חיפושים לא ממושקלים:

BFS – חיפוש לרוחב:
שלם – כן, כיוון שאם d זה רמת הפתרון, BFS לא יפתח קודקוד ברמה נמוכה יותר מרמה d לפני שהוא פיתח את כל הקודקודים ברמה זו לכן אם קיים פתרון ברמה d הוא יעצור שם לפני שימשיך לרמה הבאה.
אופטימלי – כן, הוא יגיע לפתרון שנמצא ברמה הגבוהה ביותר קודם, כיוון שהוא עובר לרוחב , רמה אחרי רמה, ולכן הפתרון הראשון שימצא הוא האופטימלי.
סיבוכיות זמן - $\mathrm{O}x(1+b+\ldots+b^d)=O(b^d)$
סיבוכיות מקום - $O(b^d)$

DFS – חיפוש לעומק:
שלם – לא, יכול להתקדם על ענף איסופי או להיתקע בלולאה.
אופטימלי – לא, עובר לעומק ויכול להיות שמצא פתרון למרות שקיים פתרון על ענף אחר בעומק נמוך יותר.
סיבוכיות זמן - $O(b^m)$ כאשר m זה עומק הפתרון שנמצא.
סיבוכיות מקום – $O(bm)$ במימוש עם מחסנית, $O(m)$ במימוש רקורסיבי.

:Limited DFS
שלם – כן אם $d\le l$ (d-הרמה שבה נמצא הפתרון, l-הגבול)
אופטימלי - לא
סיבוכיות זמן – $O(b^l)$
סיבוכיות מקום - $O(bl)$

DFID (סימלוץ BFS ע"י DFS):
שלם – כן, מכיוון שהוא מפתח את כל הקודקודים עד הcutoff ותמיד יגיע לתוצאה.
אופטימלי – כן, כמו BFS.
סיבוכיות זמן – $O((d)b+(d-1)b^2+\cdots+(1)b^d)=\ O(b^d)$
סיבוכיות מקום - $O(bd)$

חיפושים ממושקלים:

:UCS
שלם – כן, רק אם קיים חסם תחתון חיובי e לאורכי צלעות, אחרת יכול להיתקע על ענף שאורכו שואף ל1.
אופטימלי – כן, כי כאשר האלגוריתם מפתח קודקוד בהכרח נמצא המסלול הקצר ביותר לקודקוד ובנוסף אורך המסלול לא מתקצר כי משקלי הצלעות הם אי שליליים ולכן האלגוריתם מפתח קודקודים בסדר עולה לפי העלות שלהם.
סיבוכיות זמן - $\mathrm{O}(b^{\lfloor c*/e\rfloor+1})$, בהינתן ש *c היא העלות של הפתרון האופטימלי חלקי העלות המינימלית לצלע + 1 כי ייתכן שנפתח את כל הרמה האחרונה עד שנגיע ליעד.
סיבוכיות מקום - $\mathrm{O}(b^{\lfloor c*/e\rfloor+1})$.

Greedy Search – מימוש של UCS כאשר $\boldsymbol{f(n)=h(n)}$:
שלם – לא כי יכול להיתקע בענף אינסופי או בלולאה אינסופית בגלל פונקציה יוריסטית לא טובה.
אופטימלי – לא, האלגוריתם מתעלם מעלות הצלעות.
סיבוכיות זמן - $\mathrm{O}(b^m)$.
סיבוכיות מקום - $\mathrm{O}(bm)$.

A* - מימוש של UCS כאשר $f(n) = g(n) + h(n)$:
שלם – כן, אם הגרף סופי. אם הגרף אינסופי, יהיה שלם בתנאי שעלות הצלעות סופית ובעלי ערך מינימלי חיובי ובנוסף הערכים היוריסטיים סופיים ואי שליליים.
כיוון שעלות כל קשת היא לפחות e אז נגיע לקודקוד הgoal x בלכל היותר x/e צלעות.
אופטימלי – כן, רק אם הפונקציה היוריסטית היא consistent ולכן $h(n)$ לעולם לא תבצע הערכת יתר (admissible $\rightarrow h(n) \leq h^*(n)$) ובנוסף $h(n) \leq c(n,m) + h(m)$.
סיבוכיות זמן – תלוי בפונקציה היוריסטית:
<u>במקרה הגרוע</u> $f(n) = g(n)$ משמע $h(n) = 0$, ולכן זמן הריצה יהיה $O(b^{\frac{c^*}{e}+1})$ בדומה לUCS.
<u>במקרה הטוב</u> $f(n) = g(n) + h^*(n)$ ז"א שהפונקציה היוריסטית תיתן הערכה מדויקת לכל קודקוד, ולכן האלגוריתם ידע את המסלול האופטימלי ופשוט יתקדם עליו ונקבל זמן ריצה $O(bd)$.
סיבוכיות מקום – כל הקודקודים נשמרים בclosed list ולכן אקספוננציאלי ומוגבל בזיכרון.

IDA* - איטרציות עם threshold הולך וגדל:
שלם – כן בדומה לDFID (אין דרישה שהפונקציה היוריסטית תהיה admissible או consistent).
אופטימלי – כן, רק אם הפונקציה היוריסטית admissible וכך לא תיתן הערכת יתר לשום קודקוד.
סיבוכיות זמן – בדומה לA* זה תלוי בפונקציה היוריסטית, ובמקרה ויש ערכים שונים לכל הצלעות האלגוריתם עלול לייצר קודקוד חדש אחד בלבד בכל איטרציה ולכן יהיה איטי מאוד ויכול להגיע ל$O(N^2)$ בעוד שA* יפתח $O(N)$ (אסימפטוטית).
סיבוכיות מקום – לינארית כיוון שהוא מבוסס DFS: $O(b(\frac{c^*}{e}+1))$.

DFBnB – עובר על כל העץ עד לthreshold שמתקבל כקלט:
שלם – עבור $threshold = \infty$ רק אם עץ החיפוש סופי, במידה והגרף אינסופי רק אם הthreshold גדול מהעומק של הפתרון.
אופטימלי – כן עבור threshold נכון, כיוון שהאלגוריתם עובר על כל עץ החיפוש עד עומק הthreshold.
סיבוכיות זמן – <u>במקרה הטוב ביותר:</u> הפתרון הראשון שנמצא הוא האופטימלי, נקרא לו *c.
בגרף עץ יפותחו כל הקודקודים שעלותם קטנה שווה מ*c ולכן סיבוכיות הזמן תיהיה זהה לשל *A.
בגרף כללי *A יהיה יעיל יותר כיוון שהוא מזהה קודוקדים כפולים וחותך ענפים מיותרים.
<u>במציאות:</u> הthreshold יהיה גבוה יותר מ*c ולכן האלגוריתם יפתח גם קודקודים שעלותם גבוהה מ *c.
סיבוכיות מקום – לינארית כיוון שמבוססת DFS: $O(bd)$.

CSP - בעיות סיפוק אילוצים:

Node consistency – צמצום הDomains של משתנה:
נבדוק עבור כל משתנה שכל הערכים בdomain שלו מספקים את האילוצים האונרים.
נאמר שהCSP הוא NC אם כל אחת מהמשתנים הם NC.

AC-3 – Arc consistency – שמירה על AC של הDomains ושל הVariables:
נבדוק עבור כל זוג משתנים שכל הערכים בdomains מספקים את האילוצים הבינאריים.
נאמר שהCSP הוא AC אם כל אחת מהמשתנים הוא AC אם כל שאר המשתנים.
האלגוריתם מצמצם את הdomain של כל משתנה במידה וקיימים בו ערכים שלו מספקים אילוץ עם משתנה אחר.
סיבוכיות זמן - $O(n^2d^3)$

Path consistency – צמצום הDomains של זוג משתנים ביחס למשתנה שלישי:
נבדוק עבור כל זוג משתנים אם לכל השמה מספקת שלהם קיימת השמה מספקת למשתנה שלישי.

Standart Search – השמה של כל משתנה אחד אחרי השני:
גישה ישירה, מצב ההתחלה יהיה ריק ובכל רמה יהיו כל ההשמות האפשריות שלא סותרות שום אילוץ עבור משתנה בודד.
נשתמש באלגוריתם DFS לצורך ההשמה והמעבר על העץ ולכן כמות העלים בעץ תיהיה:
$$O(nd + (n-1)d + \cdots + d) = O(n!\,d^n)$$

Backtracking – השמה מספקת לקודקודים:
אלגוריתם רקורסיבי, נתחיל ממצב התחלה ריק, בכל שלב נבחר משתנה אחד באופן רנדומלי מתוך אלו שאין להם השמה ונרוץ על הdomain שלו, במידה וההשמה מספקת נאתחל אותו בהשמה החוקית ונשלח אותו ברקורסיה לאלגוריתם שיבחר את המשתנה הבאה ללא השמה.
אם הגענו להשמה מלאה נחזיר אותה, אחרת האלגוריתם יחזיר כישלון.
סדר בחירת הקודקודים הוא על פי הפונקציות הבאות:

MRV – Minimum Remaining Value – בחירת המשתנה בעל הdomain הקטן ביותר:
בכל שלב נבחר את המשתנה שאין לו השמה בעל הכמות המינימלית של ערכים בdomain.

Degree Heuristic – בחירת המשתנה עם הכמות הגדולה ביותר של אילוצים:
שובר שוויון עבור MRV, במידה ויש יותר ממשתנה אחד עם כמות מינימלית של ערכים בdomain נבחר את המשתנה שיש לו הכי הרבה אילוצים עם המשתנים שנותרו (המשתנים שעדיין ללא השמה).

LCV – Least Constraining Value – בחירת המשתנה ששולל הכי מעט ערכים לשכניו:
בהינתן משתנה שנרצה לעשות לו השמה, נבחר בערך מהdomain שישאיר את הכמות הגדולה ביותר של ערכים לשכנים שלו.

Forward Checking – הסתכלות קדימה וסינון מצבים:
לאחר כל השמה באלגוריתם נצמצם את כל הdomains עבור המשתנים שנותרו וכך נוכל לחתוך מוקדם ענפים שיביאו אותנו לכישלון (במידה ולאחר השמה מסויימת יישאר משתנה שאין לו ערכים מספקים בdomain לא נתקדם ונחזור אחורה כדי לתת לו השמה אחרת).

Improved Backtracking Search:
זהה Backtracking Search אבל משתמש בכל השיפורים הנ"ל.
נבחר משתנים בסדר מסויים לפי MRV ובמידה ונצטרך שובר שוויון לפי Degree Heuristic ונבחר ערך להשמה לפי LCV, לאחר כל השמה נקרא לאלגוריתם Inference שיבצע Forward Checking לכל הdomains.

MAC – Maintaining Arc Consistency – שמירה על AC במהלך החיפוש:
מפעיל גרסה מקוצרת של AC-3 (לא מכניס את כל הקשתות הקיימות לתור אלה רק את אלו שהושפעו מההשמה האחרונה).
<u>יתרון:</u> מזהה כישלונות בשלב מוקדם יותר.
<u>חיסרון:</u> עושה את האלגוריתם הרבה יותר איטי.

חיפוש מקומי – Local Search

Hill Climbing – יעצור כשיגיע למצב שאף אחד משכניו טוב ממנו:
מצב התחלה – השמה מלאה.
בכל שלב יעבור לשכן הטוב ביותר שלו בתנאי שהוא טוב יותר ממנו.
במידה ואין כזה האלגוריתם יעצור ויחזיר את המצב.
יתרון: מרחב חיפוש קטן, מהיר.
חיסרון: ברוב המקרים יחזיר local maximum ולא את הפתרון הטוב ביותר.

Sideways Moves – נאפשר תזוזה במישור לשכנים בעלי ערך שווה:
כדי להימנע מהאלגוריתם להיתקע על "כתף" (מישור שבהמשך שלו יש עלייה).
נצטרך להוסיף limit למספר התזוזות במישור כי במידה והמצב יהיה על מישור ולא על כתף הוא יזוז מצד לצד ויתקע על מישור.

Random Restart – נריץ Hill Climbing מספר פעמים ממצבי התחלה שונים:
נשמור את הפתרון הטוב ביותר כאשר לא נוכל להתקדם (כי הגענו לlocal maximum) ונריץ את האלגוריתם שוב ממצב התחלה רנדומלי חדש, במידה ומצא פתרון טוב יותר, נעדכן.
נצטרך להוסיף limit למספר הפעמים שנריץ את האלגוריתם וכשנגיע לlimit נחזיר את הפתרון הטוב ביותר מכל ריצות האלגוריתם.
נשים לב שהאלגוריתם לא מבטיח לנו global maximum במידה ומרחב החיפוש עצום.

Stochastic Hill Climbing – נבחר לאיזה שכן לעבור בצורה הסתברותית מבין האופציות הטובות:
יעזור לנו במצב של ridges (רכס הרים).
איטי יותר אבל לפעמים יספק תוצאות טובות יותר.

First Choice Hill Climbing – ניצור שכן אקראי ונעבור אליו במידה והוא טוב יותר מהמצב הנוכחי:
יתרון: פותר בעיות שיש להן כמות גדולה או אינסופית של שכנים, יעיל עבור ridges.

Random Walk – נאפשר בהסתברות נמוכה צעד רנדומלי לחלוטין (נרשה downhill moves):
בכל צעד, בהסתברות wp נבחר שכן רנדומלי ונעבור אליו, אחרת, בהסתברות 1-wp נבצע Hill Climbing רגיל.
בעיה: איך נבחר את ההסברות?

Min Conflict – Hill climbing עבור בעיות CSP:
<u>קלט:</u> בעיית סיפוק אילוצים עם מצב התחלה בעל השמה מלאה, משתנה max step עבור מספר הניסיונות עד שהאלגוריתם יעצור.
לולאה שרצה max step פעמים:
נבדוק האם ההשמה מספקת, אם כן נחזיר אותה.
אחרת, נבחר רנדומלית משתנה שיש לו קונפליקט מתוך המשתנים בבעיה ונבחר לו השמה שתמנן את הקונפליקט.
<u>מצב התחלה</u>: השמה רנדומלית או שנוכל להיעזר באלגוריתם גרידי שיבחר בכל פעם את המשתנה עם הכמות המינימלית של קונפליקטים.
<u>בעיה:</u> האלגוריתם יכול להיתקע בlocal maximum.
<u>איך נשפר:</u> Random-restart יכול לעזור אבל יש אפשרות טובה יותר:

Hill climbing – Min Conflict with Random Walk משופר עבור בעיות CSP:
קלט: בעיית סיפוק אילוצים עם מצב התחלה בעל השמה מלאה, משתנה max step עבור מספר הניסיונות עד שהאלגוריתם יעצור, הסתברות wp עבור Random-Walk.
לולאה שרצה max step פעמים:
נבדוק האם ההשמה מספקת, אם כן נחזיר אותה.
אחרת, נבחר רנדומלית משתנה שיש לו קונפליקט מתוך המשתנים בבעיה.
בהסתברות wp: נבחר לו השמה רנדומלית מהdomain.
אחרת, נבחר לו את הערך שממנן את כמות הקונפליקטים.
הצעד הרנדומלי באלגוריתם נקרא: conflict-direct random walk step.
הערה: הRandom Walk הוא לא רנדומלי לחלוטין כיוון שהבחירה הרנדומלית מתבצעת רק עבור משתנה שההשמה שלו אינה מספקת!

Random Hill Climbing – GSAT עבור בעיות SAT:
קלט: פורמולת SAT, משתנה max tries – מספר הפעמים שהאלגוריתם יבצע Random Restart, משתנה max flips – כמות הפעמים שנבצע flip למשתנים בפסוקית עד שנעצור.
האלגוריתם:
לולאה מ1 עד max tries:
נבחר השמה רנדומלית לפסוקית
לולאה מ1 עד max flips:
אם הפסוקית מסופקת נחזיר אותה.
אחרת, נבצע flip לאחד המשתנים בפסוקית הגורם לירידה הגדולה ביותר במספר הפסוקיות הלא מסופקות (נשים לב שזה מאפשר downhill moves כיוון שהאלגוריתם תמיד יעבור לשכן מסויימת כלומר תמיד יבצע flip למשתנה מסויים).

Random Hill Climbing – GSAT with Random-Walk משופר עבור בעיות SAT:
קלט: פורמולת SAT, משתנה max tries – מספר הפעמים שהאלגוריתם יבצע Random Restart, משתנה max flips – כמות הפעמים שנבצע flip למשתנים בפסוקית עד שנעצור והסתברות wp עבור Random-Walk.
האלגוריתם:
לולאה מ1 עד max tries:
נבחר השמה רנדומלית לפסוקית
לולאה מ1 עד max flips:
אם הפסוקית מסופקת נחזיר אותה.
אחרת, בהסתברות wp: נבצע flip למשתמנה רנדומלי הנמצא בפסוקית לא מסופקת.
אחרת, נבצע flip לאחד המשתנים בפסוקית הגורם לירידה הגדולה ביותר במספר הפסוקיות הלא מסופקות.
הערה: הRandom Walk הוא לא רנדומלי לחלוטין כיוון שהבחירה הרנדומלית מתבצעת רק עבור משתנה שנמצא בפסוקית לא מסופקת (לא נרשה לבצע flip למשתה הנמצא בפסוקית מסופקת)!

SA – Simulated Annealing – ההסתברות עבור Random Walk משתנה תוך כדי ריצת האלגוריתם:
קומבינציה של Hill Climbing with Random Walk.
ההסתברות wp משתנה באופן דינמי – האלגוריתם מוריד בהדרגתיות את התדירות לצעדים כאלו.
הרעיון: בהתחלה נרצה לבצע יותר צעדים רנדומליים כדי לברוח ממקסימום מקומי וככל שנתקרב לפתרון נרצה לבצע פחות צעדים רנדומליים כדי לא לפספס את הפתרון.
בכל שלב באלגוריתם נגריל בן, אם הוא יותר טוב מהמצב הנוכחי נעבור אליו, אחרת, נעבור אליו בהסתברות התלויה בטמפרטורה הנוכחית ובכמה המצב גרוע(ככל שהt גבוה, גדל הסיכוי שנעבור אליו, כשהטמפרטורה נמוכה הסיכוי קטן, באופן דומה אם המצב ממש גרוע יש סיכוי נמוך יותר שנעבור אליו).
הוכח כי בעבור פרמטרים טובים (α וc) האלגוריתם יגיע למצה האופטימלי.
בדרך כלל נתחיל בטמפרטורה $t = 1$, נסיים בטמפרטורה שקרובה ל0, נניח 0.000001 , stay limit 100, $\alpha =$ 0.9 ו $c = 1$.

SA-SAT גרסה של Simulated Annealing עבור בעיות SAT:
בכל איטרציה נעבור על כל הבנים (ולא נבחר אחד רנדומלית) ובהסתברות מסויימת נעבור אליו (גם אם הוא יותר טוב לא נעבור אליו בהכרח אלא רק בהסתברות).
גם כאן נגביל את מספר הflips שנעשה ובנוסף נכניס Random-Restart.

Tabu Search – נעזר בזכרון כדי להימנע ממצבים מסויימים:
האלגוריתם נעזר בזכרון שהוא מתחזק כדי לא לחזור למצבים שכבר היה בהם וכדי לגלות מקומות חדשים במרחב החיפוש.
למעשה הרעיון הוא לברוח ממקסימום מקומי ובתקווה להגיע למקסימום גלובלי.
בניגוד לSA שתלוי בהסתברות, Tabu Search הוא דטרמיניסטי.
בדומה לSA, Tabu Search מרשה downhill moves אבל רק כשהוא תקוע.
<u>קלט</u>: בעיה והיסטוריה.
לולאה: נבחר את השכן בעל הערך הגבוה ביותר, אם הוא לא בהיסטוריה נעבור אליו, במידה וזה הפתרון הטוב ביותר שלנו נעדכן ובנוסף נעדכן את ההיסטוריה לפי המעבר שעשינו.
הערה: האלגוריתם משתמש בפונקציית aspiration שמרשה לעבור לשכן גם אם הוא קיים בהיסטוריה רק אם השכן הזה ממש טוב.
תנאי עצירה אפשריים: מגבלת זמן או threshold לתוצאה שטובה לנו.
<u>עדכון ההיסטוריה</u>: לא נרשה לבקר שוב במצב שכבר היינו בו כאשר יש תאריך תפוגה למן שבו מצב נמצא בהיסטוריה (למשל אחרי 10 איטרציות נוציא אותו מההיסטוריה ונאפשר לחזור אליו).
בדרך כלל מרחב החיפוש יהיה גדול ולכן נשתמש בזיכרון לטווח קצר ובמקום לשמור את המצבים שהיינו בהם נשמור רק תכונות מסויימות שקשורות למצב.
זיכרון לטווח קצר: נגדיר את המהלכים האחרונים שביצענו כטאבו, לדוגמה נחזיק וקטור ונעדכן אותו באיזה איטרציה שינינו את המשתנה הi, ואם אנו רוצים לשנות אותו שוב נשנה רק אם הפעם האחרונה ששינינו היתה לפני יותר מ5 איטרציות.
זיכרון לטווח ארוך: ישנם מצבים בהם כל השכנים הלא-טאבו מובילים לפתרון פחות טוב, תמיד נוכל לבחור את טוב מבין הגרועים אבל הרעיון כאן הוא דווקא ללכת לשכן שהוא טאבו לפי החלטה מסויימת בעזרת הזיכרון לדוגמה נחזיק וקטור ונעלה את המקום הi בכל פעם ששינינו אותו ואז נחשב את הערך שלהם בקיזוז "עונש" עפ"י כמות הפעמים שעשינו את הפעולה הזאת ונבחר את הטוב ביותר מבין המצבים שבטאבו בקיזוז העונש.
זיכרון לטווח ביניים: כללים שנועדו להטות את החיפוש לאזורים מבטיחים יותר במרחב החיפוש, כלומר מתן עדיפות לתכונות מסויימות בזכרון שקשורות לפתרונות מובחרים יותר.

Local Beam Search – הרצה של Hill Climbing על k מצבים במקביל:
נייצר k מצבים רנדומליים, נייצר את השכנים של כולם, אם אחד מהם הוא הgoal נחזיר אותו, אחרת, נבחר את K השכנים הטובים ביותר ונשמור את המצב הטוב ביותר הנוכחי.
במה זה שונה מלהריץ את Hill Climbing עם Random-Restart k פעמים במקביל?
כאן אנו נבחר את k השכנים הטובים ביותר לעבור אליהם מבין כל השכנים, ולא נבחר שכן אחד לכל מצב.
במידה וזה יגרום לחוסר בגיוון (כי יכול להיות שכל הk שכנים שנבחר הם שכנים של אחד המצבים בלבד) נוסיף לו הסתברויות (Stochastic Beam Search).

אלגורתמים גנטיים – Genetic Algorithms:

אלגוריתם גנטי שומר על אוכלוסיית פתרונות מועמדים לבעיה העומדת בפנינו, וגורם לה להתפתח על ידי יישום איטרטיבי של סט אופרטורים סטוכסטיים.

אופרטורים סטוכסטיים - Stochastic operators:
Selection: הבחירה משכפלת את הפתרונות המוצלחים ביותר שנמצאו באוכלוסייה בשיעור פרופורציונאלי לאיכותם היחסית.
Recombination: רקומבינציה מפרקת שני פתרונות מובחנים ואז מערבבת באופן אקראי את חלקיהם ליצירת פתרונות חדשים.
Mutation: מוטציה מטרידה באופן אקראי פתרון מועמד.

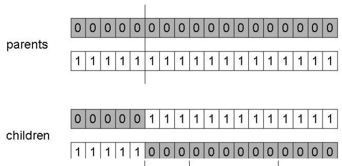
מטאפורות מעולם הביולוגיה:

Genetic Algorithm(אלגוריתם גנטי)	Nature(טבע)
Optimization problem (בעיית אופטימיזציה)	Environment (סביבה)
Feasible solutions (פתרונות אפשריים)	environment (אינדיבידואלים החיים בסביבה זו)
Solutions quality (איכות הפתרון)	Individual's degree of adaptation to its surrounding environment (מידת ההתאמה של הפרט לסביבתו)
A set of feasible solutions (קבוצת פתרונות אפשריים)	A population of organisms (אוכלוסיית אורגניזמים – מינים)
Stochastic operators (אופרטורים סטוכסטיים)	Selection, recombination and mutation in nature's evolutionary process (סלקציה, ריקומבינציה ומוטציה בתהליך האבולוציוני של הטבע)
Iteratively applying a set of stochastic operators on a set of feasible solutions (יישום קבוע של אופרטורים סטוכסטיים על קבוצת פתרונות אפשריים)	Evolution of populations to suit their environment (אבולוציה של אוכלוסיות בהתאם לסביבתן)

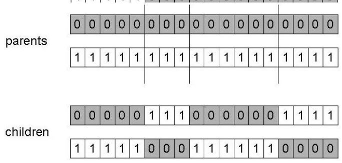
מבנה כללי של אלגוריתם גנטי Evolutionary algorithm – EA:
אתחול אוכלוסיה רנדומלית והערכת כל המצב ע"י פונקציית הערכה.
לולאה עד תנאי עצירה:
נבחר הורים, נעשה להם ריקומבינציה ליצירת בנים, נעשה מוטציה לבנים, נעריך את המצבים החדשים ונבחר את אלו שימשיכו לדור הבא ויהיו האוכלוסיה החדשה.
בחירת ההורים: בהסתברות כך שלהורה טוב יותר יהיה סיכוי גבוה יותר להעמיד צאצאים (להיבחר).
Crossover: עבור ייצוג במחרוזת בינארית אפשר פשוט לחתוך את המחרוזות של ההורים בנקודה מסויימת ולקחת חלק אחד מההורה הראשון ואת החלק הנוסף מההורה השני.
מוטציה: עבור ייצוג במחרוזת בינארית ניתן לעבור על כל ביט ובהסתברות p_c (נמוכה יחסית) לבצע flip לביט.

אלגוריתם גנטי פשוט – Simple Genetic Algorithm – SGA:
1. בחר הורים לקבוצת ההזדווגות בטכניקת גלגל הרולטה – למצב טוב יותר יהיה סיכוי גבוה יותר להיבחר (גודל קבוצת ההזדווגות = גודל האוכלוסייה).
2. ערבב את קבוצת ההזדווגות.
3. על כל זוג רצוף יש לבצע crossover בהסתברות מסויימת, אחרת העתק את ההורה לקבוצת הצאצים.
4. עבור כל צאצא יש לבצע מוטציה (flip ביט עם הסתברות p_m בלתי תלויה עבור כל ביט).
5. החלף את כל האוכלוסייה בצאצאים שנוצרו.

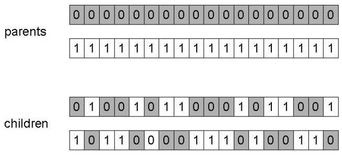
Crossover: בדרך כלל $p_c \in (0.6,0.9)$



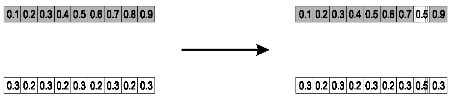
- 1-point: בחר נקודה אקראית ותחתוך את המחרוזות של ההורים בנקודה, הצאצא יקח חלק מכל הורה.
חיסרון: סיכוי גבוה יותר שלצאצא יהיו 2 גנים שסמוכים אצל ההורים ולעולם לצאצא לא יהיו 2 גנים שנמצאיםבקצוות שונים של ההורה.



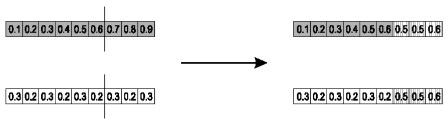
- n-point: נבחר n נקודות אקראיות לביצוע הcrossover הצאצא יקח חלק מכל הורה בצורה מתחלפת.
הכללה של 1-point.



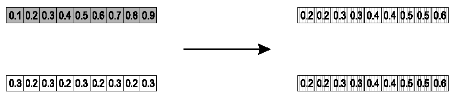
- uniform crossover: נטיל מטבע כאשר heads יסמל בחירת ביט מההורה הראשון וtails יסמל בחירת ביט מההורה השני.
נטיל מטבע לכל ביט וכך נבנה את הצאצאים.



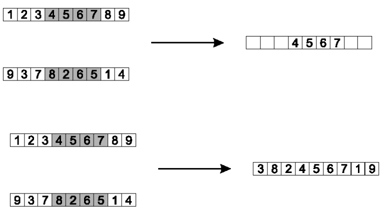
- Single Arithmetic crossover: נבחר גן רנדומלי ונשנה אותו שיהיה הממוצע (לאו דווקא ממוצע מדוייק, העיקר שיהיה קומבנציה מהערכים) של שני ההורים בגן זה.



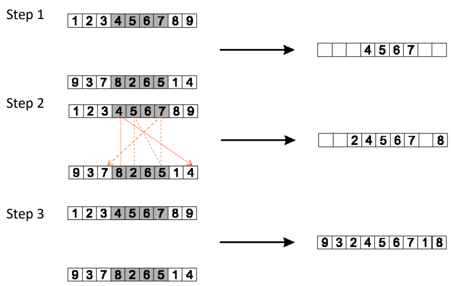
- Simple Arithmetic crossover: נבחר גן רנדומלי והחל ממקום זה נשנה את כל הגנים שיהיו הממוצע של שני ההורים בגן זה.



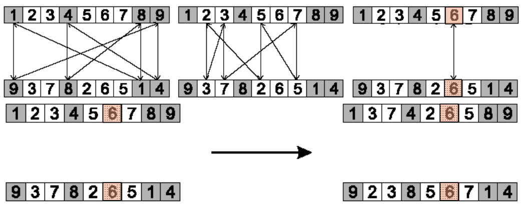
- Whole Arithmetic crossover: הcrossover הנפוץ ביותר, הצאצא יהיה ממוצע כל הערכים של כל הגנים.
כל גן יהיה ממוצע של שני הוריו.



- Order 1 crossover for permutation:
הרעיון הוא לשמור על הסדר היחסי של הפרמוטציה.
נבחר חלק שרירותי מההורה הראשון ונעתיק אותו לילד, לאחר מכן נעתיק מההורה השני את המספרים שלא נמצאים בחלק שהועתק.



- Partially Mapped crossover(PMX):
נבחר חלק שרירותי מההורה הראשון ונעתיק אותו לילד, החל מנקודת ההצלבה הראשונה חפש אלמנטים אצל ההורה השני שלא הועתקו.
לאחר שהתמודדנו עם האלמנטים מקטע ההצלבה, ניתן למלא את שאר הצאצאים מההורה השני.



- Cycle crossover: נתחיל מהגן הראשון של ההורה הראשון, נעבור ממנו למיקום שרשום בתא אצל ההורה השני וכן הלאה עד שנסגר מעגל.
נתחיל מעגל חדש מההורה השני החל מהמקום שלא נבחר במעגל הקודם.

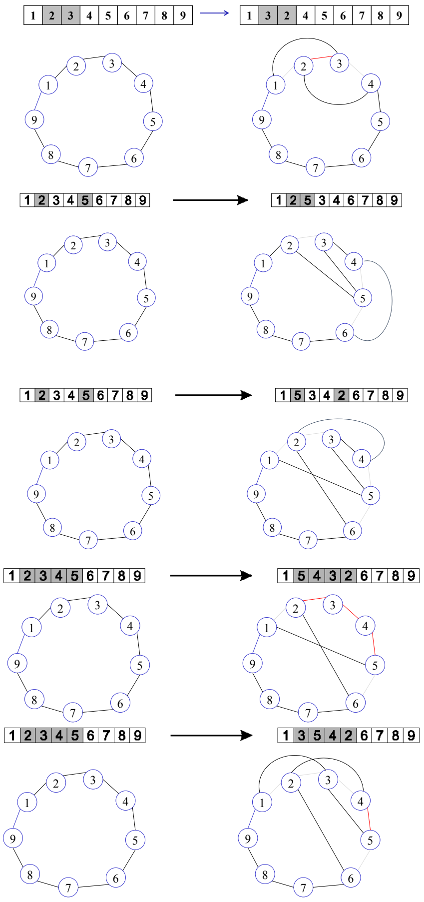
Element	Edges	Element	Edges
1	2,5,4,9	6	2,5+,7
2	1,3,6,8	7	3,6,8+
3	2,4,7,9	8	2,7+, 9
4	1,3,5,9	9	1,3,4,8
5	1,4,6+		

Choices	Element selected	Reason	Partial result
All	1	Random	[1]
2,5,4,9	5	Shortest list	[1 5]
4,6	6	Common edge	[1 5 6]
2,7	2	Random choice (both have two items in list)	[1 5 6 2]
3,8	8	Shortest list	[1 5 6 2 8]
7,9	7	Common edge	[1 5 6 2 8 7]
3	3	Only item in list	[1 5 6 2 8 7 3]
4,9	9	Random choice	[1 5 6 2 8 7 3 9]
4	4	Last element	[1 5 6 2 8 7 3 9 4]

- Edge Combination: עובד על ידי בניית טבלה המפרטת אילו צלעות קיימים אצל שני ההורים, נסמן על ידי +.
נבחר רנדומלית ממי נתחיל ונוריד את כל המופעים שלו בטבלאות.
נתבונן בטבלה שלו: אם יש צלע משותפת נעבור אליה, אחרת, נבחר את מי שהטבלה שלו היא הקטנה ביותר, במקרה של שויון נבחר רנדומלית.

חשיבות הcrossover: רק crossover יכול לשלב מידע משני ההורים.

Mutation – מוטציות:



- Adjacent-Swap mutation for permutations:
  בחר ערך אחד באופן אקראי והחלף אותו אם הערך הצמוד אליו.
  המוטציה שומרת על מרבית המידע הצמוד ועל רוב הסדר של הפרמוטציה (2 צלעות נשברו ו1 התהפכה).
- Insert mutation for permutations:
  בחר שני ערכים באופן אקראי והזז את השני שיהיה צמוד לראשון והזז את כל השאר.
  המוטציה שומרת על הסדר (3 צלעות נשברו).
- Swap mutation for permutations:
  בחר שני ערכים באופן אקראי והחלף את מקומותיהם.
  שומר על הסדר אך משבש את הסמיכויות (4 צלעות נשברו).

- Inversion mutation for permutations:
  בחר שני ערכים באופן אקראי והפוך את כל הערכים שביניהם.
  שומר על מרבית הסמיכויות (2 צלעות נשברו) אך משבש את הסדר (הופך את כל הצלעות בטווח).
- Scramble mutation for permutations:
  בחר תת קבוצה של גנים באופן אקראי,
  סדר מחדש את הערכים באופן אקראי במיקומים אלו.
  (תת הקבוצה לא חייבת להיות רצופה)

חשיבות המוטציה: רק מוטציה יכולה להשיג גן חדש, כדי להגיע לאופטימום אנו זקוקים לעיתים קרובות למזל ומוטציה יכולה לעזור לנו, מוטציה יכולה להחזיר לאוכלוסייה גן שאבד

ייצוגים אפשריים למצב:

מחרוזת בינארית, ערכים מתוך קבוצה קבועה, מספר ממשי, פרמוטציות של אלמנט, רשימות כללים או כל מבנה נתונים.

איך נבחר את הייצוג המתאים לבעיה?

שימוש במבנה נתונים קרוב ככל האפשר לייצוג הטבעי, כתיבת אופרטורים גנטיים מתאימים לפי הצורך, וידואו שכל הגנוטיפים הם פתרונות אפשריים ווידוא שהcrossover שומרים על פתרונות אפשריים.

אתחול של האוכלוסיה הראשונית:

אוכלוסייה אקראית, אוכלוסייה שנשמרה בעבר, קבוצת פתרונות המסופקת על ידי מומחה אנושי או אלגוריתם היוריסטי.

הערה: לא משתלם בהכרח להשקיע באתחול האוכלוסיה הראשונית.

בחירת ההורים:
מטרה: למקד את החיפוש באזורים מבטיחים במרחב.
השראה: "המתאים ביותר שורד" של דרווין
פשרה בין חקירה (exploration) וניצול (exploitation) של מרחב החיפוש.
<u>חסרונות</u> – super individuals גורמים להתכנסות מרחב החיפוש לאזור מסויים בשלב מוקדם מדי.
לקראת סוף האלגוריתם כל האוכלוסיה דומה ואנו מאבדים את הרלוונטיות של בחירת ההורים.
רגישות גבוהה לפונקציית ההערכה , מה לגבי הערכה שלילית למצב?

- <u>Rank-Based Selection</u>: נסיון לצמצם בעיות של FPS על ידי ביסוס הסתברויות הבחירה על כושר יחסי ולא מוחלט, נדרג את האוכלוסייה לפי ההערכה של כל אינדיבידואל ללא חשיבות לכמה הוא טוב או גרוע כאשר למקום הראשון יש הסתברות pop size ולאחרון 1.
  (נניח שהכושר של ההורים הוא: 300,20,1 אז ל300 יהיה סיכוי של $\frac{3}{6}$, ל20 יהיה סיכוי של $\frac{2}{6}$, ול1 יהיה סיכוי של $\frac{1}{6}$).
  האלגוריתם יצטרך למיין את ההורים בכל פעם, אך זה לרוב זניח בהשוואה לזמן הערכת הכושר.
  בחירה כזו נוטה להימנע מהתכנסות מוקדמת מדי של האוכלוסיה, לעומת זאת בדורות מאוחרים יותר הוא יהיה פחות יעיל הבחירה לא מתייחסת לכמה מצב יותר טוב.
- <u>Tournament Selection</u>: כל השיטות לעיל מסתמכות על נתונים סטטיסטיים של האוכלוסייה, יכול להיווצר צוואר בקבוק במיוחד במכונות מקבילות ובנוסף השיטות הנ"ל מסתמכות על פונקציות הערכה חיצונית אשר אולי לא קיימות.
  רעיון הבחירה - בחר k מצבים באופן אקראי (ללא הכנסה מחדש) ואז בחר את הטוב מבין אלה, חזור על מנת לבחור מצבים נוספים.
  איך נבחר את הטוב מבין הk? ניתן להם "לשחק" אחד נגד השני ונבחר את האחד שינצח.

Survivor Selection – בחירת השורדים (החלפת הדור הנוכחי בדור הבא):
שתי גישות עיקריות:
<u>גישה מבוססת גיל</u> – ייצור צאצאים בכמות של ההורים, החלפת כל הדור הקיים בצאצאים או ייצר צאצא אחד והחלף אותו עם ההורה הותיק ביותר.
<u>גישה מבוססת כושר</u> –דרג את ההורים ואת הצאצאים ביחד ובחר את הח הטובים ביותר, ניתן להשתמש בדירוג גם באופן הסתברותי ולא רק דטרמיניסטי (ייתכן שגודל האוכלוסיה ישתנה מדור לדור), ייצר צאצא אחד שיחליף את ההורה הגרוע ביותר או שפשוט יחליף מישהו שפחות טוב ממנו.

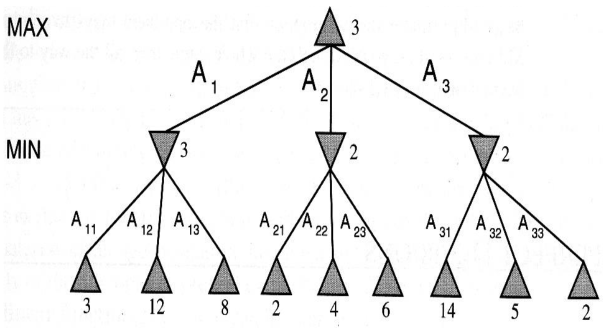
Termination – תנאי העצירה של האלגוריתם:
הגעה למצב עם כושר מספק, הגבלה של מספר דורות או זמן, הגעה לרמה נמוכה של גיוון באוכלוסיה או הגעה לרצף של דורות ללא שיפור ברמת הכושר.

Adversarial Search – חיפוש נגד יריב:

אסטרטגיה היא שיטה המספרת לשחקן כיצד לשחק בכל תרחיש אפשרי.
ניתן לתאר את האסטרטגיה בצורה מרומזת או מפורשת.
אסטרטגיה מפורשת היא תת עץ של עץ החיפוש שמסתעף רק במהלכי היריב.

Minmax algorithm:

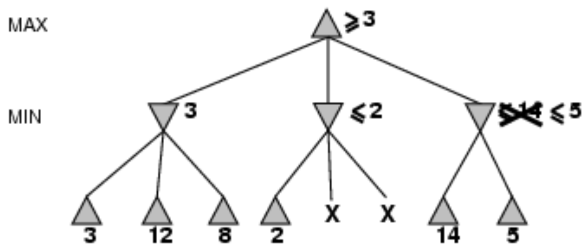
ערך הminmax של קודקוד הוא הutility של המצב בהנחה ושני השחקנים משחקים בצורה אופטימלית לכל אורך המשחק.
<u>הרעיון:</u> נבחר לעבור לקודקוד עם ערך הminmax המקסימלי – זוהי התמורה הגבוהה ביותר שנוכל להשיג אם היריב משחק באופן מושלם.
במידה ושחקן הmin (היריב) לא ישחק בצורה אופטימלית, קל לראות שהmax ישיג תוצאה טובה יותר.
קיימים אלגורתמים אחרים נגד יריבים שלא משחקים באופן אופטימלי שיתנו תוצאות טובות יותר, אך האסטרטגיות האלו יהיו פחות יעילות נגד יריבים אופטימליים.



האלגוריתם - נרד עד העלים של העץ ונחשב את הutility שלהם, שחקן הmin יקבל את הערך המינימלי של בניו ושחקן הmax יקבל את הערך המקסימלי של בניו.
סיבוכיות זמן – $O(b^m)$
סיבוכיות מקום – ע"י DFS: $O(bm)$,
אם נרצה לשמור את האופטימלי: $O(b^m)$.
<u>חיסרון</u> – נצטרך לפתח את כל עץ החיפוש.

$\alpha-\beta$:

אלגוריתם המשפר את minmax.
יחתוך ענפים בשלב מוקדם בעזרת ידע מהיורשים והמורישים שלו.



α הוא הערך של הבחירה הטובה ביותר שנמצאה עד כה לאורך המסלול של הmax.
אם קודקוד v גרוע מα הmax יתעלם ממנו ויחתוך את הענף.
β מוגדרת באופן דומה עבור הmin.
<u>תכונות האלגוריתם:</u>
חיתוך הענפים לא פוגע בתוצאה הסופית.
סידור טוב של הקודקודים ישפר את יעילות חיתוך הענף.
בסידור מושלם סיבוכיות הזמן תהיה $O\left(b^{\frac{m}{2}}\right)$ (מכפילה את עומק החיפוש).

הערכת מצב:

כיוון שעבור בעיות גדולות עם עצי חיפוש עצומים לא נוכל להגיע עד לעלים כדי לחשב ערכי minmax של כל הקודקודים שבדרך מצטרך להעריך מצב שאינו עלה (המשחק עוד לא נגמר, אין תוצאה סופית ועדיין נרצה להעריך כמה המצב טוב).
נשתמש בפונקציית הערכה יוריסטית – פונקציה שממפה מצב למספר.
נשים לב שבניגוד פונקציות יוריסטיות בבעיות חיפוש, במקרה זה לא מעניין אותנו בכמה מהלכים אנחנו רחוקים מהפתרון, אנחנו לא רוצים לנצח הכי מהר, אנחנו רוצים לנצח הכי "טוב".
דוגמה לפונקציה יוריסטית במשחק שחמט – מספר המלכות הלבנות פחות מספר המלכות השחורות.
כמובן שלכל משחק נצטרך להתאים פונקציה יוריסטית שתתאים לו.

Cutting Off Search – שיפור לאלגוריתם minmax:
כיוון שחיפוש brute-force לא רלוונטי כאן נרצה להעריך מצב מבלי להגיע עד העלים.
האלגוריתם זהה לMinmax רק שנחליף את תנאי העצירה להיות cutoff במקום עלה, ולכן נחליף את הutility בפונקציה ההערכה היוריסטית.
איך נבחר את הcutoff?
עומק קבוע או מגבלת זמן.
ניתן להשתמש בiterative deepening עד שהזמן ייגמר וכך נוכל לנצל את הידע מאיטרציה קודמת כדי לא ללכת למקומות מסויימים בחיפוש.
חסרון: אנחנו נלך למצבים שבהמשך יתבהר שהם לא טובים כי אנחנו מסתכלים רק x מהלכים קדימה.

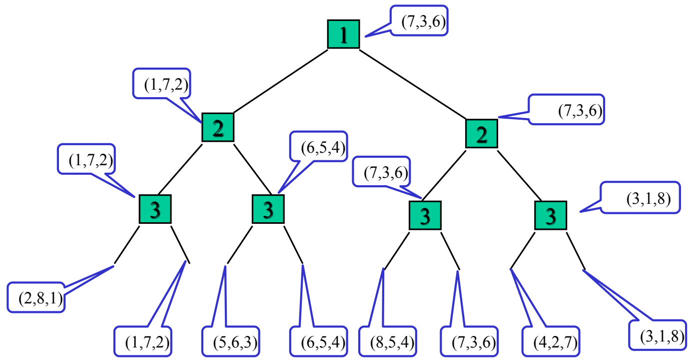
Quiescence search – חיפוש יציב/שקט:
גם כאן נחפש עד עומק מסויים, וניתן הערכה יוריסטית שתיתן הערכה למצב וציון שיגיד עד כמה היא בטוחה בהערכה שהיא נתנה למצב.
במידה והציון אומר שההערכה יציבה (בסבירות גבוהה הקודקוד טוב או רע) נסתפק בזה.
אך במידה והציון יגיד שההערכה לא יציבה (לא מדויקת) אז נעשה חיפוש משני בקודקודים אלו (נמשיך עוד רמה אחת למטה).
דרך זו משיגה הערכה יותר יציבה.

<u>שיפורים נוספים לביצועים של האלגוריתמים כשהחישוב מוגבל(cutoff):</u>
- Transposition Tables - מצבים חוזרים בעץ החיפוש עלולים לגרום לעלייה אקספוננציאלית בעלויות החיפוש. לכן נשמור את ההערכה של המצב הזה בפעם הראשונה שנגיע אליו ונחסוך את החישוב שלו בפעם הבאה שנגיע אליו.
- Opening Book and Endgame Databases - רוב משחקי הלוח מתחילים באותו מצב התחלתי. ננצל את זה לטובתנו ונעשה שימוש בטבלה של מהלכים ראשונים (מהלכי פתיחה) המבוססים על מחקר אנושי שאנו יודעים שיתנו תוצאות טובות יותר. באופן דומה ניתן להשתמש במהלכי סיום.

Selective Search – חיפוש סלקטיבי:
הסיבה המרכזית שבני אדם מתחרים במחשבים היא שבני האדם הם סלקטיביים בבחירות שלהם תוך כדי המשחק בניגוד לתוכנות מחשב שעושות חיפוש עומק קבוע לכל מצב.
חיפוש סלקטיבי יפתח רק קודקודים "מעניינים" ולא את כל העץ.
Best First minimax search מבוסס על אלגוריתם minmax וכל פעם הוא יפתח את הקודקוד הטוב ביותר, במידה ופיתוחו של הקודקוד בישר שיש אחר שהוא יותר טוב, נעדכן את הערך שלו ונפתח אחד אחר.
חסרון: חיפוש מלא בכל מרחב הבעיה נותן לנו ביטוח גבוה יותר על ההערכה של המצבים ומונע טעויות ופספוס של מצבים טובים ולכן רוב תוכנות המשחק המשתמשות בחיפושים סלקטיביים משתמשות באלגוריתם משולב המתחיל בחיפוש מלא, ואחרי עומק מסויים עוברות לחיפוש סלקטיבי.

Multi-Player Game – משחק מרובה שחקנים:
אנו יכולים להכליל את אלגוריתמי המידע המושלמים (הנחה ששני השחקנים משחקים אופטימלי) ל -2 שחקנים, במקרה של משחקי מידע מושלמים מרובי שחקנים בהנחה שאינם משתפים פעולה (כל אחד רוצה לנצח).

אלגוריתם Max^n:
הנחות – כל שחקן משחק בתורו (שחקן 1 ואז 2 ואז 3 וחוזר חלילה, אין תורות כפולים או סדר שמשתנה).
כל שחקן מנסה למקסם את התוצאה שלו.
כל שחקן אדיש לתוצאות של האחרים.
פונקציית ההערכה מחזירה n-tuple של ערכים –$(player_1, player_2, \dots, player_n)$



:Paranoid Algorithm

כאן אנו נתייחס לשאר השחקנים כיריב אחד גדול – שחקן הmin, ואני שחקן הmax.
האלגוריתם הפרנואידי יבחר בתור של שחקן הmax את מקסימום התועלת, ובתורו של שחקן הmin הוא יבחר את התועלת המינימלית.
בניגוד לMax^n , האלגוריתם הפרנואידי מניח שכולם רוצים "להרוג אותי".
במקום שכל שחקן ירצה למקסם את התוצאה שלו, הוא רוצה למנן את התוצאה שלי.
למעשה ניתן להסתכל על האלגוריתם הזה כאלגוריתם $Min^{n-1}Max$.
האלגוריתם הפרנואידי הוא הכללה טובה יותר שלminmax כיוון שהוא נותן ביטוי להערכה של השורש.
:Paranoid vs Max^n

- הפרנואידי רץ מהר יותר מכיוון שאנו יכולים לראות בו אלגוריתם minmax רגיל עם שני שחקנים בלבד. השחקן המקסימלי (אני עצמי) והשחקן המינימלי (n-1 היריבים).
- בשלב החו min נפרשים כל המהלכים של n-1 השחקנים וקח נחסכים המון צמתים פנימיים בעץ.
- שני האלגוריתמים אינם ניתנים להשוואה כיוון שהם דורשים הנחות שונות.

:Max^n עבור $\alpha-\beta$

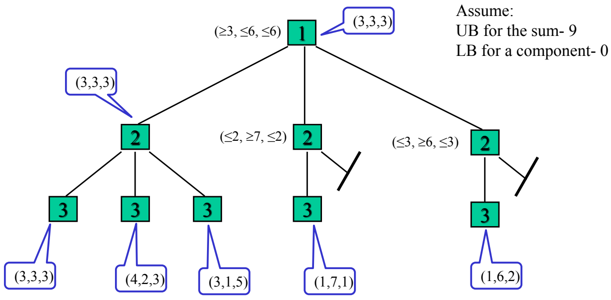
חיתוך ענפים אפשרי רק כאשר יש חסם עליון לסכום הערכים בn-tuple : $(player_1, player_2, \dots, player_n) \le x$
וחסם תחתון עבור לערך של כל שחקן : $y \le player_i \le x$.

<u>חיתוך מידי - Immediate Pruning:</u>

במידה וזהו תורו של שחקן i ולאחד הילדים שלו יש ערך ששווה לחסם העליון במקום הi כמובן שאפשר לחתוך את הענפים של כל שאר הילדים.

<u>חיתוך רדוד – Shallow Pruning:</u>

מסתמך על מידע מהאב ומהבן בלבד, בניגוד ל$\alpha-\beta$,
האלגוריתם לא יוכל להסתמך על מידע מסבא או מנכד.
כיוון שהכנסה של שחקן נוסף לעץ החיפוש משפיעה על החיתוך.



:Expectiminimax Search - הוספת הסתברות לעץ המחשב

נוכל לנתח משחקים שמעורב בהם אפקט של הגרלה
(לדוגמה שש-בש, הטלת קוביות).
נוסיף בין קודקוד של כל שחקן קודקוד של הסתברות (chance).
צמתי הmin והmax יבחרו קודקוד כמו באלגוריתם minmax,
וצומת הchance לוקח הסתברויות.
סיבוכיות זמן - $O(b^m n^m)$ כאשר n הוא מספר ההטלות השונות.
במידה וקיימת מידה על טווח הערכים האפשריים(למשל 2-12
בהטלת 2 קוביות בשש-בש) ניתן לחתוך ענפים בעזרת $\alpha-\beta$.
<u>הערה:</u> עדיין נצטרך להשתמש בcutoff ובפונקציית הערכה יוריסטית.
פונקציית הערכה – נתחיל בחיפוש $\alpha-\beta$, שישחק אלפי משחקים מול עצמו תוך שימוש בקוביות אקראיות, אחוז הזכייה שיתקבל הוכח כקירוב טוב להערכת המצב.