סיכום פרוייקט

פתרון בעיות תכנון (Planning) ויישומן בעולם הקוביות (blocks-world)

קורס: מבוא לבינה מלאכותית 67842

מגישים:
מאיר שפילריין 302749601
מיטל צוק 301850285

בעיות תכנון ועולם הקוביות

תחום התכנון הוא אחד הנושאים החשובים והעיקריים בבינה מלאכותית. בעיה כזו מאופיינת על ידי מצב התחלתי, מטרה (במילים אחרות מצב סופי) ופעולות הניתנות לביצוע על ידי הסוכן.
המתכנן מקבל על השלשה הזו ומוציא כפלט תכנון- רצף הפעולות הנדרש להשגת המצב הסופי מהמצב ההתחלתי. הפעולות הנ"ל מורכבות מכמה דברים: שם הפעולה, רשימת תנאי הקדם, רשימת העובדות המתבטלות לאחר ביצוע פעולה ורשימה של העובדות שמתווספות.

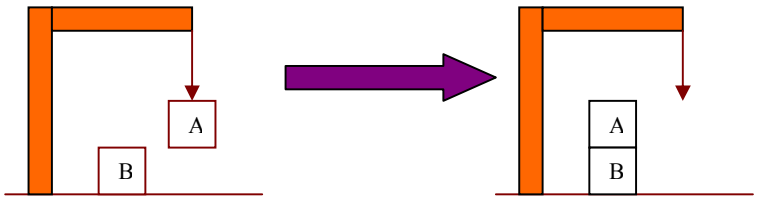
אנו בחרנו להתעסק עם בעיית הקוביות אשר מורכבת ממספר סופי של קוביות, זרוע ושולחן (אשר גדול מספיק כדי להחזיק את כמות קוביות לפי רצוננו).
המצבים שבהם יכולה להימצא קוביה: על השולחן, על קוביה אחרת, בנוסף יכולה להיות חופשייה (משמע אין עליה אף קובייה אחרת).

מצב בבעיה זו מוגדר כאוסף של עובדות שיכולות להיות:

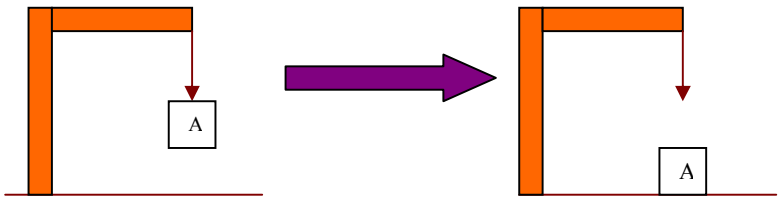
BWFreeFact- הקוביה הנ"ל פנויה (אין קוביה אחרת מעליה)
BWOnFact - הקוביה הנ"ל נמצאת מעל לקוביה אחרת.
BWOnTableFact - הקוביה הנ"ל מונחת על השולחן.

אנו יכולים לעשות את הפעולות הבאות:

להניח קוביה A נוכחית על קוביה B אחרת.
תנאי קדם: קוביה B פנויה, וקוביה A מוחזקת ע"י הזרוע.
עובדות נמחקות: קוביה B פנויה, והזרוע תפוסה.
עובדות חדשות: כעת קוביה B כבר לא פנויה, קוביה A על קוביה B, כמו כן, הזרוע כרגע פנויה, וקוביה A פנויה.

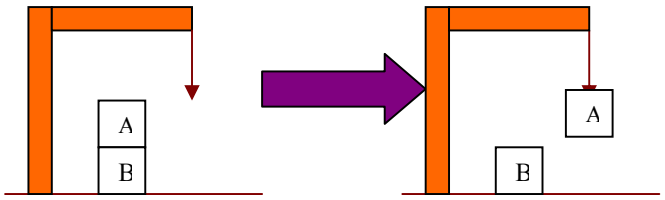


להניח קוביה A על השולחן:
תנאי קדם: קוביה A מוחזקת על ידי הזרוע.
עובדות נמחקות: זרוע תפוסה.
עובדות חדשות: קוביה A על השולחן, קוביה A פנויה.

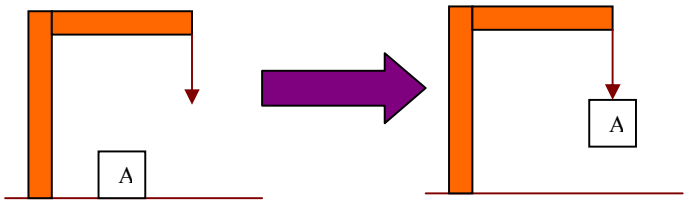


הרם קוביה A :
תנאי קדם: הזרוע פנויה וכמובן שקוביה A תהיה חופשיה.
עובדות שנמחקות: הזרוע פנויה.
עובדות נוספות: הזרוע מחזיקה את קוביה A, אם הקוביה הייתה על אחרת, קוביה זו כעת תהיה חופשיה.

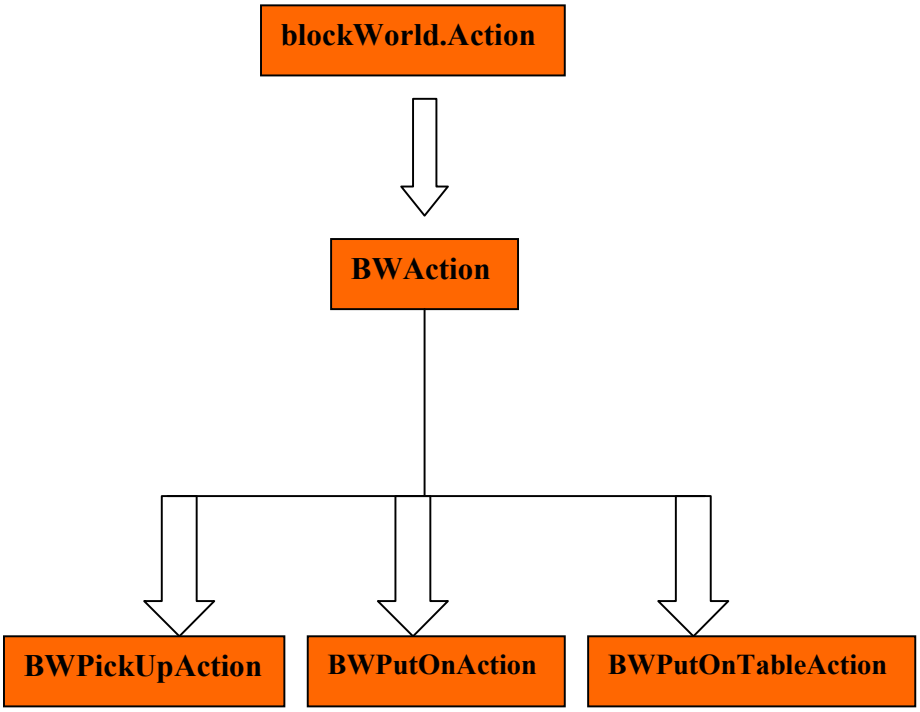
.1



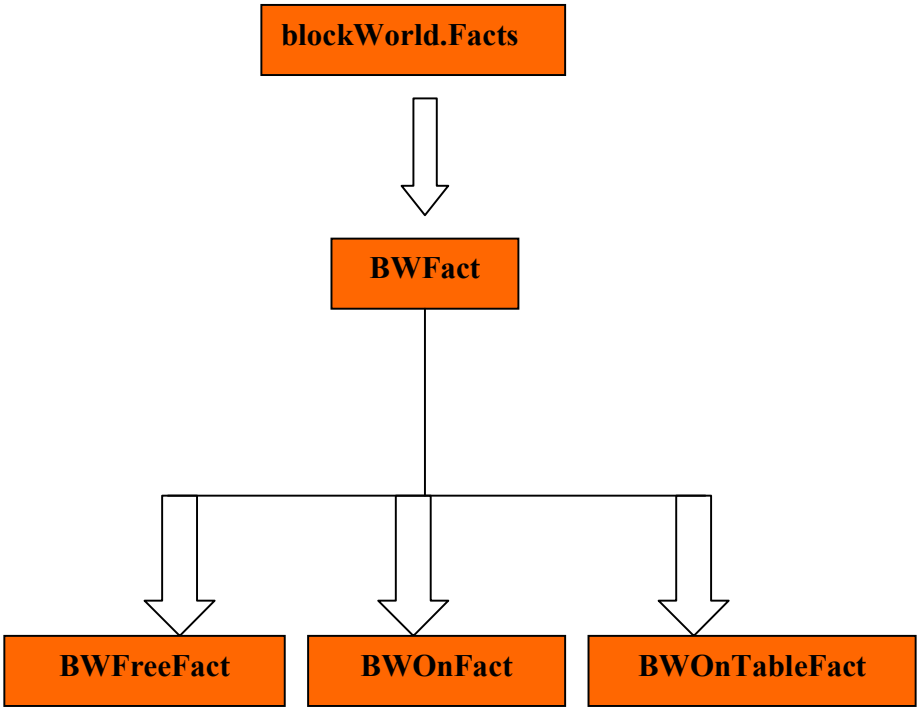
.2



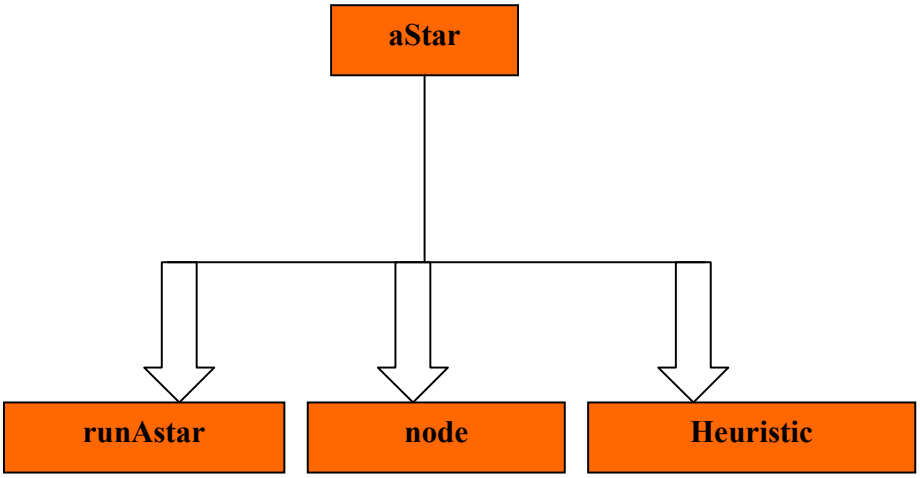
הארכיטקטורה:



blockWorld.Action - מתאר את הפעולות שאפשר לעשות על קוביה.
BWAction - Abstract class - המתאר את הפעולה. כל פעולה תממש אותו.
BWPickUpAction - זוהי פעולה של הרמת הקוביה על ידי הזרוע.
BWPutOnAction - זוהי פעולה של הנחת הקוביה על קוביה אחרת.
BWPutOnTableAction - זוהי פעולה של הנחת הקוביה על השולחן.



blockWorld.Action- מתאר את הפעולות שאפשר לעשות על קוביה.
BWFact - Abstract class - המתאר את העובדה. כל עובדה תממש אותו.
BWFreeFact - זוהי עובדה שאומרת שעל הקוביה הנוכחית אין אף קוביה אחרת.
BWOnFact - זוהי עובדה שאומרת שהקוביה הנוכחית נמצאת על קוביה אחרת.
BWOnTableFact - זוהי עובדה שאומרת שהקוביה הנוכחית נמצאת על השולחן.



aStar- אלגוריתם החיפוש עליו נפרט בהמשך.
runAstar - מריץ את האלגוריתם.
node - abstract class - כל מצב שנרצה להשתמש בו בתוכנית יצטרך לממש אותו.
BWState מממש אותו.
Heuristic- abstract class - כל יוריסטיקה צריכה לממש אותו.

אלגוריתם החיפוש - A^*

זהו אלגוריתם חיפוש שבהנתן גרף, מצב התחלתי ומטרה, מוצא את המסלול הזול ביותר שמתחיל במצב ההתחלתי ומסתיים במטרה הנתונה. את המסלול הנ"ל האלגוריתם מוצא באמצעות פתיחת קודקודים שמהם אנו עשויים להגיע ביתר קלות אל המטרה. בכל שלב, הוא בוחר לבקר בקודקוד שהכי סביר שממנו נגיע למטרה, והקודקוד שנבחר מהווה את המסלול הזול ביותר כרגע המוביל אותנו הכי קרוב למטרה.
על מנת לבחור את הקודקוד הבא שאליו נעבור, משתמשים בפונקציה יוריסטית h(x), שכל מטרתה היא לדרג את הקודקודים באופן שבו קודקוד עם דירוג גבוה יותר יהיה קודקוד בעל סבירות גבוהה יותר שיוביל אותנו אל המטרה דרך מסלול קצר יחסית.לערך הפונקציה הנ"ל הוא מוסיף את המרחק של הקודקוד x עם המצב ההתחלתי (את המרחק נסמן ב- g(x) ואת הסכום ב- f(x).

האלגוריתם מחזיק תור של קודקודים שממוין על פי ערך ה- f(x) כאשר לאורך הריצה הוא בוחר לעבור אל הקודקוד עם ערך מינימאלי שלו. את הקודקודים שעדיין לא ביקרנו בהם הוא מחזיק בתור ממוין שנקרא unvisited. כאשר האלגוריתם עובר אל קודקוד חדש הוא בודק האם הגענו למטרה, אם כן – עוצר, אחרת מוסיף את כל הבנים ל-unvisited ומחשב את שלושת הערכים הנ"ל עבור על אחד מהקודקודים. באופן הזה, אם הפונקציה היוריסטית היא admissible (כלומר לא משערכת יתר על המידה את המרחק מהקודקוד אל המטרה) מובטח לנו שהמסלול שיחזיר האלגוריתם הזה (אם קיים פתרון) הוא המסלול האופטימלי המוביל אל המטרה.

אנו מימשנו כמה יוריסטיקות שיפורטו בהמשך.

בגישה זו, החיפוש, ייצגנו מצב של קוביות במישור כקודקוד בגרף שאת הקשתות שלו הגדרנו באופן הבא: עבור קודקוד x ו y הקשת (x,y) שייכת לקבוצת הקשתות בגרף אמ"מ ניתן להגיע מהמצב של x למצב של y (בעולם הקוביות) ע"י פעולה אחת שתנאי הקדם שלה מתקיימים במצב x . בתנאים הנ"ל נאמר כי המצב של y ישיג מהמצב של x. בנוסף נזכיר כי כל קודקוד בגרף (כלומר מצב מסוים של הקוביות במישור) מיוצג במימוש שלנו ע"י אוסף של עובדות.

כעת, את אלגוריתם החיפוש הרצנו על גרף המצבים הנ"ל כאשר תחילת הריצה היא בקודקוד שמייצג את המצב ההתחלתי והמטרה היא הקודקוד שמייצג את המצב הסופי בעולם הקוביות. כמובן שמעבר מקודקוד לקודקוד מתאפשר רק אם יש מסלול שמעביר אותנו בעולם הקוביות.

הפונקציות היוריסטיות:

MisssingHeuristic

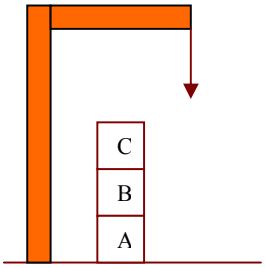
זוהי יוריסטיקה שמקבלת מצב נוכחי ואת המצב הסופי ומחשבת כמה עובדות נמצאות במצב הסופי שלא נמצאות במצב הנוכחי. היא תספור כל עובדה כזאת ותסכום את מספר העובדות החסרות. לפי זה נעריך כמה צעדים יש לנו להשגת המטרה.

FactDistanceHeuristic

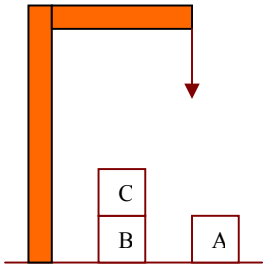
זוהי יוריסטיקה שמקבלת מצב נוכחי ואת המצב הסופי ומחשבת לכל קוביה שלא נמצאת באותו מקום, כמה קוביות יש מעליה, מה שאומר כמה קוביות נצטרך להזיז כדי להגיע למיקום במצב הסופי.
בהגיענו לקוביה שלא נמצאת באותו מיקום במצב הסופי, ובמצב הנוכחי יש לפחות קוביה אחת מעליה, נוסיף 2 למספר הצעדים הכולל. מה שאומר שגם הקוביה שנמצאת מעליה לא תהיה באותו מיקום, לכן אם יש עליה גם קוביה נוסיף עוד 2 צעדים. וכך הלאה.
אם היינו מחשיבים כל קוביה בנפרד קיימת האפשרות שזאת לא תהיה admissible function, לכן מימשנו זאת כך.

דוגמא:

המצב הנוכחי:



המצב הסופי:

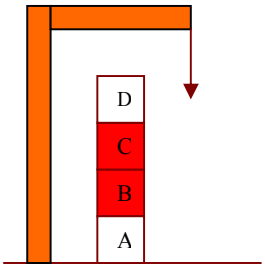


במקרה זה אפשר לראות שבמצב הסופי קוביה A לא אמורה להיות באותו מיקום במצב הסופי. מה שאומר שכדי להזיזה יש לעשות שני צעדים להזזת B שנמצאת ממש מעליה (הרמת B והנחת B). כמובן ש-B לא נמצאת באותו מיקום (שכן היא אמורה להיות מונחת על השולחן) לכן כדי להזיזה יש להרים את C ולהניח אותה במקום אחר. מכאן שנוסיף עוד שני צעדים.

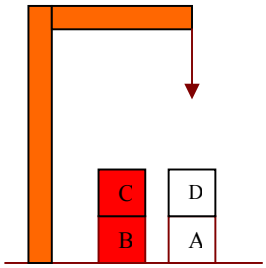
ReduceProblemHeuristic

זוהי פונקציה המקבלת מצב סופי ומצב נוכחי ומחפשת קוביות שנמצאות אחת על השנייה גם במצב הנוכחי וגם במצב הסופי (מגדל). למשל ש-A נמצאת מעל B.
פונקציה זו גורמת להם להיות כקוביה אחת ובכך עוזרת לנו בפתרון של בעיות קטנות יותר, בהתייחסות אליהם כקוביה אחת.
את הבעיה היותר קטנה היא פותרת באמצעות היוריסטיקה הקודמת שהסברנו. הקטנת הבעיה התבצעה על ידי כך שהורדנו את הקוביות שנמצאות מעל המגדל להיות מעל הקוביה התחתונה ביותר במגדל.

המצב הנוכחי:

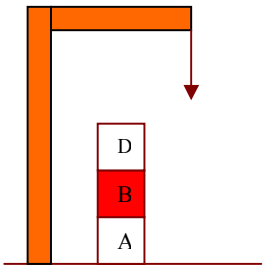


המצב הסופי:

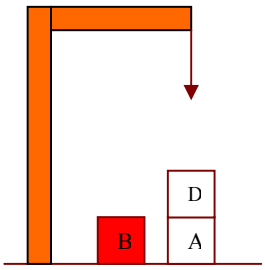


ניתן לראות ש-C נמצאת מעל B גם במצב הנוכחי וגם במצב הסופי (המגדל מסומן באדום). לכן נתייחס אליהם כאל קוביה אחת הנקראת B ויווצר המצב הבא:

המצב ההתחלתי:



המצב הסופי:



כעת נפתור זאת כפי שאנחנו פותרים עם היוריסטיקה הקודמת בעיה של 3 קוביות.

השוואה בין היוריסטיקות

השערות:

השוואה על פי מספר איטרציות:

היוריסטיקה שעובדת הכי פחות טוב מבחינה זו היא BWMissingHeuristic מכיוון שהיא נותנת הערכה מאוד לא מדוייקת יחסית למרחק האמיתי.

לדוגמא:

מצב התחלתי: מצב סופי:

ניתן לראות שהיוריסטיקה תתן 1 (B אמור להיות על השולחן במצב הסופי, כל שאר הדברים נשארים אותו דבר). אך לעומת זאת, המצב האמיתי הוא שאנו צריכים לעשות 6 פעולות כדי להגיע למצב הסופי:

להרים את C
להניח את Cעל השולחן
להרים את B
להניח את B על השולחן
להרים את C
להניח את C על B

אנו מעריכים שהפונקציה BWFactDistHeuristic תפעל יותר טוב מהקודמת אך פחות טוב מהשלישית. משום שההערכה על המרחק הרבה יותר טובה וקרובה למרחק האמיתי. למשל בדוגמא הקודמת, היורסטיקה תתן 4 (להזזת B תתן 2, אך כדי להזיז את B, יש להזיז את C שזה עוד 2). ניתן לראות שזה כבר יותר קרוב לאמת מאשר היוריסטיקה הקודמת.

היוריסטיקה השלישית שלנו היא BWReduceProblemHeuristic שהיא הטובה ביותר מבחינת מספר איטרציות. היא מדוייקת יותר משאר היוריסטיקות. למשל בדוגמא שלנו הפונקציה תחבר את B ו-C לקוביה אחת, אבל ברגע שנזיז את הקוביה הזו נחשיב זאת כ-6 צעדים. מה שבמקרה הזה יהיה בדיוק אותו מספר צעדים.

השוואה על פי זמן ריצה:

BWReduceProblemHeuristic תרוץ הכי לאט, כי היא פותרת המון בעיות קלות יותר מהבעיה המקורית אבל עדיין פתרון הבעיות הקלות יכול לקחת הרבה זמן. יש לציין שניסינו להקטין את זמן הריצה ע"י פתרון הבעיות הקטנות לפני החיפוש, ושמירתן בזכרון, אך לא היה לנו מספיק זיכרון כדי לשמור את כל הבעיות. חשבנו להשתמש בdata base אך מפאת חוסר זמן לא עשינו זאת. אנו חושבים שדבר זה יכול להיות מעניין בניסוי עתידי.

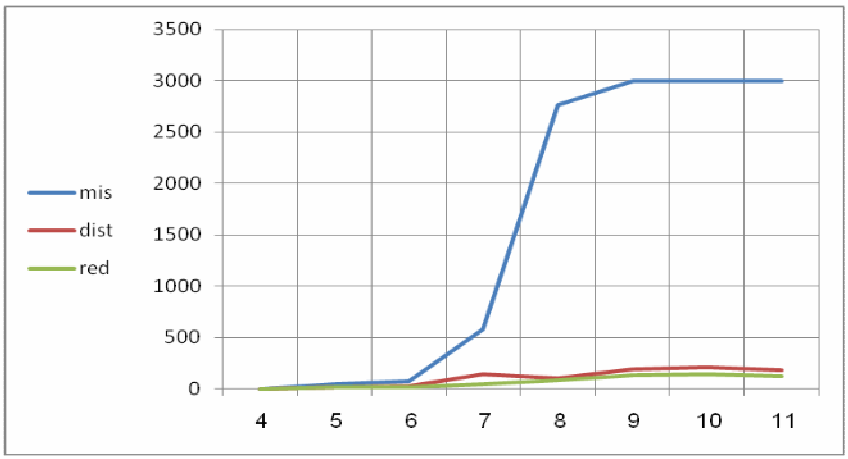
מבחינת שתי היוריסטיקות האחרות, קשה לקבוע איזה מהן תרוץ מהר יותר, מכיוון ששתיהם בסופו של דבר תרוץ על רשימת העובדות ולכן זה יהיה פחות או יותר אותו דבר, עם נטייה קלה ל BWMissingFact כי יש שם פחות חישובים.
כמובן שכל אלה הם הזמן ריצה שלוקח לחישוב היוריסטיקה עצמה כאשר יש הרבה קוביות זמן הריצה תלוי גם במספר הפעמים שמחשבים את היוריסטיקה כלומר ב node expanded
במקרה כזה קשה להעריך מה ירוץ הכי מהר והכי קל זה פשוט לבדוק זאת

מבחינת פתרון אופטימלי, כולן יחזירו פתרון אופטימלי כי כפי שציינו הם admissible functions.

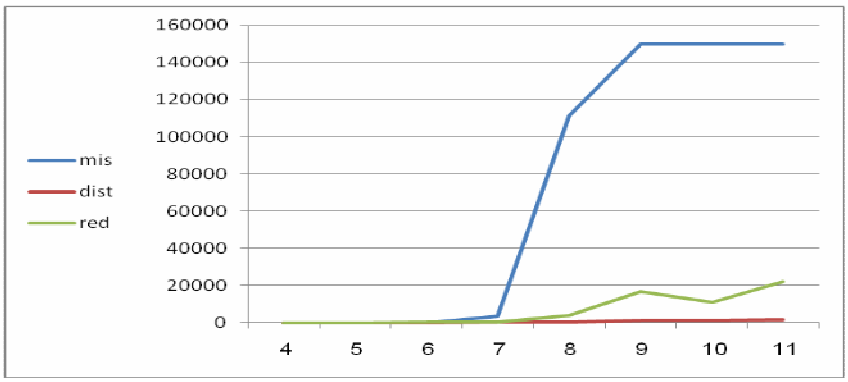
תוצאות בשטח

התוצאות שיצאו לנו בבדיקות מוכיחות את אשר הנחנו למעלה

גרף מספר איטרציות למספר קוביות



גרף זמן ריצה למספר קוביות

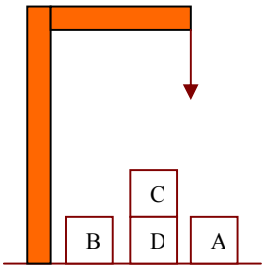


The Gridy algorithm

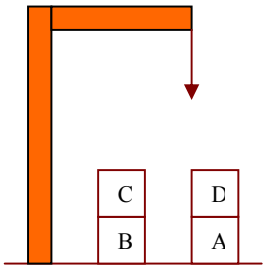
האלגוריתם הגרידי מטפל מגדל מגדל. בטיפול במגדל מסויים הוא יסיים לבנות אותו ורק אז יעבור למגדל הבא. הוא יעשה את כל הצעדים הדרושים גם אם זה אומר לפרק מגדלים אחרים שכמעט בנויים.
הוא לא אלגוריתם אופטימלי, הוא מוצא פיתרון לא אופטימלי. בחלק מהמקרים הוא אפילו יגיע למצב שהוא שם את כל הקוביות על השולחן ומתחיל לבנות מחדש את הבניינים.

יתרון על פני ה*A: הגרידי הרבה יותר מהיר (כי אין צורך במציאת מסלול מהיר ביותר בגרף המצבים אלא בבנייה בלבד) ומסוגל לפתור בעיות של- *A יקח זמן לא רציונלי לפתור אותן.
חסרון: הפתרון הוא לא אופטימלי. לדוגמא:

מצב התחלתי:



מצב סופי:



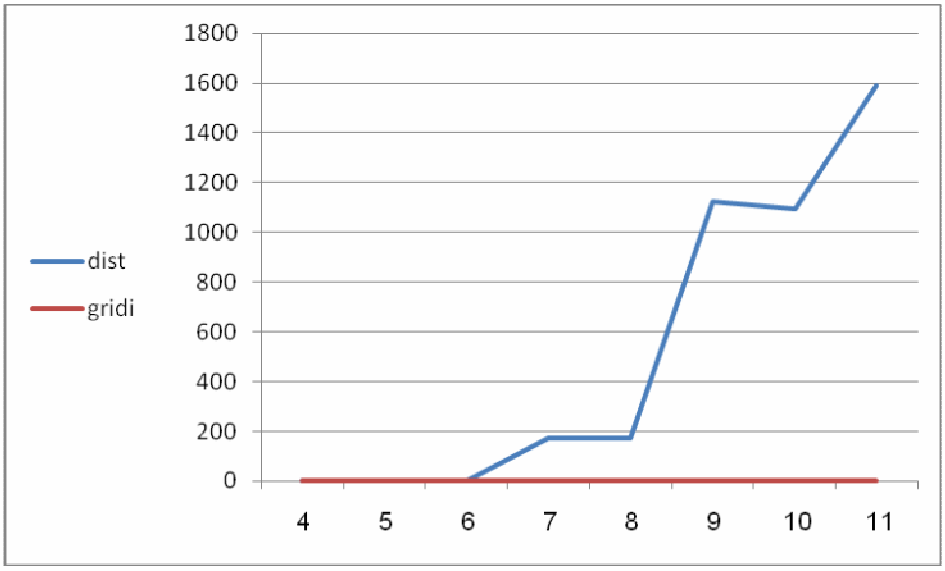
הפתרון שהגרידי יביא הוא:
להרים את C
להניח את C על השולחן
להרים את D
להניח את D על A
להרים את C
להניח את C על B

לעומת זאת, הפתרון האופטימלי הוא:
להרים את C
להניח את C על B
להרים את D
להניח את D על A

הערה: ברור שאפשר לעשות אופטימזציות שבמקרה זה האלגוריתם יביא פתרון אופטימלי, אך לא ניתן לעשות זאת עבור כל מקרה.

<u>גרף זמן ריצה למספר קוביות</u>

(השוואה בין זמן הריצה של האלגוריתם הגרידי לזמן הריצה של *A עם היוריסטיקה הכי מהירה שיש)



The hill climbing algorithm

ניסיון נוסף שלנו היה לממש את האלגוריתם הנ"ל. הרעיון הוא לתת לכל עובדה שקיימת במצב הסופי 1+ ועל כל עובדה שלא קיימת 1-. כמובן שהמקסימום יהיה כאשר נגיע למצב הסופי. לכן נבחר את המצב הבא אליו הולכים כך שערכו יהיה הגדול ביותר מכל השכנים.
אחרי הרבה ניסיונות הגענו למסקנה שזה לא יעיל ולא עובד. מכיוון שאם נותנים לאלגוריתם לטעות הרבה כלומר לבחור צעדים לא טובים אז מקבלים פתרון ארוך מאוד. ואם נותנים לו לטעות מעט אז הוא נתקע לזמנים ארוכים ולא מסיים.
גם ניסיון לשנות את פונקציית הגובה לאחת היוריסטיקות ב- *A לא הצליח, אמנם התוצאות ישתפרו מעט אך עדיין לא מספיק טובות.
הגענו למסקנה שהאלגוריתם לא מתאים לבעיה זו. ההשערה היא שהאלגוריתם לא מצליח להתמודד עם מצבים סימטריים וחוזר על מצבים שוב ושוב.

<u>סיכום:</u>

אם רוצים למצוא פתרון אופטימלי לבעיית הקוביות נשתמש ב *A עם היוריסטיקה distFacyHur אך מכיוון שזמן הריצה של *A הוא אקספוננציאלי לא יהיה רלוונטי להשתמש בוא על מספר רב של קוביות ולכן במקרה כזה נשתמש באלגוריתם הגרידי כי נעדיף לפחות למצוא פתרון אפילו אם הוא לא אופטימלי.