הראשון וכל ליבה קיבלה חצי אחר שלו.

דוגמאות הקלט נתונות בקבצי Bean Shell, אלו סקריפטים "בשפת" Java אשר מפורשים בזמן ריצה. גם כאן, ניתן היה להשתמש ב-Groovy הידועה יותר אך אין בבחירה זו או אחרת משום הבדל משמעותי.

בנספח מדריך למשתמש הרחבה למפתח המעוניין בהוספת סוגי שריגים אחרים אל 10,000 שורות הקוד.

השריגים שנבחנו הם:

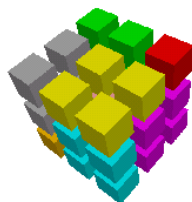
- אורתוגונוליים-דו-מימדיים (2D) ותלת-מימדיים (3D) - שתוארו בהקדמה.
  - פירמידת כדורים
  - פירמידת משושים-כולל תכונה על תאים בפזל (מקרה פשוט של צירוף מפתחות ומנעולים)
  - טנגרם
  - גליל דיסקיות – לבחינת מימוש כללי של צירוף מפתחות ומנעולים.
- בטבלה מסוכמות התוצאות עבור מדגם ממקרי המבחן שנבדקו (המקרים היסודיים מצורפים בנספח ד'), ומיד אחריה פתרון אקראי.

<sup>16</sup> בפיצול שכזה, כמובן שהשיפור אינו לינארי במדויק במספר הליבות, לדוגמה: עבור DL 5x4x3 - ליבה אחת: זמן 65 שניות, עבור 2 ליבות: 42 שניות (לעומת התיאוריה – 32.5 שניות) והוא תלוי בסדר החלקים הנתון.

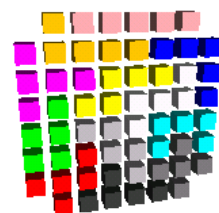
זמן (שניות)	ענפים	אלגוריתם	פתרונות סימטריות	יחודיים	גודל MC	צמצום	חלקים סו"כ	מקורי	גודל	פול שם	שריג
23	23,524,900	BT	*8	2170	1916	56	56	12	60	8 X 8 - Corners	2D Ort.
17	12,199,877	BT+ST									
67	23,525,610	MC									
45	18,278,672	MC+ST									
6.6	1,322,540	MC+SZ									
10	1,306,946	MC+SZ+ST									
3.3	1,691,373	BT(+ST)	*24	1	297	21	21	9	27	Slothouber-Graatsma	3D Ort.
1.7	498,714	MC(+SZ)									
2.2	1,694,454	MC(+ST)									
27	15,806,635	BT	*12	49,160	1,095	53	53	10	37	Cluster	Hex
24	9,720,154	BT+ST									
41	15,818,006	MC									
34	13,234,306	MC+ST									
12	1,116,474	MC+SZ									
11	994,216	MC+SZ+ST									
0.2	3,358	MC(+ST)	*12	18	123	56	56	6	20	Pyramid II	Spheres
0.1	397	MC+SZ(+ST)									
1,088	40,257,073	BT+ST	*1	284	924	77	77	12	48	Locks	Rings
< 3 יממות	?	MC									
1	1,329	BT(+ST)	*1	1	104	18	18	7	32	Diamond	Tangram
1	1,538	MC(+ST)									
0.1	292	MC+SZ(+ST)									



פתית שלג



Graatsma



8x8 ללא הפינות

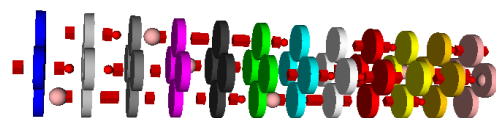
פתרון יחיד !



פירמידת כדורים



טנגרם (\*)



דיסקיות - מנעולים ומפתחות

## - איור 11 -

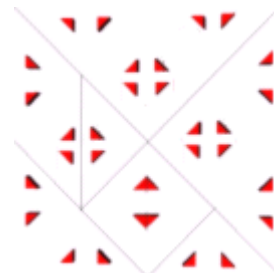
(\*) עבור טנגרם מומשה הצגה בסיסית להדגמת טיפול בתא עם direction:

```

  A A      A A
G      A A      B
G      A A      B
  G C      B B
  G C      B B
  E      F F      B
  E      F F      B
  E E      D D

```

טנגרם קלאסי - טקסטואלית



טנגרם קלאסי - גרפית (ניתן לשיפור)



כלל הפזלים שנבדקו (מלבד המוזכרים בנספח ד'):

- שני מימדים

- חלקי ה- pentominoes

- $20 \times 3$  – 2 פתרונות יחודיים;

- $15 \times 4$  – 368 פתרונות;

- $12 \times 5$  – 1010 פתרונות;

- Dudeney  $8 \times 8$  בתוספת החלק  $2 \times 2$  – 16,146 פתרונות;

- Eureka – 8 חלקים, פזל  $8 \times 8$  – 32 פתרונות;

- שלושה מימדים

- Red Happy Cube – 4 (ה-Happy Cube הקשה ביותר – מספר הפתרונות הנמוך);

- חלקי ה- pentominoes

- Big T – 146 פתרונות;

- Big C – 22 פתרונות;

- Whole Stairs – 364 פתרונות;

- Soma Cube – 7 חלקים, פזל  $3 \times 3 \times 3$  – 240 פתרונות;

- Bedlam Cube – 13 חלקים, פזל  $4 \times 4 \times 4$  – 19,186 פתרונות;<sup>14</sup>

- Live Cube – 5,040 פתרונות;

- "Gaya" Cube – 7,921 פתרונות;

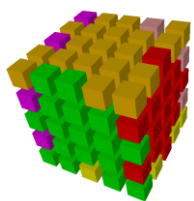
- Pairs –  $4 \times 4 \times 5$ , החלקים הם כל 10 צירופי 2 בלוקים של  $1 \times 2 \times 2$  – 25 פתרונות.

- טנגרם

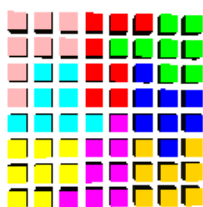
- קלאסי, ריבוע – 1 פתרון.

---

<sup>14</sup> הפזל ה"קשה" ביותר, 52 מיליון ענפים ו-9 דקות לכלל הפתרונות עבור MC+SZ לעומת כדקה בפזל  $5 \times 4 \times 3$ . גודל גודל המטריצה 3,774.



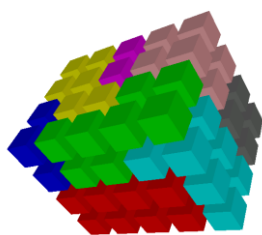
Red Happy Cube



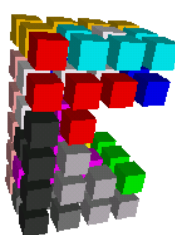
Eureka



20x3



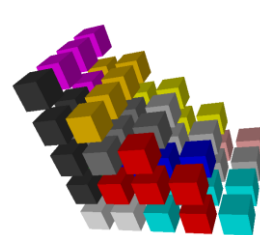
Pairs



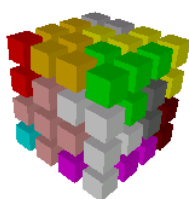
Big C



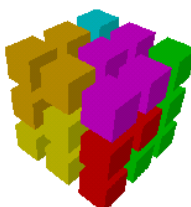
Big T



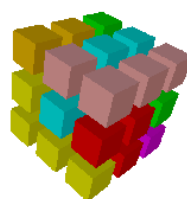
Stairs- simple version



Bedlam

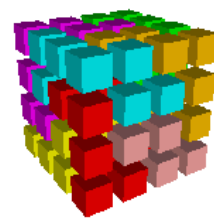


Soma



"גאיה"

חלקים 2D



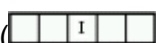
Live cube

חלקים 3D

## - איור 12 -

הרחבה על מקצת מן העמודות:

גודל – מספר התאים בפזל.

חלקים – סך אוריינטציות מופחתות – גודל השריג מכתוב היתכנות של אוריינטציה כלשהי, לדוגמה: לפנטומי "I" () 3 אוריינטציות אפשריות (בתלת-מימד) אך בשריג אורתוגנלי דו-מימדי 20x3 רק אחת מהן אפשרית (עבור אותו מופע של השריג, דהיינו 20x3 אינו 3x20).

גודל MC – מספר האפשרויות להשמת כלל החלקים (מספר השורות במטריצה עבור אלגוריתם MC). זהו מדד קורלטיבי למספר הענפים בחיפוש, אשר יכול לרמז על קושי הפזל.

פתרונות ייחודיים – תואר בהרחבה לעיל.

אלגוריתם: BT – Back Track, נסיגה לאחור; ST – Stranded; MC – Matrix Cover; SZ – היוריסטיקת גודל התת-עץ המינימלי.

ענפים – סך הצמתים בחיפוש. לדוגמה: עבור הפזל באיור 2 בסעיף תיאור הבעיה, ייתכנו במקרה הגרוע  $96 = 3! \times (8 \times 2 \times 1)$ , או מכפלת כמות האוריינטציות של כל חלק במספר האפשרויות לסדרן.

הזמנים הנתונים הינם במונחי single core.

הערה: שימוש ב-Java גרסת 64 סיביות ובגרסה הנסיונית של Java 7 – לא שיפר את התוצאות. בגרסה זו המקביליות אינה סטטית אלא דינמית, ובכל רגע בו מעבד אחר במצב idle, הוא מקבל אליו "נתח" חדש מעץ החיפוש.

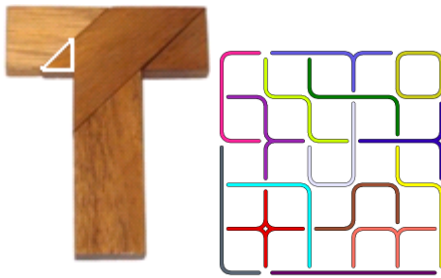
ניתן להוסיף מדדים נוספים כגון מספר העדכונים בסריקת עץ החיפוש (מספר הקריאות ל- cover/uncover באלגוריתם MC).

התוצאה הבולטת הינה עליונותה של שיטת כיסוי המטריצה בתוספת היוריסטיקת גודל התת-עץ המינימלי על-פני החיפוש הנאיבי. במקרים רבים טריק ה-Dancing Links משפר בקבוע את זמן הריצה. כמובן שהיוריסטיקת Stranded אף היא תורמת שיפור מה לצמצום מרחב החיפוש.

חשוב ביותר לתת את הדעת למקרים בהם שיטת כיסוי המטריצה אינה טובה בהשוואה לחיפוש הנאיבי. בשריג הדיסקיות, כיוון שכל החלקים זהים, מלבד צירוף המנעולים והמפתחות המולבש עליהם, אין כלל משמעות להפעלת היוריסטיקת ה-Size או אפילו שימוש ב-Dancing Links. דווקא כאן, השיטה הפשוטה מהירה לאין ערוך. שכן מטבעה היא מכסה את שריג המטרה באופן סדרתי, וכך מתאפשרת בדיקת צירוף המנעולים והפתרונות לאחר כל כיסוי. בדיקה זו מצמצמת את מרחב החיפוש בהתאם לפוטנציאל האמיתי שלו, אשר קטן בהרבה מסך הבדיקות שה-Exact Cover מבצע (החלקים מושמים ללא סדר לקסיקוגרפי בשריג המטרה, כך שיש לחכות לפתרון על-מנת לוודא שהוא אכן פתרון).



בעבודה נבחנו שתי שיטות לפתירת פזלים: האחת בהתאם לטכניקה שהוצעה בעבר Dancing Links עבור פתרון Matrix Cover והשנייה נסיגה לאחור נאיבית. השיטות נועדו להדגים היתכנות הרעיון על שריג כלשהו, ונבחנו בסוגים שונים, כולל צירוף מנעולים ומפתחות.



שיפורים נוספים:

- תמיכה בשריגים נוספים כמו "אותיות" (מסומן החלק הגרעיני), Polyamonds, Poly-sticks (כמו טנגרם, לכל משולש תכונת כיוון לימין או לשמאל), Super Tangram (כל האריאציות לחיבור 4 משולשים שווי-שווקיים שהם חצי ריבוע) ועוד.
  - אופטימיזציות ברמת השריג (לדוגמה, דרך plugins). יש לזכור כי יצירת צמידות לשריג מסוים ע"י חקירת תכונותיו ומימוש היוריסטיקות שונות הנובעות מהן לא הייתה הדגש כאן, אלא חיפוש המכנה המשותף הנובע מאופן הייצוג הכללי, ובדיקת כדאיותו, כך שכל היוריסטיקה תתאים לכל שריג שהוא.
  - מימוש היוריסטיקת התת-עץ המינימלי גם בנסיגה לאחור הטריויאלית. מאידך, החסרון הוא המאמץ בבדיקת מספר ההשמות החוקיות של החלק הנוכחי במשבצת מסוימת, שהיא לאו דווקא נקודת המוצא של אותו החלק. יש כאן trade off שכלל אינו בטוח שישתלם.
  - ייעול תהליכי האתחול כך שירוצו פעם אחת. כיום כל thread בפיצול עץ החיפוש עובר אתחול בנפרד.
  - הכללת היוריסטיקת התת-עץ המינימלי על צירוף מנעולים ומפתחות, כך שלא ייגזמו ענפי הפתרונות, וגם הם ייהנו מפירות טכניקת ה-Dancing Links.
  - שיפור חוויית המשתמש עבור הגדרת הפזל. במקום למצוא מודל מתאים שלו ולחבר סקריפט המתאר את הקשתות בין התאים (רגיש מאוד לטעויות על אף מכלול הבדיקות שמבוצעות), לצייר את החלקים והפזל על-פני canvas של התאים האפשריים.
- הרחבה נוספת ומשמעותית תהא התאמת האלגוריתמים (הטריויאלי וה-Matrix cover) ל-Burr Puzzles – הפזלים בהם חשוב סדר ההרכבה.
- חובבי הפזלים בכל גיל יכולים למצוא ביישום ההיתכנות שימוש חביב, כפותר חידות הרכבה שונות, דוגמת צירופי הטנגרם הפופולרי.



- [Sc58] Scott, D.S.: Programming a combinatorial puzzle, Technical Report 1, Dept. of Electrical Engineering, Princeton University (1958)
- [Kn75] Knuth, D.E.: Estimating the efficiency of backtrack programs, *Mathematics of ours. Computation* 29, 121 - 136 (1975)
- [GB65] Golomb, S.W. and Baumart, L.D.: Backtrack programming, *J. of the ACM*, 12, 516 - 524, (1965)
- [Br71] De Bruijn, N.G.: Programmeren van de pentomino puzzle. *Euclides* 47, 90–104, (1971–1972)
- [Fl65] Fletcher, J.G.: A program to solve the pentomino problem by the recursive use of macros. *Comm. of the ACM* 8, 621–623 (1965)
- [To69] Torbijn, P. J.: Polyamonds, *Journal of Recreational Mathematics* 2, 216 - 227 (1969)
- [Le78] Lewis, H.R.: Complexity of solvable cases of the decision problem for the predicate calculus, 19th Ann. Symp. on Foundations of Computer Science, Ann Arbor, MI, 35-47 (1978)
- [DD07] Demaine, E.D., Demaine, M.L.: Jigsaw puzzles, edge matching, and polyomino packing: Connections and complexity. *Graphs and Combinatorics* 23 (suppl.), 195–208 (2007)
- [Ko98] Korf, R.E.: Artificial intelligence search algorithms, *CRC Handbook of Algorithms and Theory of Computation*, CRC Press, Boca Raton, FL, 36-1 to 36-20 (1998)
- [GJ79] Garey, M. and Johnson, D.S.: *Computers and Intractability: A Guide to the Theory of NP-completeness*, Freeman, San Francisco (1979)
- [KPS08] Kendall G., Parkes A. and Spoerer K: A Survey of NP-Complete Puzzles, *International Computer Games Association Journal (ICGA)*, 31(1), 13-34 (2008)
- [Kn00] Knuth, D.E.: Dancing links, in: *Millennial Perspectives in Computer Science* (Davies, J., Roscoe B., and Woodcock J., eds.), Palgrave Macmillan, England, 187-214 (2000), <http://arxiv.org/abs/cs/0011047>





## נספח א' שריגים ומקרים מיוחדים

בכל שריג קיים סט שגרות משותף. להלן תיאור העיקריות שבהן ע"פ מורכבות מימוש השריג.

### שריג אורתוגונלי דו-מימדי

מתואר בסעיף תיאור הבעיה.

### שריג אורתוגונלי תלת-מימדי

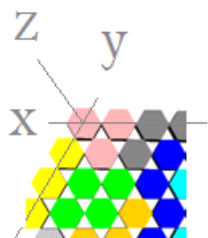
פשוט וטריויאלי. ראה בסעיף תיאור הבעיה, איור 1.

בהיבט הקשת: 4 קשתות כמו בשריג דו מימדי ו-2 קשתות נוספות למימד הנוסף: קדימה ואחורה.

לצורך חישוב נקודת המוצא, נגדיר (למען נוחיות המימוש, לעומת סדר לקסיקוגרפי) כי התקדמות דרומה משמעותה הוספת 10,000 נקודות לציון התא, התקדמות קדימה – 100 נקודות, והתקדמות מזרחה – 1 נקודה. כפועל יוצא, המשמעות היא כי גודלו המירבי של הפזל הוא  $100 \times 100 \times 100$  – אחרת, שני תאים שונים יקבלו ציון זהה.

בהיבט התא הבודד: הסיבובים הם Rotate ו-Flip, אך הפעם ב-3 הצירים האפשריים.

### משושה



ראה בנספח ד', איור 3. בהיבט הקשת: 6 קשתות – יוצאת מכל צלע, וקשתות קדימה ואחורה.

מקדם המרחק בין תאים הינו  $\sin(60)$  – נועד להצגה הגרפית.

לצורך חישוב נקודת המוצא, היחס הוא כמו לשריג תלת-מימדי.

- איור 13 -

לצורך הבחנה בין "צבע" החלקים כפי שנדרש בפזל המטרה, נוספה תכונת "צבע" שחור/ לבן על כל חלק וחלק. כמובן שבנסיון שיכון בודקים מידית את התאמת צבע התא בחלק, לצבע התא בפזל.

מימוש ישיר, נוסף דגל special עבור תא. בשיכון נבחנת התאמת special בין התא המארח בפזל לתא המתארח מהחלק.

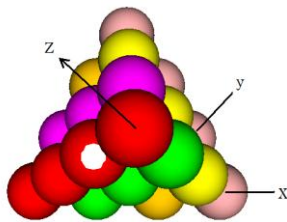
בהיבט התא הבודד: הסיבובים הם Rotate-ו Flip בציר XY (שכן אין סימטריה בתא עצמו, בין המישור XY למישור XZ).

### גליל דיסקיות

ראה בסעיף מנעולים ומפתחות. כהשלמה נציין כי הסיבובים הם Rotate-ו Flip יחיד. וכן שלחישוב נקודת המוצא (למעשה אין לכך משמעות בפזל זה בהם כל אין חלק המורכב מיותר מדיסקית אחת) ניתנת נקודה בהתאם לרבע הדיסקית עליה נמצאים, וכן 100 נקודות בעבור כל התקדמות לדיסקית אחרת.

### פירמידת כדורים

בהיבט הקשת: קיימות 18 (!) קשתות, חצי הן כמובן סימטריות. 6 קשתות עבור 6 השכנים במישור, ו-3 קשתות למעלה ולמטה. שים לב שכל קומה שונה מהקומה השכנה. ניקח כדוגמה את החלק האדום המסומן בלבן, עבורו קיימת קשת "צפונה ואחורה", אך כלפי מעלה, קשת זו אינה חוקית, המקבילה לה היא קשת "דרומה וקדימה".



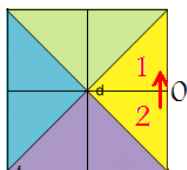
- איור 14 -

לחישוב נקודת המוצא, היחס הוא שוב כמו לשריג תלת- מימדי.

בהיבט התא הבודד: הסיבובים הם Flip כמובן, וכן 6 סיבובים סביב הצירים.

פירוט אופן חישוב המרחק בין התאים עבור התצוגה הגרפית מצורף בקובץ sphereMath.gif בחבילת התוכנה.

### טנגרם (שריג דו-מימדי, אין מימד "עומק")



- איור 15 -

כאמור בתקציר, התא הבסיסי הינו משולש עם תכונת כיוון הסיבוב שלו. על-כ, לראשונה נוספה כאן תכונה ברמת התא – כיוון הסיבוב ( $0^\circ$ ,  $45^\circ$ ,  $90^\circ$ , ...) או 0..7 (מסומנים תאים 1,2 באדום והלאה בכיוון השעון). ברמת הקשת, מספר הקשתות הוא גם 8 (מסומנת קשת 0 בשחור והלאה נגד כיוון השעון). כל חלק מורכב ממספר תאים.

מקדם המרחק בין התאים לצורך התצוגה הגרפית הוא  $\sqrt{2}$ .

לחישוב נקודת המוצא, היחס הוא כמעט כמו לשריג דו-מימדי: לכל ריבוע המכיל תא בסיסי ציון ממש כמו בשריג הדו-מימדי. לציון זה מוסיפים את כיוון הסיבוב של אותו תא.

בהיבט התא הבודד: הסיבובים הם Rotate ו-Flip.

## מקרים מיוחדים

### מדרגות בשריג אורתוגונלי תלת-מימדי

בפזל זה 55 תאים, ומאידך חלקיו הם ה-pentominoes דהיינו 60 תאים יחידים. אחד מהחלקים (לא ידוע מי) אינו נמצא בפזל.

ניתן כמובן לפתור 12 סוגי פזל, כל פעם חלק אחר ייגרע מסט החלקים. באופן זה אמנם ייפתר הפזל, אך הזמן הדרוש לפתירתו יהיה רב יותר, וכמובן שהבעיה לפתרון כבר אינה המקורית.

קל מאוד בשיטת הנסיגה לאחור להתגבר על ה"סתירה" לכאורה. פשוט בודקים שמספר החלקים שנותרו פנויים הינו 1 (במקום 0) וכמובן, שלא נותרו תאים פנויים בפזל. כך גם ב-Matrix Cover ללא היוריסטיקות.

לעומת זאת, בהיוריסטיקת גודל התת-עץ המינימלי, ייתכן כי ייחתך תת עץ (שאינו מינימלי כמובן) שמכיל פתרון, ומאידך, שהתת-עץ שנבחר לא מכיל כלל פתרונות. בפועל לעולם לא נבחר תת עץ שכזה ולא נמצא ולו פתרון אחד. זהו בדיוק טבעה של היוריסטיקה – כלל אצבע אשר אמור לעבוד, אך לא חייב.

### פזל happy cube

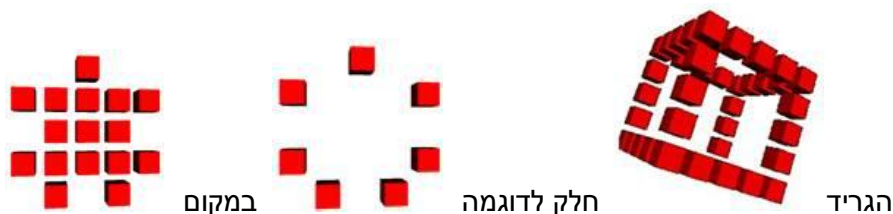


פזל זה מוגדר רק עבור פאות הקוביה. למעשה אין כאן מקרה מיוחד מלבד אולי שימוש לא שגרתי במוסכמה פרטית עבור הגדרת כל פאה, אשר תאפשר להגדיר את מגוון הפזלים ממשפחה זו בקלות יחסית.

- איור 16 -

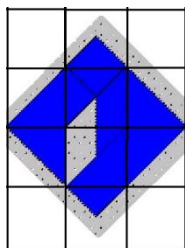
לדוגמה, 1/0 עבור קיום/ אי-קיום התא בפאה. כך שתיאור הפאה העליונה ב-Java הוא  
`add(create('A', new String[] { "0101", "0100", "0100", "0010" })))`

לאחר מכן מבוצע parse לבניית כל הקשתות והצמתים בפאה זו.



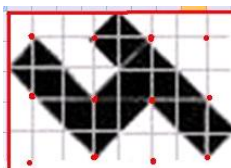
#### טנגרם שאינו מיושר לשריג

השיטות שתוארו (חיפוש טריויאלי או Matrix Cover) דורשות את היכולת לחשב את נקודת המוצא של כל חלק באופן אוטומטי כפי שבחרנו. הבחירה באופן החישוב יכולה להשפיע רבות על היכולת לפתור פזלים מאותו סוג שריג.



פזל זה לדוגמה (בכחול בלבד) דורש "כיסוי" ריק (place holder בקוד) כדי לשמור על יישור קו עם פתרון פזל הטנגרם המקורי. בבניית פזל זה לדוגמה, השורה הראשונה תהיה `buildSquares(4,3)` ורק לאחר מכן הורדת ה-place holders. בפזל המקורי אין צורך במעקף זה.

כך גם בטנגרם הספרה 4.



פזלים אחרים דורשים יישור מלאכותי עוד יותר, שכן הקשתות בין התאים אינן מתאימות לקשתות הקיימות. כאן ניתן לדוגמה ליישר את כף רגלו הימנית של האיש כך שהחלקים יתאימו לשריג (וכך גם הקשתות). עדיין צריך דרך נוחה לחישוב נקודת המוצא של הפזל (כמו שבירה למשבצות).



לעתים הדבר דורש את ידיעת הפתרון מראש, שכן ללא ידיעה זו, לא ניתן להגדיר את הפזל כנדרש (כזכור, גרף - תאים וקשתות ביניהם).

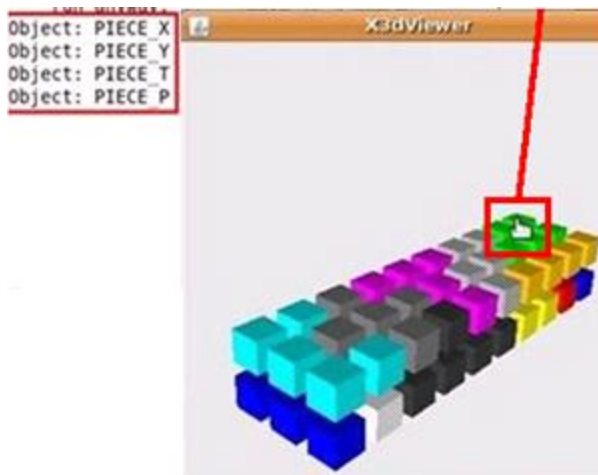
בנוסף, השינוי הנדרש יכול להיות גדול מדי ולגרום ליותר על היכולת לפתור את הפזל כמות שהוא:

חובה להדגיש כי מימוש הטנגרם נועד רק לרמז על כוחה של שיטת הייצוג לפתרון פזלים, ולא להציף את מגבלותיה.

## נספח ב' מדריך הפעלה

### התקנה

פתיחת miniDist.zip אל הספרייה המבוקשת. התקנת Java Runtime Environment /Java Dev Kit 6.  
לצפייה בפלט/ קלט גרפי של פתרונות/ פזל/ חלקים יש להתקין x3d viewer – קישור בספריית windows.



הרצת GraphLayout.jar או GraphLayout.bat

בלינוקס, הפעלה ע"י GraphLayout.sh. במהלך הצפייה הגרפית קליק על תא יראה את ה-id שלו. לשם כך יש להתקין את התוסף java3d.

### קבצי תצורה

myLog.Properties – הגדרת מוד עבור הלוג, האם לשמור הודעות שגיהא בלבד או גם הודעות verbose למעקב אחר פעולת התוכנה. העקרון: בכל שורה רשום ה-class ומוד הלוג שלה.

myLog.Properties.Log4J – הגדרת מוד הלוג, במידה ומשתמשים ב-log4j. כיום התוכנה משתמשת בספריית הלוג המסופקת עם Java.

myPzl.Properties – הגדרות אתחול (בצירוף ברירות המחדל):

- `Generate by All = false`, האם להשתמש בחלק ייחודי בעל אוריינטציות מעטות (לדוגמה, F בפנטומינוס)
- `Engine Type = 0`, סוג המנוע.
- `0` עבור Matrix Cover by Dancing Links - כדקה למציאת 3,940 פתרונות בשריג אורתוגונולי תלת-מימדי 5x4x3;
- `1` עבור מנוע Back Track רקורסיבי (הטריויאלי) – כשעה;

- BT 2 איטרטיבי.
- Full Output = false, האם להדפיס את כל הפתרונות או רק את הראשון
- Threads = 1, מספר המעבדים בהם ייעשה שימוש:
- 0 עבור זיהוי אוטומטי (במחשב dual core לדוגמה זהה ל-2);
- 1 עבור שימוש ב-CPU יחיד;
- 2 במחשב dual core, שימוש בכל עוצמת החישוב. אל דאג, המחשב עדיין יהיה זמין לפעולות אחרות שכן הריצה היא בעדיפות נמוכה על פני שאר התהליכים במערכת;
- 4 ב-quad core, וכו' (רצוי לבחור 3 על-מנת להשאיר ליבה פנויה לחלוטין).
- Auto Debug = true, האם להיכנס מידית למצב דיבוג.
- Size Heuristic = true, כיום עבור Dancing Links בלבד, שימוש בהיוריסטיקה- מיון ענפי עץ החיפוש לפי גודלם.
- Stranded Heuristic = true, שימוש ביוריסטיקה "אי בודד".
- Auto Graph it = true, צפייה אוטומטית בפתרון הראשון.
- Graph for All = false, יצירת הקבצים לצפייה גרפית עבור כל הפתרונות.
- Internal Viewer = false, שימושי בעיקר בסביבת לינוקס, אם כי כעת קיים כבר viewer חיצוני.
- Results.properties – עבור כל מקרה, שמירת מספר הפתרונות הייחודיים והכלליים. חיוני למתן שערור של זמן ריצה נותר.

## קבצי פלט

בספריית tmp/<config>\_<args> נשמרים עם <Engine Type> הקבצים הבאים:

Engine\_Console.log – פלט שנוצר במהלך הריצה והוצג למשתמש.

Engine.log – לוג, פלט "מאחורי הקלעים" של התקדמות הריצה. לדוגמה, איזה thread מצא איזה פתרון.

x3d – פתרון גרפי עבור <solution index>, מכיל בתחילתו את סדר החלקים בפתרון, את הפזל הסופי בצורה שטוחה (שורה אחת) ע"פ יחס סדר על התאים בשריג, את הפזל בצורה טקסטואלית ואת מספר הפתרון. לדוגמה:

Solution #2 Parts order H7 G3 F5 J7 I5 K7 A0 B0 C0 D0 E0

H G G F F H H J J I K J I K K H A G B F I C G F I K J D D E A A A B B C C B B E C D E D E

H H I I K

G H K K

G J J

F J

F

....

config בהתאם לתצורה שנבחרה, args אפיון ספציפי של המקרה (לדוגמה, מימדי x y z).

saves – שמירת שבוצעו. פורמט בינרי. שם הקובץ <args>\_<config>\_state\_puz.bin. נוצרות אוטומטית כל דקה כאשר המנוע הנבחר הינו האיטרטיבי.

## תצורות מוכנות מראש – הסקריפטים

### ספריית config

מכילה סקריפט הגדרות שריג וחלקים. כרגע תמיכה במשפחות: קובייה, כדור ומשושה, טנגרם וגליל טבעות (הדגמת מקרה אמיתי של מנעולים).

validateBshs.bat – מוודא תקינות הסקריפט.

שם הקובץ <config name>\_grid.bsh ו- <config name>\_parts.bsh.

דוגמה לשריגים פשוטים

#### 3D- sizes x y z

```
grid = new Grid3D();
```

```
grid.build3d(5,5,5);
```

#### sphere pyramid

```
grid = new GridSpheresPyramid();
```

```
grid.buildPyramid(2);
```

דוגמה מורכבת לחלקים: spheres\_parts.bsh ולציון מפתחות ומנעולים: 3d\_parts.bsh.

עבור כל חלק, הגדרת שם, מספר התאים בו, וקשתות מרכיבות (זוג התאים המעורבים, וכיוון הקשת ביניהם). סוגי הקשתות מפורטים בנספח שריגים. ניתן לראות ב-src את קבצי Arc\*.java.

```
part = new PartsSphere('E');
```

```
part.prepareRotations(3);
```

```
part.addArc(1, ArcBall.EAST, 2);
```

```
part.addArc(2, ArcBall.EAST_120, 3);
```

```
parts.add(part);
```

```
part = new PartsSphere('F');
```

```
part.prepareRotations(1);
```

```
part.add(part);
```

להלן דוגמה לסכימת מנעולים ומפתחות עבור פזל הדיסקיות

```
final int KEYS=2; final int LOCKS=1; //final int MAX=Math.max(KEYS,LOCKS); // USER DEFINITION
```

```
boolean [][] matches = new boolean [KEYS+1][];
```

```
/****** USER DEF... *****/
```

```
// lock without a key
```

```
matches[0] = new boolean[LOCKS+1]; matches[0][1] = true;
```



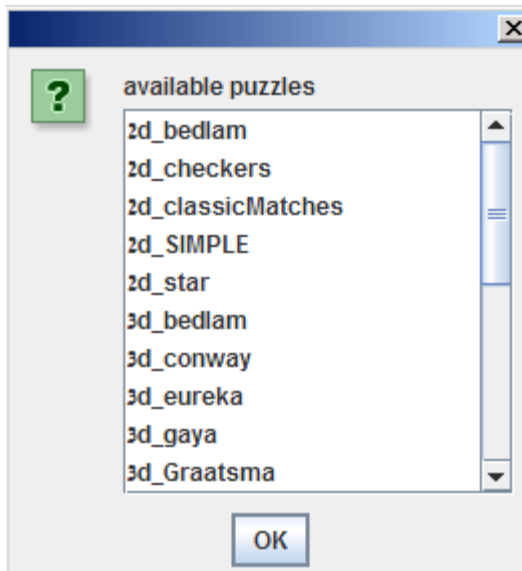
```
// invalid case - implicit declaration
matches[0][0] = false;
// key=1
matches[1] = new boolean[LOCKS+1];
// key without a lock: matches[1][0] = true
matches[1][1] = true;
// key=2
matches[2] = new boolean[LOCKS+1];
matches[2][1] = true;
/***** ...USER DEF *****/
return matches;
```

התוכנה מריצה בדיקות לוגיות על ההגדרות. שגיאות יופיעו בצירוף error:

- error: no unique is used – לא נבחר חלק ייחודי על-פיו נקבעים פתרונות ללא חזרות (וכך החיפוש מהיר יותר)
- error: you should choose other unique – כאשר החלק הייחודי שנבחר לא ממקסם את הייחודיות. לדוגמה, בחלקי הפנטומינוס ופזל מסוג שריג אורתוגונלי דו-מימדי, בחירת L 4) אוריינטציות (לעומת F 8)
- error: parts already contains such id – חלק זהה כבר קיים
- error: invalid config, size of parts xx vs. grid yy – פתרון אינו אפשרי, אין התאמה
- error: IGrid::remove - already deleted – כאשר מגדירים פזל עם "חור"
- error: self reference – הוגדרה קשת בין תא לעצמו, ייתכן וטעות, תבוצע התעלמות
- error: part X already exists as Y – חלק זהה קיים כבר. כיום יש להשתמש ב-one(i) another על כל חלק זהה וב-one(amount) another בסוף הכנסת amount חלקים זהים. דוגמה ב-graatsma\_parts.bsh

## הרצה

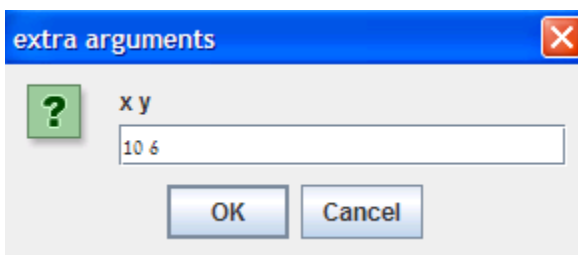
waiting for user input...



בהפעלת התוכנה מופיעים החלונות הבאים:

בחירת פזל

ארגומנטים (אופציונלי)



ה-Console מאפשר הזנת פקודות דיבוג (אם Auto Debug = true). סגירתו תסגור את התוכנה. ה-console מופיע רק בהפעלת התוכנה דרך GraphLayout.bat. הרחבה בהמשך.

### DEBUGGING.

Examples: `parts.get('F').rotate(); grid.show(); parts.show('F');` to stop: `exit` or `kill`

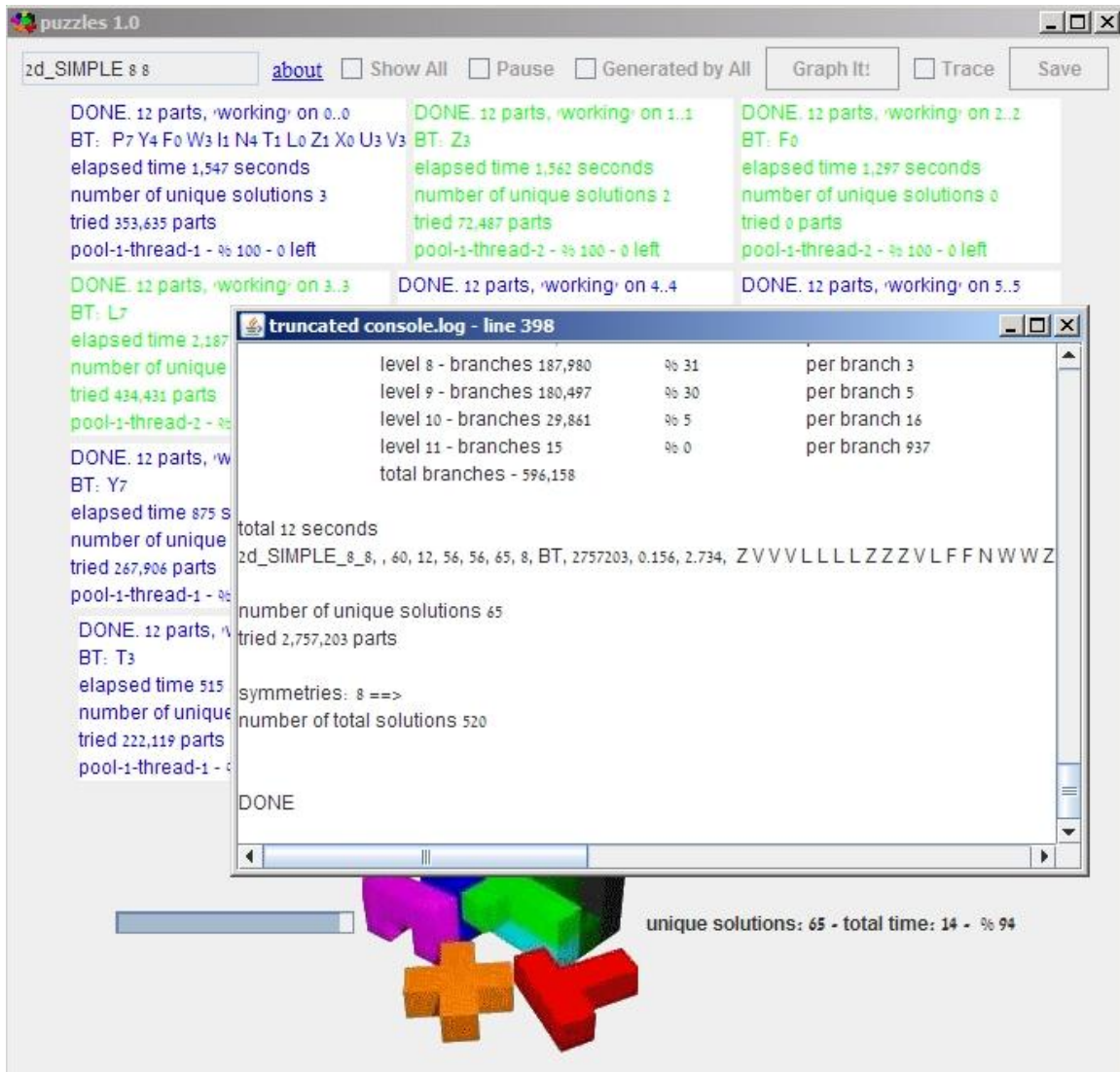
**>>parts.show('F')**

Part F 5 cells: F #0 edges [ RIGHT:1>>1 DOWN:3>>3] F #1 edges [ LEFT:0>>0] F #2 edges [ RIGHT:1>>3] F #3 edges [ LEFT:0>>2 UP:2>>0 DOWN:3>>4] F #4 edges [ UP:2>>3] rotations: 8

GraphIt.x3d generated, from Engine0, for solution #0

tmp/customPartIndex\_0.x3d

>>



הבקרים ב-GUI:

### סטטוסים

Main args[] מראה את בחירת המשתמש



Load disabled/ Loaded – האם בוצעה טעינת מצב שמור.  
במידה והמשתמש בחר שוב config + args עבורם קיימת  
שמירה תופיע שאלה למשתמש. אפשרי רק במנוע BT  
איטרטיבי ועבור מס' thread-ים = 1.

Progress bar – צפי לסיום, במידה וחושב בעבר (כאמור, נשמר ב-results.properties)

סטטוס ריצה, זמן ריצה, מספר פתרונות יחודיים (או כלליים, בהתאם ל-Generate by All), מספר נסיונות וזמן ריצה משוער לסיום. במידה ו-1 > threads השורה העליונה תציין על איזה חצי (או רבע) מהבעיה עובדים. זמן הריצה מתבסס על מספר הפתרונות (ולא הענפים שכן הוא משתנה בהתאם להיוריסטיקות שנבחרו וסוג המנוע).

### אפשרויות בקרה

Show All – פלט של כל הפתרונות, כולל אוריינטציות של כל אחד ואחד. לדוגמה, בפזל ריבועי דו-מימדי עבור פתרון ייחודי ייתכנו עוד 7 העתקים שלו (סיבוב ושיקוף).

Pause – במידה ו-1 > threads ההפעלה תחליף מעגלית בין המצבים Not Paused/Pause/Pause 2/Pause 1- כאשר 1 ו-2 מציינים את id המעבד שבמצב השהייה.

Generate by All – בהתאם ל-myPzl.Properties

Save – שמירה, דריסת מצב שמור קודם.

Trace – מאפשר לראות את השיכונים המבוצעים ב-console. טוב לדיבוג.

Graph it! – הצגה גרפית של הפתרון הבא שיימצא (אין שמירה של הפתרון הנכחי). כל הצגה תדרוס את הקודמת (לא את הקובץ השמור ב-tmp).

בסוף הריצה תופיע טבלת סטטיסטיקה ובסופה מספר הענפים הכללי, מספר הסימטריות ועוד.

בקובץ הלוג מופיעים פרטים סטטיסטיים נוספים: מספר האוריינטציות הכללי לפני ואחרי צמצום כתלות בשריג, זמן לפתרון ראשון וזמן כללי, ועוד.

### **דיבוג**

לאחר טעינת הפזל, למשתמש מוצגים מקצת מפרטי קלט שבקובץ: חלקים, סוג הפזל שנפתר ועוד...

Solving Type: examples.cube.dimension3.CellPart3D, parts: 7, grid cells: 27

לאחר מכן יוצג >> prompt למשתמש. ניתן להזין כל פקודה חוקית ב-Java. התוכנה מדפיסה את תוצאת כל פקודה (לכן אין טעם להוסיף System.out.print במפורש).

#### הפקודות השימושיות הן:

- parts.showPart('X') – הצגת החלק (שם, מספר תאים וקשתות ביניהם – כולל מזהה הקשת, ומספר אוריינטציות) ותצוגה גרפית שלו באוריינטציה הנוכחית שלו  
Part X 4 cells: X #0 arcs [ DOWN:3>>3] X #1 arcs [ UP:2>>2] X #2 arcs [ RIGHT:1>>3 DOWN:3>>1]  
X #3 arcs [ LEFT:0>>2 UP:2>>0] rotations: 64
- parts.getParts().get(2).rotate() – סיבוב החלק השני, עד לגמר האוריינטציות הקיימות וחוזר חלילה.
- grid.show() – תצוגת הפזל
- kill – יציאה מיידית, במידה והמשתמש מבין שחלה טעות בהגדרת הפזל או החלקים
- Exit – להמשך ההפעלה

#### רישוי ושונות

source code of Hadoop DLX ,Pine Coast swirlViewer ,BeanShell – כולן ברישוי חופשי.

סביבת הפיתוח – YourKit Java Profiler, Eclipse/ IntelliJ Idea, התוכנה תואמת Java 6. הקוד ניתן להתאמה ל-Java 5 אם משתמשים ב-synchronized<sup>18</sup> במקום ב-Concurrent Map לשמירת מבנה הנתונים המשותף בין המעבדים, אשר נועד למניעת כפילויות – Shared.uniqsFoundSolutions.

---

<sup>18</sup> ניתן להשתמש גם בספריית backport-util-concurrent.jar

## דגשים למפתח

כדי לספק התקנה קטנה ככל האפשר, לא צורף ה- JUnit framework הידוע (כלי לבדיקות יחידה), במקום זה מומשה מחלקה MyTestCase אשר מממשת את העיקר, החזרה ל-JUnit בקלות – ירושה מ- TestCase.

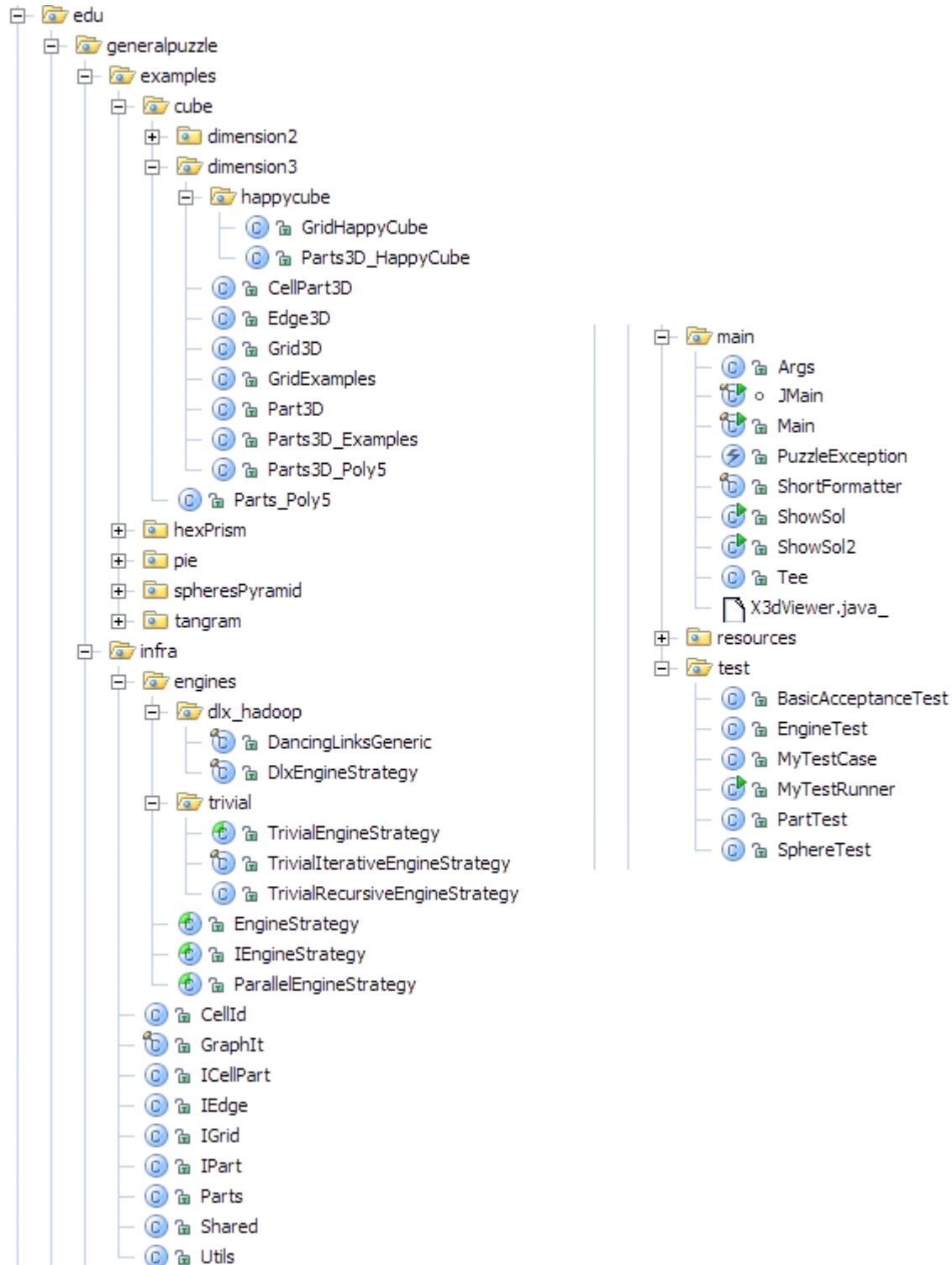
BTrace – מאפשר לבצע trace פרימיטיבי על מתודות בדומה ל-aspect. הדבר נעשה ע"י byte code manipulation. מצ"ב סקריפט דוגמה, המדגים כיצד לבצע trace (על אף שהאופציה קיימת ב-UI). על-מנת שהדוגמה תעבוד, יש להריץ את BTrace עם safe mode = false (במצב true, מובטח כי ה-trace לא יכול לפגוע בנכונות התוכנה).

```
@BTrace public class BTrace {
    @OnMethod(clazz="edu.generalpuzzle.infra\..\*", method="newCanPut.*/*")
    public static void m(@Self Object self, @ProbeClassName String probeClass,
        @ProbeMethodName String probeMethod, AnyType[] args) {
        // print(strcat("entered ", probeClass));
        // println(strcat(".", probeMethod));
        print(((IGrid)self).toString2());
        print(((IPart)args[0]).toString2());
        // printArray(args); // args are the Part to put
    }
}
```

תהליך ה-build הריץ בעבר כברירת מחדל גם את ה-test-ים הקיימים (מלבד ה-acceptance test). על-מנת לספק קובץ התקנה מינימלי נעשה שימוש בכלי Ant build ישן שאינו תומך בהרצת tests. המעוניין יכול להריצם עצמאית.

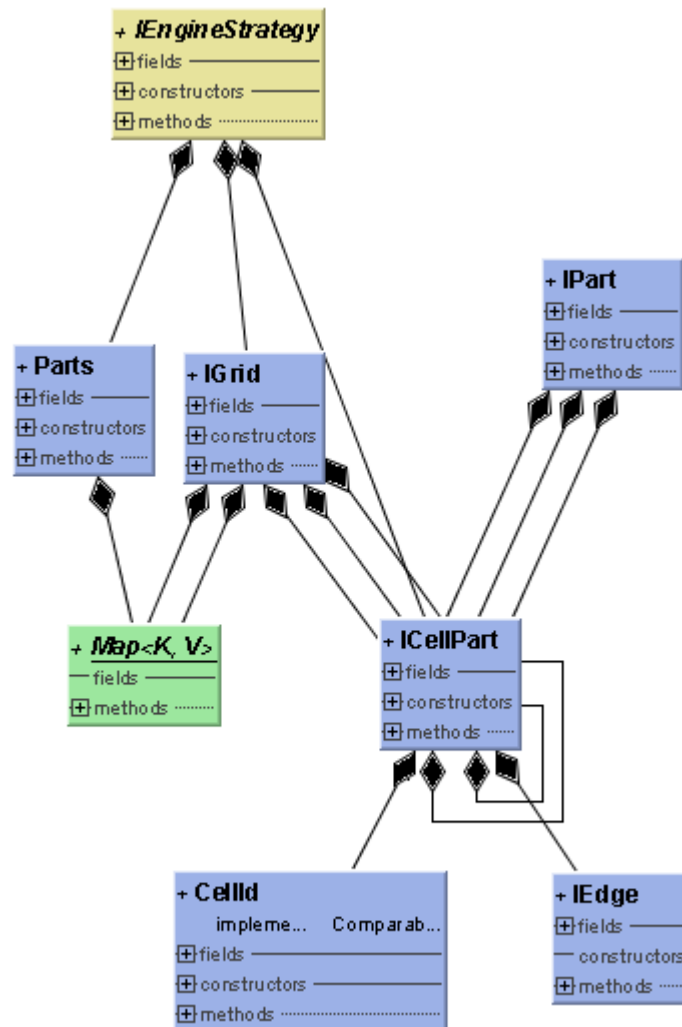
התוצאות רוכזו בעזרת doltAll.bat אשר מריץ עבור כל config את כל השיטות.

## מבנה הפרוייקט:

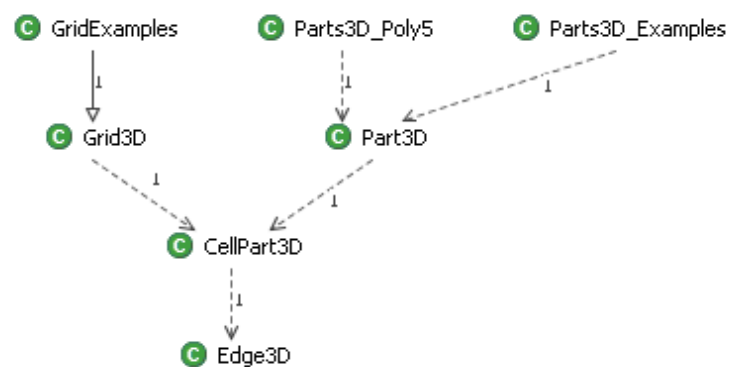


להלן דיאגרמות שנוצרו בעזרת התוסף Simple UML והכלי לניתוח סטטי Structure 101:

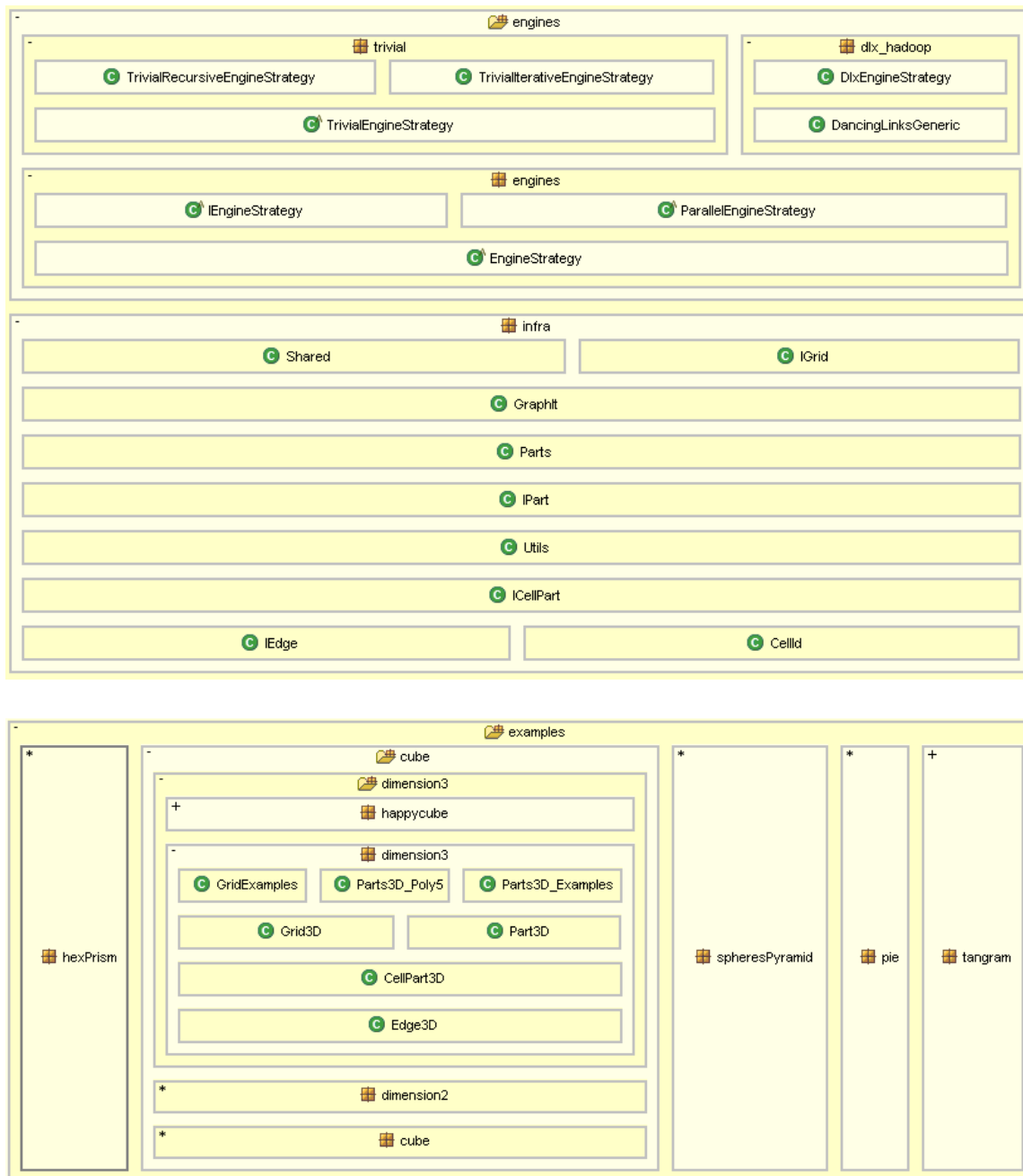
דיאגרמת הכלה



גרף תלויות לדוגמה, עבור שריג אורתוגונלי תלת-מימדי







Main.manualCustom מאפשר לדלג על שלב בחירת ה-config, טוב לפיתוח סוגי שריגים חדשים לפני המרת מקרי המבחן שלהם לסקריפטים.



## נספח ג' מראי מקום לקוד

### הייצוג

"חישוב נקודות המוצא" – `IPart.computeAnchorIndex` בוחר הצומת עם הסימון המינימלי, "בכל טיול על קשת מעודכן הסימון" – `IPart.markGridIds`

### אוריינטציות

"ראשית, מחושב מספר האוריינטציות הנגזרות מכל סוג טרנספורמציה" – `IPart.rotationCycle`  
"לאחר מכן, מחושבות אוריינטציות אלו כפול..." – `IPart.completeRotations`  
"השלב המסכם הינו הורדת הכפילויות" – `IPart.checkDuplicity` אשר קורא ל- `Utils.dfsComparePart`  
ול- `Utils.dfsUncompare`

### החלק הייחודי

"נכונות החלק הייחודי" – `Parts.checkUnique`  
אם החלק אכן ייחודי, חישוב `implied` ב- `EngineStrategy.helper`

### הפתרונות הייחודיים

"כאשר נמצא פתרון" – `EngineStrategy.solved`  
בדיקה לפני גוף ההרצה – `EngineStrategy.preSolved` הכוללת ביצוע `IGrid.verify` – צירוף מנעולים/מפתחות חוקי.  
"יש לסווג, האם מדובר בתמורה זרה" – `Parts.getSamePartExist` ואז בדיקת `sameSolution`, או בפתרון סימטרי (אפשרי רק אם ה- `uniqueInImplied` קיים)  
"פוטנציאל לכפילות" – הרשימה `EngineStrategy.uniqsFoundSolutions` אשר משותפת בין ה- `thread-`ים הפותרים. והמערך `uniqueInImplied`.

## פתרון הפזל

### נסיגה לאחור

"השיטה הראשונה והטריויאלית לפתרון" – `IEngineStrategy.solve` ו-  
`TrivialRecursiveEngineStrategy.putRecursive` – זהו מימוש הפסידו-קוד "שיכון"  
"ניתן לשמור את היתכנות השיכון באופן סטטי" – `EngineStrategy.removeImpossible` שכמובן קורא ל-  
`IGrid.canPut`. `IGrid.canPut` משתמשת ב-`IGrid.dfsPut`, ואם השיכון מצליח, מתקדמת בעזרת  
`IGrid.goForward` או נסוגה ומסירה את החלק עם `IGrid.remove`.  
כיוון שההיתכנות שמורה סטטית, נסיון ההשמה הינו עם `IGrid.new2CanPut`.  
"כאשר הסריקה מצליחה... ועוגן הפזל "מתקדם" – `IGrid.goForward` ולאחר מכן  
`TrivialEngineStrategy.track` לעדכון מבני עזר כמו סטטיסטיקות, ה-`partsIndices` שנותרו לשיכון  
"תהליך זה חוזר על עצמו.. נמצא פתרון" – הבדיקה `leftParts == partsInSolution` (לרוב 0)  
"בכל מקרה, מבצעים הסרה" – `IGrid.removeLast` וחזרה לצומת הנכחי לשיכון עם  
`TrivialEngineStrategy.backTrack` וכן `IGrid.setCurrCellIndex`

### *Matrix cover by Dancing Links*

הפסידו-קוד `DancingLinksGeneric.coverColumn`, `uncover`, `solveDLX`, `cover`, `unCover` בשגרות  
`.search`.  
פיצול לעבודה מקבילית מתאפשר ע"י בניית ראשית עץ החיפוש, שגרות `DancingLinksGeneric.split`,  
`.solve`, `rollback`, `advance`.  
המרת הייצוג מפזל למטריצה ב- `DlxEngineStrategy.regularGeneratorRows`

### היוריסטיקות

#### אי בודד

"ניתן לחתוך ענפים בעץ החיפוש" – `TrivialEngineStrategy.rollback`  
"חישוב גודל האי" – `EngineStrategy.islandSize` וכן `islandRecursive` שהינו DFS

## תת-עץ מינימלי

הבדיקה EngineStrategy.S\_HEURISTIC ב- TrivialRecursiveEngineStrategy.putRecursive (מימוש חלקי).

בשיטת כיסוי המטריצה, בחירת העמודה לשיכון עם מספר ה-1-ים הקטן ביותר - DancingLinksGeneric.findBestColumn.

## שיפורים

חישוב trips – Ipart.calcTrips שקורא ל-Ipart.dfsPut.

שמירה מדויקת של ההשמה – EngineStrategy.removeImpossible

## מפתחות ומנעולים

"סכימת התאמה" – Parts.matches[keys][locks] (לדוגמה, סקריפט 2d\_classicMatches\_matches.bsh). הבדיקה ב-IGrid.match.

"עבור BT, בזמן השיכון, נבדקת ההתאמה" - ב- TrivialEngineStrategy.rollback קוראים ל-IGrid.verifyMiddle

"ב- Dancing Links ... מתבצעת בדיקה" - IGrid.verifyMiddleDLX.

בשתי השיטות, קוראים ל-IGrid.verify כחלק מה-EngineStrategy.preSolved אשר נקרא לפני ה-solved ומוודא עמידה בסכימת המנעולים.



- [Du08] Dudeney, H.E.: 74.—The broken chessboard. In: The Canterbury Puzzles, 90–92 (1908)
- [Fl65] Fletcher, J.G.: A program to solve the pentomino problem by the recursive use of macros. *Comm. of the ACM* 8, 621–623 (1965)
- [Ga57] Gardner, M.: Mathematical games: More about complex dominoes, plus the answers to last month's puzzles. *Scientific American* 197, 126–140 (1957)
- [GJ79] Garey, M., Johnson, D.S.: *Computers and Intractability: A Guide to the Theory of NP-completeness*. Freeman, San Francisco (1979)
- [Go54] Golomb, S.W.: Checkerboards and polyominoes. *American Mathematical Monthly* 61, 675–682 (1954)
- [Go65] Golomb, S.W.: *Polyominoes*, Scribners, New York (1965); 2nd edn. Princeton University Press, Princeton (1994)
- [GB65] Golomb, S.W., Baumart, L.D.: Backtrack programming, *J. of the ACM* 12, 516–524 (1965)
- [HH60] Haselgrove, C.B., Haselgrove, J.: A computer program for pentominoes. *Eureka* 23, 16–18 (1960)
- [Ha74] Haselgrove, J.: Packing a square with Y-pentominoes. *J. of Recreational Mathematics* 7, 229 (1974)
- [Kn00] Knuth, D.E.: Dancing links. In: Davies, J., Roscoe, B., Woodcock, J. (eds.) *Millennial Perspectives in Computer Science*, pp. 187–214. Palgrave Macmillan, England (2000), <http://arxiv.org/abs/cs/0011047>
- [Le78] Lewis, H.R.: Complexity of solvable cases of the decision problem for the predicate calculus. In: 19th Ann. Symp. on Foundations of Computer Science, Ann Arbor, MI, pp. 35–47 (1978)
- [Me73] Meeus, J.: Some polyomino and polyiamond problems. *J. of Recreational Mathematics* 6, 215–220 (1973)
- [Sc58] Scott, D.S.: Programming a combinatorial puzzle, Technical Report 1, Dept. of Electrical Engineering, Princeton University (June 1958)

We experimented with many puzzles on these lattices, and report here (see Table 1) on only a few of these puzzles. The *size* of a puzzle is the number of cells it contains. For parts we provide four numbers: the number of original parts, the total number of different oriented parts, the number of oriented parts that can fit into the puzzle, and the total number of options to position oriented parts in the puzzle. (The latter is a good measure of the complexity of a puzzle.) We provide two counts of solutions: essentially different and the total number of solutions. The acronyms BT and MC stand for the direct back-tracking and matrix cover methods, respectively. The ST (stranding) heuristic can be applied to both methods, while SZ (the size heuristic) is relevant to matrix cover only. The notation “X(+Y)” means that applying the heuristic “Y” did not improve the running time relative to using only the method “X.” *Branches* are decision points in which the back-tracking algorithm places a part or the matrix-cover algorithm chooses a column. Finally, we report the times needed to find (on MS Windows) either a single solution or all solutions to the puzzle. Figure 6 shows representative solutions to these puzzles.

As can be easily observed from the data in Table 1, the matrix-cover algorithm, coupled with the dancing-links “trick” and using the branch size-ordering heuristic, is superior to the direct back-tracking algorithm. (Due to the inherent representation of the problem in the two algorithms, we do not have any heuristic for the latter algorithm that is similar to the size heuristic for the former algorithm.) The stranding heuristic can modestly improve both algorithms, but it doesn’t change the superiority of the matrix-cover algorithm.

## 5 Conclusion

We present two puzzle-solving algorithms and identify one of them as the method of choice. We experimented with many puzzles in different lattices. Our future goals are to extend the algorithms to other lattices, add more restrictions to the puzzles (e.g., attaching knobs and holes to the parts), implement “SZ” for the back-tracking method, and improve the efficiency of our implementation.

## Acknowledgment

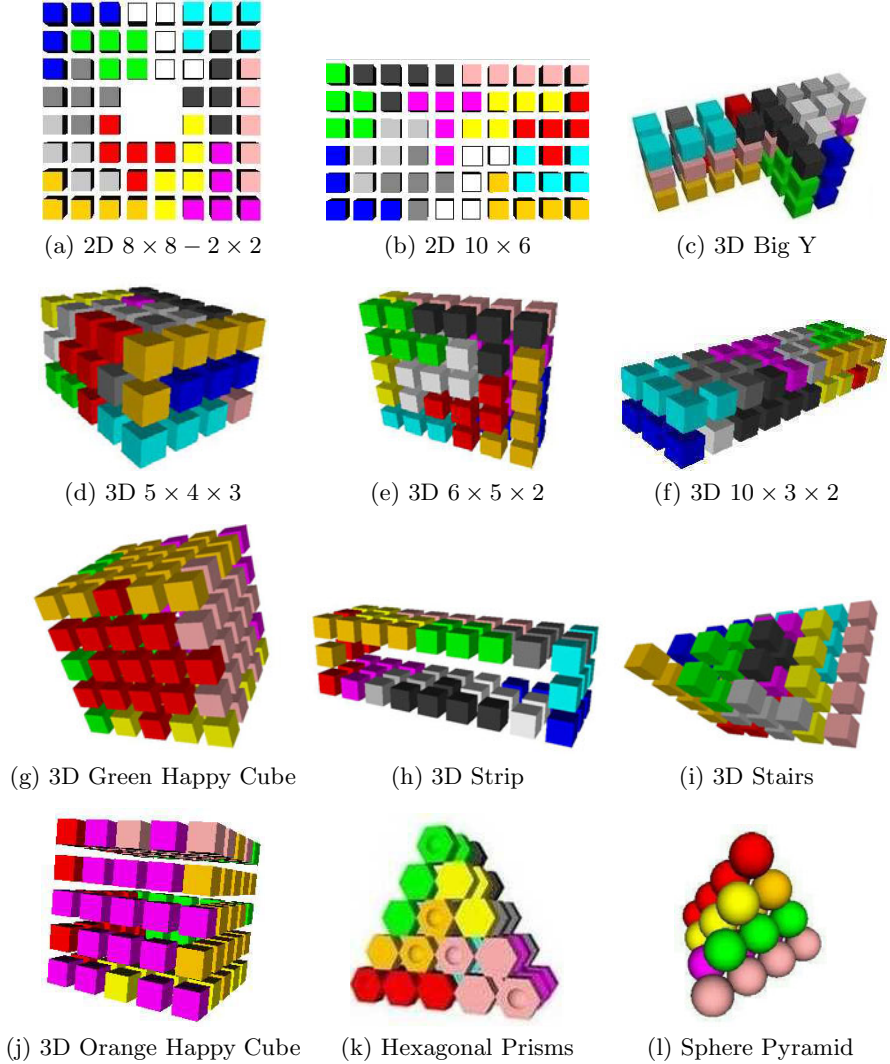
The authors wish to thank an anonymous reviewer and Rudolf Fleischer for many valuable comments on the paper. Part of this research was performed while the first author was on sabbatical at Tufts University, Medford, MA.

## References

- [Br71] De Bruijn, N.G.: Programmeren van de pentomino puzzle. *Euclides* 47, 90–104 (1971–1972)
- [DD07] Demaine, E.D., Demaine, M.L.: Jigsaw puzzles, edge matching, and polyomino packing: Connections and complexity. *Graphs and Combinatorics* 23(suppl.), 195–208 (2007)



three dimensions. Figure 5(a) shows a sample puzzle on the 3D lattice. The third lattice type is the “packed-spheres” lattice. Figure 5(b) shows a sample puzzle on this lattice. In this lattice, the repeated pattern is a node of degree 12. A complete characterization of the group of transformations in this lattice will be provided in the full version of the paper. The fourth lattice type is the “hexagonal-prism” lattice, whose structure is also described in the introduction. Figure 5(c) shows a sample puzzle on this lattice.



**Fig. 6.** Solutions to various puzzles

the user to save the current search state in the direct back-tracking method and to reload it later for a warm-restart of the program.

The first two lattice types which we experimented with were the two- and three-dimensional orthogonal lattices. The formal structure of the 2D lattice is described in the introduction of this paper. It is straightforward to generalize it to

**Table 1.** Statistics of puzzle solving

Lattice	Puzzle		Parts				Solutions		Method & Heuristic(s)	Branching Points	Time (Sec.)	
	Name	Size	Orig.	Total	Reduced	Poses	Unique	Sym.			First	All
2D Ort.	$8 \times 8 - 2 \times 2$	60	12	56	56	1,290	65	*8	BT	2,757,203	0.2	5.7
									BT+ST	1,523,328	0.5	3.5
									MC	2,910,535	0.2	12.4
									MC+ST	2,383,728	0.3	9.8
									MC+SZ	117,723	0.2	1.1
									MC+SZ+ST	81,880	0.1	0.8
2D Ort.	$10 \times 6$	60	12	57	57	1,758	2,339	*4	BT	10,595,621	0.1	17.9
									BT+ST	5,802,074	0.1	14.4
									MC	10,606,305	0.2	54.4
									MC+ST	9,242,529	0.1	43.3
									MC+SZ	1,732,537	0.2	11.8
									MC+SZ+ST	1,480,505	0.1	11.2
3D Ort.	$10 \times 3 \times 2$	60	12	168	68	1,416	12	*8	BT	1,315,930	0.2	2.3
									BT+ST	861,525	0.1	2.0
									MC(+ST)	1,325,957	0.3	6.3
									MC+SZ(+ST)	74,947	0.2	0.7
3D Ort.	$5 \times 4 \times 3$	60	12	168	142	2,168	3,940	*8	BT	1,969,089,192	0.4	5,191
									BT+ST	1,259,189,714	1.1	4,165
									MC	1,969,152,287	0.6	5,874
									MC+ST	1,969,152,287	0.8	5,912
									MC+SZ	10,107,229	0.2	68
									MC+SZ+ST	10,108,090	0.3	65
3D Ort.	$6 \times 5 \times 2$	60	12	168	93	1,916	264	*8	BT	107,590,605	4.8	223
									BT+ST	67,714,344	4.2	190
									MC	107,620,901	1.9	395
									MC+ST	107,620,901	0.8	421
									MC+SZ	759,343	0.2	7.2
									MC+SZ+ST	693,970	0.4	6.7
3D Ort.	Green Happy Cube	98	6	91	91	182	20	*24	BT	1,262	0.2	0.2
									BT+ST	1,215	0.1	0.2
									MC(+ST)	3,318	0.2	0.3
									MC+SZ(+ST)	234	0.0	0.2
3D Ort.	Orange Happy Cube	98	6	79	79	158	2	*24	BT	863	0.1	0.4
									BT+ST	846	0.1	0.3
									MC(+ST)	2,235	0.1	0.3
									MC+SZ(+ST)	146	0.0	0.3
3D Ort.	Strip	60	12	168	85	965	6	*4	BT	187,883	0.0	0.7
									BT+ST	99,272	0.2	0.6
									MC	203,197	0.3	1.7
									MC+ST	188,506	0.1	1.6
									MC+SZ	14,274	0.0	0.4
									MC+SZ+ST	13,904	0.0	0.3
3D Ort.	Stairs	55	12 <sup>a</sup>	186	186	1,573	640	*1	BT	3,143,814	0.1	12.9
									BT+ST	2,088,970	0.1	10.8
									MC	3,143,784	0.1	23
									MC+ST	3,100,112	0.1	18.8
									MC+SZ	178,350	- <sup>a</sup>	1.8
									MC+SZ+ST	159,868	- <sup>a</sup>	1.4
3D Ort.	Big Y	60	12	186	157	1,640	14	*1	BT	210,454,691	52	536
									BT+ST	161,584,456	26	513
									MC	210,502,803	29	529
									MC+ST	210,502,803	2.8	560
									MC+SZ	205,230	0.3	1.6
									MC+SZ+ST	205,524	0.3	1.6
Spheres	Pyramid	20	6	52	31	85	1	*12	BT+(ST)	634	0.0	0.1
									MC+(ST)	1,994	0.1	0.2
									MC+SZ(+ST)	171	0.1	0.1
Hex Prism	Hex prisms	45	11	104	98	232	2	*6	BT	4,675	0.0	0.2
									BT+ST	4,029	0.0	0.2
									MC+(ST)	5,902	0.1	0.2
									MC+SZ(+ST)	422	0.1	0.2

Legend: BT - back tracking; MC - matrix-cover (with dancing links); ST - stranding heuristic; SZ - size heuristic.

<sup>a</sup>In this puzzle one (a priori unknown) part was redundant. This caused the SZ heuristic to not be able to find any solution.

**Dancing Links.** The algorithm can be sped up tremendously by an elegant pointer-manipulation trick [Kn00]. In a doubly-connected linked list, an element  $x$  is removed by executing the two operations  $\text{Next}[\text{Prev}[x]] := \text{Next}[x]$  and  $\text{Prev}[\text{Next}[x]] := \text{Prev}[x]$ . It is a good programming practice not to access the storage of an element  $x$  after it is deleted, since it may already serve a different purpose. However, if it is guaranteed that the area allocated to  $x$  is not altered even after it is deleted, one can easily undo the deletion of  $x$  by making the links “dance”: The pair of operations  $\text{Next}[\text{Prev}[x]] := x$  and  $\text{Prev}[\text{Next}[x]] := x$  readily return  $x$  to its original location in the linked list.

**Efficient Branching (Size Heuristic).** A very efficient heuristic, which reduces the running time significantly, is ordering the branches of the algorithm according to their (anticipated) size. The speed-up of the algorithm naturally depends on the quality of the prediction of the sizes of branches. In our setting, branching occurs at the selection of an additional column. Recall that in a valid solution, exactly one of the rows comprising the submatrix contains the value 1 in this column. (This means that exactly one part covers the puzzle cell corresponding to this column.) Since no a priori information is known (or computed), a reasonable choice is to explore first columns with the *least* number of 1-entries. This is because we will have fewer parts among which to choose the one covering this cell of the puzzle.

## 4 Experimental Results

The two algorithms were implemented in Java on a dual-core 2.2GHz PC with Sun’s Virtual Machine 5+, under the MS Windows XP and Linux operating systems. The two algorithms allowed simple parallelism for using the dual-core CPU by splitting the search tree into two branches. All the running times reported below were measured on MS Windows XP without parallelism. (Using the dual-core CPU reduced the running time by half in practically all cases.) Each puzzle was solved twice (with a random order of parts), and the reported value for each puzzle is the average of the two measurements. The software consists of about 10,000 lines of code. In addition, we implemented a feature which allowed

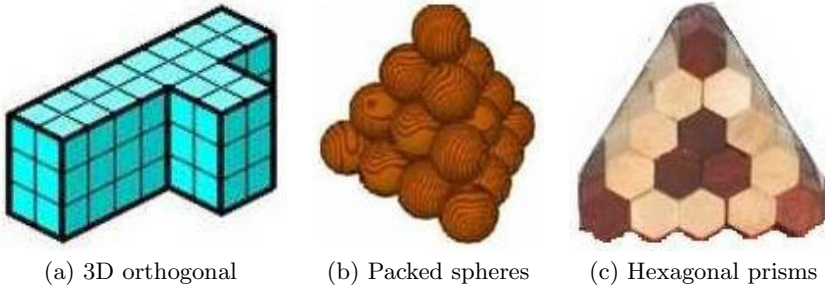


Fig. 5. Lattice puzzles

### 3.2 Matrix Cover

Our second approach to solving the puzzle problem is by a reduction to *matrix cover*, and speeding up the algorithm that solves the latter problem by using the “dancing links” technique [Kn00].

**Reduction.** The puzzle problem is represented by a binary matrix in the following manner. We create an  $M \times N$  matrix, where  $M$  is the total number of options to position parts (in all orientations) in the puzzle, and  $N$  is the number of cells of the graph. By “part-positioning” options we mean all possible mappings of the anchor of a part graph to a node in the puzzle graph, such that the part will partially cover the puzzle and will not exceed its boundary. An entry  $(i, j)$  in the matrix contains the value 1 if the  $j$ th node of the puzzle is covered by some node in the  $i$ th part-positioning option; otherwise it is 0.

Naturally, solving the puzzle amounts to choosing a subset of the rows in which every column contains a single 1 with 0 in all other entries. The chosen rows represent the choice of parts, while the requirement for a single 1 per column guarantees that every cell of the puzzle will be covered by exactly one part.

We *may* want to ensure that every part will be used exactly once, among all its possible orientations and positions in the puzzle. This is easily achieved by adding more columns to the matrix, one for each part. In each such column, we put 1 in all the rows that correspond to the same part, and 0 elsewhere. This guarantees that exactly one part orientation and position is chosen.

For example, consider the simple 3X3 puzzle shown in Figure 4(a). The three puzzle parts are shown in Figure 4(b). The left part has eight different orientations (allowing flipping), in each of which it can be positioned in the puzzle in two ways, for a total of 16 options. The middle part has two different orientations, in each of which it can be positioned in the puzzle in three ways, for a total of 6 options. Finally, the right part has only one orientation, which can be positioned in the puzzle in nine ways. Thus, the matrix we need to cover is made of 31 rows (the total number of part-positioning options) and 12 columns (9 puzzle cells plus three original parts). One possible solution to the puzzle is represented by the 3-row submatrix shown in Figure 4(c): The 9 left columns show the exact covering of the puzzle, while the 3 right columns show that each part is used exactly one.

The matrix-cover algorithm is also back-tracking in nature, and so its details are not provided here. However, its running time is reduced significantly as described below.

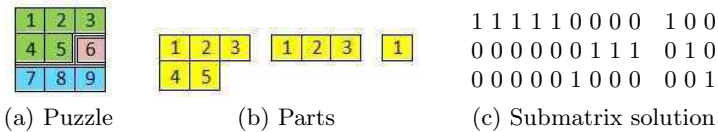


Fig. 4. A puzzle solution represented by a submatrix

graph. A simultaneous and identical (edge-label-wise) traversal is performed in the puzzle graph. This yields precisely which portion of the puzzle is covered by the part. An attempt to position a part in the puzzle fails if either the traversal of the puzzle graph reaches an “occupied” cell, or it gets out of the graph (that is, the boundary of the puzzle is about to be crossed).

If the part-positioning is successful, we mark the part as “used,” mark all the nodes of the puzzle graph that are matched to nodes of the positioned part as “occupied,” and proceed to the next part-positioning step as follows. First, we set the anchor of the puzzle to be the new lexicographically-minimal “empty” node in the puzzle graph. This is achieved by scanning the puzzle lexicographically, starting from the previous anchor and looking for a free node. Then, we attempt to position a new “free” part (according to a predefined order) in the puzzle by identifying its origin with the new anchor and proceeding as above. The new anchor must be the origin of the next positioned part for the same reason as for the first positioned part.

This part-positioning process continues until one of two things happen: Either (a) The puzzle graph is fully covered (this situation is identified by not being able to reset the anchor); or (b) The algorithm is stuck in a situation in which the puzzle graph is not fully covered, yet no new part can be positioned. In the former situation the algorithm declares a solution and terminates. In the latter situation the algorithm back-tracks: The last positioned part is removed from the puzzle, restoring its status to “free” and marking again the covered puzzle nodes as “empty.” Then, if another orientation of the same part exists, the algorithm attempts to position it as above. Otherwise, the algorithm proceeds to the next available part (according to the predefined order) and attempts to position it in the same manner.

In case we like to find *all* solutions to the puzzle, a minor step of the algorithm is modified: When the algorithm finds a solution, the latter is reported, but then refers to the last part-positioning step as a failure and then back-tracks. In such a situation, the algorithm terminates when no back-tracking options remain: This happens when all options for the *first* part-positioning are exhausted.

We conclude the description of the algorithm by referring to the dangling half-edges in part graphs. In principle, they should be matched too to half-edges in the puzzle graph, in order to ensure proper *neighborhoods* between parts positioned in the puzzle. However, this was redundant in all the lattices that we handled. Thus, we ignored all dangling half-edges in the part-positioning operations. A cover of the puzzle graph was actually only an exact cover of the *nodes* of the graph. Edges in the puzzle graph, that represent neighborhood relations between parts, were not covered. Exact match of labels was enforced only between *inter-part* edges and the corresponding edges in the puzzle graph.

**Stranding.** A simple pruning method is to consider the size of the connected component of empty nodes, that include the anchor, in the puzzle graph. If all free parts are larger than this component, then the algorithm may back-track without any further checks. This heuristic can also be applied for the second algorithm described below.

we compute all the possible orientations of each part. Specifically, we apply all possible transformations (and combinations of them) to the half-edges outgoing from the origin cell. Applying a single combination to the origin changes the orientations of all of its neighbors (manifested by new edge labels), and the process continues recursively in a depth-first manner to all the cells of the part.

Naturally, the graph that represents a part may contain cycles, in which case “back-edges” are encountered in the course of the search. Note that the formal definition of a lattice does *not* ensure that back-edges are consistent edge-labeling-wise. Such an inconsistency simply means that a specific transformation (the analogue of a rotation) is not allowed in the dealt-with lattice. However, in all lattices we experimented with, such a situation can never occur; therefore, we did never check for consistency of back-edges.

After all orientations of a part have been found, two steps should be taken: (a) Recomputing the origin cell of each oriented copy; (b) Removing duplicate copies. In fact, both steps can be performed simultaneously. Moreover, the removal of duplicates may be avoided if we precompute the symmetries of the original part. Nevertheless, all these operations are performed in a preprocessing step, before running the main puzzle-solving algorithm (which takes the main bulk of the running time), so any approach will practically do.

**Essentially-Different Solutions.** In most cases we are not interested in finding multiple solutions that are inherently the same, up to some transformation defined for the specific lattice. Instead of computing all the solutions and look for repetitions (an operation which might render the algorithm infeasible if there are too many solutions), we applied a simple pruning method. Suppose that  $S$  is a solution to a puzzle. One only need to observe that if the entire puzzle has some symmetry realized by the transformation  $T$ , then  $T(S)$  is also a solution to the puzzle. Thus, if we discard all copies of an arbitrary part, obtained by transformations that realize symmetries of the puzzle, we guarantee that only essentially-different solutions to the puzzle will be found. To maximize efficiency, we choose the part with the maximum number of copies. (Usually, this is the part with the least number of symmetries.)

**Solving the Puzzle.** Our first method is a classical back-tracking algorithm. We attempt systematically to *cover* the puzzle graph with the part graphs. A part graph covers exactly a portion of the puzzle graph by an injective mapping of the nodes of the part to the nodes of the puzzle, subject to adjacency relations in the two graphs, while the labels of the (half-)edges of the part fully match those of the covered portion in the puzzle.

Initially, all parts are “free,” and all nodes in the puzzle graph are “empty.” We initialize a variable, called the *anchor*, to be the origin cell of the puzzle. Naturally, it should be covered by some cell  $c$  of some part  $p$ . Moreover,  $c$  must be the origin of  $p$ , otherwise, after positioning  $p$  in the puzzle, the origin of  $p$  will occupy a puzzle cell which is different from the origin of the puzzle, which is a contradiction to the lexicographic minimality of the puzzle origin. Thus, positioning a part in the puzzle (in one of its possible orientations) is achieved by identifying the anchor with the origin of the part and traversing the part

Some freedom can be given to the set of parts, so as to form variants of the puzzle problem. We may have a single copy of a given part, or an unlimited amount of copies. Different types of transformations may be defined for different parts of the puzzle.<sup>2</sup> While solving a puzzle, we may want to compute one solution (thus, determine whether or not the puzzle is solvable), or to find the entire set of solutions to the puzzle.

### 3 Algorithms

In the full version of the paper we illustrate a simple reduction from bin-packing (which is known to be an NP-Complete problem [GJ79]) to puzzle solving, showing that the latter is also NP-Complete. The work [DD07] and references therein provide alternative proofs.

In this section we describe two puzzle-solving algorithms: Direct back-tracking and matrix cover. Since both paradigms are well-known, emphasis is put not on these methods but rather on the data structures and heuristics used to expedite the algorithms in the setting of *general-lattice* puzzles.

#### 3.1 Back-Tracking

Our first approach to solving the puzzle problem is by direct back-tracking.

**Puzzle and Part Data Structures.** As mentioned above, the puzzle and parts are represented by graphs. Edges of the graphs are labeled with numbers: for each node  $v$ , the outgoing half-edges of  $v$  are labeled  $0, \dots, d(v) - 1$ , where  $d(v)$  is the degree of  $v$ . Dangling half-edges of both puzzle and parts (that is, half-edges beyond their boundaries), are omitted.

One specific node of each part is marked as the *origin* of the part. For example, in the two-dimensional orthogonal lattice, one may fix the origin at the leftmost cell of the topmost row of the part. Note that the edge labels have to induce a total order on the neighbors of a cell, thereby, on all the cells of the puzzle or its parts. In a general lattice, we simply choose the lexicographically-smallest cell, induced by this order, as the origin of the part. Similarly, we mark the origin of the puzzle. For example, in the two-dimensional orthogonal lattice, a cell is always “smaller” than its bottom and left neighbors, and “greater” than its top and right neighbors. This implies the chosen origin in this lattice.

**Part Orientations.** A special data structure stores the transformations allowed for each part of the puzzle. As mentioned above, a transformation is a 1-1 mapping between the set of possible labels and itself. In a preprocessing step,

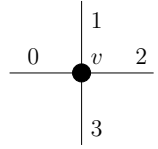
---

<sup>2</sup> For example, consider the *flip* transformation in a two-dimensional orthogonal-lattice puzzle. Its analogue in three dimensions is the *mirror* transformation, which is mathematically well defined but hard to realize with physical puzzle pieces made of wooden cubes. In such a puzzle, whether or not to allow the mirror transformation is a matter of personal taste.

## 2 Statement of the Problem

Let  $\mathcal{L}$  be a *lattice* generated by a repeated pattern. To avoid confusion, let us refer to the dual of  $\mathcal{L}$ , that is, to its adjacency graph, and use interchangeably the terms “cell” (of the lattice) and “node” (of the graph). A “pattern,” in its simplest form, is a single node  $v$  and a set of labeled half-edges adjacent to it. (Each undirected edge  $e$  of the graph is considered as a pair of half-edges, each of which is outgoing from one endpoint of  $e$ .) The half-edges incident to  $v$  are labeled 0 through  $d(v) - 1$ , where  $d(v)$  is the degree of  $v$ . In addition, there is a specification of how copies of  $v$  connect to each other to form  $\mathcal{L}$ . For a 1-node pattern, this specification is simply a pairing of the labels. In addition, there is a set of transformations defined on  $v$ . A transformation is a permutation on the set of labels, i.e., a 1-to-1 mapping between the set  $\{0, \dots, d(v) - 1\}$  and itself.

For example, the orthogonal two-dimensional lattice is generated by the pattern shown in Figure 2. The pattern contains a single node,  $v$ , of degree 4. The four half-edges adjacent to  $v$  are labeled ‘W’=0, ‘N’=1, ‘E’=2, and ‘S’=3, which may be coupled exactly with ‘E,’ ‘S,’ ‘W,’ and ‘N,’ respectively. The pairing specification is, then,  $\{(0, 2), (1, 3)\}$ . Only one transformation, so-called *rotate left* (RL, in short) is defined on  $v$ . The mapping defined by RL on the half-edges incident to  $v$  is  $RL(i) = (i + 1) \bmod 4$ . The composition of one to four RL transformations brings  $v$  to any possible orientation.



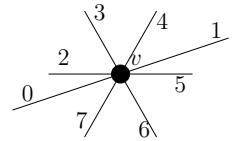
**Fig. 2.**  
Repeated  
pattern of  $\mathbb{Z}^2$

Another example is shown in Figure 3. The repeated pattern of this 3D lattice (see Figure 5(c)) is a prism with a hexagonal cross-section. The degree of the single-node pattern  $v$  is 8. The eight half-edges, with labels 0, ..., 7, are coupled by  $\{(0, 1), (2, 5), (3, 6), (4, 7)\}$ . Two transformations are defined on  $v$ :

(a) An “x-flip”:  $F(0|1|2|3|4|5|6|7) = (1|0|2|7|6|5|4|3)$ ; and

(b) A “yz-rotate”:  $R(0|1|2|3|4|5|6|7) = (0|1|3|4|5|6|7|2)$ .

Compositions of  $F$  and  $R$  bring  $v$  (with repetitions) to all possible orientations in this lattice.



**Fig. 3.** The  
hexagonal-prism  
pattern

A lattice puzzle  $\mathcal{P}$  is a finite subgraph of  $\mathcal{L}$ . Dangling half-edges (that is, half-edges that are *not* coupled with half-edges of other copies of the repeated pattern) are marked as the *boundary* of the puzzle. Puzzle parts are also finite subgraphs of  $\mathcal{L}$ , and they undergo a similar procedure. A solution to the puzzle is a *covering* of  $\mathcal{P}$  by a collection of parts, such that:

1. All nodes of  $\mathcal{P}$  are covered by nodes of the parts;
2. The labels of dangling half-edges of parts match the labels of the respective half-edges of  $\mathcal{P}$ ; except
3. Dangling half-edges of parts may extend out of the boundary of  $\mathcal{P}$ .



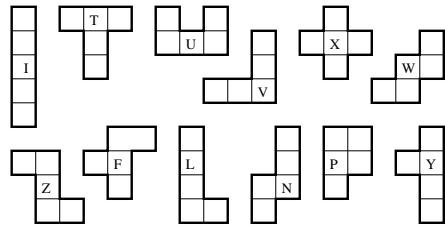
discussed by de Bruijn [Br71]. If pentominoes are not allowed to be flipped upside-down, then there exist 18 such 1-sided pentominoes. Golomb [Go65] provided one tiling of a  $9 \times 10$  rectangle by all these pentominoes, while Meeus [Me73] credited Leech for finding all the 46 tilings of a  $3 \times 30$  rectangle by the set of 1-sided pentominoes. Haselgrove [Ha74] found one of the 212 essentially-different tilings of a  $15 \times 15$  square by 45 copies of the “Y” polyomino. Golomb [Go65] provided many other polyomino puzzles in his seminal book “Polyominoes.” See also [Kn00] for a listing of other well-studied puzzles. The Java applet “Jerard’s Universal Polyomino Solver”<sup>1</sup> is highly optimized for puzzles on a regular orthogonal lattice, and can supposedly solve any puzzle on this lattice.

It is well-known that finite-puzzle problems on an orthogonal lattice are NP-Complete. This is usually shown by a reduction from the Wang tiling problem, which is also known to be NP-Complete [Le78]. It is not surprising, then, that no better method than back-tracking is used for solving puzzles. Many works, e.g., [Sc58, GB65, Br71], suggested heuristics for speeding-up the process, usually by taking first steps with the least number of branches (possible next steps).

Knuth [Kn00] suggested the *dancing links* method, which allowed solving several problems, including orthogonal and triangular lattice puzzles, much more efficiently than before. The main idea is a combination of a link-handling “trick” that enables easy and efficient “unremove” operations of an object from a doubly-connected list (a key step in back-tracking), and an abstract representation of the problems as a matrix-cover problem: Given a 0/1-matrix, choose a subset of its rows such that each column of the shrunk matrix contains exactly one ‘1’ entry. The reader is referred to the cited reference for more details about this extremely elegant method.

In this paper we present two back-tracking approaches to solving *general lattice* puzzles. We formulate the puzzle problem in general terms, so that it would fit various types of puzzles. Since solving a puzzle even on a two-dimensional orthogonal lattice is NP-Complete, we cannot hope (unless  $P=NP$ ) for subexponential algorithms for the problem. We fully implemented two algorithms and a few heuristics to expedite them, and ran them on many puzzle problems which lie on several types of lattices.

This paper is organized as follows. In Section 2 we define the puzzle-solving problem. In Section 3 we describe two algorithms to solve it, and present in Section 4 our experimental results. We end in Section 5 with some concluding remarks.



**Fig. 1.** Twelve essentially-different 2-dimensional pentominoes

<sup>1</sup> Available at <http://www.xs4all.nl/~gp/Site/Polyomino-Solver.html>

# Solving General Lattice Puzzles

Gill Barequet<sup>1</sup> and Shahar Tal<sup>2</sup>

<sup>1</sup> Center for Graphics and Geometric Computing  
Dept. of Computer Science  
Technion—Israel Institute of Technology  
Haifa 32000, Israel

`barequet@cs.technion.ac.il`




<sup>2</sup> Dept. of Computer Science  
The Open University  
Raanana, Israel  
`shahar_t@hotmail.com`

**Abstract.** In this paper we describe implementations of two general methods for solving puzzles on *any* structured lattice. We define the puzzle as a graph induced by (finite portion of) the lattice, and apply a back-tracking method for iteratively find all solutions by identifying parts of the puzzle (or transformed versions of them) with subgraphs of the puzzle, such that the entire puzzle graph is covered without overlaps by the graphs of the parts. Alternatively, we reduce the puzzle problem to a submatrix-selection problem, and solve the latter problem by using the “dancing-links” trick of Knuth. A few expediting heuristics are discussed, and experimental results on various lattice puzzles are presented.

**Keywords:** Polyominoes, polycubes.

## 1 Introduction

Lattice puzzles intrigued the imagination of “problem solvers” along many generations. Probably the most popular lattice is the two-dimensional orthogonal lattice, in which puzzle parts are so-called “polyominoes” (edge-connected sets of squares), and the puzzle is a container which should be fully covered without overlaps by translated, rotated, and, possibly, also flipped versions of the parts.

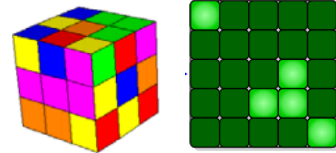
One very popular set of parts is the 12 “pentominoes” (5-square polyominoes), which is shown in Figure 1. We describe here a few examples of the many works on pentomino puzzles. Such games were already discussed in the mid 1950’s by Golomb [Go54] and  Gardner [Ga57]. Scott [Sc58] found all 65 essentially-different (up to symmetries) solutions of the  $8 \times 8$  puzzle excluding the central  $2 \times 2$  square.  Covering the entire  $8 \times 8$  square  with the twelve pentominoes and one additional  $2 \times 2$  square part, without insisting on the location of the latter part, is the classic Dudeney’s puzzle [Du08]. The Haselgrove couple [HH60], as well as Fletcher [Fl65], computed the 2,339 essentially-different solutions to the  $6 \times 10$  pentomino puzzle. Other pentomino puzzles were





## ***Abstract***

In this thesis we describe general methods for solving puzzles on any *structured* lattice. The puzzles are of a "put together" type, compared to combinatorial puzzles like the Rubik's Cube, in which an operation on 'part' of the puzzle bring it to a new permutation.




We define the puzzle as a graph induced by the lattice, and apply a back-tracking method for iteratively finding all solutions by identifying parts of the puzzle (or transformed versions of them) with sub-graphs of the puzzle, such that the entire puzzle graph is covered without overlaps by the graphs of the puzzle parts.

Alternatively, we reduce the puzzle problem to a sub-matrix-selection problem, and solve it using the "dancing-links" technique as described by Knuth [Kn00].

Experimental results on various lattice puzzles are presented.

Here is an example of a puzzle, on a 2-dimensinal orthogonal lattice, made of 12 Pentominoes. The cell (node) is a single square, which can be flipped and rotated. Its neighboring squares are those lying on its North, South, East and West.



A complex example is the well-known Tangram set. The lattice consists of small triangles with 8 different orientations. The basic cell  is marked on the target. This cell does not exist in the set, but in fact, each part is composed of these basic cells (but with different orientations).

This is a rather complex example, usually we deal with puzzles assembly.



## ***Table of content***

Abstract	pg. 84
Preface	9
Statement of the problem	11
NP complete	15
Algorithms - Presentation	17
Part Orientations	19
The Unique Part	21
Different Solutions	23
Solving the Puzzle – Back Track	25
Matrix Cover	29
Heuristics: Size and Stranded	33
Improvements	35
Keys and Locks	37
Experimental results	39
Conclusions	45
References	47
Appendix A. Grids	49
Appendix B. Operation Guide	53
Appendix C. Code Reference	67
Appendix D. Paper – published at FAW 2010	82



**The Open University of Israel**

**Department of Mathematics and Computer Science**

# **Solving General Put-Together Puzzles**

Thesis submitted as partial fulfillment of the requirements  
towards an M.Sc. degree in Computer Science  
The Open University of Israel  
Computer Science Division

**By**

**Shahar Tal**

Prepared under the supervision of

Prof. Gill Barequet, The Technion

Dr. Jack Weinstein, The Open University

**September 2010**