

# 以 TSP 為載體進行簡易形式驗證與演算法擴增

## 1. 引言 (Introduction)

本報告旨在學習較為日常可見之 NP 問題的演算法原理，並自形式驗證 (Formal Verification, FV) 領域的相關文獻之研讀發想<sup>1</sup>，先以 Verilog 語言製作 TSP 問題的模型，再以開源工具(abc tools)進行驗證，最後嘗試實作基因遺傳演算法之基礎模型，並以更為複雜的 TSP 問題作為測試資料，以進行改良。

由上，本文選定旅行推銷員問題 (Traveling Salesman Problem, TSP) 為主要研讀與實作主題。其定義為：給定一系列城市以及各城市之間的距離，目標是找出訪問每一座城市恰好一次，並最終返回出發城市的最短可能路徑。該問題已被證明為 NP-hard 問題，這意味著隨著城市數量的增加，找到絕對最優解所需的時間會呈指數級增長，使得暴力窮舉法對於實際規模的問題變得不可行。

### 1.2 納入比較的演算法 (Algorithms Included in Comparison)

本次專題先簡單探討以下五種 TSP 求解演算法

- 基因遺傳演算法 (Genetic Algorithm, GA)  
一種模擬生物進化過程的元啟發式搜索演算法。
- Held-Karp (Dynamic Programming, DP)  
一種能夠在特定條件下找到最優解的精確演算法。
- 模擬退火演算法 (Simulated Annealing, SA)  
一種基於物理退火過程的概率性元啟發式演算法。

---

<sup>1</sup> H. Saafan, M. W. El-Kharashi and A. Salem, "SoC connectivity specification extraction using incomplete RTL design: An approach for Formal connectivity Verification," 2016 11th International Design & Test Symposium (IDT), Hammamet, Tunisia, 2016, pp. 110-114, doi: 10.1109/IDT.2016.7843024. keywords: {Formal verification;Documentation;Pins;Standards;Libraries;Hardware design languages;Data mining;Bus interface;Connectivity;Formal Verification;Integration;IP Reuse;IP-XACT;System-on-Chip},

- 蟻群最佳化演算法 (Ant Colony Optimization, ACO)  
一種模擬螞蟻覓食行為的元啟發式演算法。
- 最小生成樹的近似演算法 (MST-based Approximation Algorithm, MST)  
一種利用圖的最小生成樹特性來構造近似解的經典演算法。

選擇這五種演算法進行比較，是因為它們代表了求解最佳化問題的不同思路、技術範疇和問題範圍。DP 在各類小範圍的問題中是最常見的，也是教科書內最常出現的演算法之一；最小生成樹近似法則是因其在特定條件下具有時間複雜度保證的方法；而遺傳演算法、模擬退火和蟻群最佳化則是近期較為廣泛應用的元啟發式方法。藉由對這一系列演算法的比較與實作，我大致獲得了在問題規模、期望解的品質以及可用計算資源之間進行權衡的思維訓練。

## 2. 問題模型化與基本測試

本節旨在提供對指定日常問題的 Verilog 電路的建模與基本的 SAT 形式測試。

### 2.1 基礎模型 3-SAT 問題

為熟悉驗證操作與驗證的正確性，故依照經典的 3-SAT 問題進行建模，並依序對 abc tools 中的 bmc、bmc2、bmc3、int 和 pdr 進行測試，同時額外輸出 log，以便進行進一步的效能判斷。

Assertion

```
assign clause_out = l1 || l2 || l3;
```

BMC

```
Converting 1 flops from don't-care to zero initial value.
Running "bmc". AIG: PI/PO/Reg = 7/1/1. Node = 18. Lev = 8.
Time-frames (120): PI/PO = 840/120. Node = 2142. Lev = 8. Time = 0.00 sec
CNF: Variables = 1676. Clauses = 3692. Literals = 10834. Time = 0.01 sec
Solving output 0 of frame 0 ...
Solved 1 outputs of frame 0. Conf = 0. Imp = 2. T = 0.00 sec
Solving output 0 of frame 1 ...
Solved 1 outputs of frame 1. Conf = 0. Imp = 1676. T = 0.00 sec
Output 0 of miter "3sat" was asserted in frame 1. Time = 0.02 sec
```

BMC2

```
Converting 1 flops from don't-care to zero initial value.
Running "bmc2". AIG: PI/PO/Reg = 7/1/1. Node = 18. Lev = 8.
Params: FramesMax = 120. NodesDelta = 2000. ConfMaxOne = 0. ConfMaxAll = 0.
0 : F = 81. 0 = 0. And = 1440. Var = 1045. Conf = 0. 0 MB 0.01 sec
Output 0 of miter "3sat" was asserted in frame 1. Time = 0.01 sec
```

BMC3

```
Converting 1 flops from don't-care to zero initial value.
Running "bmc3". PI/PO/Reg = 7/1/1. And = 18. Lev = 8. ObjNums = 17.
Params: FramesMax = 120. Start = 0. Conflimit = 0. TimeOut = 0. SolveAll = 0.
0 + : Var = 1. Cla = 0. Conf = 0. Learn = 0. 0 MB 4 MB 0.01 sec
1 + : Var = 13. Cla = 29. Conf = 0. Learn = 0. 0 MB 4 MB 0.01 sec
Runtime: CNF = 0.0 sec (0.1 %) UNSAT = 0.0 sec (0.0 %) SAT = 0.0 sec (0.2 %) UNDEC = 0.0 sec (0.0 %)
LStart(P) = 10000 LDelta(Q) = 2000 LRatio(R) = 80 ReduceDB = 0 Vars = 1000 Used = 1000 (100.00 %)
Buffs = 0. Dups = 0. Hash hits = 0. Hash misses = 6. UniProps = 0.
Output 0 of miter "3sat" was asserted in frame 1. Time = 0.01 sec
```

## INT (插值)

```
Converting 1 flops from don't-care to zero initial value.
AIG: PI/PO/Reg = 7/1/1. And = 18. Lev = 8. CNF: Var/Cla = 17/32.
✓ Step = 1. Frames = 1 + 1. And = 0. Lev = 0. Time = 0.00 sec
| I = 1. Bmc = 2. IntAnd = 0. IntLev = 0. Conf = 0. Time = 0.00 sec
Found a real counterexample in frame 1.
Running "bmc3". PI/PO/Reg = 7/1/1. And = 18. Lev = 8. ObjNums = 17.
✓ Params: FramesMax = 0. Start = 1. ConfLimit = 100000000. TimeOut = 0. SolveAll = 0.
| 1 + : Var = 13. Cla = 29. Conf = 0. Learn = 0. 0 MB 4 MB 0.01 sec
Runtime: CNF = 0.0 sec (0.1 %) UNSAT = 0.0 sec (0.0 %) SAT = 0.0 sec (0.2 %) UNDEC = 0.0 sec (0.0 %)
LStart(P) = 10000 LDelta(Q) = 2000 LRatio(R) = 80 ReduceDB = 0 Vars = 1000 Used = 1000 (100.00 %)
Buffs = 0. Dups = 0. Hash hits = 0. Hash misses = 6. UniProps = 0.
Runtime statistics:
Rewriting = 0.00 sec ( 0.00 %)
CNF mapping = 0.00 sec ( 0.40 %)
SAT solving = 0.00 sec ( 0.05 %)
Interpol = 0.00 sec ( 0.00 %)
Containment = 0.00 sec ( 0.19 %)
Other = 0.01 sec ( 99.35 %)
TOTAL = 0.01 sec (100.00 %)
✓ Output 0 of miter "3sat" was asserted in frame 1. Time = 0.02 sec
```

## PDR

```
Converting 1 flops from don't-care to zero initial value.
VarMax = 300. FrameMax = 10000. QueMax = 0. TimeMax = 0. MonoCNF = no. SkipGen = no. SolveAll = no.
Frame Clauses
Max Queue Flops Cex Time
| 1 : 0 0 2 0 0.00 sec
Block = 1 Oblig = 3 Clause = 0 Call = 3 (sat=66.7%) Cex = 0 Start = 0
SAT solving = 0.00 sec ( 0.35 %)
| unsat = 0.00 sec ( 0.00 %)
| sat = 0.00 sec ( 0.35 %)
Generalize = 0.00 sec ( 0.21 %)
Push clause = 0.00 sec ( 0.00 %)
Ternary sim = 0.00 sec ( 0.06 %)
Containment = 0.00 sec ( 0.00 %)
CNF compute = 0.00 sec ( 0.07 %)
Refinement = 0.00 sec ( 0.05 %)
TOTAL = 0.01 sec (100.00 %)
Output 0 of miter "3sat" was asserted in frame 1. Time = 0.01 sec
```

對於輸出結果，全數符合預想輸出，僅為測試故不分析數據

## 2.2 TSP 問題模型 N = 4

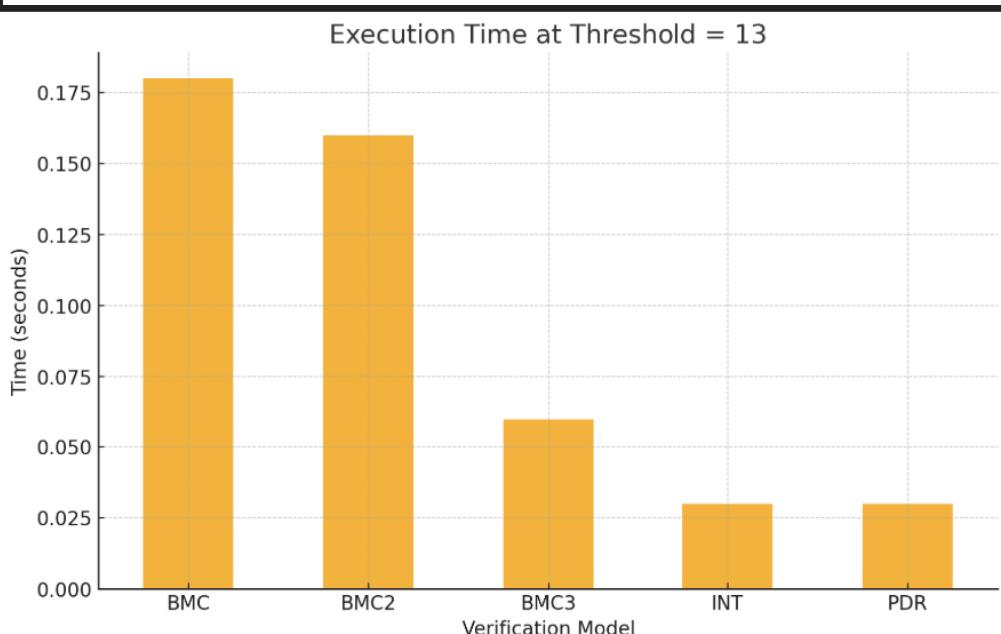
此為對 TSP 問題模型後的 FSM 運作圖

FSM-style TSP verifier for 4 cities; single property output p  
 如果都沒有 → foundLower = 0 → p = 1

編號	路徑 (CITY 0 起點)	總成本計算
--	--	--
1	0 → 1 → 2 → 3 → 0	3 + 5 + 6 + 2 = **16**
2	0 → 1 → 3 → 2 → 0	3 + 1 + 6 + 4 = **14**
3	0 → 2 → 1 → 3 → 0	4 + 5 + 1 + 2 = **12**
4	0 → 2 → 3 → 1 → 0	4 + 6 + 1 + 3 = **14**
5	0 → 3 → 1 → 2 → 0	2 + 1 + 5 + 4 = **12**
6	0 → 3 → 2 → 1 → 0	2 + 6 + 5 + 3 = **16**

測試 p = 0 的情況，只需要降低 THRESHOLD 值，例如設成 THRESHOLD = 13，因為有成本 = 12 的路徑存在，foundLower 就會被設成 1，最後 p = 0。

以下為 TSP 模組使用 BMC、BMC2、BM3、INT、PDR 後所得之數據



雖然由於 abc tools 並不支持印出 cex 的功能，但單就效率測量上，在已知 13 為必定會觸發 assertion 的前提下，可明顯看出各模組的差距。

對於此，課堂上對於各類方式的講解已相當完善，故不在此贅述其原因。

## 2. 形式驗證 – 演算法

一開始本來是想以大型日常 NP 問題建立模型並驗證，但由於根據大型的 TSP 問題建出 FSM 模組過於複雜後放棄，改為轉向利用其他方式運行形式驗證。

### 2.1 遺傳演算法 (Genetic Algorithm, GA)

#### 2.1.1 核心思想

遺傳演算法 (GA) 是一種啟發式搜索技術，其靈感來源於生物進化論中的自然選擇和遺傳機制。它將潛在解視為「個體 (individuals)」，每個個體擁有一套「染色體 (chromosomes)」來編碼解的特徵 (例如，在 TSP 中，染色體可以是一條城市訪問序列)。通過「適應度函數 (fitness function)」(例如，TSP 中的路徑總長度) 來評估每個個體的優劣。通過遺傳操作，產生具有更高適應度的新一代個體群體，逐步逼近最優解。

#### 2.1.2 步驟

典型的遺傳演算法包含以下步驟

##### 1. 初始化族群 (Initialization)

隨機生成一定數量的初始解 (個體)，構成初始族群。

##### 2. 適應度評估 (Fitness Evaluation)

根據適應度計算族群中個體的適應度。

##### 3. 選擇 (Selection)

根據個體的適應度值，選出優良的個體，使其有更大的機會進入下一代。

##### 4. 交叉 (Crossover)

從選出的父代個體中配對，並使其染色體片段進行交換，從而產生子代個體。

## 5. 突變 (Mutation)

子代個體以一定的較小機率發生隨機變動，

## 6. 重複步驟 2 至 5，直到滿足預設的終止條件，例如達到最大進化代數或解的品質不再顯著提升。

# 2.2 Held-Karp 動態規劃 (Dynamic Programming, DP)<sup>1</sup>

## 2.2.1 核心思想

Held-Karp 動態規劃演算法是解決 TSP 的一種精確演算法，對於 TSP，DP 的核心思想是將問題分解為一系列重疊的子問題，並儲存這些子問題的解以避免重複計算。狀態通常定義為  $dp[j]$ ，表示從指定的起點城市出發，訪問過城市子集  $S$  中的所有城市，並最終停留在城市  $j$  的最短路徑長度 1。

## 2.2.2 演算法步驟<sup>2</sup>

### 1. 初始化 (Initialization)

設定起始狀態。將狀態  $dp[mask][pos]$  定義為  $(cost, prev\_city)$ ，其中  $mask$  是一個位元遮罩表示已訪問的城市集合， $pos$  是當前所在的城市。

### 2. 迭代與狀態轉移 (Iteration and State Transition)

依序遍歷所有可能的已訪問城市子集  $mask$  (從只包含一個城市到包含所有城市)。對於每個  $mask$ ，遍歷  $mask$  中所有已被訪問的城市  $po$ ，並更新  $dp[new\_mask][nxt]$  為  $(new\_cost, pos)$ 。

### 3. 計算最終結果 (Final Result)

當所有城市都被訪問過時 (即  $mask$  為 FULL\_MASK，所有位均為 1)， $dp[j]$  儲存了遍訪所有城市並停留在城市  $j$  的最低成本。最終解的成本是枚舉所有可能的終點城市  $j$ ，並取其中的最小值。

### 4. 路徑重建 (Path Reconstruction)

利用在  $dp$  表中儲存的前一個城市 ( $prev\_city$ ) 的資訊，從最終選擇的終點開始，反向回溯，直到回到起點，從而重建出完整的最佳路徑。

### 2.2.3 簡要分析

Held-Karp DP 演算法的顯著優點是它能夠保證找到 TSP 的最優解。然而，其代價是較高的計算複雜度。該演算法的時間複雜度為  $O(n^2 \cdot 2^n)$ ，空間複雜度為  $O(n \cdot 2^n)$ ，其中  $n$  是城市的數量。這種指數級的複雜度使得 DP 演算法僅適用於城市數量較少的情況。

## 2.3 模擬退火演算法 (Simulated Annealing, SA)<sup>3</sup>

### 2.3.1 核心思想<sup>4</sup>

模擬退火 (SA) 演算法是一種概率性的元啟發式方法，其設計靈感來源於冶金學中固體材料的退火過程。在退火過程中，金屬被加熱至高溫然後緩慢冷卻，使得其內部分子能夠排列到低能量的穩定晶體結構。SA 演算法模擬這一過程，允許在搜索解空間的過程中，不僅接受能夠改進當前解的移動 (即走向成本更低的解)，而且以一定的機率接受可能使解變差的移動。這種接受「壞解」的機制使得演算法有能力跳出局部最優解的束縛，從而有更大的機會找到全局最優解。接受差解的機率與一個稱為「溫度」的控制參數相關，溫度越高，接受差解的機率越大；隨著溫度的逐漸降低，演算法趨向於只接受優解，最終收斂。

### 2.3.2 演算法步驟

#### 1. 初始化 (Initialization)

- 隨機生成一個初始解。
- 設定一個較高的初始溫度  $T_{start}$  和一個冷卻率  $\alpha$

#### 2. 迭代搜索 (Iterative Search)

在達到預設的迭代次數或溫度降至某個值之前，重複以下步驟

- 產生鄰近解 (Generate Neighboring Solution)

對當前解施加一個微小的擾動，產生一個新的鄰近解。在 TSP 中，常用的鄰域操作是 2-opt 交換，即隨機選取路徑中的兩個邊，斷開它們，然後以不同的方式重新連接以形成一條新路徑。

- 計算成本差 (Calculate Cost Difference,  $\Delta E$ )

計算新解的成本與當前解的成本之差

$$\Delta E = \text{cost}(\text{new\_solution}) - \text{cost}(\text{current\_solution})$$

- 接受準則 (Metropolis Criterion):

- 如果  $\Delta E < 0$  (即新解更優) , 則接受新解作為當前解。
- 如果  $\Delta E \geq 0$  (即新解更差或相同) , 則以機率  $P = e^{-\Delta E/T}$  接受新解。其中  $T$  是當前溫度。

### 3. 冷卻 (Cooling)

按照預定的冷卻策略降低溫度，例如  $T=T\cdot\alpha$ 。

### 4. 結束 (Termination)

當滿足終止條件時，演算法結束，回傳在整個搜索過程中找到的最佳解。

#### 2.3.3 簡要分析

模擬退火演算法的有效性在很大程度上取決於其參數的設定，特別是初始溫度、冷卻速率和迭代次數。初始溫度需要足夠高，以確保在搜索初期有足夠的探索能力，能夠廣泛地搜索解空間。冷卻速率  $\alpha$  控制著降溫的速度， $\alpha$  越接近 1，降溫越慢，搜索越精細，但需要更多的計算時間。

Metropolis 接受準則  $e^{-\Delta E/T}$  是 SA 的核心機制。在早期高溫階段，即使新解比較差，接受的機率也相對較高，鼓勵演算法探索。隨著溫度的降低，接受差解的機率減小，演算法逐漸將搜索集中在已發現的較優解附近，進行更精細的利用。如果過早收斂，表現會類似於 greedy；如果冷卻過慢，則可能浪費計算資源。

## 2.4 蟻群最佳化演算法 (Ant Colony Optimization, ACO)<sup>5</sup>

### 2.4.1 核心思想

蟻群最佳化 (ACO) 演算法是一種模擬自然界中螞蟻覓食行為的元啟發式演算法。真實的螞蟻在尋找食物的過程中，會在走過的路徑上釋放一種稱為「費洛蒙 (pheromone)」的化學物質。其他螞蟻在選擇路徑時，會傾向於選擇費洛蒙濃度較高的路徑。這種正回饋機制使得整個蟻群能夠逐漸發現從蟻巢到食物源的最短路徑。

ACO 將這一生物學現象應用於解決組合最佳化問題，如 TSP。在 ACO 中，「虛擬螞蟻」負責建構問題的解 (即 TSP 路徑)。對於 TSP 來說距離越短，吸引力越大。

### 2.4.2 演算法步驟

#### 1. 初始化 (Initialization)

- 在所有可能的路徑 (城市間的邊) 上設置少量且均勻的初始費洛蒙濃度。
- 將一定數量的虛擬螞蟻放置在隨機選擇的起始城市，或者都從同一個城市出發。

#### 2. 螞蟻建構路徑 (Ant Solution Construction)

- 每一隻螞蟻獨立地、逐步地建構一條完整的 TSP 路徑。
- 在路徑的每一步，螞蟻  $k$  從當前城市選擇下一個未訪問的城市的機率通常由邊上的費洛蒙濃度、邊的啟發式資訊值和控制費、洛蒙啟發式資訊，在決策中的相對重要性的參數。

#### 3. 費洛蒙更新 (Pheromone Update):

- 當所有螞蟻都完成路徑的建構後，對所有路徑上的費洛蒙進行更新。
- 費洛蒙蒸發 (Evaporation):

所有邊上的費洛蒙濃度都會按照一定的比例  $\rho$  ( $0 < \rho \leq 1$ ) 減少。蒸發有助於避免演算法過早收斂到次優解，並鼓勵探索新的路徑。

#### 4. 費洛蒙沉積 (Deposition)

每一隻螞蟻會根據其建構路徑的品質 (通常是路徑長度的倒數) 在其走過的邊上增加費洛蒙。路徑越短，增加的費洛蒙越多。

#### 5. 迭代 (Iteration)

重複步驟 2 和 3，直到達到預設的終止條件 (如最大迭代次數或解的品質長時間沒有改善)。整個過程中找到的最短路徑即為最終的近似最優解。

### 2.4.3 簡要分析:

ACO 的核心在於其通過環境實現所謂的「標記」(stigmergy)。單個螞蟻的行為相對簡單，但群體的集體行為卻能導致複雜問題的有效解決。費洛蒙的蒸發和沉積機制是 ACO 的關鍵組成部分。

參數的設定對 ACO 的性能有顯著影響。兩者分別決定費洛蒙的歷史經驗和啟發式信息的影響程度。適當平衡這兩者，以及調整螞蟻數量、迭代次數和費洛蒙蒸發率等參數，才能獲得更優質的解解。

## 2.5 基於最小生成樹的近似演算法 (MST-based Approximation Algorithm)<sup>6</sup>

### 2.5.1 核心思想

基於最小生成樹 (MST) 的近似演算法是一種經典的 TSP 求解方法，尤其適用於滿足三角不等式 (triangle inequality) 的度量 TSP (Metric TSP)。

該演算法的核心思想是利用圖的最小生成樹。MST 的總權重是連接所有城市的一個子圖的最小可能權重，因此它天然地構成了 TSP 最優路徑長度的一個下界 (儘管 TSP 路徑是一個環路，而 MST 是一棵樹)。對於滿足三角不等式的 TSP，通過對 MST 進行特定操作，可以構造出一條 TSP 路徑，其長度被證明不超過最優 TSP 路徑長度的兩倍 (即 2-近似保證)。

## 2.5.2 演算法步驟

### 1. 建立最小生成樹 (Construct MST)

給定所有城市及其間的距離，使用標準的 Prim 演算法或 Kruskal 演算法找到連接所有城市的最小生成樹。

### 2. 前序走訪 (Preorder Traversal)

從 MST 的任意一個節點 (城市) 開始，對該 MST 進行一次前序深度優先搜索 (DFS)，並記錄節點被訪問的順序。

### 3. 形成哈密瓜迴路 (Form Hamiltonian Circuit):

- 將前序走訪得到的節點序列視為一條初步的 TSP 路徑。
- 更簡單的理解是，按照前序走訪的順序連接城市，如果某個城市已經在當前構建的路徑中 (除了作為路徑的終點去閉合環路)，則跳過它，直接連接到序列中的下一個新城市。
- 最後，將序列中的最後一個城市與第一個城市相連，形成一個完整的哈密頓迴路。

## 2.5.3 簡要分析:

MST 近似演算法的主要優勢在於其理論上的性能保證和計算效率 (Prim 或 Kruskal 演算法通常具有多項式時間複雜度，如  $O(N^2)$  或  $O(E\log N)$ ，其中  $N$  是城市數， $E$  是邊數)。這使得它成為一種快速獲得有界解的方法。

然而，這個性能保證嚴格依賴於三角不等式的成立。如果問題不滿足三角不等式，MST 近似演算法的理論保證將完全失效，其產生的解的品質可能會非常差，遠超最優解的兩倍。

即使在滿足三角不等式的情況下，2-近似也意味著解的成本可能高達最優解的兩倍，這在許多實際應用中可能仍然不夠理想。另外，演算法的實際性能還取決於 MST 的結構與最優 TSP 路徑結構的相似程度。

## 2.6 實作與搜索

此節主要為實作各類演算法，但此次報告並不會據此解析各算法優劣，故不贅述

### 3. 改良演算法 – 基因遺傳演算法(GA)

在各類演算法中，我選擇了基因遺傳演算法作為後續改良的載體，結合簡單的自適應迴路，讓 GA 演算法能在此實驗環境中模擬用於 SAT 求解的過程。

#### 3.1 實驗設置與數據來源 (Experimental Setup and Data Sources)

本報告的實驗基礎框架來自於 TSP-Genetic-Algorithm<sup>2</sup>，數據分析基於四個選自 TSPLIB 的標準測試案例：

att 48.tsp

att 532.tsp

ch130.tsp

ulysses16.tsp

#### 3.2 模組擴充

由於 GA 演算法核心原理源於自然選擇和遺傳機制，包含

族群 (population)

染色體 (chromosome，解決方案的編碼)

適應度函數 (fitness function，目標評估)

選擇 (selection)

交叉 (crossover，重組)

突變 (mutation) …等要素。

遺傳演算法擅長探索廣闊且複雜的搜索空間，使其適用於那些解空間崎嶇或特性尚不明確的形式驗證問題。其基於族群的特性允許多路徑並行探索並發現多樣化的解決方案。

---

<sup>2</sup> <https://github.com/RenatoMaynard/TSP-Genetic-Algorithm/tree/main>

故我將基礎的程式碼框架進行擴展，加上了自適應迴路，讓其除了「找到最小 cost」的功能外，額外增加了對目標 cost 進行求解，若無法(由於該指定 cost 不在 set 中或被演算法遺棄)，則輸出所能找到的最近解。

### 3.2 自適應機制

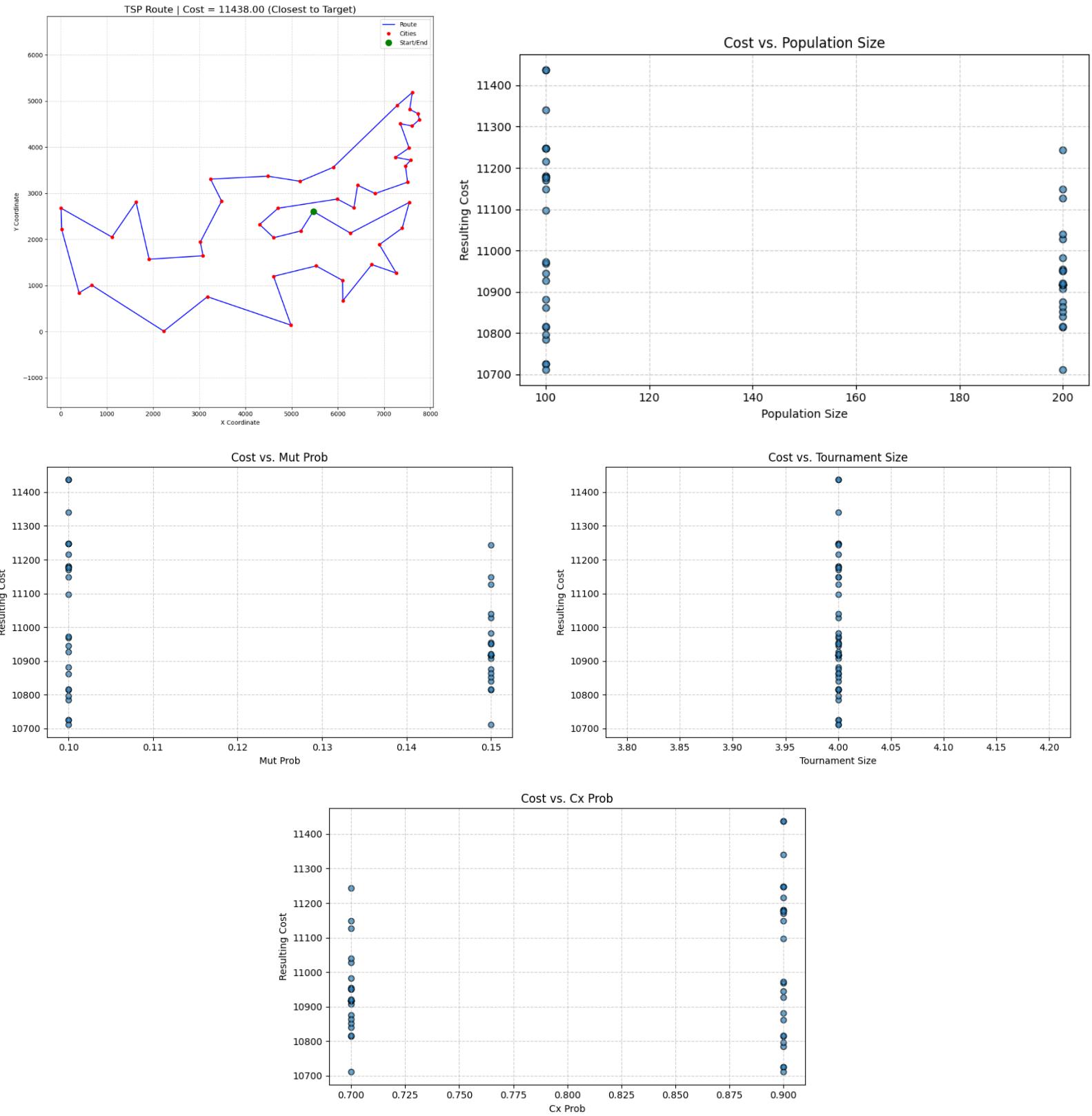
仔細閱讀 GA 的核心思想與具體實現的演算法後，我們不難發現 GA 中的幾項參數可被視為自適應過程中的變量，此幾項為

```
n: int  
population_size: int = 100  
generations: int = 1000  
mutation_rate: float = 0.02  
seed: int = random
```

其中，由於 n 為測資指定，無法更動，所以將其餘四項納入自適應所能更動的範圍內，再搭配激勵機制，一定程度上避免陷入 GA 的缺點，也就是「過早收斂讓指定解被踢出範圍」的情況，讓整個類 SAT 的求解過程中能有更高的正確性。

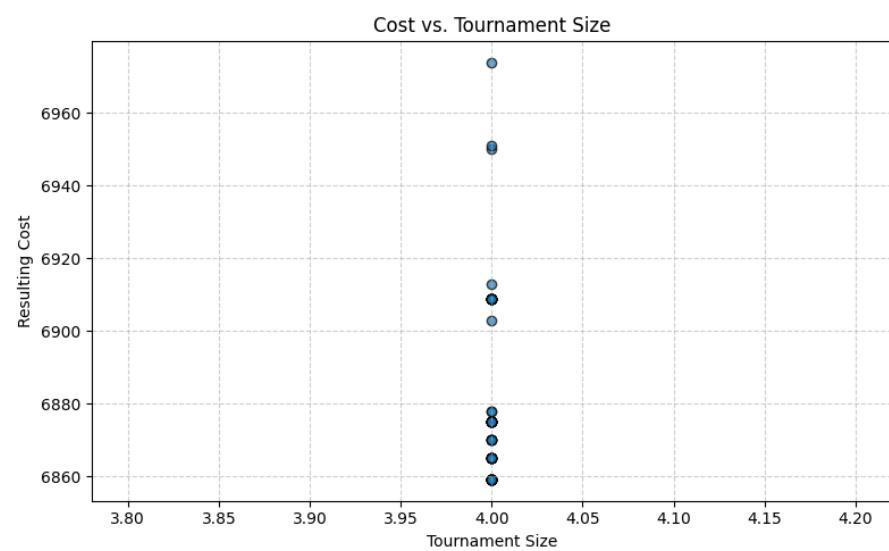
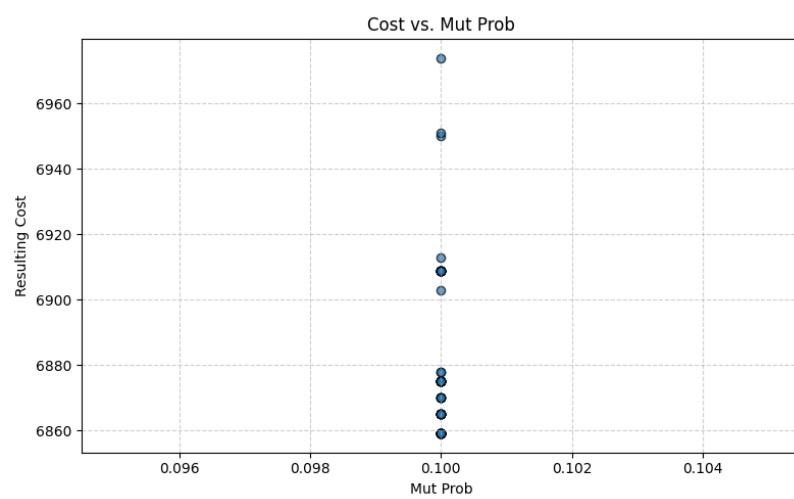
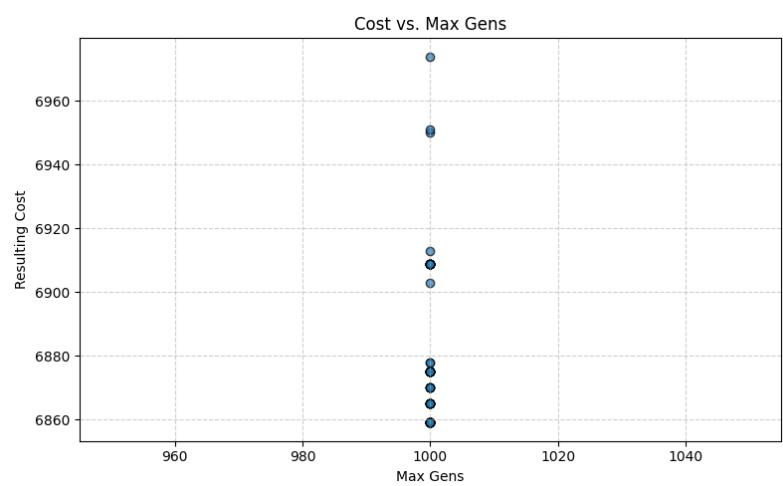
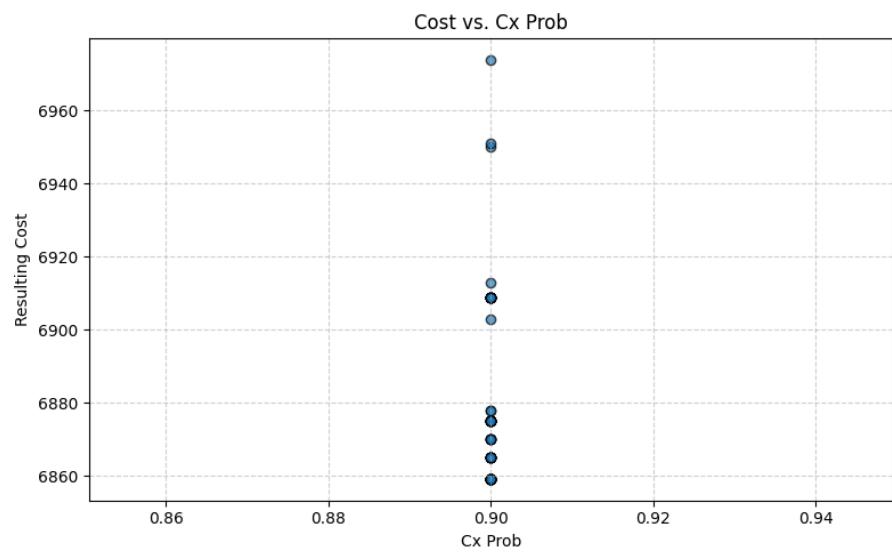
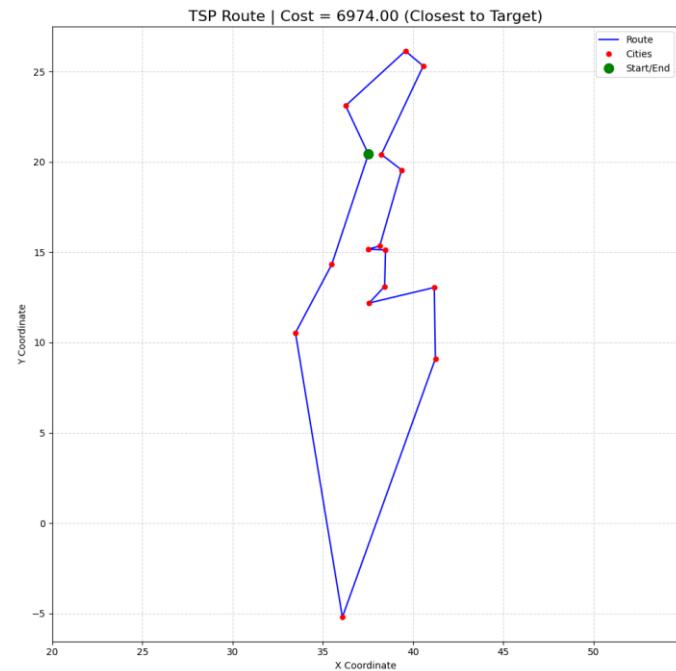
### 3.3 數據分析 (詳細.log 與.json 皆於 tsp/log 中)

CASE (att48) Target = 14628, Official Opt = 10628, Result = 11438, Attempt = 50



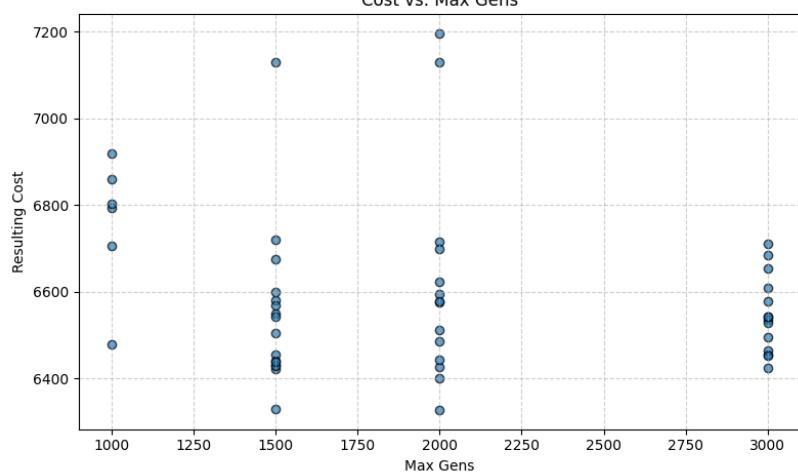
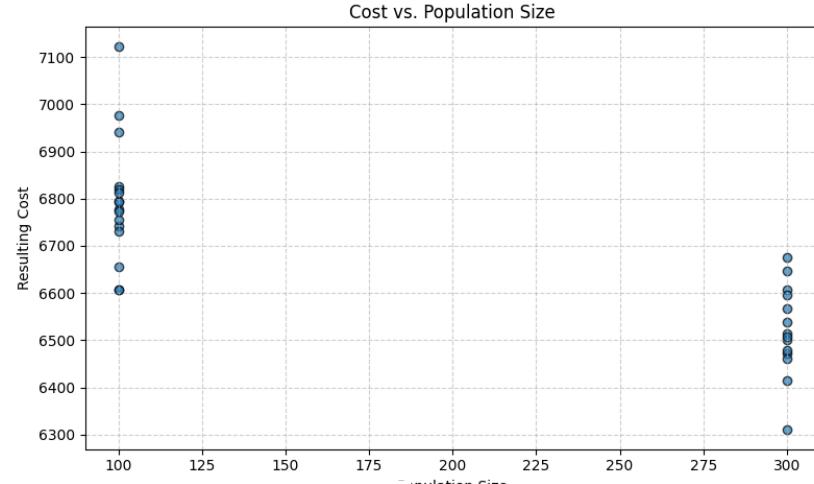
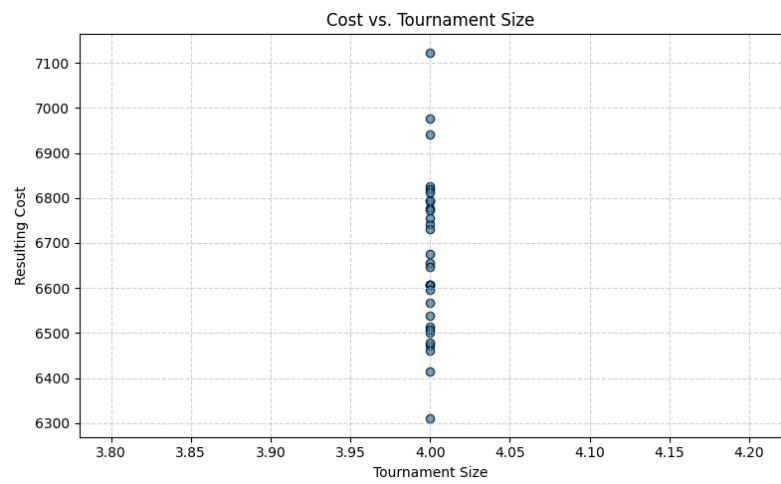
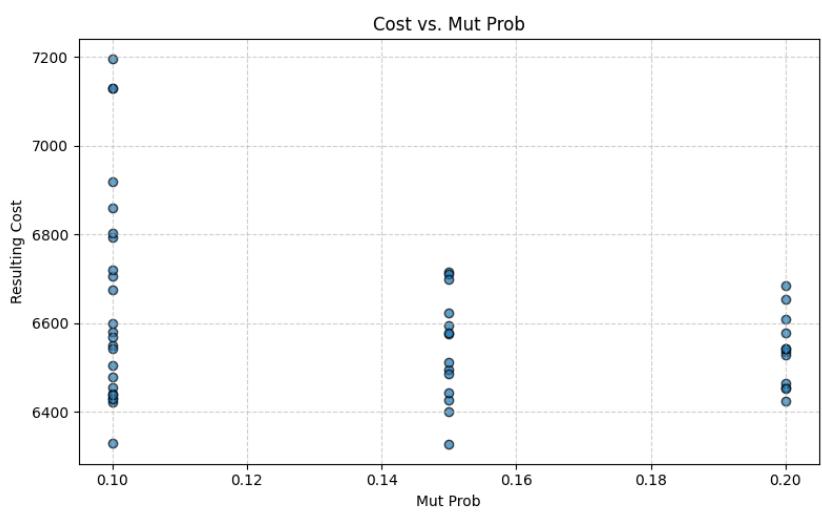
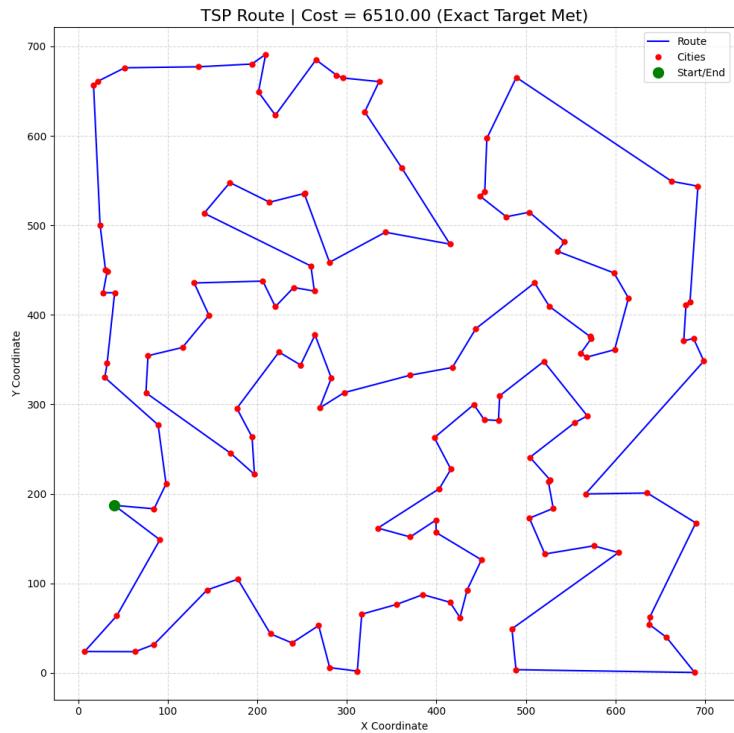
CASE (ulysses16) Target = 7000, Official Opt = 6859, Result = 6974,

Attempt = 50



CASE (ch130) Target = 6510, Official Opt = 6110, Result = 6327

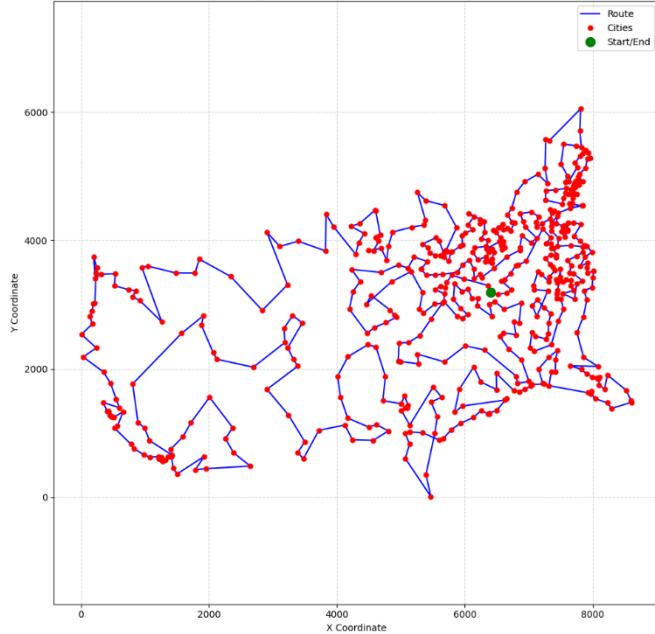
Attempt = 50



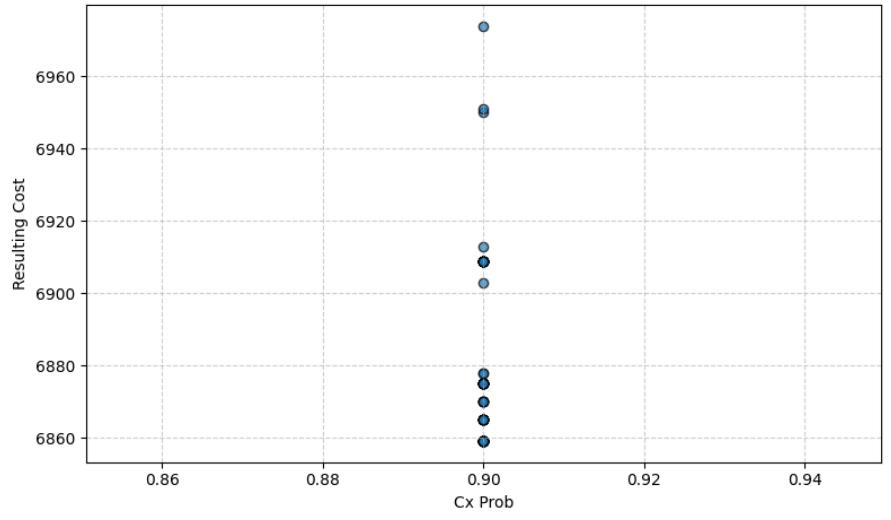
CASE (att532) Target = 31686, Official Opt = 27686, Result = 31400

Attempt = 50

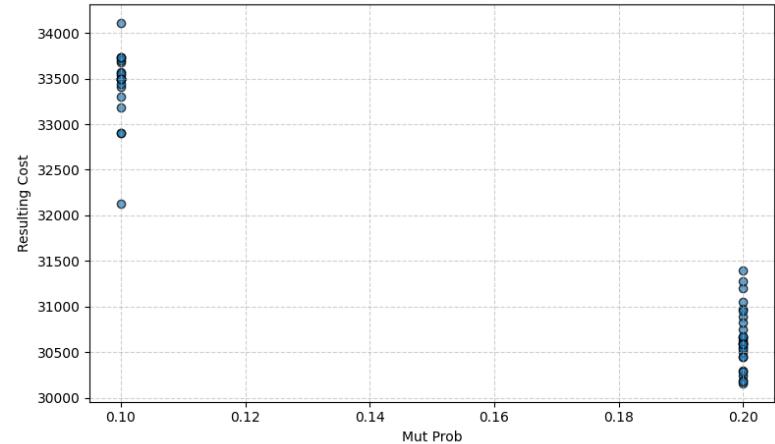
TSP Route | Cost = 31400.00 (Closest to Target)



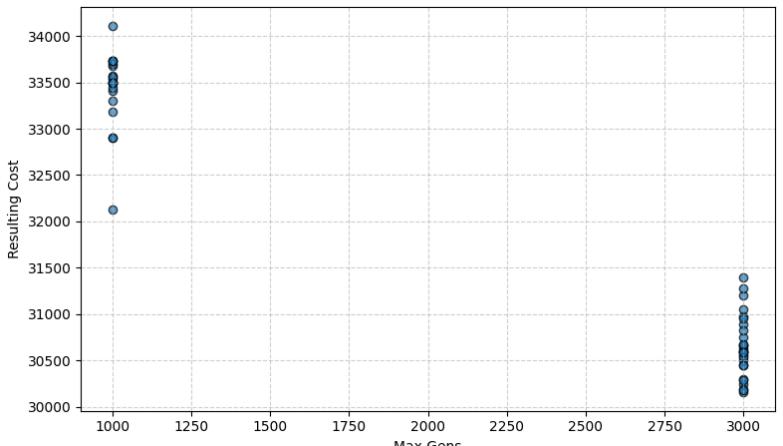
Cost vs. Cx Prob



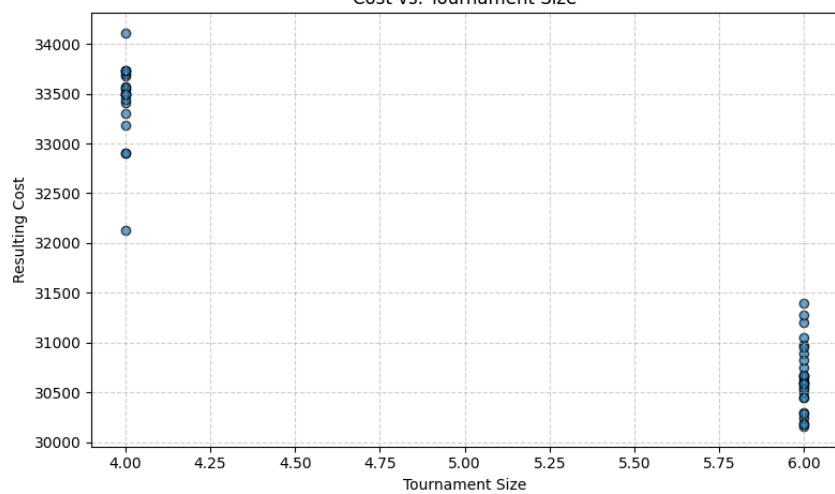
Cost vs. Mut Prob



Cost vs. Max Gens

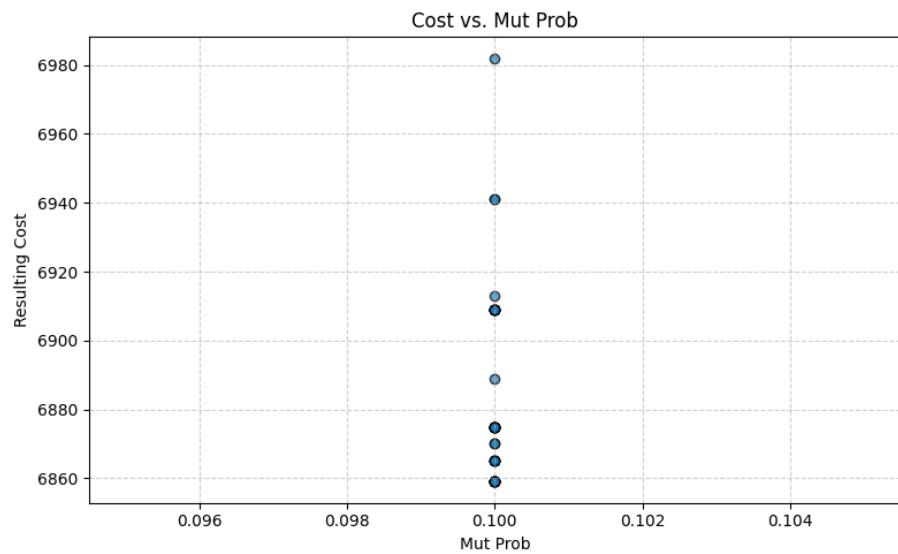
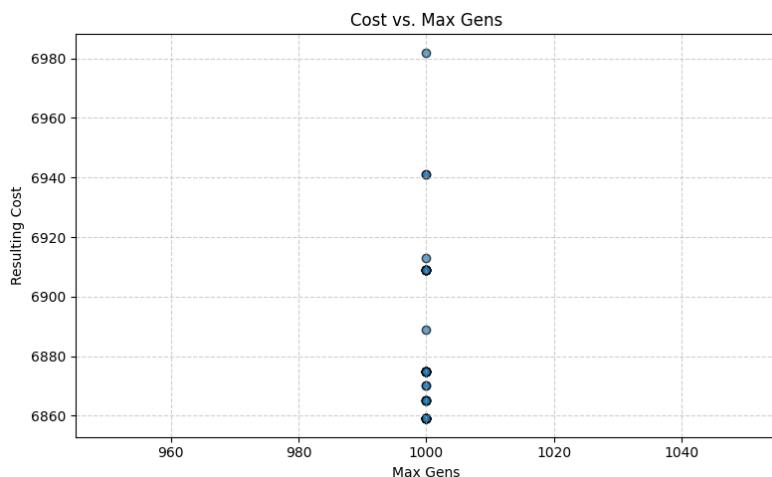
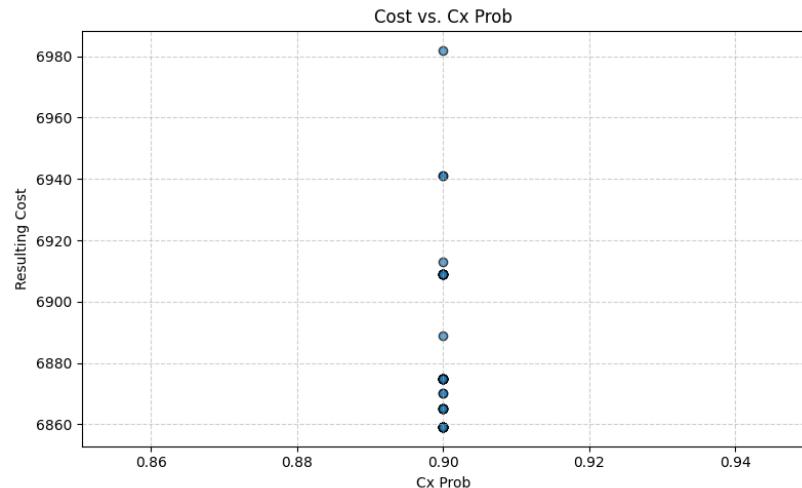
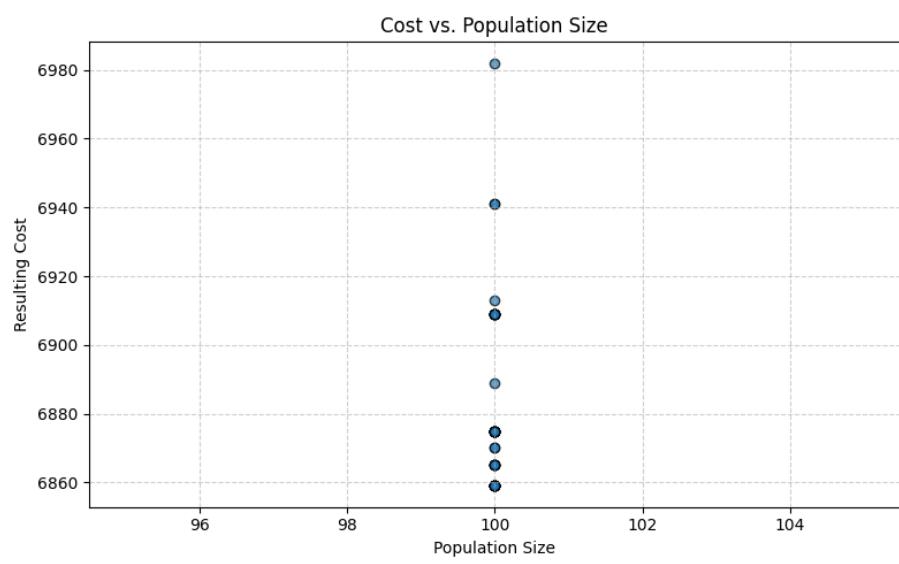
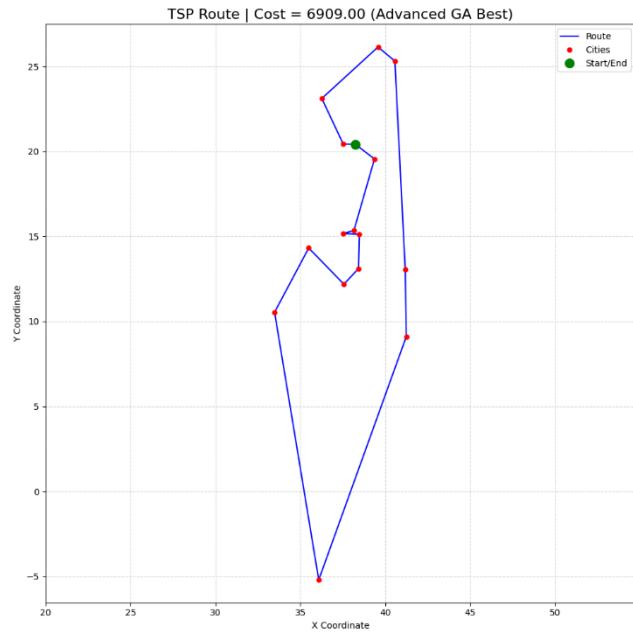


Cost vs. Tournament Size



CASE (ulysses16) Target = 6939, Official Opt = 6859, Result = 6941

Attempt = 50



## 4. 結論 (Conclusion)

由前一章節所得出的統計數據，其實我們可以大概得到各個參數對於實際 cost 與適應迴路判斷的影響力，而對於兩個 case ulysses16 的測試，第二個測試相對而言應為較為貼近真實 SAT 情況時的數據，這是由於其目標數 6939 為第一次 ulysses16 於 log 中所測試得到的其中一個 current cost，這代表 6939 這個數字對於該模型理應為有解，但最後在 50 次 attempt 的情況卻還是找不到，這也真實反映了 TSP 問題的候選解範圍有多大。

即便是在  $N = 16$  的情況下，不同的 TSP 結構變化使得候選解的範圍變得更加廣大，而 GA 如果過早收束，還是有可能將此解踢出候選解範圍，以至於最後呈現 UNSAT 的情況。

故由此狀況可知，為避免目標解過早被踢出考慮範圍，激勵制度的存在為必須。

另外，由於 TSP 的問題難度本身就隨著  $N$  有著指數性的增加趨勢，再加上內部複雜的資料結構與權重分配，這使得大型 TSP 問題建模的困難度上升。不過另一方面卻也代表著假設建模成功，抑或是有其他能被切割或抽象成 TSP 問題的電路構造，GA 本身的決策豐富度優勢，再加上自適應迴圈的方法，只要挑對演算法，我們其實可以有效的進行形式化驗證。

Table 1: 演算法在數位電路形式驗證中的影響比較

演算法	主要應用	影響
SAT/SMT	有界模型檢查、等價性檢查	高效處理大型電路，支援複雜理論
BDD	符號模型檢查	適合中小型電路，記憶體限制大型應用
插值	歸納性驗證	減少狀態空間探索，提升效率
機器學習 演化式演算法	策略選擇、屬性優先順序化 設計優化、測試生成	自動化驗證，減少時間 提供全局探索，輔助驗證

## 5. 演算法在數位電路形式驗證中的應用擴展

### 5.1 遺傳演算法在硬體驗證中的「雙層架構」模式

#### 5.1.1 外層搜尋器與內層驗證器的協同

根據 Vasíček & Sekanina 的研究，遺傳演算法在數位電路後合成優化中採用「外層 GA／內層 Model Checker」的架構模式。在每次迭代中，GA 負責產生候選電路設計，而 SAT 求解器作為內層驗證器判斷功能等價性。這種模式的核心優勢在於將全域搜尋能力與嚴格的形式驗證相結合，確保優化過程中不引入功能錯誤。

### 5.2 模擬退火在驗證流程優化中的關鍵作用

#### 5.2.1 BDD 變數排序優化

由於 BDD 變數排序問題一直都被視為是優化 SAT 的其中一種方式。, 而 SA 提供了有效的近似解決方案<sup>7</sup>。實驗結果顯示，SA 優化的變數排序可以使 BDD 節點數下降一個數量級，大幅降低記憶體消耗並提升驗證效率。

#### 5.3.2 強化學習增強的 SAT 求解

GQSAT 系統通過分析問題實例的結構特徵，在搜尋初期做出更佳的分支決策，並能泛化到比訓練集大 5 倍的問題規模。

#### 5.3.3 神經網路輔助的模型檢查

Neural Model Checking 技術代表了形式驗證的最新發展方向。該方法使用神經網路作為線性時序邏輯的形式證明證書，通過隨機執行軌跡訓練神經排名函數，然後利用 SAT 求解器驗證其有效性。實驗結果表明，在 194 個標準硬體模型檢查問題上，該方法平均比學術工具多完成 60 個任務，比商業工具多完成 11 個任務。

## 6. 額外補充

一開始在規劃期末計畫時，我參考了一些比較近期且有關於 RTL 電路驗證的論文，其中”*SoC connectivity specification extraction using incomplete RTL design: An approach for Formal connectivity Verification*” 的內容是我比較感興趣的。

而又由於我自身在這學期的期許是學習演算法，所以我選擇了演算法為主題的報告內容。一開始其實考慮過自作 NP-hard 相關議題的 RTL 模型，並透過 abc 工具進行 UBMC、INT、PDR 的驗證，並根據數據分析各驗證方式的優劣。但由於模組化的過程受挫，因此轉向了使用其他方式進行演算法與形式驗證相關議題的專題題目。

Final Work :

Github : <https://github.com/jimmy01081122/SOCV-Final-Project>

---

<sup>14</sup> Held, M. and Karp, R. (1962) A Dynamic Programming Approach to Sequencing. Journal for the Society for Industrial and Applied Mathematics, 10, 1-10. <https://doi.org/10.1137/0110015>

<sup>15</sup> Travelling Salesman Problem using Dynamic Programming  
<https://www.geeksforgeeks.org/travelling-salesman-problem-using-dynamic-programming/>

<sup>16</sup> Kirkpatrick, S., Gelatt, C.D. and Vecchi, M.P. (1983) Optimization by Simulated Annealing. Science, 220, 671-680.  
<http://dx.doi.org/10.1126/science.220.4598.671>

<sup>17</sup> Luping Fang, Pan Chen, and Shihua Liu. 2007. Particle swarm optimization with simulated annealing for TSP. In Proceedings of the 6th Conference on 6th WSEAS Int. Conf. on Artificial Intelligence, Knowledge Engineering and Data Bases - Volume 6 (AIKED'07). World Scientific and Engineering Academy and Society (WSEAS), Stevens Point, Wisconsin, USA, 206–210.

<sup>18</sup> M. Dorigo, V. Maniezzo and A. Colorni, "Ant system: optimization by a colony of cooperating agents," in IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics), vol. 26, no. 1, pp. 29-41, Feb. 1996, doi: 10.1109/3477.484436.

keywords: {Ant colony optimization;Traveling salesman problems;Intelligent robots;Feedback;Distributed computing;Simulated annealing;Robustness;Artificial intelligence;Computational modeling;Data structures},

<sup>19</sup> Approximate solution for Travelling Salesman Problem using MST  
<https://www.geeksforgeeks.org/approximate-solution-for-travelling-salesman-problem-using-mst/>

<sup>7</sup> B. Bollig and I. Wegener, "Improving the variable ordering of OBDDs is NP-complete," in IEEE Transactions on Computers, vol. 45, no. 9, pp. 993-1002, Sept. 1996, doi: 10.1109/12.537122.