

# Sokoban Solver - Implementation Report

學號：p13922006

日期：2025/10/03

=====

=====

作業要求回答

=====

=====

1. Briefly describe your implementation.
2. What are the difficulties encountered in this homework? How did you solve them?
3. What are the strengths and weaknesses of pthread and OpenMP?

=====

=====

1. Briefly describe your implementation (實作說明)

=====

=====

本專案實作了一個平行化的 Sokoban 求解器，使用 A\* 演算法配合 Intel TBB (Threading Building Blocks) 進行並行搜索。

【核心設計】

### (1) CompactState - 記憶體優化的狀態表示

- 使用 `uint16_t` 編碼位置 ( $y * \text{COLS} + x$ )
- 只儲存箱子位置陣列和玩家位置
- 相較於完整 `board` 表示節省超過 90% 記憶體

```
struct CompactState {  
  
    vector<uint16_t> boxes;    // 排序過的箱子位置  
  
    uint16_t player_pos;      // 玩家位置  
  
};
```

### (2) 並行搜索架構 (Parallel A\* Search)

使用 TBB 的無鎖並行容器：

- `tbb::concurrent_priority_queue`：thread-safe 的 open set
- `tbb::concurrent_unordered_map`：thread-safe 的 visited set

Worker threads 採用 batch processing：

- 每個 thread 從 priority queue 取出一批狀態 (`batch_size=4`)
- 平行展開後繼狀態
- 使用 `atomic flags` 同步解的發現

### (3) 啟發式函數 (Heuristic Function)

採用自適應策略：

- 5-15 個箱子：Hungarian Algorithm ( $O(n^3)$ ) - 精確配對
- 其他情況：Greedy Matching ( $O(n^2)$ ) - 快速近似

- 基於 **Manhattan Distance**，保證 **admissible**

```
int calculateHeuristicCompact(const CompactState &compact) {  
    int n = compact.bboxes.size();  
    if (n >= 5 && n <= 15) {  
        return hungarian(cost_matrix); // 最優匹配  
    } else {  
        return greedy_matching();      // 快速近似  
    }  
}
```

#### (4) 死鎖檢測 (Deadlock Detection) - 三層防禦

第一層：**Simple Deadlock** (預計算)

- **Corner deadlock**：箱子被兩面牆夾住
- **Corridor deadlock**：無目標的封閉走廊

第二層：**Early Pruning** (立即剪枝)

- 在 **tryMove()** 之前就檢查 **corner deadlock**
- 避免生成註定失敗的狀態

第三層：**Freeze Deadlock** (運行時檢測)

- 遞歸檢查箱子是否被「凍結」
- 水平與垂直方向都無法移動

## (5) Player Movement Optimization (玩家移動優化)

- 使用 **BFS** 預計算玩家可達的所有位置
- 合併 **player-only moves**：只記錄推箱動作
- 大幅減少狀態空間 (原本每步 4 個方向 → 只展開有效推箱)

=====

=====

## 2. What are the difficulties encountered? How did you solve them?

(困難與解決方案)

=====

=====

### 【困難 1】State Space Explosion (狀態空間爆炸)

問題描述：

- 10 個箱子的地圖有  $10! \approx 362$  萬種排列
- 加上玩家位置，狀態數達百萬級
- 樣本 24/25 在 30 秒限制內難以求解

解決方案：

1. **CompactState** 壓縮 → 記憶體減少 90%，Hash 加速
2. **Player movement merging** → 狀態數減少 70%
3. **Early deadlock pruning** → 剪枝 40%無效分支

效果：

- 簡單案例 (1-3 箱): <2 秒
  - 中等案例 (5-7 箱): 2-10 秒
  - 困難案例 (10 箱): 30-60 秒
- 

## 【困難 2】Synchronization Overhead (同步開銷) ★重要優化點★

問題描述：

- 原本使用 `std::mutex` 保護所有共享資料
- Lock contention 導致 CPU 利用率 <50%
- 平行效率低於串行版本

解決方案：Lock-Free Containers

改用 **TBB** 的並行容器：

**// Before: Heavy locking**

**mutex mtx;**

**priority\_queue<PQItem> pq;**

**unordered\_map<State, int> visited;**

**lock\_guard<mutex> lock(mtx); // Bottleneck!**

**// After: Lock-free with TBB**

```

tbb::concurrent_priority_queue<PQItem> pq;

tbb::concurrent_unordered_map<State, int> visited;

pq.push(item);                // No lock needed!

visited.insert({state, idx}); // Thread-safe!

```

效果：

- CPU 利用率: 50% → 85%
- 加速比 (6 threads): 3.2x → 4.8x

這是最關鍵的優化！從有鎖改為無鎖大幅提升並行效率。

---

### 【困難 3】Heuristic Accuracy vs. Speed Trade-off ★效能瓶頸★

問題描述：

- Hungarian Algorithm 提供精確 heuristic，但  $O(n^3)$  很慢
- Greedy Matching 快速但不精確，導致 A\* 展開更多節點
- 10 個箱子時 Hungarian 每次要 1000 次操作

實驗結果：

策略 | 樣本 24 (10 箱) | 樣本 22 (7 箱) | 結論

-----|-----|-----|-----

只用 Greedy	TIMEOUT	TIMEOUT	heuristic 太弱
Hungarian (4-9 箱)	TIMEOUT	25 秒	閾值太窄
Hungarian (5-15 箱) ✓	58 秒	8 秒	最佳平衡

最終策略：

- 5-15 箱: Hungarian (精確引導)
- 其他: Greedy (快速計算)

這個 **trade-off** 是效能優化的關鍵決策點。

-----

#### 【困難 4】Deadlock False Positives (死鎖誤判)

問題描述：

- 樣本 21 有脆弱地板 (@)，預計算的 `deadCellMap` 會誤判
- 使用 `deadCellMap` 立即剪枝導致樣本 21 超時

解決方案：**Conservative Pruning** (保守剪枝)

只剪枝絕對安全的 **corner deadlock**：

```
if (enableDeadCheck && !targetMap[ty][tx]) {
    bool up = isWall(ty - 1, tx);
    bool down = isWall(ty + 1, tx);
```

```

    bool left = isWall(ty, tx - 1);

    bool right = isWall(ty, tx + 1);

    if ((up && left) || (up && right) ||

        (down && left) || (down && right)) {

        continue; // 絕對安全的剪枝
    }
}

```

效果：

- 樣本 21: TIMEOUT → 2 秒 ✓
- 不會誤剪 corridor deadlock

---

## 【困難 5】Load Balancing (負載平衡)

問題描述：

- **A\*** 搜索深度不均，某些 **threads** 提前結束
- 單個大狀態展開時，其他 **threads** 閒置

解決方案：**Batch Processing**

每個 **thread** 一次處理多個狀態：

```
const int batch_size = 4;
```



```

vector<PQItem> batch;

for (int i = 0; i < batch_size; ++i) {

    PQItem item;

    if (pq.try_pop(item)) {

        batch.push_back(item);

    }

}

```

效果：

- Thread 閒置時間: 30% → 15%
- 整體吞吐量提升 ~20%

=====

=====

### 3. What are the strengths and weaknesses of pthread and OpenMP?

(Pthread 與 OpenMP 的優缺點比較)

=====

=====

#### 【Pthread (POSIX Threads)】

優點 (Strengths)：

##### 1. 精細控制 (Fine-grained Control)

- 完全控制 thread 生命週期
- 可實現複雜的同步模式 (condition variables, barriers)

## 2. 跨平台相容性 (Cross-platform)

- POSIX 標準，Linux/Unix 原生支援

## 3. 底層優化 (Low-level Optimization)

- 手動管理 thread affinity
- 可調整 scheduling policy

## 缺點 (Weaknesses) :

### 1. 學習曲線陡峭 (Steep Learning Curve)

- 需要手動管理 mutex, condition variables
- 容易出現 deadlock, race condition

### 2. 程式碼冗長 (Verbose Code)

```
pthread_t threads[NUM_THREADS];  
  
pthread_mutex_t mutex;  
  
pthread_mutex_init(&mutex, NULL);  
  
pthread_create(&threads[i], NULL, worker, &data);  
  
pthread_join(threads[i], NULL);  
  
pthread_mutex_destroy(&mutex);
```

### 3. 容易出錯 (Error-prone)

- 忘記 unlock → deadlock
- 忘記 join → memory leak

---

## 【OpenMP】

### 優點 (Strengths) :

#### 1. 語法簡潔 (Simple Syntax)

```
#pragma omp parallel for  
  
for (int i = 0; i < n; ++i) {  
  
    work(i);  
  
}
```

一行 `pragma` 即可平行化

#### 2. 自動管理 (Automatic Management)

- 編譯器處理 `thread` 創建/銷毀
- 自動負載平衡 (dynamic scheduling)

#### 3. 適合資料平行 (Good for Data Parallelism)

- Loop parallelization 極簡單
- Reduction operations 內建支援

### 缺點 (Weaknesses) :

#### 1. 控制受限 (Limited Control)

- 難以實現複雜同步模式

- 無法精細控制 **thread** 行為

## 2. Fork-Join 開銷 (Fork-Join Overhead)

- 每個 **parallel region** 都重建 **threads**
- 不適合 **irregular parallelism**

## 3. 不適合任務平行 (Poor for Task Parallelism)

- 本專案的 **A\*** 搜索是 **dynamic task graph**
- **OpenMP task** 支援有限且效能不佳

---

### 【為何本專案選擇 Intel TBB】

本專案的平行化挑戰：

- × 不是規則的 **loop parallelism**
- ✓ **Dynamic task parallelism** (**A\*** 搜索樹)
- ✓ 需要 **concurrent data structures**

**TBB** 優勢：

- **Lock-free concurrent containers** (priority queue, hash map)
- **Thread-safe operations** 不需要手動加鎖
- 原生支援 **dynamic task parallelism**

比較表：

特性	Pthread	OpenMP	TBB
----- ----- ----- -----			
Concurrent Queue	需手動實現	無	✓ 內建
Concurrent HashMap	需手動實現	無	✓ 內建
Dynamic Task	複雜	有限	✓ 原生支援
Code Simplicity	×	✓	✓
Performance	手動優化最高	中等	✓ 高

結論：

對於本專案的 **A\*** 並行搜索，**TBB** 提供了最佳的平衡：

- 比 **pthread** 更簡潔 (不需手動管理鎖)
- 比 **OpenMP** 更強大 (支援 **concurrent containers**)
- 效能接近手動優化的 **pthread**

=====

=====

效能總結 (Performance Summary)

=====

=====

【關鍵優化技術】

1. ✓ **Compact State** → 記憶體減少 **90%**

2. ✓ Player movement merging → 狀態數減少 70%
3. ✓ Hungarian heuristic → 更好的 A\* 引導
4. ✓ Early deadlock pruning → 剪枝 40% 無效分支
5. ✓ TBB lock-free containers → 6 核心加速 4.8x

#### 【剩餘瓶頸】

1. 狀態爆炸：10 箱案例的組合空間仍然過大
2. Heuristic 成本：Hungarian  $O(n^3)$  在密集搜索時累積開銷
3. Deadlock 檢測：Freeze deadlock 遞歸檢查較慢

#### 【未來可能改進方向】

1. Bi-directional A\*：從起點終點同時搜索
2. Pattern Database：預計算子問題的精確代價
3. Iterative Deepening：限制搜索深度避免無效展開

=====

=====

#### 結論 (Conclusion)

=====

=====

本專案深入探索了 Sokoban 求解器的平行化實作，從演算法設計 (A\*)、啟發式函數 (Hungarian)、死鎖檢測、到平行化架構 (TBB)，每個環節都經過仔細的權衡與優化。

透過此作業學到的關鍵經驗：

- 演算法選擇比程式碼優化更重要
- 記憶體效率與計算速度同樣關鍵
- **Lock-free programming** 是並行效能的關鍵
- 並行化不是萬靈丹，錯誤的同步策略反而降低效能