הטכניון – מכון טכנולוגי לישראל

הפקולטה להנדסת חשמל ומחשבים

המעבדה ל-VLSI

דו"ח סיכום פרוייקט ב'

בנושא:

VLSI Routing Optimization Using Dynamic Programming for Exact RSMT

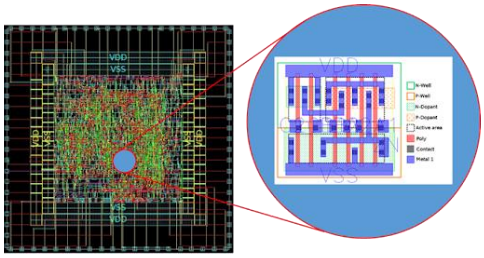
מבצעים:
אלירם עמרוסי 319040325
אלירז קדוש 315675090

מנחה:
אמנון סטניסלבסקי

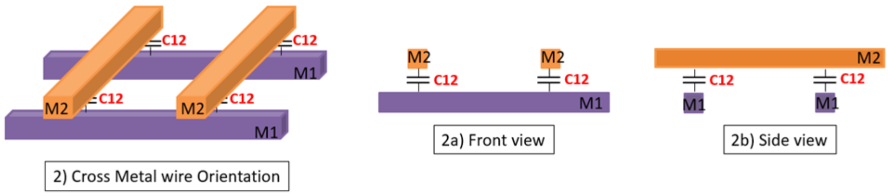
תוכן עניינים

מבוא

תחום ה-(Very Large Scale Integration) **VLSI** עוסק בתכנון ושילוב של **מיליוני ואף מיליארדי רכיבים** לוגיים על גבי שבב אחד. אחד מהאתגרים המרכזיים בתכנון מעגלים משולבים הוא נושא ה-Routing – שלב בתהליך הפיזי של תכנון השבב שבו יש לחבר בין הרכיבים (שערים לוגיים) באמצעות רשת של חיבורים מתכתיים, כך שהאותות יזרמו בצורה נכונה ויעילה.



בשל מגבלות טכנולוגיות, תהליכי הייצור בליתוגרפיה מותאמים **לקווים ישרים** בלבד. שכבות המתכת בפרוסת הסיליקון מאורגנות לרוב כך ששכבות זוגיות מוקצות לניתוב אופקי, ואילו שכבות אי-זוגיות לניתוב אנכי. סידור זה תורם לצמצום צפיפות ומקל על מימוש Vias בין שכבות. בנוסף, קווים ישרים מאפשרים ניתוחים חשמליים מדויקים יותר ומפחיתים תופעות כגון השהיות לא צפויות, רעש וקיבול הדדי בין מסלולים. גם מההיבט האלגוריתמי, השימוש בקווים ישרים מאפשר פתרון יעיל באמצעות מודלים מבוססי Grid וכלים סטנדרטיים בתחום ה־EDA. לכן, Routing בקווים ישרים מהווה סטנדרט מקובל ויעיל בתעשיית ה-VLSI.



חשיבות אופטימיזציה במסלולים של אותות

בשל ה**צפיפות** הרבה של רכיבים על גבי השבב והדרישות ההולכות וגוברות למהירות, הספק וזמן פיתוח, ישנה חשיבות קריטית לאופטימיזציה של מסלולי האותות. תכנון לא מיטבי עלול לגרום לעיכובים בזמן הגעת האותות (Timing Violations), לריבוי קונפליקטים בין מסלולים (Congestion), לבזבוז שטח יקר, לעלייה בהספק ולפגיעה באמינות המעגל. לכן, שימוש באלגוריתמים מתקדמים שמבצעים אופטימיזציה של המסלולים הוא **חיוני להשגת ביצועים טובים** יותר של המערכת.

מטרות הפרויקט

מטרת פרויקט זה היא **לפתח כלי תוכנה** אשר יבצע **אופטימיזציה של מסלולי Routing** בשלב הפיזי של תכנון VLSI. הכלי יאפשר טעינה של קבצי תכנון בפורמטים סטנדרטיים (כגון LEF, DEF, Verilog), יבצע ניתוח של הרשתות החשמליות הדרושות לחיבור, ויישם אלגוריתם למציאת מסלולים קצרים, תוך שיקולי יעילות מרחבית אשר יובילו כמובן להפחתת השהייה.

הכלי יספק גם **ממשק גרפי** למעקב אחרי תוצאות התכנון. בנוסף, הכלי יספק **קבצי פלט** בפורמט נוח לקריאה אשר יאפשרו מימוש קל ויעיל של המסלולים בתהליך הפיזי של ייצור השבב.

עצי שטיינר

מבוא

בפרק זה נסקור את בעיית עצי שטיינר, נבחן סוגים שונים של הבעיה ונעמוד על מורכבותה החישובית. נתמקד במיוחד בגרסה הרלוונטית לפרויקט שלנו - בעיית **RSMT** (Rectilinear Steiner Minimum Tree).

בעיית עץ שטיינר

לאחר 26 שנות מלחמה, הקיסר הסיני, Qinshi Huangdi, איחד את סין בשנת 221 לפנה"ס ורצה לבנות רשת דרכים קצרה ככל האפשר שתבוסס על דרכים קיימות ותחבר ערים עיקריות.

הבעיה הזו ניתנת להמרה לבעיה על גרף לא מכוון, כאשר ערים הן קודקודים (**V**), דרכים הן קשתות (**E**), והאורך הוא משקל (**w**).

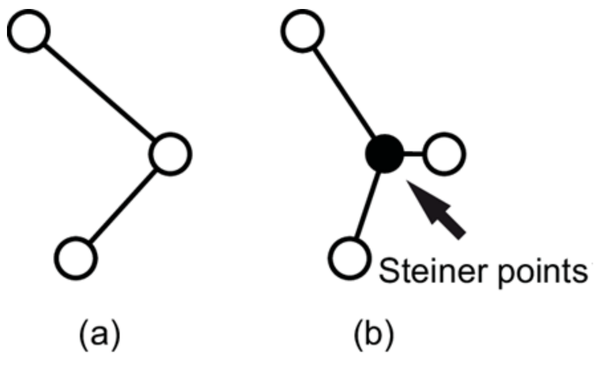
הערים שצריך לחבר מהוות קבוצת טרמינלים (**Y**), ובעיית Steiner Tree Problem - **STP** מוגדרת כך:

בהינתן גרף לא מכוון (**V**,**E**,**w**) = **G** ותת-קבוצה **V** ⊆ **Y**, מטרת בעיית עץ שטיינר היא למצוא תת-עץ ב־**G** שמחבר את כל הנקודות ב־**Y** עם סכום משקלים מינימלי.

ההבדל בין MST לבין STP

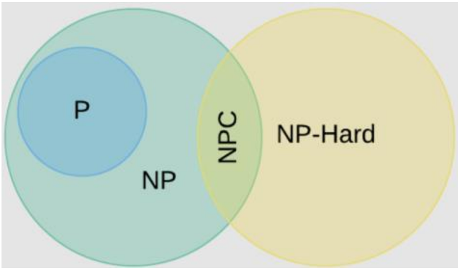
ההבדל בין MST (Minimum Spanning Tree) לבין STP (Steiner Tree Problem) טמון בשאלה: האם מותר להוסיף **נקודות עזר** (Steiner points) כדי לחסוך באורך.

	STP	MST
נקודות חיבור	טרמינלים **ומותר להוסיף נקודות עזר** (Steiner Points) לעץ	**טרמינלים בלבד**
מטרה	עץ באורך מינימלי שמחבר את הטרמינלים **כולל תוספת נקודות אם צריך**	עץ באורך מינימלי שמחבר את הטרמינלים **ללא תוספות**
מורכבות חישובית	קשה (בעיה מסוג NP-Hard)	קלה (זמן ריצה פולינומי)
אורך הפתרון	תמיד קטן (או שווה) מהפתרון של MST	תמיד גדול (או שווה) מהפתרון של STP



(a) MST (b) STP

תזכורת למחלקות סיבוכיות



דיאגרמת ון של מחלקות סיבוכיות

P (Polynomial time) – בעיות שניתן לפתור אותן בזמן פולינומי. כלומר, קיימים אלגוריתמים יעילים שמוצאים פתרון.
דוגמה: מיון מערך של מספרים.

NP (Nondeterministic Polynomial time) – בעיות שלמרות שלא בהכרח ניתן לפתור בזמן פולינומי, אפשר לבדוק אם פתרון מוצע הוא נכון בזמן פולינומי.
דוגמה: בעיית הספיקות (SAT – Satisfiability) – קשה למצוא פתרון, אבל קל לבדוק אם פתרון הוא תקין בזמן פולינומי.

NP-Hard – בעיות שקשות לפחות כמו הבעיות הכי קשות ב-NP, אך לא בהכרח שייכות ל-NP (כלומר, אולי אפילו אי אפשר לבדוק פתרון בזמן פולינומי).
דוגמה: גרסה של Traveling Salesman Problem (TSP) שמבקשת את המסלול הקצר ביותר (ולא רק לבדוק אם קיים מסלול באורך נתון). לא ניתן לאמת תשובה (כלומר: לוודא שהיא המסלול הקצר ביותר) בזמן פולינומי.

NPC (NP-Complete) – קבוצה של בעיות שהן גם ב-NP וגם NP-Hard:
דוגמה: גרף המילטון (Hamiltonian Path).

יש שני סוגי בעיות כאשר מדברים על עצי שטיינר:

בעיית אופטימיזציה – למצוא פתרון מינימלי לפי פונקציית עלות.

בעיית החלטה – ניסוח בינארי: למשל "האם קיים עץ שטיינר באורך קטן מ-K?".

הגרסה הכללית של בעיית עץ שטיינר (Steiner Tree Problem) היא NPC, ויש גרסאות מסוימות שהיא NP-Hard. יש לה שימושים נרחבים במגוון תחומים מדעיים וטכנולוגיים, כגון תכנון VLSI , תקשורת, ותכנון תחבורה.

היסטוריה של בעיית עץ שטיינר

מקורה של הבעיה הוא בבעיה גיאומטרית שהוגדרה במאה ה־17 על ידי פייר פרמה:

בעיה 1: מצא נקודה p בתוך משולש, כך שסכום המרחקים מהנקודה הזו לשלוש קודקודי המשולש יהיה מינימלי.

הבעיה נפתרה ע"י Torricelli והוכח שהזוויות בין הקווים הן בדיוק 120°.

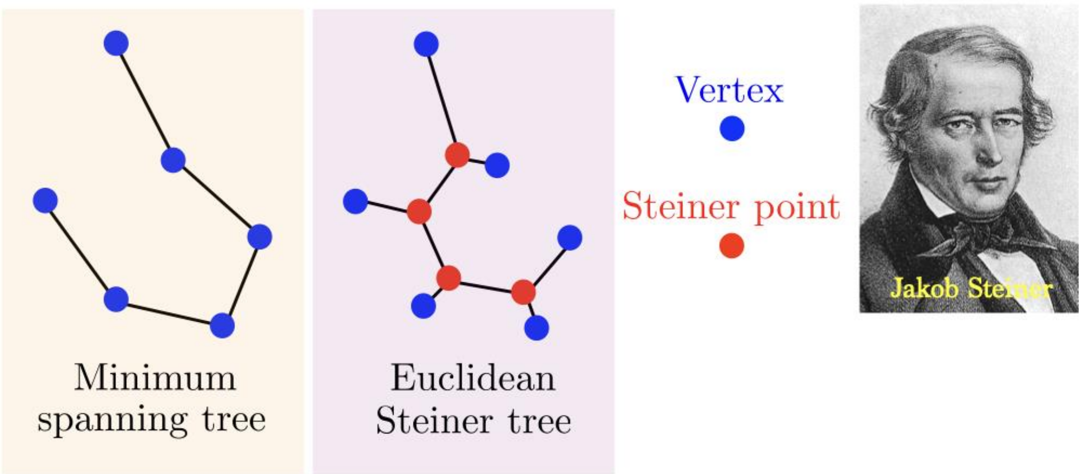
בהמשך, סימפסון הכליל את הבעיה לנקודות כלליות במישור:

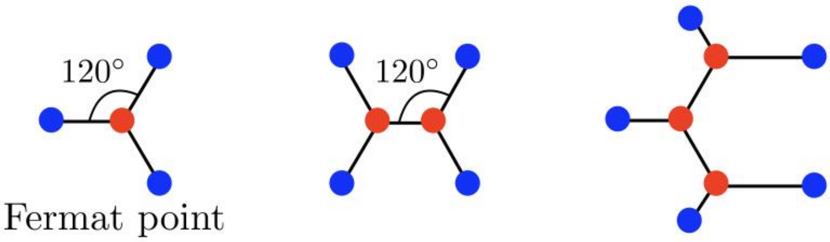
בעיה 2: מצא נקודה p במישור כך שסכום המרחקים מנקודה זו ל-n נקודות נתונות יהיה מינימלי.

מתמטיקאים רבים התעניינו בבעיה של סימפסון. בשנת 1934, Jarnik ו-K¨ossler הציגו סוג נוסף של בעיה - בעיית רשת הקישור הקצרה ביותר (shortest network), המוגדרת כך:

<u>בעיה 3:</u> מצא רשת קצרה ביותר שמקשרת בין n נקודות נתונות במישור.

בעיית רשת הקישור הקצרה ביותר שונה מבעיית פרמה, למעט במקרה של n=3.
Courant ו-Robbins שילבו את שתי הבעיות והשתמשו בשם "בעיית שטיינר" כדי לתאר את בעיית רשת הקישור הקצרה ביותר, על שמו של המתמטיקאי הגרמני הדגול יאקוב שטיינר, בזכות תרומתו לגיאומטריה פרויקטיבית - אף כי תרומתו הישירה לבעיית עץ שטיינר אינה ברורה.
Gilbert ו-Pollak התייחסו בהמשך לבעיית רשת הקישור הקצרה ביותר בתוך גרף כאל בעיית עץ שטיינר, משום שהרשת הקצרה ביותר בגרף צריכה להיות בצורת עץ.





בעיות קשורות

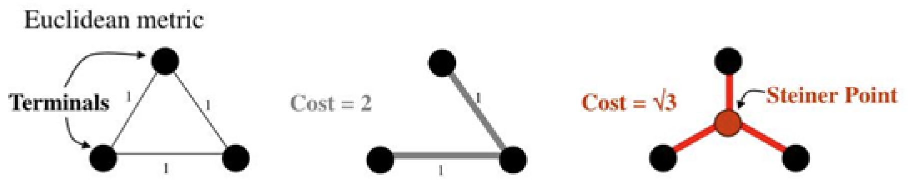
מספר וריאציות של הבעיה נחקרו, ביניהן:

- <u>(**ESTP**) Euclidean Steiner tree Problem - עץ שטיינר אוקלידי:</u>
המרחקים נמדדים לפי ℓ_2 (אוקלידי), מותר להוסיף נקודות בכל מקום במישור.

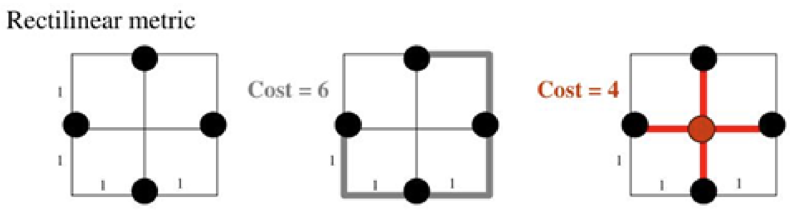
- <u>(**RSTP**) Rectilinear Steiner Tree Problem - עץ שטיינר אורתוגונלי:</u>
המרחקים נמדדים לפי ℓ_1 (מנהטן), מותר להוסיף נקודות רק **בקווים אופקיים ואנכיים**.

- <u>(**DST**) Directed Steiner Tree - עץ שטיינר מכוון:</u>
גרף מכוון עם שורש, צריך שיהיה מסלול מהשורש לכל טרמינל.

בתמונה הבאה ניתן לראות דוגמה של עץ פורש מינימלי לעומת עץ שטיינר כאשר המטריקה היא **אוקלידית**.



בתמונה הבאה ניתן לראות דוגמה של עץ פורש מינימלי לעומת עץ שטיינר כאשר המטריקה היא **אורתוגונלית**.



אלגוריתמים לבעיית עץ שטיינר

קיימים אלגוריתמים מדויקים (Exact) – לרוב זמן ריצה אקספוננציאלי.
אלגוריתמים בקירוב (Approximation) – פתרון קרוב למיטבי.
היוריסטיקות – פתרונות מהירים ללא הבטחה על איכות.

מאחר ואנו עוסקים בתחום ה-VLSI, אנו נתמקד בבעיית RSTP ובפרט בבעיית האופטימיזציה שלה, נגדיר זאת במדויק בפסקה הבאה.

הבחנה בין RSTP, RSMT ו־optimal RSMT

המונח (Rectilinear Steiner Tree Problem) **RSTP** מתאר את **בעיית ההחלטה**:
האם קיים עץ רקטיליניארי (המבוסס על קווים אופקיים ואנכיים בלבד, עם אפשרות להוסיף נקודות שטיינר) שמחבר את כל הטרמינלים כך שאורכו הכולל אינו עולה על ערך נתון k?

בעיית האופטימיזציה המקבילה של RSTP שואפת למצוא את **העץ הקצר ביותר האפשרי** בתנאים אלה – כלומר את הפתרון האופטימלי. פתרון זה נקרא (Rectilinear Steiner Minimum Tree) **RSMT**.

עם זאת, בשיח פחות מדויק (למשל בהקשרים תעשייתיים או בקוד), המונח RSMT משמש לעיתים גם עבור פתרונות שמתקבלים מאלגוריתמים היריסטיים או מקורבים, שאינם בהכרח נותנים את האורך המינימלי האמיתי. במקרים אלו, העץ שמתקבל אמנם רקטיליניארי ותקף, אך **אינו בהכרח אופטימלי**.

לכן, במאמרים מדעיים ובספרות פורמלית, נהוג להדגיש את המונח **optimal RSMT** כדי לציין שמדובר בפתרון שהתקבל באמצעות **אלגוריתם מדויק** (**Exact**), ושאורכו הוא **באופן ודאי הקצר ביותר האפשרי** – כלומר, **הפתרון האופטימלי של בעיית האופטימיזציה של RSTP**. לסיכום:

RSMT – עץ רקטיליניארי תקף שמחבר את הטרמינלים, ייתכן שהתקבל מאלגוריתם קירוב או היריסטי, ולכן **לא בהכרח אופטימלי**.
optimal RSMT – הפתרון **הקצר ביותר** האפשרי ל־RSTP, שהתקבל על ידי אלגוריתם מדויק (Exact).

סקירת המאמר

בפרק זה נסקור את המאמר עליו מתבסס הפרויקט שלנו ששמו המלא הינו:
A Faster Dynamic Programming Algorithm for Exact Rectilinear Steiner Minimal Trees
מאת: Joseph L. Ganley ו-James P. Cohoon, המחלקה למדעי המחשב, אוניברסיטת וירג'יניה.

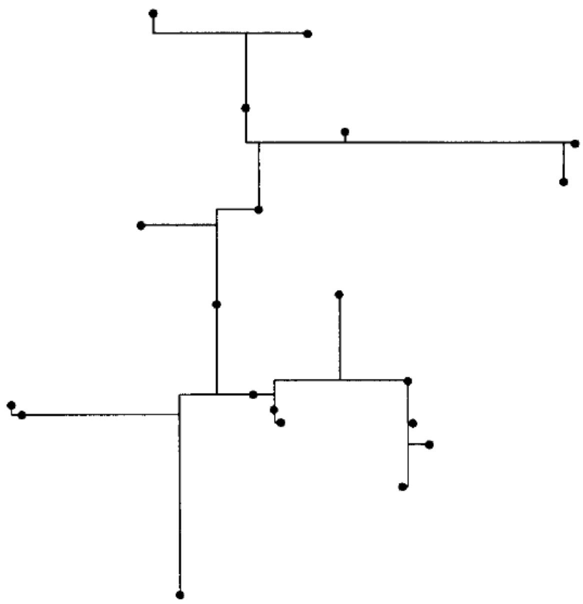
או בתרגום חופשי לעברית:
אלגוריתם תכנות דינמי מהיר יותר ומדוייק (Exact) עבור עצי שטיינר מלבניים (אורתוגונליים) מינימליים.

הצגת המאמר

המאמר מציג אלגוריתם מדויק למציאת עץ שטיינר מלבני מינימלי, אשר משפר את סיבוכיות הזמן והזיכרון בהשוואה לחסמים קודמים. כמו כן, מוצגות תוצאות המדגימות שהאלגוריתם מתפקד היטב גם בפועל.

מבוא

כזכור, בעיית עץ שטיינר מלבני מינימלי (Rectilinear Steiner Minimal Tree, RSMT) מוגדרת כך: בהינתן קבוצת נקודות במישור הנקראות טרמינלים (terminals), יש למצוא אוסף קטעים אופקיים ואנכיים שסכום אורכם מינימלי ושמחבר בין כל הטרמינלים. בעיית RSMT דומה לבעיית העץ הפורש המינימלי (Minimum Spanning Tree) המוכרת, עם הבדל חשוב אחד: בעץ פורש מינימלי, החיבורים מותרים רק בין הטרמינלים הנתונים, בעוד שבבעיית RSMT אפשר להוסיף נקודות שטיינר (Steiner points) כך שאורך העץ הכולל יתקצר. איור 1 ממחיש עץ RSMT אופטימלי עבור קבוצה של 20 טרמינלים. Garey ו-Johnson הוכיחו שבעיית RSMT היא NP-שלמה, דבר המרמז כי אלגוריתם בזמן פולינומי לפתרון מדויק של הבעיה כנראה לא קיים. יישום מרכזי של אלגוריתמי RSMT הוא ב-Routing של מעגלי VLSI. ביישומים כאלה, מספר הטרמינלים הוא בדרך-כלל קטן, ולכן אלגוריתם מדויק ויעיל עשוי להיות ישים הלכה למעשה. במאמר מוצג אלגוריתם תכנות דינמי המחשב עץ RSMT מדויק, עם סיבוכיות זמן וזיכרון טובות יותר מכל החסמים הקודמים הידועים במקרה הגרוע.



איור 1: RSMT אופטימלי עבור 20 טרמינלים

עבודות קודמות

עבודות קודמות לפתרון בעיית RSMT (Rectilinear Steiner Minimal Tree) מתחלקות לשני סוגים:

1. גישות ישירות לבעיה הגיאומטרית.

2. גישות עקיפות באמצעות הפחתת הבעיה לבעיה על גרפים.

גישות ישירות לבעיה הגיאומטרית:

1. Yang ו-Wing: פיתחו אלגוריתם Branch-and-Bound עם מורכבות זמן במקרה הגרוע של $O(2^{k^2})$ כאשר k מספר הטרמינלים. האלגוריתם נבדק על קבוצות של עד 9 טרמינלים בלבד.

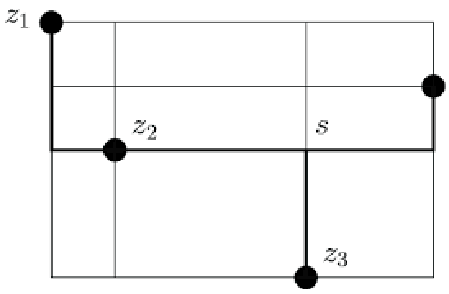
2. Wong ו-Pecht: הציעו אלגוריתם אחר המבוסס על גישה ממצה של השמת קשתות (Edge-Embedding), הסיבוכיות אינה מוגדרת, אך משוערת להיות לפחות אקספוננציאלית במספר הקשתות בגרף Hanan שמספרן הוא $O(K^2)$. האלגוריתם ישים עד 15 טרמינלים.

3. Thomborson, Deneen, ו-Shute: הציגו אלגוריתם עם מורכבות זמן של $O(2^{\sqrt{(k)}\log k})$ אך יחד עם זאת האלגוריתם שלהם מוגבל בכך שהאופטימליות או הזמן תלויים בפרמטרים הסתברותיים.

4. Salowe ו-Warme: הציגו אלגוריתם שמצליח לפתור בעיות של עד 30 טרמינלים באופן יעיל, אך החסם הידוע היחיד לסיבוכיות הזמן שלו היא $O(2^{2^k})$.

גישות עקיפות באמצעות הפחתת הבעיה לבעיה על גרפים:

Hanan הוכיח שעבור כל קבוצה של טרמינלים, קיים עץ שטיינר מלבני מינימלי שמורכב אך ורק מתת-קטעים של קווי הרשת האנכיים והאופקיים (המכונה רשת Hanan) שעוברים דרך הטרמינלים. לכן ניתן לבנות גרף רשת (grid graph) שקדקודיו הם הטרמינלים ונקודות החיתוך של קווי הרשת הללו (כלומר, נקודות שטיינר פוטנציאליות), ובו כל שני קדקודים סמוכים לאורך קו רשת מחוברים בקשת שמשקלה הוא המרחק המלבני ביניהם.



Hanan grid example for n = 4 terminals (only line segments within the bounding rectangle are drawn). A Steiner minimum tree (SMT) is drawn with bold lines. Note that the single Steiner point s shares coordinates with the terminals z2 and z3.

מדוע רשת Hanan חשובה?

מתאוריית Hanan נובע שהפתרון האופטימלי לבעיית שטיינר בגרף Hanan הוא גם פתרון אופטימלי לבעיה הגיאומטרית המקורית. בהתאם לכך, גישה נפוצה לפתרון RSMT היא שימוש באלגוריתם עבור בעיית עץ שטיינר בגרפים המופעל על גרף Hanan.

TECHNION
Israel Institute of Technology

נכון לזמן פרסום מאמר זה (1994), האלגוריתם עם החסם הידוע הטוב ביותר למשימה זו הוא אלגוריתם התכנות הדינמי של Dreyfus ו-Wagner, בעל המורכבות הטובה ביותר למקרה הגרוע עם סיבוכיות זמן של $O(k^2 3^k + (k^2\ logk)2^k)$ (כאן k הוא מספר הטרמינלים ו-n הוא מספר נקודות שטיינר המועמדות).

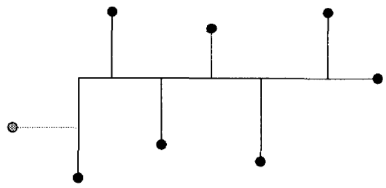
מכיוון שאלגוריתם Dreyfus ו-Wagner נותן את החסם הידוע הנמוך ביותר לזמן ריצה במקרה הגרוע לפתרון מדויק של RSMT והוא גם דומה במידת מסויימת לאלגוריתם המוצע במאמר זה – ההשוואות שייעשו יהיו ביחס אליו.

הצגת וניתוח האלגוריתם

לפני תיאור האלגוריתם, נגדיר מספר מושגים:
Full Set: תהא T קבוצה של טרמינלים. נאמר על T שהיא קבוצה מלאה (full set) אם בכל RSMT אופטימלי של T, כל מסוף ב-T הוא עלה (leaf) בעץ.
Full Tree: RSMT של Full Set ייקרא עץ מלא (full tree).

Hwang הוכיח שעץ מלא יכול להתקיים רק באחת משתי טופולוגיות פשוטות:

טיפוס I: הוא עץ שבו ישנו מקטע "עמוד שדרה" (backbone) הצמוד לאחד הטרמינלים הקיצוניים, כאשר יתר הטרמינלים מתחברים לסירוגין (למשל משני צדדיו) אל עמוד השדרה באמצעות מקטעים קצרים.
טיפוס II: דומה לטיפוס I, אך כולל מסוף קיצוני נוסף שמחובר אל אותו מקטע שמקשר את המסוף הקיצוני הנגדי אל עמוד השדרה (ראו איור 2).



איור 2: המקטע והטרמינל המוצללים עשויים להופיע או לא.

בעזרת המשפט של Hwang, ניתן לחשב בזמן לינארי את RSMT האופטימלי עבור כל Full Set, על ידי בדיקת שתי הטופולוגיות הפשוטות הללו. לבסוף, משפט ידוע לגבי RSMT קובע שכל RSMT מורכב ממספר עצים מלאים הנפגשים בטרמינלים מדרגה שתיים או יותר (טרמינל עם דרגה X = מחובר לX קשתות).

מן העובדות הנ"ל נובע כי עבור כל קבוצה של טרמינלים, עץ ה-RSMT האופטימלי הוא או עץ מלא המקיים את תנאי המשפט של Hwang, או שניתן לחלק אותו לשני עצים קטנים יותר המחוברים זה לזה דרך טרמינל משותף.

תובנה זו מובילה ישירות לפיתוח האלגוריתם המוצג במאמר, שהוא **אלגוריתם תכנות דינמי**.

האלגוריתם פועל בגישת תכנות דינמי: הוא מחשב את הפתרון (האורך הקצר ביותר של עץ RSMT) לכל תת־קבוצה של טרמינלים, לפי סדר עולה של גודל הקבוצה – כלומר, הוא מתחיל מתתי־קבוצות בגודל 2, אחר כך בגודל 3, וכך הלאה. עבור כל תת־קבוצה כזו, האלגוריתם בודק שתי אפשרויות: (1) האם ניתן לבנות לה עץ מלא לפי טופולוגיה פשוטה (עפ"י משפט של Hwang), או (2) האם כדאי לפרק אותה לשתי קבוצות קטנות יותר שמתחברות דרך טרמינל משותף. מכיוון שהאלגוריתם מתקדם לפי גודל תת־הקבוצות, הפתרונות לכל הקבוצות הקטנות כבר חושבו ונשמרו קודם – ולכן ניתן להשתמש בהם מחדש מבלי לחשב שוב. כך הוא בונה בהדרגה את הפתרון לכלל הטרמינלים בקלט, תוך חיסכון משמעותי בזמן ובזיכרון.

TECHNION Israel Institute of Technology

האלגוריתם הנ"ל נקרא FDP (Full-Set Dynamic Programming) ולהלן מוצג הפסואודו קוד שלו כפי שמופיע במאמר:

```
(1)  For m = 2 to |T|
(2)    For each C ⊆ T such that |C| = m
(3)        L[C] = FullTree(C)
(4)        For each i ∈ C
(5)            For each F ⊂ (C - {i})
(6)                L' = L[F ∪ {i}] + L[C - F]
(7)                L[C] = min(L[C], L')
```

(הפונקציה `FullTree` מחזירה את אורכו של עץ מלא אופטימלי עבור הקבוצה הנתונה, בהתאם למשפט של Hwang.)

ניתוח סיבוכיות הזמן והמקום של האלגוריתם:

נתחת תחילה את סיבוכיות הזמן והמקום של כל שורה באלגוריתם ולאחר מכן נציג את התוצאה הכוללת.

```
(1)  For m = 2 to |K|
```

<u>הסבר</u>: לולאה שמריצה את האלגוריתם לכל גודל אפשרי של תת־קבוצות טרמינלים, מ־2 ועד k (מס' הטרמינלים).
<u>סיבוכיות זמן</u>: $O(k)$ (קבוע לעומת החלקים הפנימיים).
<u>סיבוכיות מקום</u>: זניחה, לא תורמת לצריכת זיכרון משמעותית.

```
(2)    For each C ⊆ K such that |C| = m
```

<u>הסבר</u>: עובר על כל תתי־הקבוצות בגודל m. סך הכל לאורך כל השלבים עובר על כל $O(2^k)$ תתי־הקבוצות.
<u>סיבוכיות זמן</u>: $O(2^k)$.
<u>סיבוכיות מקום</u>: יש להחזיק ערך אחד לכל תת־קבוצה: $O(2^k)$.

```
(3)        L[C] = FullTree(C)
```

<u>הסבר</u>: מחשב את אורך העץ המלא של הקבוצה לפי טופולוגיות פשוטות (לפי Hwang).
<u>סיבוכיות זמן</u>: O(k) לכל קבוצה ולכן $O(k2^k)$ בסך הכל.
<u>סיבוכיות מקום</u>: שומר ערך יחיד (מספרי) לכל קבוצה – כבר נכלל בסעיף קודם.

```
(4)        For each i ∈ C
```

<u>הסבר</u>: בוחר טרמינל שדרכו תת־הקבוצה תתפצל – משמש כצומת חיבור.
<u>סיבוכיות זמן</u>: $O(k2^k)$.
<u>סיבוכיות מקום</u>: אין אחסון נוסף – רק משתנה לולאה.

```
(5)            For each F ⊂ (C - {i})
```

<u>הסבר</u>: בודק את כל החלוקות של הקבוצה לשתי קבוצות זרות שמתחברות דרך i.
<u>סיבוכיות זמן</u>: עד $O(2^k)$ חלוקות אפשריות לכל קבוצה ולכן סך הכל $O(3^k)$.
<u>סיבוכיות מקום</u>: אין אחסון קבוע – מבוצע בלולאה בלבד. זניח.

```
(6)                L' = L[F ∪ {i}] + L[C - F]
```

<u>הסבר</u>: מחשב את אורך החיבור של שני עצים שכבר חושבו קודם.
<u>סיבוכיות זמן</u>: $O(1)$.
<u>סיבוכיות מקום</u>: זניח – ערך זמני.

```
(7)                L[C] = min(L[C], L')
```

<u>הסבר</u>: עדכון האורך המינימלי אם נמצא פירוק טוב יותר.
<u>סיבוכיות זמן</u>: $O(1)$ לכל איטרציה ולכן סך הכל $O(3^k)$.
<u>סיבוכיות מקום</u>: ללא השפעה נוספת – הערך כבר קיים ב־L[C].
<u>סה"כ סיבוכיות הזמן של האלגוריתם:</u>

First loop (line 1): L_1 = each value of m

Second loop (line 2): $L_2 = \binom{k}{m}$

Third loop (line 4): $L_4 = m * L_2 = m * \binom{k}{m}$

Fourth loop (line 5): $L_5 = 2^{m-1} * L_4 = \binom{k}{m} * m * 2^{m-1}$

$$\sum_{m=2}^{k} \binom{k}{m} * m * 2^{m-1} = k * 3^{k-1}$$

הפונקציה Full Tree פועלת בזמן ליניארי לכל קבוצה כלומר $O(k)$ ולכן כאשר היא מופעלת על כל $O(2^k)$ תתי־הקבוצות של הטרמינלים, היא מוסיפה זמן כולל של $O(k2^k)$ לאלגוריתם כולו.
ולכן **סיבוכיות הזמן** של אלגוריתם הFDP הוא:

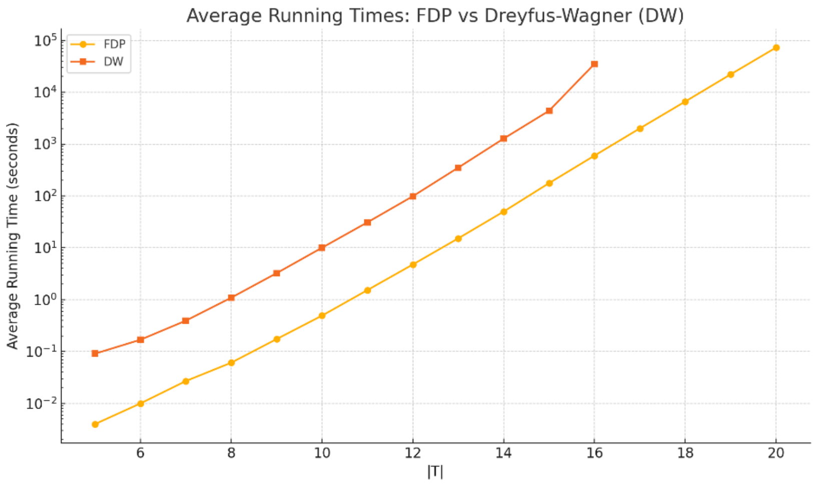
$$\boldsymbol{O(k3^k + k2^k)}$$

הערה: תוצאה זה כמובן שקולה באופן אסימפטוטי ל- $\boldsymbol{O(k3^k)}$ אך אנו שומרים על האיבר הנוסף לצורך עקביות עם האלגוריתם של **Dreyfus ו- Wagner** ומכיוון שעבור קבוצות קטנות של טרמינלים, האיבר השני משמעותי.

סה"כ סיבוכיות המקום של האלגוריתם:

האלגוריתם מאחסן רק את האורך של העץ האופטימלי עבור כל תת-קבוצה, ישנן 2^k תתי-קבוצות, ולכן **סיבוכיות המקום היא $\boldsymbol{O(2^k)}$.**
בפועל, ייתכן שירצו לאחסן גם את הפירוק האופטימלי של כל תת-קבוצה יחד עם האורך שלה.
שינוי זה לא משנה את סיבוכיות הזמן של האלגוריתם, אך מבטל את הצורך במעבר שני (Second Pass) לחישוב ה-RSMT בפועל, ובכך מקטין באופן משמעותי את זמן הריצה בפועל.
שינוי זה מגדיל את סיבוכיות המקום ל- $\boldsymbol{O(k2^k)}$ שעדיין מהווה שיפור לעומת האלגוריתם של Dreyfus-Wagner.

תוצאות אמפיריות



על מנת לבחון את בפועל את יעילות האלגוריתם FDP, Ganley ו-Cohoon מימשו את שני האלגוריתמים (FDP ודרייפוס-ויגנר) בשפת C והריצו אותם על תחנת עבודה Sun SPARC-20. הגרף מציג את זמני הריצה (בסקלת לוגריתם) של שני האלגוריתמים כפונקציה של מספר הטרמינלים - |T|. ניתן לראות בבירור כי האלגוריתם FDP גדל אקספוננציאלית אך בצורה מתונה יותר, וממשיך לספק תוצאות עד T=20.
האלגוריתם DW הופך לאיטי מאוד כבר מ-T=13, ולא מצליח להתמודד עם ערכים גבוהים יותר (אין תוצאות מ-T=17).
המשמעות: FDP הרבה יותר סקיילבילי עבור ערכים גדולים של טרמינלים.
הערה: אנחנו מימשנו ב-python את האלגוריתם. נציג את תוצאות הריצה שלנו בפרק "ניתוח תוצאות".

מסקנות

המאמר הציג אלגוריתם תכנות דינמי בשם Full set Dynamic Programming (FDP), אשר מחשב עצי שטיינר מלבניים מינימליים באופן מדויק, ואשר סיבוכיות הזמן והזיכרון במקרה הגרוע שלו, טובה יותר משל כל האלגוריתמים הקודמים. בפרט, האלגוריתם משפר את החסם הטוב ביותר שהיה ידוע קודם לכן – האלגוריתם הדינמי של דרייפוס וויגנר.

אלגוריתם דרייפוס-ויגנר:

- סיבוכיות זמן: $O\left(k^2 3^k + \left(k^2 \log k\right) 2^k\right)$
- סיבוכיות מקום: $O\left(k^2 2^k\right)$

אלגוריתם FDP:

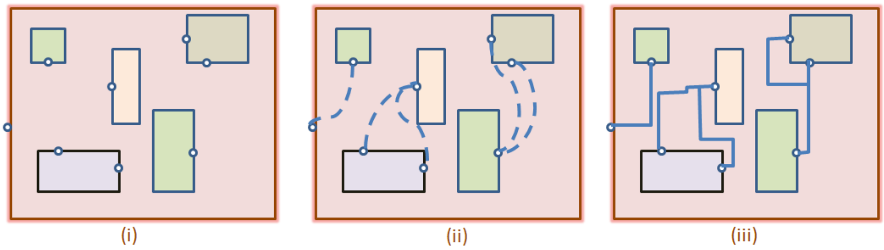
- סיבוכיות זמן: $O\left(k 3^k + k 2^k\right)$
- סיבוכיות מקום: $O\left(2^k\right)$

שלבי המימוש וניתוח הקוד

בפרק זה נפרט את שלבי המימוש של התוכנה, החל משלב תחילת העבודה והבנת הדרישות, דרך תכנון מבנה האלגוריתם על בסיס הפסאודו-קוד, ועד לאתגרים שבהם נתקלנו במהלך המימוש.

תחילת העבודה והבנת הדרישות

הפרויקט שלנו עוסק בבעיה מרכזית בעולם תכנון השבבים – שלב **ה-Routing** בתהליך ה-Place and Route של תכנון פיזי (Physical Design). לאחר שממקמים את השערים הלוגיים על גבי השבב (Placement), יש צורך לחבר ביניהם בעזרת קווים ממתכת (metal layers), באופן שיבטיח מסלולים קצרים, חסכוניים בשטח, ובעלי השהיה נמוכה ככל האפשר.



An example showing – (i) Placement of standard cells, (ii) After global routing, (iii) After detailed routing

אחת מהבעיות הידועות בשלב זה היא חישוב מסלולים אופטימליים בין קבוצות של נקודות, והיא ידועה כבעיה NP-שלמה. יש גישות שונות העושות שימוש בגישות מקורבות (Heuristics או Approximation), אך אנחנו בפרויקט זה בחרנו ליישם אלגוריתם מדויק (**Exact Algorithm**), המבוסס על המאמר המצורף בתיאור הפרויקט.
האלגוריתם מאפשר חישוב של עץ שטיינר (Steiner Tree) בצורה מדויקת, תוך מציאת מבנה חיבור שממזער את סך אורך החוטים הנדרש – כולל שימוש בנקודות שטיינר אם הדבר משפר את התוצאה.

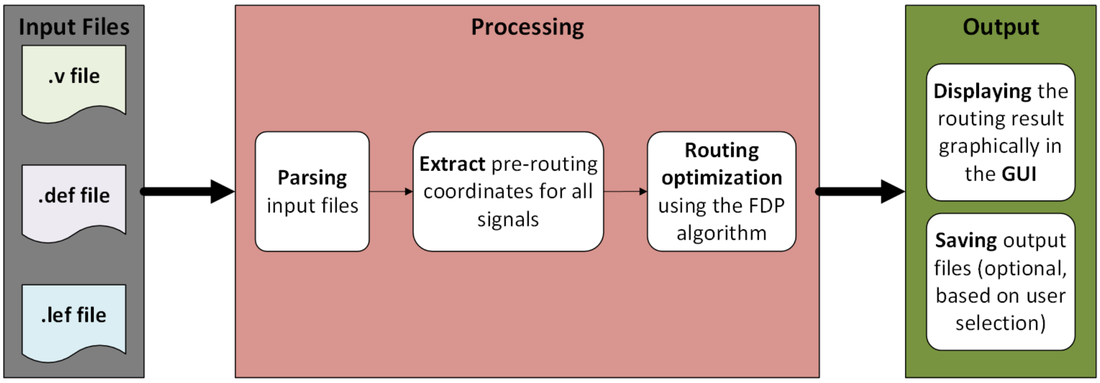
בעוד שבפרויקט א' עסקנו בתהליך RTL2GDS מהיבט רחב וכללי, פרויקט ב' אפשר לנו להתעמק באופן ממוקד בשלב ה־Routing של תכנון השבב – כלומר, בחיווט הפיזי של החיבורים בין השערים הלוגיים.

בשלב ראשון, חקרנו את תהליך ה־Physical Design Flow, והבנו שהפרויקט שלנו ממוקם עמוק בתוך השלב הפיזי – לאחר יצירת ה־Netlist וביצוע Placement, ולפני שלבי הבדיקות הסופיים (DRC, LVS). הבנה זו סייעה לנו לגבש תמונה ברורה של הדרישות מהמערכת: אילו נתונים עליה לקבל, כיצד לעבד אותם, ואיזה פלט עליה להפיק.
כחלק מהעמקה זו, נדרשנו להבין אילו קבצי קלט לדרוש לצורך יישום האלגוריתם:

- קובץ Verilog – לתיאור לוגי של המעגל.
- קובץ DEF – לפריסת הרכיבים והפינים.
- קובץ LEF – למאפייני התאים הסטנדרטיים, כולל מיקום וגודל הפינים.

בהתבסס על הקבצים הללו בנינו מבני נתונים המתארים את הרכיבים והקשרים ביניהם, כדי להפעיל עליהם את אלגוריתם שטיינר ולהפיק פתרון אופטימלי. בהמשך הוספנו גם ממשק גרפי בסיסי (GUI) להצגת העץ המחושב, בהתאם לדרישות הפרויקט.

התרשים הבא מציג את האסטרטגיה שעל פיה בנינו את הכלי התוכנתי – מרגע קליטת הקבצים ועד להפקת התוצאה הסופית.



תרשים זרימה של שלבי עיבוד המידע – מקבצי הקלט ועד להצגת הפלט

בפרק הבא נסביר איך התרשים בא לידי ביטוי מבחינת מבנה הקוד.

ניתוח הקוד

הקוד העוסק בעיבוד המידע מחולק לשני חלקים עיקריים:

1. פיענוח הקלט והפקת נקודות החיבור הפיזיות (Pre-routing).
2. ביצוע החיווט האופטימלי (באמצעות אלגוריתם עץ שטיינר).

נפרט כעת על כל אחד מהם באמצעות.

1. פיענוח הקלט והפקת נקודות החיבור הפיזיות

שלב זה עוסק בשליפת מידע מהקבצים הסטנדרטיים של תכנון שבבים (Verilog, DEF, LEF), במטרה לבנות את הקלט המדויק לאלגוריתם עץ שטיינר.
במסגרת שלב זה, פותחו מספר פונקציות שמטפלות בשלבים שונים של עיבוד הקבצים והמרת המידע הגולמי למבני נתונים מוכנים לשימוש.

תהליך זה דרש מאיתנו להבין לעומק את פורמט הקבצים השונים, ואת הדרך בה ניתן למזג ביניהם על מנת לבנות תיאור מלא של המעגל — הן מבחינה לוגית והן מבחינה פיזית. הבנו שעלינו להתחשב במיקומים אבסולוטיים של הפינים, ובמיקומי הרכיבים עצמם – זאת כדי להכין את הקלט הנכון לאלגוריתם שטיינר שיבוא לאחר מכן.

קובץ Verilog (.v): מגדיר את הקשרים הלוגיים בין הרכיבים (netlist). מזהה את שמות הרכיבים והסיגנלים המחברים ביניהם.

קובץ DEF (.def): מתאר את הפריסה הפיזית של הרכיבים על גביי השבב – כולל מיקום של כל רכיב וקואורדינטות מדויקות של הפינים (inputs/outputs).

קובץ LEF (.lef): מספק את המאפיינים הגיאומטריים של כל תא לוגי סטנדרטי, כולל מיקומי הפינים בתוך כל תא, גודלו הפיזי, השכבות שבהן הוא משתמש ועוד.

נפרט את השלבים על דוגמה לדיזיין פשוט בשם "module not(a*b)+b":

שלב 1: איסוף וסיווג סיגנלים
(מבוצע על ידי הפונקציה build_signals_array)

- סריקת קובץ ה־Verilog לזיהוי כל הסיגנלים (wires).

```
module not(a*b)+b
     input wire a,
     input wire b,
     output wire y
);

wire n1, n2;

and U1 (n1, a, b);
not U2 (n2, n1);
or U3 (y, n2, b);

endmodule
```

- ספירת מספר המופעים של כל סיגנל לצורך הקצאת מערכים בגודל מתאים.

```
module not(a*b)+b
     input wire a,
     input wire b,
     output wire y
);

wire n1, n2;

and U1 (n1, a, b);
not U2 (n2, n1);
or U3 (y, n2, b);

endmodule
```

- יצירת מערך עבור כל סיגנל, כאשר כל תא עתיד להכיל מיקום אחד של אותו סיגנל במעגל.

שלב 2: מיקום ראשוני לפינים מסוג input/output
(מבוצע על ידי הפונקציה fill_pin_positions)

- סריקת קובץ ה־DEF לשליפת פינים המוגדרים כ־input או output.

```
PINS 3 ;
   - a + NET a + DIRECTION INPUT + USE SIGNAL
     + LAYER M1 ( 0 0 ) ( 10 10 )
     + PLACED ( 50 200 ) N ;
   - b + NET b + DIRECTION INPUT + USE SIGNAL
     + LAYER M1 ( 0 0 ) ( 10 10 )
     + PLACED ( 50 300 ) N ;
   - y + NET y + DIRECTION OUTPUT + USE SIGNAL
     + LAYER M2 ( 0 0 ) ( 10 10 )
     + PLACED ( 650 200 ) N ;
END PINS
```

- עדכון התא הראשון במערך הסיגנלים עם קואורדינטות מוחלטות לפינים החיצוניים.

TECHNION
Israel Institute
of Technology

שלב 3: יצירת מערך רכיבים

(מבוצע על ידי הפונקציה build_components_array)

- סריקת קובץ ה־DEF לשליפת רשימת רכיבים (Components), כולל שם, סוג השער ומיקום הפיזי.

```
VERSION 5.6 ;
DIVIDERCHAR "/" ;
BUSBITCHARS "[]" ;
DESIGN two_metal_layers ;
UNITS DISTANCE MICRONS 1000 ;

COMPONENTS 3 ;
   - U1 AND2_X1
      + PLACED ( 100 200 ) N ;
   - U2 NOT_X1
      + PLACED ( 300 200 ) N ;
   - U3 OR2_X1
      + PLACED ( 500 200 ) N ;
END COMPONENTS
```

- יצירת מבנה נתונים עבור כל רכיב, הכולל:
  - מיקום הרכיב על השבב.
  - שם הרכיב וסוג השער.
  - שדות ריקים למיקומי הפינים A, B ו־Y (שיושלמו בשלב הבא).

שלב 4: חישוב מיקומים מוחלטים לפינים בתוך הרכיבים

(מבוצע על ידי הפונקציה fill_components_array)

- סריקת קובץ ה־LEF לזיהוי הגדרות המאקרו של כל שער ומיקומי הפינים היחסיים (RECT).

```
VERSION 5.6 ;
BUSBITCHARS "[]" ;
DIVIDERCHAR "/" ;

MACRO AND2_X1
  CLASS CORE ;
  FOREIGN AND2_X1 ;
  ORIGIN 0 0 ;
  SIZE 1.0 BY 1.0 ;
  SYMMETRY X Y R90 ;

  PIN A
    DIRECTION INPUT ;
    USE SIGNAL ;
    PORT
      LAYER M1 ;
      RECT 0.1 0.9 0.2 1.0 ;
    END
  END A
```

- חישוב נקודת המרכז של כל pin לפי ממוצע קואורדינטות ה־RECT.

- חישוב המיקום האבסולוטי של pin על ידי חיבור המיקום היחסי של ה-pin למיקום הרכיב בקובץ DEF.

דוגמה:

Pin A נמצא בטווח המלבני שיוצרות הקואורדינטות (0.1 ,0.9) עד (0.2 ,1.0), ולכן מיקומו המרכזי הוא (0.15 ,0.95). הרכיב ממוקם ב-(100 ,200), לכן המיקום האבסולוטי של הפין יהיה (100.15 ,200.95).

שלב 5: השלמת מערך הסיגנלים
(מבוצע על ידי הפונקצייה complete_signals_array)

- סריקת קובץ ה־Verilog החלק מהקטע בו מוגדרים החיבורים (שורה 9).

dot v (Verilog) file example.txt | dot def (Design Exchange Format) file example.txt

```
1  module not(a*b)+b
2      input wire a,
3      input wire b,
4      output wire y
5  );
6
7  wire n1, n2;
8
9  and U1  (n1, a, b);
10 not U2  (n2, n1);
11 or U3  (y, n2, b);
12
13 endmodule
14
```

- שליפת מידע ממערך הרכיבים עבור כל מופע.

- עדכון מערך הסיגנלים בערכים מתאימים.

דוגמה:
n1 הוא ה-output של הרכיב U1, נחפש במערך הרכיבים את U1 ונחפש את המידע לגבי ה-output שלו ונכניס אותו למקום הפנוי הבא במערך של הסיגנל n1, וכך נעשה גם עבור הסיגנלים a, b.

שם הפונקציה	קובץ קלט	תיאור הפעולה	פלט / תוצאה
build_signals_array	Verilog (.v)	סריקת קובץ ה־ Verilog לשליפת כל הסיגנלים (wires), ספירת מופעים ויצירת מערך לכל סיגנל	מערך סיגנלים ריק בגודל מתאים לכל wire
fill_pin_positions	DEF (.def)	סריקת קובץ ה־ DEF למציאת פינים מסוג input/output ועדכון התא הראשון במערכי הסיגנלים	מיקום ראשוני של קואורדינטות עבור סיגנלים מהסביבה החיצונית
build_components_array	DEF (.def)	סריקת הרכיבים בקובץ ה־ DEF ושליפת מיקום האבסולוטי על השבב	מערך רכיבים עם מידע ראשוני: שם, סוג, מיקום
fill_components_array	LEF (.lef)	סריקת קובץ ה־ LEF לצורך חישוב מיקום מוחלט של הפינים בכל רכיב	עדכון מערך הרכיבים עם מיקומי הפינים A, B, Y
complete_signals_array	Verilog + DEF + LEF	סריקת הקבצים ליצירת מיפוי מלא בין סיגנלים לפינים בכל רכיב	מערך סיגנלים מלא, כולל כל נקודות הקצה עם קואורדינטות מוחלטות

לאחר השלמת שלבים אלו, נוצר **מערך נתונים הכולל את כל נקודות החיבור הדרושות להפעלת אלגוריתם שטיינר**, כאשר כל נקודה מתוארת באמצעות קואורדינטות מדויקות על השבב.

2. ביצוע החיווט האופטימלי

המימוש של הקוד לחישוב ה-RSMT האופטימלי הוא מהמרכיבים המאתגרים יותר בתוכנה שלנו. לכן, בחרנו לממש אותו בקובץ עצמאי בשם FDP.py, כדי לאפשר עבודה יעילה יותר ודיבוג נוח. המימוש תוכנן כך שניתן להזין אליו קלט של טרמינלים ישירות מהטרמינל של לינוקס, ובכך לעקוף את שלב פיענוח הקלט והפקת נקודות החיבור הפיזיות.

כלומר ניתן להפעיל אותו כך:

```
python3.10 FDP.py "[(0,0), (0,10), (5,5)]"
```

ונקבל את הפלט הבא:

```
Terminals[(5 ,5) ,(10 ,0) ,(0 ,0)] :
Total RSMT length: 15.0
Steiner points: [(0.0, 5.0)]
Edges:
  ((0, 0), (0.0, 5.0))
  ((0, 10), (0.0, 5.0))
  ((5, 5), (0.0, 5.0))
```

בקובץ זה עשינו שימוש בספריית **GeoSteiner** (קרדיט מתאים מצורף בפרק האחרון) לצורך חישוב האורך של עץ שטיינר המינימלי (RSMT) עבור תת-קבוצות של טרמינלים. הספרייה מופעלת בצורה נקודתית על מנת להחזיר את אורך העץ האופטימלי בלבד עבור קבוצה נתונה, ואינה משמשת ככלי לבניית העץ האופטימלי עבור כלל הקלט.

GeoSteiner מהווה רכיב **ניתן להחלפה** בפרויקט. תפקידו מסתכם בהחזרת עלות מדויקת של פתרון עץ שטיינר עבור קבוצה קטנה של נקודות, ולכן ניתן להחליפה בעתיד באלגוריתם אחר, תוכנה שונה, או חישוב פנימי, ללא צורך לשנות את מבנה האלגוריתם הדינמי.

תיאור כללי של הפונקציה compute_rsmt ופעולתה

הפונקציה compute_rsmt מיועדת לחשב את העץ השטיינר המינימלי הרקטילינארי (Rectilinear Steiner Minimal Tree – RSMT) עבור קבוצת טרמינלים (נקודות קלט במישור עם קואורדינטות x,y). בעיה זו דורשת למצוא עץ בעל אורך כולל מינימלי, המחבר את כל הנקודות הנתונות באמצעות קשתות אופקיות ואנכיות בלבד (מרחק מנהטן), וניתן להוסיף בו נקודות שטיינר – נקודות נוספות (שאינן טרמינלים מקוריים) שבהן מסלולי הקשתות יכולים להיפגש כדי לקצר את אורך העץ. הפונקציה מחזירה שלושה פרמטרים: אורך כולל מינימלי, רשימת נקודות שטיינר, רשימת הקשתות בעץ. האלגוריתם ממומש באמצעות תכנות דינמי הסוקר את כל תתי-הקבוצות של הטרמינלים ובונה בהדרגה עץ מינימלי לכל תת-קבוצה. הרעיון המרכזי הוא לחשב עבור כל תת-קבוצה של טרמינלים את עלות העץ המינימלי שלה בשתי דרכים אפשריות:

1. <u>עץ מלא (Full Tree)</u> – פתרון ישיר של בעיית שטיינר עבור תת-הקבוצה כולה, המניב את עץ השטיינר המינימלי לאותה קבוצה (יתכן עם נקודות שטיינר משלו).
2. <u>פיצול בקודקוד</u> – חלוקת התת-קבוצה לשתי קבוצות קטנות יותר שנפגשות בטרמינל משותף אחד, וחישוב העלות כסכום העצים המינימליים של שתי הקבוצות המחוברות דרך אותו טרמינל. הפיצול נבדק עבור **כל טרמינל אפשרי** בתת-הקבוצה כנקודת המפגש (join point), ועבור **כל אפשרות חלוקה** של שאר הנקודות מסביבה לשתי קבוצות (כל תת-קבוצה בלתי-ריקה ובלתי-שלמה מתוך שאר הנקודות).

הפונקציה compute_rsmt מבצעת מספר שלבי חישוב עיקריים:

בדיקת מקרי בסיס: אם מספר הטרמינלים n הוא 0, מוחזר עץ ריק (אורך 0 וללא קשתות או נקודות שטיינר). אם n הוא 1, אין צורך בחיבור – מוחזר אורך 0 ללא נקודות שטיינר או קשתות (כי נקודה בודדת אינה דורשת עץ).

אתחול ספריית GeoSteiner: הספרייה הגיאומטרית GeoSteiner משמשת כאן כפונקציה חיצונית לפתרון בעיית שטיינר בתת-מרחב של טרמינלים.

הכנת מבני נתונים: הפונקציה שומרת את רשימת הטרמינלים ב־coords (רשימת זוגות (x,y)) להנגשה נוחה. בנוסף, מחושבות גדלי מקסימום למבני נתונים שידרשו ל-GeoSteiner: מספר נקודות שטיינר מרבי (max_sps = n-2) ומספר קשתות מרבי (max_edges = 2n-3) – אלו נובעים מתכונות עץ שטיינר (לעץ שטיינר עם n טרמינלים מקסימום n-2 נקודות שטיינר ו-2n-3 קשתות בעץ מלא). בהתאם לכך מוקצים באורך קבוע מערכים עבור נקודות שטיינר (sps_buf) וקשתות (edges_buf), וכן משתנים לאחסון אורך העץ (length_buf), מונה נקודות שטיינר (nsps_buf) ומונה קשתות (nedges_buf). מבנים אלה מוגדרים בסיוע ספריית ctypes בפייתון, כך שניתן יהיה להעבירם למתודות בספריית GeoSteiner שפועלות בקוד C.

פונקציית עזר לחישוב עץ מלא: בתוך compute_rsmt מוגדרת פונקציה פנימית full_tree_cost(sub_indices), המחזירה את אורך העץ השטיינר המינימלי המלא עבור תת-קבוצה נתונה של טרמינלים (מבלי לפרק לתת-חלקים). הפונקציה הזו אורזת את קואורדינטות תת-הקבוצה (רשימת אינדקסים sub_indices) למערך C רציף של נקודות (terms_array), ואז קוראת לפונקציה gst_rsmt של GeoSteiner כדי לפתור את בעיית ה-RSMT עבור הנקודות הללו. היא בודקת את ערך החזרה והסטטוס של הקריאה – אם התרחשה שגיאה, תיזרק RuntimeError; אחרת, היא קוראת את אורך העץ המינימלי מתוך length_buf ומחזירה אותו. שימו לב: בפונקציה זו אנו לא אוספים עדיין את פרטי נקודות השטיינר או הקשתות – רק את האורך. הדבר נעשה לשם ייעול התכנות הדינמי: חישוב האורך מספיק כדי להחליט על מינימום, ואת מבנה העץ נשחזר רק בשלב מאוחר יותר עבור העץ האופטימלי הסופי.

טבלת תכנות דינמי (DP): האלגוריתם משתמש במילון (dict) בשם dp_cost למיפוי תת-קבוצה של טרמינלים לעלות המינימלית שמצאנו עבורה, ובמילון dp_choice למיפוי תת-קבוצה לבחירת הפיצול האופטימלית (אם היה פיצול) שעזר להשיג את העלות הזו. תת-קבוצה מיוצגת באמצעות מסכה של ביטים (bit mask) מסוג מספר שלם: לכל טרמינל יש ביט במיקום הייחודי לו (לפי אינדקס הטרמינל ברשימה), וביט זה מוגדר ל-1 אם הטרמינל בתת-הקבוצה. כך, כל תת-קבוצה מזוהה ע"י מספר ייחודי (מסכה) שבו הביטים הדולקים מייצגים את האינדקסים של הטרמינלים הכלולים. בהתחלה, לכל תת-קבוצה בגודל 1 (מסכה עם ביט בודד) מוקצית עלות 0 בטבלת dp_cost והבחירה None בטבלת dp_choice (כי אין צורך בעץ או פיצול עבור נקודה יחידה).

לולאת תכנות דינמי – חישוב עלויות עבור כל תת-קבוצה: האלגוריתם מחשב את עלות עץ השטיינר המינימלי עבור כל תת-קבוצת טרמינלים מהקטנה לגדולה (גודל 2 עד n).

1. קודם כל, עבור כל תת-קבוצה C, מחושב האורך של עץ שטיינר מלא בעזרת קריאה ל-GeoSteiner. התוצאה נשמרת כעלות הטובה ביותר הנוכחית.

2. אם גודל הקבוצה C הוא 3 ומעלה, נבדקות כל אפשרויות הפיצול האפשריות:

- בוחרים כל טרמינל i בקבוצה כנקודת חיבור.

- עבור כל תת-קבוצה F מתוך שאר הנקודות (ללא i), מחושבת עלות של פיצול – כלומר עלות של עץ המחבר את F ביחד עם i, ועוד עץ המחבר את שאר הנקודות (כולל i).

- אם הפיצול נותן עלות קטנה יותר מהעץ המלא – שומרים אותו כפתרון הטוב ביותר לקבוצה הזו.

3. לבסוף שומרים לכל תת-קבוצה את העלות המינימלית ואת אופן החיבור (האם זו קבוצה אחת או פיצול לשתי קבוצות).

תוצאה אופטימלית כוללת: לאחר סיום הלולאות, החישובים הושלמו עבור כל תת-קבוצה – ובפרט עבור תת-הקבוצה שהיא כל הטרמינלים (גודל n). מהטבלה נשלפת העלות האופטימלית total_length = dp_cost[full_mask]. זהו אורך ה-RSMT המבוקש. בשלב זה יש לנו גם את כל הנתונים כדי לשחזר את מבנה העץ עצמו.

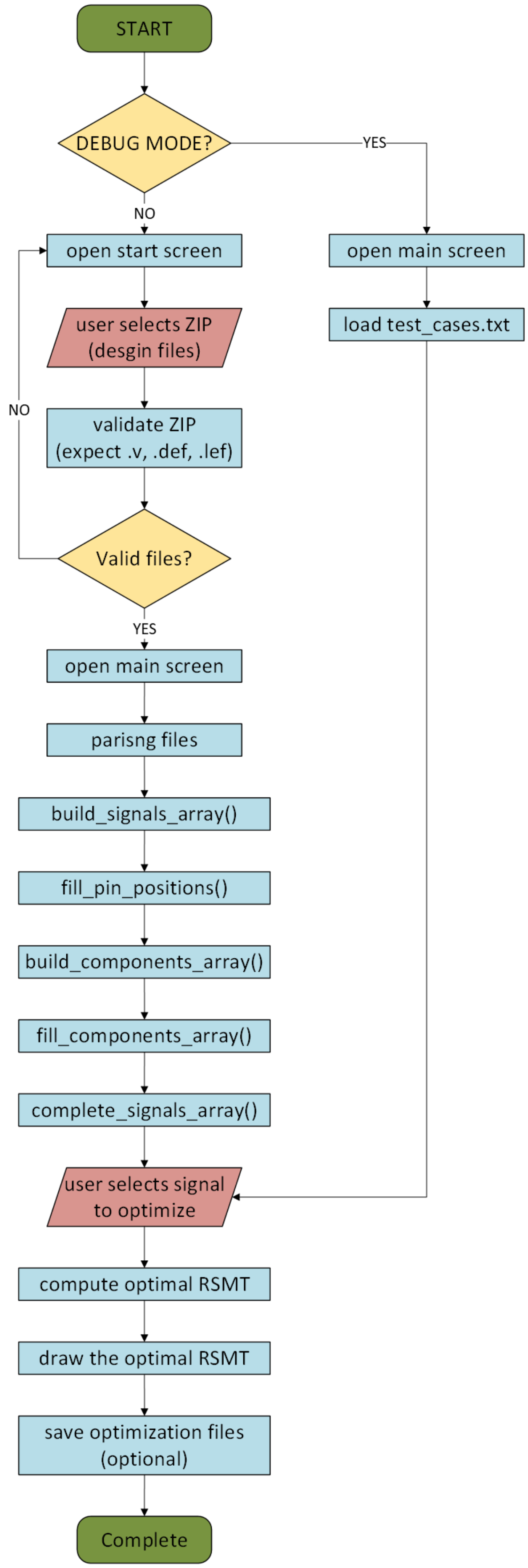
שלב השחזור – בניית העץ בפועל: אחרי שמחושבת העלות המינימלית לכל תת-קבוצה, משחזרים את מבנה העץ עצמו – כלומר את נקודות השטיינר ואת הקשתות.

אם הפתרון לתת-הקבוצה היה עץ מלא (כלומר ללא פיצול), מבצעים קריאה ל-GeoSteiner פעם נוספת, הפעם כדי לקבל את מבנה העץ בפועל – נקודות שטיינר וקשתות.

אם הפתרון היה פיצול – מבצעים רקורסיה על שתי הקבוצות שהתקבלו מהפיצול, ובונים עץ מכל אחת מהן.

לאחר מכן מאחדים את הקשתות והנקודות משני תתי-העצים, ומסננים כפילויות.

:Program Flowchart



אתגרים במימוש

במהלך העבודה על המערכת נתקלו מספר אתגרים טכנולוגיים ופרקטיים, אשר טופלו באמצעות פתרונות מותאמים, תכנון מחדש של חלקים מהמערכת, ואינטגרציה עם כלים חיצוניים. להלן עיקרי האתגרים:

1. **ייצוג הנתונים בפורמט קריא ונגיש**: בתחילה השתמשנו בקובץ Excel לניהול וארגון מיקומי הטרמינלים. עם הזמן נמצא כי הפורמט הזה מסורבל לקריאה ונדרש עיבוד מיותר. לפיכך, הוחלט לעבור לפורמט JSON, אשר מאפשר ייצוג היררכי נוח יותר, קריא למכונה ולאדם, ותומך בצורה טבעית באובייקטים כמו מערכים ומילונים.

2. **בעיות בצילום המסך לאחר אופטימיזציה**: בעת ניסיון לבצע צילום מסך אוטומטי של המסך הכולל את מבנה העץ (RSMT), נתקלנו בבעיה לפיה הקווים המחברים בין הנקודות לא הופיעו בתמונה. לאחר חקירה נמצא כי הצילום מתבצע לפני סיום ציור האלמנטים הגרפיים על גבי הקנבס. פתרנו זאת באמצעות הוספת השהייה (wait) קצרה לאחר פעולות הציור ולפני הצילום.

3. **מעבר מסביבת Windows לסביבת Ubuntu**: במהלך הפיתוח התברר כי חלק מהספריות, הכלים החיצוניים (כגון GeoSteiner), או יכולות המערכת אינם נתמכים היטב ב-Windows. לכן הוחלט לבצע מעבר לפיתוח בסביבת Ubuntu, דבר שהקל על ההרצה, ההתקנה, והתמיכה בספריות קוד פתוח.

4. מימוש אלגוריתם FullTree באופן עצמאי: בשלבים הראשונים ניסינו לממש בעצמנו את הפונקציה FullTree, אשר מחשבת את עץ שטיינר האופטימלי עבור קבוצה של טרמינלים. השקענו מאמץ רב בפיתוח הפונקציה, תוך התבססות על רעיונות גיאומטריים כגון חיבור זוגות נקודות, מציאת נקודות אמצע (medians), ופתרונות pairwise. עם זאת, לאחר מחקר מעמיק של הספרות האקדמית בתחום, התברר כי הבעיה מורכבת בהרבה ממה שנראה במבט ראשון, ודורשת טיפול במספר רב של טופולוגיות אפשריות.

במאמר שקראנו הפונקצייה מתוארת בתור "קופסה שחורה" (black box), כלומר: מדובר במודול חישוב עצמאי, יעיל, אך לא טריוויאלי ליישום ידני. בהתאם לכך, בחרנו להתייחס לפונקציה זו באותו האופן – ולהשתמש בספריית GeoSteiner לצורך חישוב נקודתי של עלות ועץ שטיינר אופטימלי עבור קבוצות קטנות, כחלק מהאלגוריתם הדינמי הכולל שלנו.

5. באגים בהתנהגות הממשק הגרפי: נתקלנו בתקלות שונות בממשק המשתמש, לדוגמה פתיחה של חלונות כפולים במקרים מסוימים. באגים אלו טופלו באמצעות מנגנון מעקב אחר חלונות פתוחים (open_windows) אשר מונע פתיחה חוזרת של אותו חלון.

6. בעיות של נקודות מחוץ למסך: כאשר נקודות הטרמינלים היו ממוקמות ב־(0,0), הן הופיעו בקצה המסך ולעיתים אף חרגו מגבולות הקנבס. הבעיה זוהתה ונפתרה על ידי התאמת הגריד הגרפי והזזת נקודות בעת הצורך.

7. פיצול מבני בין חישוב האופטימיזציה לפרסינג: בתחילה פותח קובץ יחיד אשר טיפל גם בניתוח הקבצים (DEF/LEF/Verilog) וגם בביצוע האלגוריתם. עם התקדמות הפרויקט, נמצא כי הפרדה בין שלב ה־Parsing לשלב חישוב העץ תשפר את הקריאות, הניהול והתחזוקה של הקוד. לפיכך, המערכת פוצלה לשני רכיבים: main.py (אינטגרציה ו-GUI) ו־FDP.py (לוגיקת חישוב RSMT).

TECHNION
Israel Institute
of Technology

סביבת עבודה (Software Environment)

מטרת פרק זה היא לתאר את סביבת העבודה הנדרשת להפעלת התוכנה שבנינו, כולל הכלים והתוכנות שבהם נעשה שימוש, וכן את מבנה התיקיות והקבצים המרכיבים את המערכת. פרק זה נועד לספק תמונה ברורה של התשתית התוכנתית, לצורך הפעלה, תחזוקה או המשך פיתוח עתידי של הכלי.

התקנות וכלי תוכנה נדרשים

חשוב לציין כי סביבת העבודה של הפרויקט מבוססת על מערכת ההפעלה Linux, ואינה תומכת בהרצה ישירה על Windows.

לצורך הרצת התוכנה, ודאו כי התוכנות והספריות הבאות מותקנות וזמינות לשימוש בסביבה בה אתם עובדים:

- **Python**: שפת התכנות הראשית, המשמשת לארגון הממשק הגרפי, הקריאות לקבצי קלט ופלט, ולחישובי האופטימיזציה. מומלץ גרסה 3.10 ומעלה.

- **Tkinter**: ספריית GUI של Python ליצירת ממשק משתמש גרפי אינטראקטיבי.

- **GeoSteiner**: ספרייה חיצונית (נכתבה בשפת C) לחישוב מדויק של עצי שטיינר רקטילינאריים (RSMT). משולבת בפרויקט בעזרת ctypes. בפרוייקט עשינו שימוש בגרסה 5.3.

ספריות נוספות:
- zipfile, os, re, json, shutil, datetime, tempfile – לניהול קבצים ונתיבים.
- mss, mss.tools – לצילום מסך ולשמירת תיעוד גרפי של האופטימיזציה.
- ast – לפענוח קלט בפורמט מילולי של Python.

התקנת GeoSteiner-5.3

<u>דרישות מוקדמות:</u>

- מערכת לינוקס (Debian/Ubuntu) .
- מותקנים gcc, make, libtool, tar

אם חסר משהו, ניתן להתקין עם:

```
sudo apt update
sudo apt install build-essential libtool
```

1. חילוץ קבצי ההתקנה (בהנחה שאתה נמצא בתיקיית הפרויקט שלך):

```
tar -xf geosteiner-5.3.tar
cd geosteiner-5.3
```

2. הפעלת configure עם תמיכה ב־-fPIC (נחוץ לשלב הבא):

```
./configure CFLAGS="-fPIC"
```

ניתן להתעלם מאזהרה unrecognized options: --enable-shared אם היא מופיעה.

3. בניית הספריות עם תמיכה ב־fPIC:
קומפילציה של GeoSteiner :

```
make clean
make -j$(nproc)
```

4. יצירת הספרייה הדינמית libgeosteiner.so:
עבור לתיקיית האובייקטים:

```
cd .libs
```

צור את הקובץ הדינמי:

```
gcc -shared -o libgeosteiner.so *.o ../lp_solve_2.3/*.o -lm
```

העבר אותו לתיקייה הראשית (לשימוש נוח בסקריפטים):

```
cp libgeosteiner.so ..
```

5. סיום: כעת תוכל להריץ את הקוד src/main.py

התקנות לטובת הרצת הסקריפט validate.sh

1. וודא שאתה בתיקייה geosteiner-5.3.

2. נבנה את הקבצים הבינאריים:

```
make rfst bb
```

3. נבדוק שהם קיימים:

```
ls rfst bb
```

4. באותו טרמינל שבו אתה מריץ את validate.sh תוכל להוסיף את מיקום הקובץ libgeosteiner.so למשתנה LD_LIBRARY_PATH :

```
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:../geosteiner-5.3
```

5. סיום: כעת תוכל להריץ את הסקריפט. דוגמת הרצה:

```
./validate.sh "[(53, 266), (480, 271), (423, 142), (137, 95), (1009, 23)]"
```

6. הפתרון בסעיף 4 הוא זמני, יש במקומו תיקון קבוע אם רוצים:
6.1. פתח את קובץ הקונפיגורציה של ה-shell שלך (תלוי ב-shell שלך):

```
nano ~/.bashrc
```

אם אתה משתמש ב-Zsh , תכניס את הפקודה הבאה במקום הקודמת:

```
nano ~/. zshrc
```

6.2. הוסף את השורה הבאה בסוף הקובץ (כמובן שהנתיב תלוי במבנה התיקיות שלך):

```
export
LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/home/os/Documents/VLSI_Routing_Opti
mizer_Tool/geosteiner-5.3
```

6.3 שמור וצא מהקובץ.

6.4 הפעל את השינויים שעשית על ידי:

```
source ~/.bashrc
```

TECHNION
Israel Institute
of Technology

מבנה תיקיות וקבצים

המבנה של תיקיית הפרויקט VLSI_Routing_Optimizer_Tool הוא כדלקמן:

```
VLSI_Routing_Optimizer_Tool/
 ──├src               /
 ──├    │main.py
 ──├    │FDP.py
 ──├    │validate.sh
 ──└    │tmp_files/
 │
 ──├tests
 ──└    │test_cases.txt
 │
 ──├Input_Files
 ──├    │DEBUG_MODE/
 │          ──└    │DEBUG_MODE.zip
 ──├    │Design_1/
 │          ──└    │Design_1.zip
 ──└    │Design_2/
 │          ──└     Design_2.zip
 │
 ──├Output_Files
 ──├    │DEBUG_MODE_2025-01-01_10-50-00/
 │        _3 ──├    │points_test_1/
 │          │          _3 ──├    │ points_test_1_optimization_data.json
 │          │          _3 ──└    │ points_test_1_optimization_data.png
 │          ──└    │log.txt
 │          │
 ──├    │Design_1_2025-01-01_10-50-00/
 │          ──├    │signal_a/
 │          │          ──├    │ signal_a_optimization_data.json
 │          │          ──└    │ signal_a_optimization_data.png
 │          │          │
 │          ──├    │signal_b/
 │          │          ──├    │ signal_b_optimization_data.json
 │          │          ──└    │ signal_b_optimization_data.png
 │          │          │
 │          ──└    │log.txt
 │          │
 ──└    │Design_2_2025-01-01_10-50-00/
 │          ──├     signal_a/
 │                     ──├    │ signal_a_optimization_data.json
 │                     ──└    │ signal_a_1_optimization_data.png
 │          ──└     log.txt
 │
 ──└geosteiner-5.3
```

<u>פירוט תכולת התיקיות</u>:
src - תיקייה המכילה את קבצי הפייתון להרצת התוכנית.
main.py - הקובץ הראשי להפעלת הממשק הגרפי והאופטימיזציה.
FDP.py - מימוש אלגוריתם FDP בעזרת ספריית GeoSteiner.
Input_Files - תיקייה לקבצי קלט (ZIP עם קבצי Verilog/DEF/LEF)
Output_Files - תיקייה לשמירת קבצי תוצאה ואופטימיזציה (תיווצר אוטומטית)
geosteiner-5.3 - ספריית GeoSteiner כולל libgeosteiner.so
test_cases.txt - קובץ המכיל טסטים (לטעינה במצב DEBUG)

<u>הערה חשובה</u>: להפעלת הכלי במצב DEBUG, יש לוודא שבתוך /Input_Files קיים קובץ בשם DEBUG_MODE.zip.
מצב זה מאפשר דילוג על מסך בחירת הקלט וטעינה אוטומטית של קובץ מוגדר מראש לצורך בדיקות.

ממשק משתמש (GUI)

ממשק המשתמש הגרפי (GUI) נבנה באמצעות ספריית Tkinter בשפת Python, ומטרתו לספק למשתמש דרך אינטואיטיבית וידידותית להזין קבצים, להפעיל את האלגוריתם המבצע אופטימיזציית routing עבור סיגנל שהמשתמש בחר, ולבחון את תוצאות החישוב על גבי ייצוג גרפי ברור.
הממשק כולל שני שלבים עיקריים: מסך פתיחה ומסך ראשי, אשר מוצגים למשתמש לפי סדר הפעולה.

הסבר על הממשק

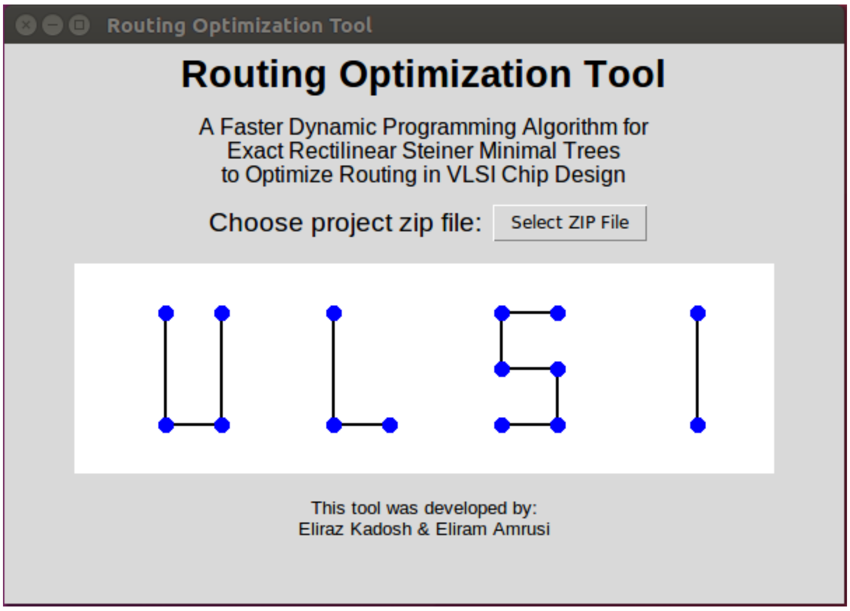
1. **מסך פתיחה**
הפעלת התוכנה נעשית על ידי הכנסת שורת הפקודה הבאה בטרמינל:

```
python3.10 main.py
```

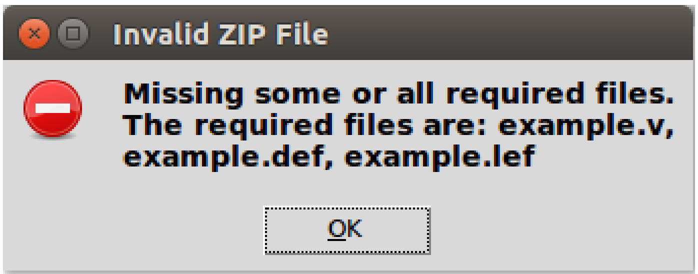
עם עליית התוכנה, מוצג למשתמש מסך פתיחה פשוט הכולל כפתור:

- "Select ZIP file": מאפשר למשתמש לבחור קובץ ZIP המכיל את קבצי העיצוב (Verilog, DEF, LEF).

לאחר בחירה תקינה, המערכת מחלצת את הקבצים וממשיכה למסך הראשי.

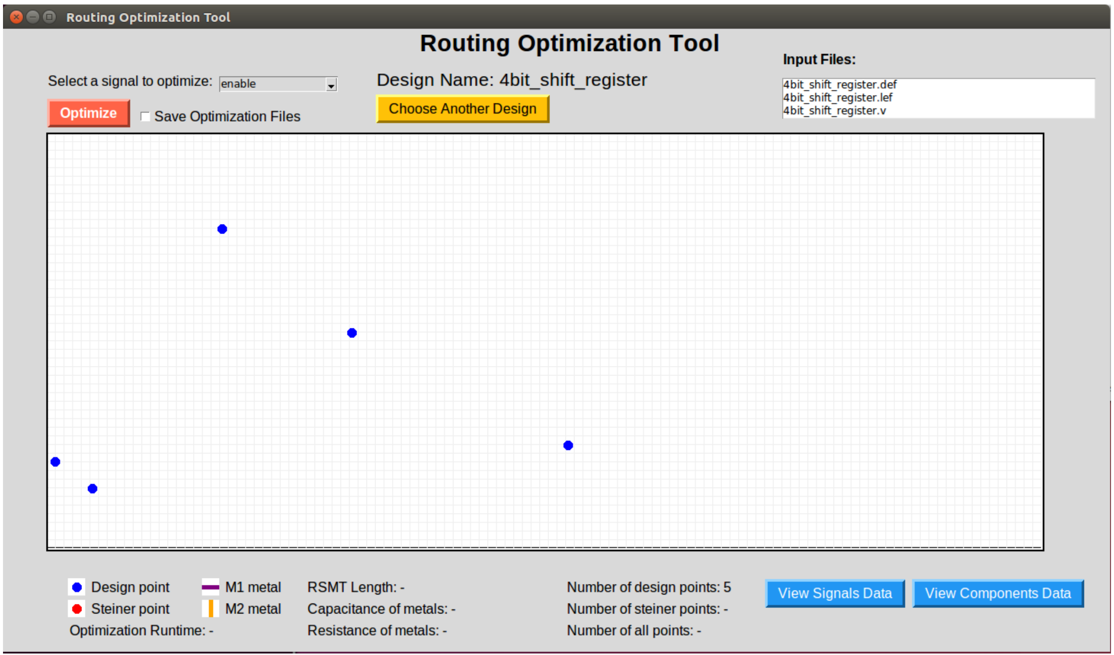


במידה וקובץ הZIP שנבחר אינו תקין, תופיע השגיאה הבאה:

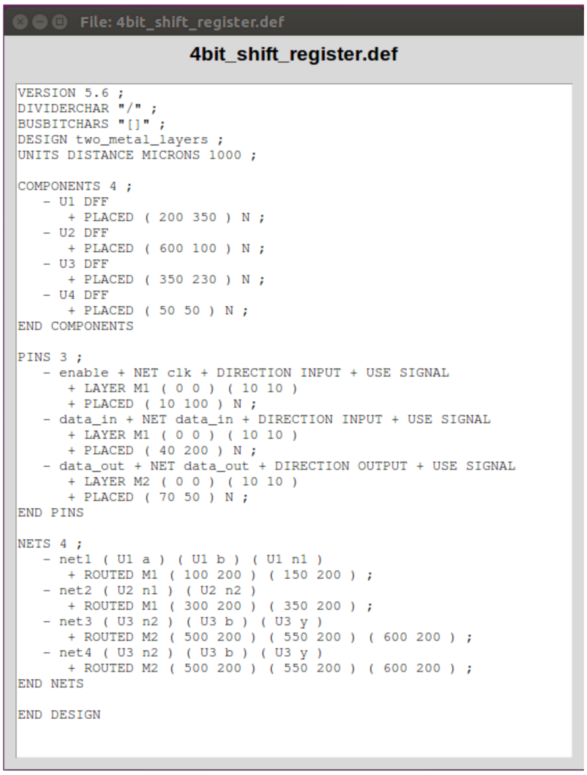


2. **מסך הראשי**

לאחר טעינת הפרויקט, מוצג המסך המרכזי, הכולל את הרכיבים הבאים:



- רשימת Input Files של ה-design הנבחר מכיל. לחיצה על כל אחד מהם תפתח חלון ובו מוצג הקובץ. למשל בלחיצה על קובץ הDEF יופיע החלון:



File: 4bit_shift_register.def

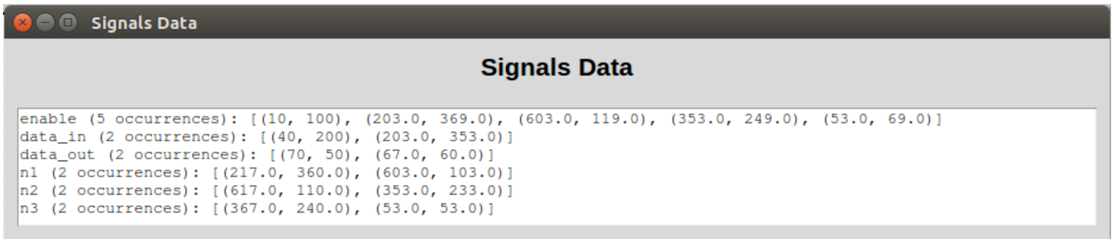
4bit_shift_register.def

```
VERSION 5.6 ;
DIVIDERCHAR "/" ;
BUSBITCHARS "[]" ;
DESIGN two_metal_layers ;
UNITS DISTANCE MICRONS 1000 ;

COMPONENTS 4 ;
   - U1 DFF
      + PLACED ( 200 350 ) N ;
   - U2 DFF
      + PLACED ( 600 100 ) N ;
   - U3 DFF
      + PLACED ( 350 230 ) N ;
   - U4 DFF
      + PLACED ( 50 50 ) N ;
END COMPONENTS

PINS 3 ;
   - enable + NET clk + DIRECTION INPUT + USE SIGNAL
      + LAYER M1 ( 0 0 ) ( 10 10 )
      + PLACED ( 10 100 ) N ;
   - data_in + NET data_in + DIRECTION INPUT + USE SIGNAL
      + LAYER M1 ( 0 0 ) ( 10 10 )
      + PLACED ( 40 200 ) N ;
   - data_out + NET data_out + DIRECTION OUTPUT + USE SIGNAL
      + LAYER M2 ( 0 0 ) ( 10 10 )
      + PLACED ( 70 50 ) N ;
END PINS

NETS 4 ;
   - net1 ( U1 a ) ( U1 b ) ( U1 n1 )
      + ROUTED M1 ( 100 200 ) ( 150 200 ) ;
   - net2 ( U2 n1 ) ( U2 n2 )
      + ROUTED M1 ( 300 200 ) ( 350 200 ) ;
   - net3 ( U3 n2 ) ( U3 b ) ( U3 y )
      + ROUTED M2 ( 500 200 ) ( 550 200 ) ( 600 200 ) ;
   - net4 ( U3 n2 ) ( U3 b ) ( U3 y )
      + ROUTED M2 ( 500 200 ) ( 550 200 ) ( 600 200 ) ;
END NETS

END DESIGN
```

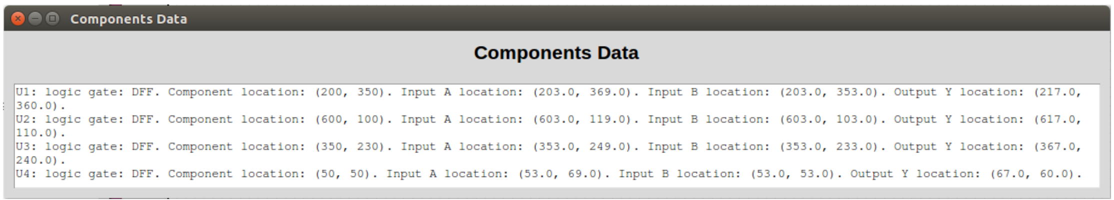
- רשימת signals – מציגה את כל הסיגנלים הקיימים. המשתמש יכול לבחור סיגנל אחד לניתוב.

* כפתור "Choose Another Design" – משמש להחלפה של קבצי design. מחזיר את המשתמש למסך הפתיחה.

* כפתור "View Signals Data" – מציג את כל הסיגנלים עם הקואורדינטות הנדרשות לחיווט (Design points).



* כפתור "View Components Data" – מציג את כל הרכיבים (שערים לוגיים) עם הנתונים שלהם.



* כפתור "Optimize" – מפעיל את אלגוריתם FDP על הסיגנל שנבחר וה-RSMT האופטימלי מחושב. סימון הcheckbox שסמוך אליו ישמור צילום של צורת העץ וקובץ JSON עם כל הנתונים בתיקיית Output files.

* קנבס גרפי – מציג את התוצאה החזותית של עץ שטיינר, כולל הטרמינלים, הקווים המחברים, ונקודות שטיינר.

קלט/פלט מהמשתמש

קלט מהמשתמש:

* קובץ ZIP המכיל קבצי Verilog, LEF, DEF

* בחירת סיגנל מהרשימה

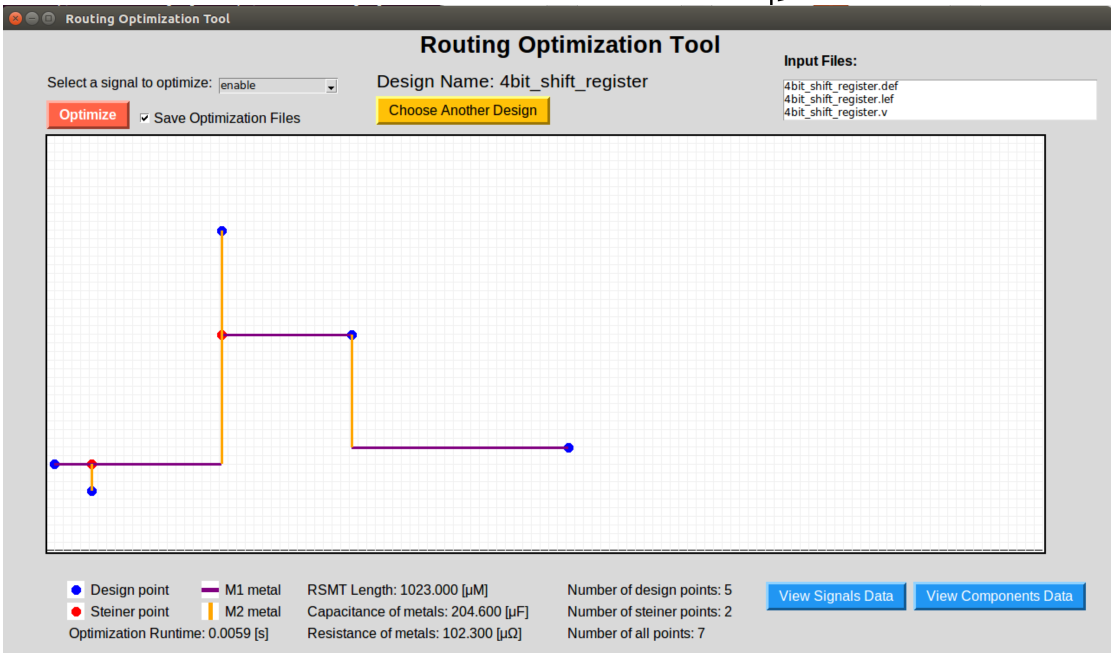
* הפעלת כפתור "Optimize"

<u>הבהרה לגבי הכלי שלנו:</u>
בתהליך Physical Design אמיתי, שלב ה-Routing מתבצע רק לאחר CTS, כלומר לאחר בניית עץ השעון. למרות זאת, הכלי שלנו מאפשר למשתמש לבצע Routing גם לרשת השעון (Clock Net). זה נעשה לצורכי הדמיה, ניתוח או תרגול, אך חשוב להבין שבפועל, Routing של אותות (ובפרט של ה-Clock) מתבצע רק לאחר CTS.

TECHNION
Israel Institute
of Technology

פלט למשתמש:

- הצגה גרפית של עץ שטיינר.



- הצגת מיקומי נקודות ה-design ונקודות ה-Steiner, אורך העץ, התנגדות וקיבול המתכות (layers), ומשך זמן הריצה של האופטימיזציה.

- הדפסת ושמירת הלוג – החל משלב הפעלת הכלי, התוכנה מציגה הודעות מערכת והדפסות לטרמינל: סטטוס טעינת הקבצים, התקדמות חישוב, והתראות שגיאה, כמו כן, נשמר קובץ log באופן אוטומטי בתיקייה.

ההדפסה למסך ותוכן הlog:

```
Total RSMT length: 1023.000 [μM]

Wire details:
P(10,100) connects to P(53,100): length=43.000 [μM], capacitance=8.600 [μF], resistance=4.300 [μΩ]
P(53,100) connects to P(53,69): length=31.000 [μM], capacitance=6.200 [μF], resistance=3.100 [μΩ]
P(53,100) connects to P(203,100): length=150.000 [μM], capacitance=30.000 [μF], resistance=15.000 [μΩ]
P(203,100) connects to P(203,249): length=149.000 [μM], capacitance=29.800 [μF], resistance=14.900 [μΩ]
P(203,249) connects to P(203,369): length=120.000 [μM], capacitance=24.000 [μF], resistance=12.000 [μΩ]
P(203,249) connects to P(353,249): length=150.000 [μM], capacitance=30.000 [μF], resistance=15.000 [μΩ]
P(603,119) connects to P(353,119): length=250.000 [μM], capacitance=50.000 [μF], resistance=25.000 [μΩ]
P(353,119) connects to P(353,249): length=130.000 [μM], capacitance=26.000 [μF], resistance=13.000 [μΩ]

Total M1 length: 593.000μm, capacitance: 118.600μF, resistance: 59.300Ω
Total M2 length: 430.000μm, capacitance: 86.000μF, resistance: 43.000Ω
M1 Wires (Horizontal):
M1 | (10,100) to (53.0,100.0): length=43.000 [μM], capacitance=8.600 [μF], resistance=4.300 [μΩ]
M1 | (53.0,100.0) to (203.0,100.0): length=150.000 [μM], capacitance=30.000 [μF], resistance=15.000 [μΩ]
M1 | (203.0,249.0) to (353.0,249.0): length=150.000 [μM], capacitance=30.000 [μF], resistance=15.000 [μΩ]
M1 | (603.0,119.0) to (353.0,119.0): length=250.000 [μM], capacitance=50.000 [μF], resistance=25.000 [μΩ]
M2 Wires (Vertical):
M2 | (53.0,100.0) to (53.0,69.0): length=31.000 [μM], capacitance=6.200 [μF], resistance=3.100 [μΩ]
M2 | (203.0,100.0) to (203.0,249.0): length=149.000 [μM], capacitance=29.800 [μF], resistance=14.900 [μΩ]
M2 | (203.0,249.0) to (203.0,369.0): length=120.000 [μM], capacitance=24.000 [μF], resistance=12.000 [μΩ]
M2 | (353.0,119.0) to (353.0,249.0): length=130.000 [μM], capacitance=26.000 [μF], resistance=13.000 [μΩ]

Optimization Runtime: 0.0067 seconds

finished optimization for signal: enable
```

- יצירת תיקייה בשם הסיגנל שעליו נעשתה האופטימיזציה, התיקייה מכילה קובץ PNG עם צורת העץ וקובץ JSON בפורמט הבא:


enable_optimization_data.json


enable_optimization_route.png


```json
{
    "Design_Name": "4bit_shift_register_20250330_120057",
    "Signal_Name": "enable",
    "Optimization_Runtime": "0.0043 seconds",
    "Points": {
        "Design_Points": {
            "Number": 5,
            "Desgin points coordinates": {
                "P1": [
                    10,
                    100
                ],
                "P2": [
                "P3": [
                "P4": [
                "P5": [
            }
        },
        "Steiner_Points": {
            "Number": 2,
            "Steiner points coordinates": {
                "S1": [
                    53.0,
                    100.0
                ],
                "S2": [
            }
        }
    },
    "Wires": {
        "Wires_total_length_(RSMT)": 1023.0,
        "Wires_total_capacitance": 204.6,
        "Wires_total_resistance": 102.3,
        "M1_Wires": {
            "Number": 4,
            "M1_wires_total_length": 593.0,
            "M1_wires_total_capacitance": 118.6,
            "M1_wires_total_resistance": 59.3,
            "M1_wires_coordinates": {
                "wire_1": {
                    "Start": [
                    "End": [
                    "Length": 43.0,
                    "Capacitance": 8.6,
                    "Resistance": 4.3
                },
                "wire_2": {
                "wire_3": {
                "wire_4": {
            }
        },
        "M2_Wires": {
            "Number": 4,
            "M2_wires_total_length": 430.0,
            "M2_wires_total_capacitance": 86.0,
            "M2_wires_total_resistance": 43.0,
            "M2_wires_coordinates": {
                "wire_1": {
                "wire_2": {
                "wire_3": {
                "wire_4": {
            }
        }
    }
}
```

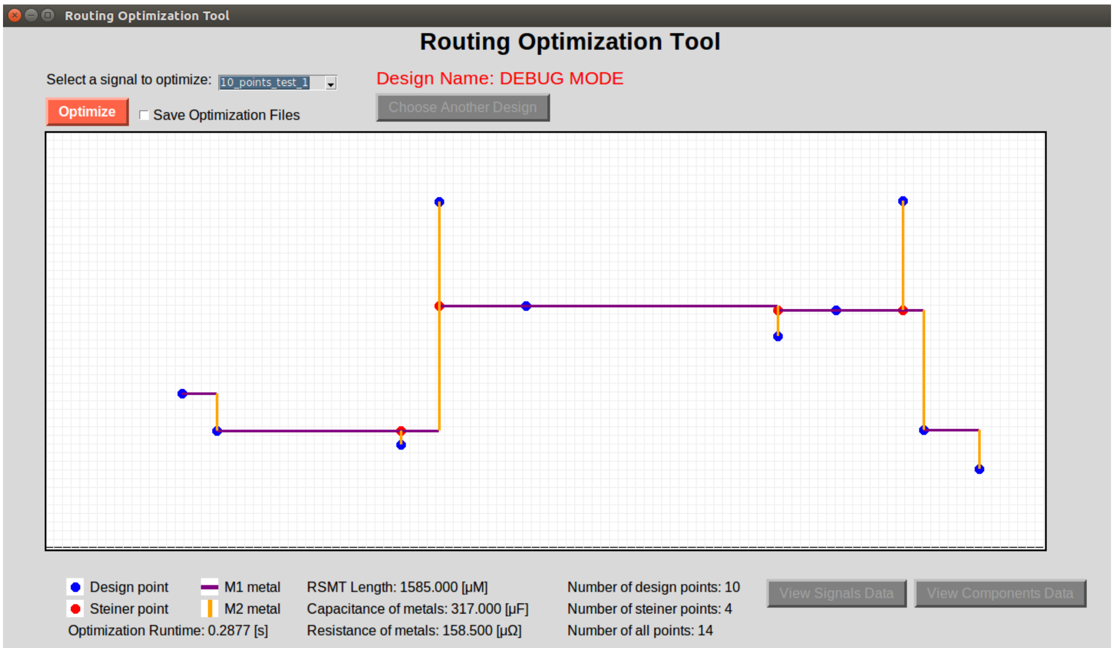
DEBUG MODE – מצב דיבאג

הממשק כולל מצב דיבאג, אשר ניתן להפעילו לצורך בדיקות פנימיות, ניתוח בעיות והבנת התנהגות האלגוריתם.
מצב דיבאג מופעל על ידי הכנסת שורת הפקודה הבאה בטרמינל:

```
python3.10 main.py -D
```

עם עליית התוכנה, מוצג למשתמש מיידית המסך הראשי וכפתורים שאינם רלוונטיים למצב זה - הם במצב Disabled.

הסיגנלים שנטענים הם מתוך קובץ בשם test_cases.txt.



ולידציה

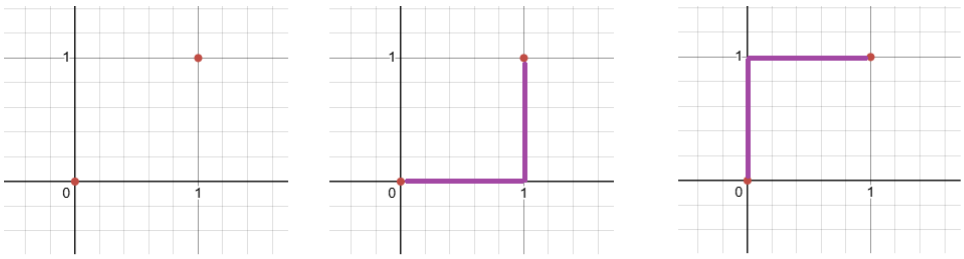
בכדי להעריך את איכות האלגוריתם שפותח ולוודא את תקינותו, בוצעו מספר שלבים של ולידציה והשוואה מול פתרונות ידועים.

תיאור הקריטריונים להשוואה

לצורך הערכת איכות הפתרונות, הוגדרו הקריטריונים הבאים:

- **דיוק**: נמדד על ידי השוואת אורך העץ שהתקבל מהתוכנה שלנו לעומת אורך העץ האופטימלי, כפי שמתקבל מ-GeoSteiner. חשוב לציין כי עץ Steiner אינו קנוני – כלומר, יכולים להתקיים מספר עצים שונים (במבנה ובמיקום נקודות Steiner) בעלי אותו אורך כולל. לפיכך, לעיתים התקבל עץ באורך זהה אך בצורת חיבור שונה מזו של GeoSteiner.

  קל לראות כי עבור הנקודות: (1,1) (0,0) יש שתי פתרונות אופטימליים מאוד ברורים עם אורך עץ זהה אך צורה שונה:



המחשה כי עץ שטיינר אינו קנוני

- **זמן ריצה**: נמדד עבור כל מקרה מבחן, במטרה להעריך את סקלאביליות האלגוריתם ככל שמספר הטרמינלים עולה. השוואה של התוצאה שלנו מול התוצאות מהמאמר ומול DW תוצג בפרק "ניתוח תוצאות".
- **שלמות הפתרון**: נבדק האם האלגוריתם מצליח למצוא פתרון תקף עבור כל קלט (כלומר עץ המקשר את כל הטרמינלים באמצעות קווים רקטיליניאריים בלבד).
- **יציבות**: נבדקה עקביות התוצאה במקרים של קלטים דומים או זהים.

בדיקות על קלטים עם פתרון ידני

לצורך אימות ראשוני של נכונות האלגוריתם, נבחרו מספר מקרי מבחן פשוטים (2–5 טרמינלים), עבורם ניתן לחשב פתרון אופטימלי באופן ידני.
האלגוריתם הופעל על קלטים אלו, והתוצאות הושוו באופן ישיר לפתרונות הידניים.
הבדיקות אישרו כי האלגוריתם מספק פתרונות תקפים עם אורך זהה לפתרון הידני, מה שמעיד על נכונות הבסיס של הפתרון.

השוואת תוצאות מול GeoSteiner

לצורך ולידציה רחבת היקף, השתמשנו ב-GeoSteiner – ספרייה מתקדמת לפתרון בעיות Steiner – להשוואה בין הפתרונות שהתקבלו מהאלגוריתם שלנו לבין פתרונות אופטימליים. מעבר לשימוש שעשינו ב-GeoSteiner כחלק מהפיתוח של הכלי שלנו (באמצעות ממשק Python), הספרייה מאפשר גם חישוב עצמאי ומדויק של עצי Steiner עבור מספר וריאנטים של הבעיה, ביניהם:

- Euclidean Steiner Tree Problem
- Rectilinear Steiner Tree Problem
- Uniformly-Oriented Steiner Tree Problem

יכולת זו של GeoSteiner לחשב פתרונות אופטימליים בצורה עצמאית נוצלה כדי להשוות את תוצאות האלגוריתם שלנו, גם מבחינת אורך העץ וגם מבחינת זמן הריצה.

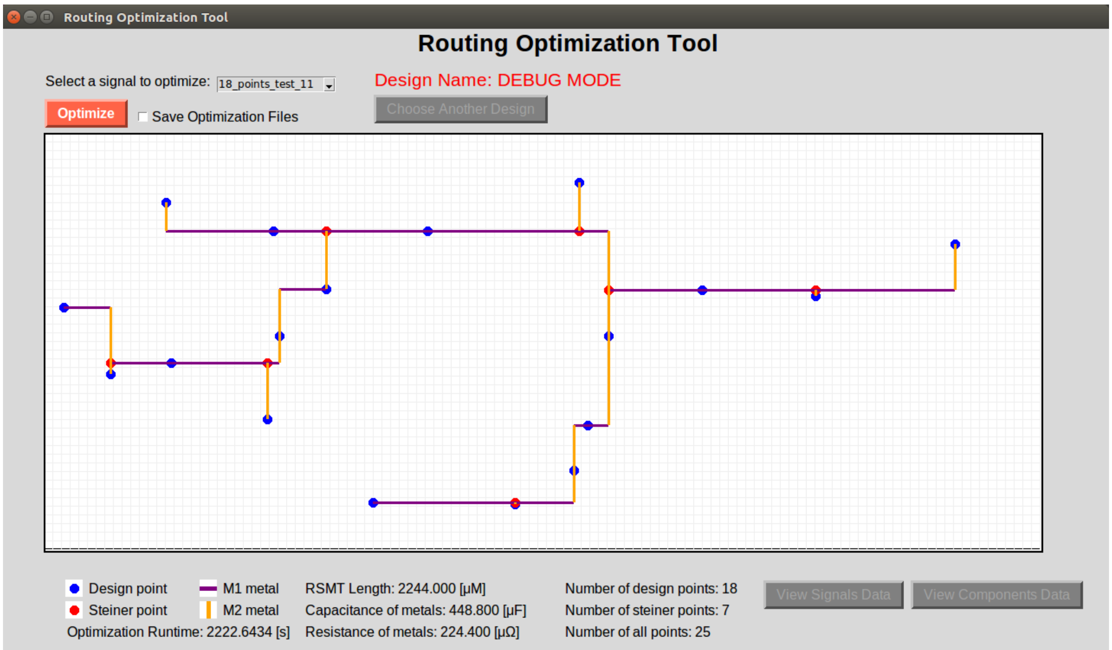
ביצענו הרצה של האלגוריתם על סט מקיף של מקרי מבחן, שכלל קבוצות טרמינלים בגדלים שונים – החל ממקרים פשוטים עם מספר מצומצם של טרמינלים ועד למקרים מורכבים הכוללים 20 טרמינלים. הקואורדינטות נבחרו בצורה רנדומלית, במטרה ליצור שונות גבוהה וייצוג רחב של תרחישים.
עבור כל מקרה, הופעל גם האלגוריתם של GeoSteiner לשם השוואה.

מבחינת אורך העץ האופטימלי, נמצא כי האלגוריתם שלנו מספק תוצאות הזהות לתוצאות של GeoSteiner.
מבחינת ביצועים, נרשם יתרון משמעותי לאלגוריתם של GeoSteiner, במיוחד בקלטים בינוניים וגדולים – בהם זמן הריצה היה גבוה באופן משמעותי מאוד בכלי שלנו.
כאמור, ראוי לציין כי מאחר שעץ Steiner אינו קנוני, לעיתים התקבל עץ שונה בצורתו (מבחינת מבנה או מיקום נקודות Steiner), אך באורך זהה.

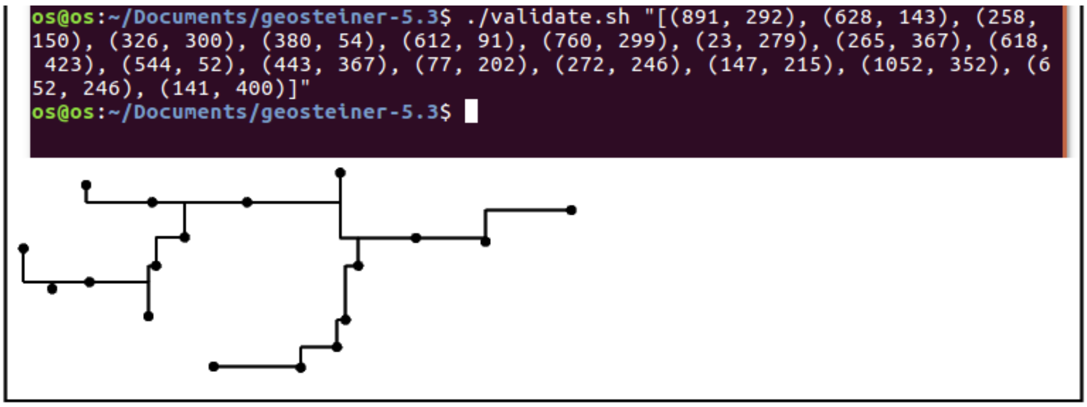
לטובת הרצה מהירה של GeoSteiner כתבנו סקריפט בשם "validate.sh", המקבל בשורת הפקודה את הטרמינלים לחישוב.

השוואה של קלט בגודל 18 טרמינלים:

<u>תוצאת הכלי שלנו:</u>



<u>תוצאת GeoSteiner:</u>



Rectilinear SMT: 18 points, length = 2244, 0.00 seconds

הפתרון שהתקבל זהה באורכו לפתרון האופטימלי, אך שונה בצורתו עקב אי-קנוניות של עץ Steiner.

ניתוח תוצאות

לצורך ניתוח ביצועים, השווינו את זמני הריצה הממוצעים של האלגוריתם שפיתחנו (המבוסס על FDP) מול שלושה מקורות נוספים:

1. **DW (1971, measured in 1994)** – אלגוריתם Dreyfus-Wagner הקלאסי, שתוצאותיו נמדדו במאמר משנת 1994.

2. **FDP (measured in 1994)** – אלגוריתם Full Set Dynamic Programming כפי שנמדד באותו מאמר.

3. **GeoSteiner** – ספריית GeoSteiner.

הסבר לגרף:

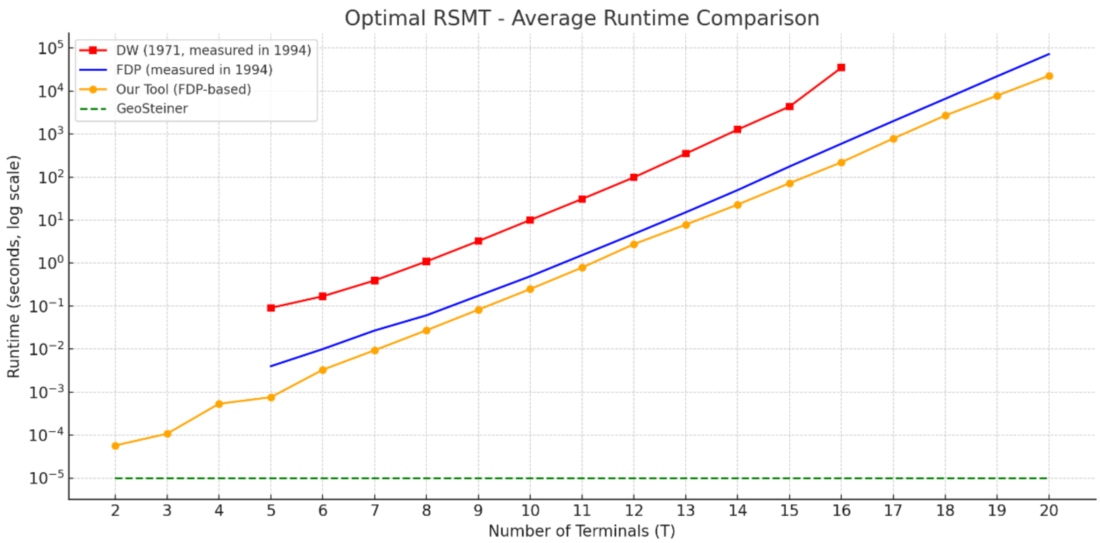
בגרף מוצגות ארבע עקומות המתארות את זמן הריצה הממוצע כפונקציה של מספר הטרמינלים, T, כאשר T נע בין 2 ל־20:

1. DW (1971, measured in 1994) – זמן הריצה עולה באופן אקספוננציאלי ונהיה לא פרקטי מ־17 ואילך.

2. FDP (measured in 1994) – ביצועים טובים בהרבה מ־DW, אך עדיין איטיים יחסית, בעיקר בגלל מגבלות חומרה באותה תקופה.

3. GeoSteiner – מחשב את הפתרון מיידית ומופיע כקו קבוע בתחתית.

4. Our Tool (FDP-based) – מראה שיפור ברור לעומת התוצאות של 1994, הודות למחשוב מודרני. עם זאת, איטי יותר מ-GeoSteiner עקב שכבות עיבוד נוספות ואלמנטים נוספים שאנו לא מודעים אליהם.



מסקנות:

- זמן הריצה של GeoSteiner הוא הנמוך ביותר, מאחר שמדובר בספרייה ממומשת היטב בשפת C, עם אופטימיזציות רבות. הוא משמש אותנו כ"קופסה שחורה" לצורך חישוב עלויות מדויקות.

- התוכנה שפיתחנו, על אף ה־overhead הנוסף (קריאות חוזרות, ניהול זיכרון, רקורסיה וממשק Python), הציגה ביצועים טובים משמעותית מהאלגוריתמים שנמדדו במאמר מ־1994.

- חשוב להדגיש: מטרתנו לא הייתה לשבור שיאי ביצועים או "לנצח" את התוצאות של המאמר. המטרה הייתה ללמוד את האלגוריתם, להבין כיצד הוא מתנהג בפועל, לבנות כלי שמממש אותו בצורה מודולרית, ולראות את תוצאותיו המעשיות על קבוצות טרמינלים שונות.

- מבחינתנו, הצלחנו להפוך חומר תיאורטי לכלי שעובד באמת, וזה מה היה העיקר שלנו בפרוייקט: להבין, לנסות, לראות תוצאות, וללמוד מהן.

- התוצאה היא כלי גמיש שמאפשר להמשיך לשחק עם הרעיונות, לבדוק קבוצות חדשות, לשלב קבצים ממעגלים אמיתיים, ואפילו לחשוב הלאה – איך משפרים או משלבים את זה בפרויקטים עתידיים.

TECHNION
Israel Institute
of Technology

מסקנות

הפרויקט אפשר לנו לחקור לעומק את תחום ה-VLSI, תוך התנסות מעשית באתגרים הקיימים בשלבי התכנון והאופטימיזציה של חיבורים בין רכיבים. נדרשנו לשלב ידע תיאורטי עם יישום פרקטי, מה שאִפשר לנו להבין טוב יותר את החשיבות של כל שלב בתהליך, החל מאיסוף נתונים וכלה בהפקת תוצאה סופית. השימוש בספרייה קיימת (GeoSteiner) הדגים את היכולת לנצל כלים קיימים כדי לייעל תהליכים, לחסוך זמן, ולהתמקד בפיתוח רכיבים ייחודיים. כמו כן, ראינו עד כמה חיוני תכנון מוקדם ומדויק – תכנון לקוי עלול להוביל לקשיים משמעותיים בשלבים מאוחרים יותר של הפיתוח.

תובנות שלמדנו מהפרויקט

במהלך הפרויקט צברנו מספר תובנות מרכזיות, אשר תרמו להעמקת הידע והמיומנויות המקצועיות שלנו:

- רכישת יכולת לקרוא ולהבין מאמרים אקדמיים בתחום טכנולוגי מתקדם כדוגמת Rectilinear Steiner Minimal Tree (RSMT), ולתרגם את התוכן התיאורטי לכדי יישום פרקטי במסגרת תהליך הפיתוח.
- פיתחנו מיומנויות דיבוג, תוך התמודדות עם תרחישים מורכבים, הבנת תיעוד של קוד חיצוני, וניתוח מעמיק של זרימת הנתונים במערכת.
- הבנו את החשיבות של עבודת צוות אפקטיבית, לרבות חלוקה שיטתית של משימות, תיאום, ושיתוף ידע, אשר הובילו להתקדמות טובה.
- התחדדה ההבנה לגבי הצורך בתכנון מקדים ומבוסס, תוך בחינת תרחישים עתידיים ושמירה על מבנה קוד מודולרי וגמיש, המאפשר תחזוקה והרחבה קלה בעתיד.
- רכשנו ניסיון בשימוש בכלים תוכניתיים קיימים, תוך הבנת יתרונותיהם ומגבלותיהם, והטמעתם כחלק אינטגרלי ממערכת הפיתוח לצורך ייעול תהליכים ושיפור התוצאה הסופית.

מקורות וקרדיטים

1. A faster dynamic programming algorithm for exact rectilinear Steiner minimal trees

J.L. Ganley, J.P. Cohoon

https://ieeexplore.ieee.org/document/289962

2. Exact Algorithms for the Steiner Tree Problem

Xinhui Wang

https://research.utwente.nl/files/6039875/thesis_Wang_Xinhui.pdf

3. GeoSteiner 5.3
This project uses GeoSteiner 5.3, developed by David M. Warme, Pawel Winter, and Martin Zachariasen.

The software is licensed under the Creative Commons Attribution-NonCommercial 4.0 International License, and is intended for non-commercial use only.

The distribution includes third-party components such as triangle and lp_solve_2.3, which are subject to their own licenses. This project does not use these components directly.

http://www.geosteiner.com/

http://www.geosteiner.com/geosteiner-5.3-manual.pdf