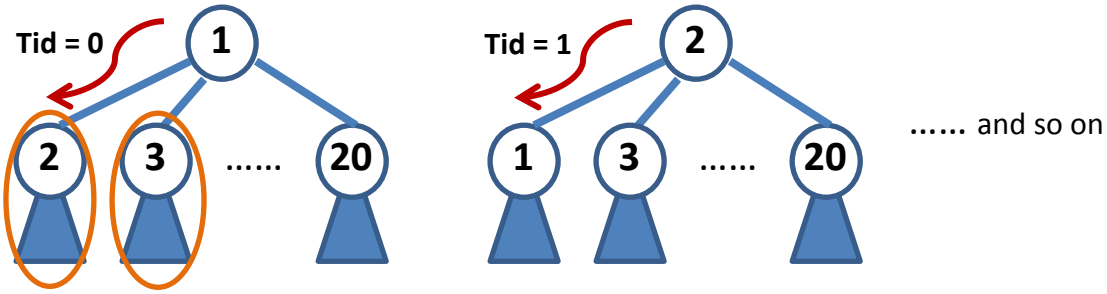
壹、Implementation

一、Algorithm



如上圖所示，我主要採用 DFS 暴力搜尋狀態樹來解這道問題。

由上左圖的橘色圈圈明顯看出，那兩棵子樹除了是從 City1 走過來外，並沒有其它相依關係，所以非常容易做平行化！因此我是<u>利用切割子樹進行平行化</u>。

二、Optimization

因為爆搜這棵樹的複雜度為 O(n!)，假設 n = 20 時，基本上就無法在時間內完成，因此我採取了以下幾個優化方法。

(1) 剪枝

由題目所給的條件：**"each edge is associated with a positive integer to present the distance between two cities"**，可以知道在我搜索這棵樹越深時（代表著我拜訪了更多的城市），我所**累積的路徑距離會遞增**！

由這個觀察可以得到以下結論：

==========

令 tmp_len = 走到目前城市的累計距離

ans_len = 截至目前的最佳解

if (tmp_len >= ans_len) **then** 那就停止往下搜索。

(因為在往下走距離會更大，已經不可能比最佳解好)

else 繼續往下層搜索。

==========

所以我可以藉由這個剪枝條件來減少搜索次數！

(2) Job queue

根據上面的剪枝優化，會發現這棵樹會變得**不平衡**，而造成每個 Thread 的工作量會不同，導致有些 Thread 會閒置形成資源的浪費，因此我加了 Job queue 的機制。除了可以達到 load balance 之外，又可提升程式可 Scalability 的程度！

其中切割子樹的方式是，我定義了一個 Path 的結構，裡面儲存了路徑走法跟他的花費。我在開 thread 平行執行之前，可以先跑 2 ~ 3 層的點，並記錄路徑走法丟到 Job queue 中。每當 thread 從 Job queue 取出某段路徑來做 DFS 暴搜時，就可以接續走下去。

Ex. 先跑兩層，得到 1->2、1->3 … 1->20、2->1 … etc
然後我從 Job queue 中取到了 2->1 這段路徑來處理，那 DFS 就會從 3 開始往下搜(因為 1、2 已經都走過了)，然後跑 4 以此類推。

三、Synchronization problem

程式中需要被所有 thread 讀取的共同變數主要有四個：連接矩陣、最佳解（長度 & 路徑）、Job queue。

而其中只有後面三者需要進行更新的動作，所以需要保護以免產生問題。其中他們個別更新資料的時機點為

- **最佳解**：更新答案
- **Job queue**：從 queue 中取出 job

所以只需要個別保護這些區域即可！而因為這些資源其實都只有一個，所以我採用 mutex 去保護。

貳、Experiment & Analyze

一、Motivation：

主要關注於「到底適不適合做平行化?」，而此項又可以分為以下幾個問題，然後根據這些問題去跑實驗來進行評估。

Q1. 做完剪枝優化後得到了多少的好處?
Q2. Mutex lock 的 overhead 佔了多少?
Q3. 如果將 Job 切割的越細，能提高多少 Load balance? 效能提升多少?

二、Time measure function

我採用以下的函數測量時間，精準到 us (10^(-6))

Time Measure

```
1  unsigned long long start_utime, end_utime;
2  struct timeval tv1, tv2;
3  gettimeofday(&tv1, NULL);
4  start_utime = tv1.tv_sec * 1000000 + tv1.tv_usec;
5
6  // do something
7
8  gettimeofday(&tv2, NULL);
9  end_utime = tv2.tv_sec * 1000000 + tv2.tv_usec;
10 printf("thread %d's runtime = %llu\n", tid, end_utime - start_utime);
```

三、Experiment

(1) Execution times after cut

使用 sequential 的版本進行檢測

・City = 5

	Before	Lv1	Lv2	Lv3
執行次數	120	115	84	70
減少比例		4.17%	30%	41.67%

・City = 10

	Before	Lv1	Lv2	Lv3
執行次數	3628800	12698	14435	4953
減少比例		99.65%	99.60%	99.86%

・City = 20

	Before	Lv1	Lv2	Lv3
執行次數	$24.33*10^{17}$	$21.68*10^{8}$	$92.23*10^{6}$	$47.63*10^{6}$
減少比例		99.99%	99.99%	99.99%

由上面圖表很容易發現，當#City 越大的時候，優化的效果越明顯，而且幾乎佔了快 100%的比例。

(2) Mutex lock overhead

試跑幾筆測資結果截圖如下(單位為 us)

・20lv1 #thread = 10 (Load Balance)

```
thread 5'sruntime = 68997248 , lock overhead = 5394
thread 8'sruntime = 71072041 , lock overhead = 11926
thread 7'sruntime = 71653679 , lock overhead = 8255
thread 4'sruntime = 72284045 , lock overhead = 289
thread 3'sruntime = 72436607 , lock overhead = 28531
thread 1'sruntime = 73508949 , lock overhead = 8737
thread 2'sruntime = 73752395 , lock overhead = 783
thread 9'sruntime = 73768145 , lock overhead = 742
thread 6'sruntime = 74382203 , lock overhead = 214
thread 0'sruntime = 74615559 , lock overhead = 9229
454:20,16,11,4,10,3,2,14,13,8,1,5,15,9,6,12,18,7,19,17
```

・20lv1 #thread = 20 (Unload Balance)

```
thread 5'sruntime = 51659986 , lock overhead = 8524
thread 4'sruntime = 53284478 , lock overhead = 9277
thread 8'sruntime = 55775181 , lock overhead = 126
thread 15'sruntime = 57158954 , lock overhead = 256
thread 11'sruntime = 57931119 , lock overhead = 1544
thread 14'sruntime = 58122484 , lock overhead = 292
thread 16'sruntime = 58688104 , lock overhead = 314
thread 6'sruntime = 59547534 , lock overhead = 291
thread 10'sruntime = 60138579 , lock overhead = 194
thread 9'sruntime = 60187972 , lock overhead = 26325
thread 1'sruntime = 60180217 , lock overhead = 388
thread 0'sruntime = 60438965 , lock overhead = 390
thread 13'sruntime = 60583847 , lock overhead = 220
thread 19'sruntime = 60726909 , lock overhead = 849
thread 17'sruntime = 60849576 , lock overhead = 510
thread 18'sruntime = 60863930 , lock overhead = 5606
thread 3'sruntime = 61068702 , lock overhead = 650
thread 2'sruntime = 61165152 , lock overhead = 115
thread 7'sruntime = 61235206 , lock overhead = 953
thread 12'sruntime = 61313832 , lock overhead = 1817
454:20,16,11,4,10,3,2,14,13,8,1,5,15,9,6,12,18,7,19,17
```

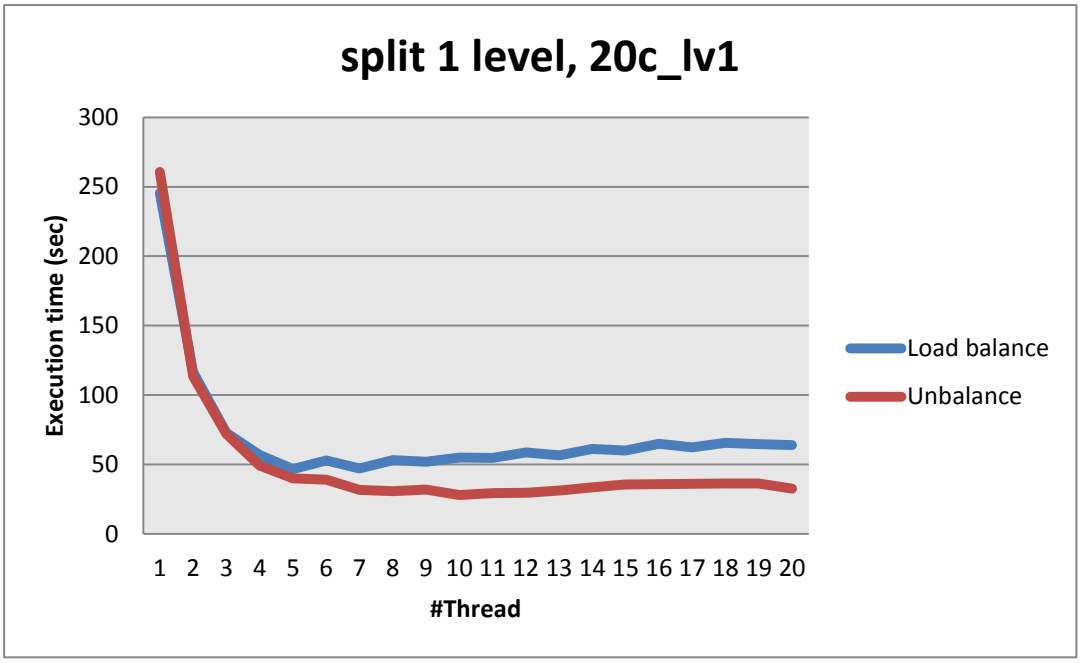
上面兩筆數據比較再不同#thread 下競爭 mutex lock 的情形，可以看出 lock overhead 跟實際執行時間是沒有相關性的，因為 lock 是要到全部 City 走完才要判斷，Runtime 則是看到底 trace 了幾個 node，所以很合理。

計算 lock overhead 所佔總體時間比例最多不過 0.04%而已，所以幾乎可以忽略這個因素！

(3) Load Balance effect

由上面兩組數據可以看出，做了 Job queue 之後 Load balance 的確有明顯的改善！其他更詳細的數據分布可參考附件。(在 appendix 資料夾下面)

以下為 Load balance version 跟 Unload balance version 的比較圖

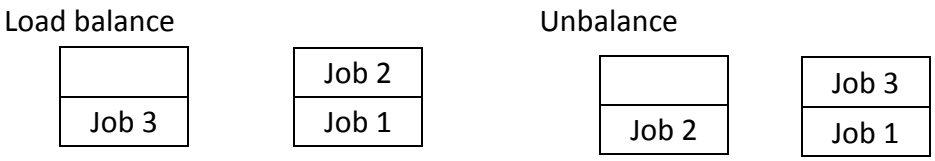


此圖 split 1 level 的意思是預先走完一層，再去跑暴搜（預先執行層數的意義可以參照 P2, Job queue 內文有定義）

圖中發現 Unbalance 的執行結果比有 Load balance 處理要來的好。分析可能的情況是因為切割出來的 Job 不夠細，所以可能會產生以下的 case：

假設 Job1: 3 sec、Job2: 6 sec、Job3: 4 sec

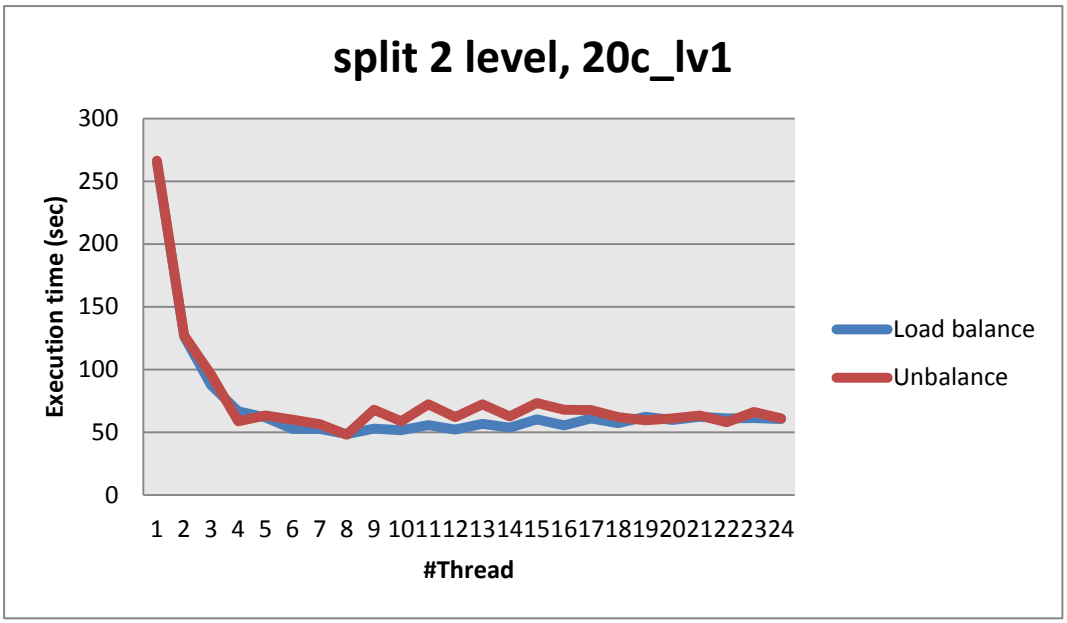
每個方框代表著一個 Thread



那很明顯可以看出在 Unbalance 的情況下比 Load balance 要來的好 (3 + 6 = 9 > 7 = 3 + 4)，所以應是遇到這種情況。

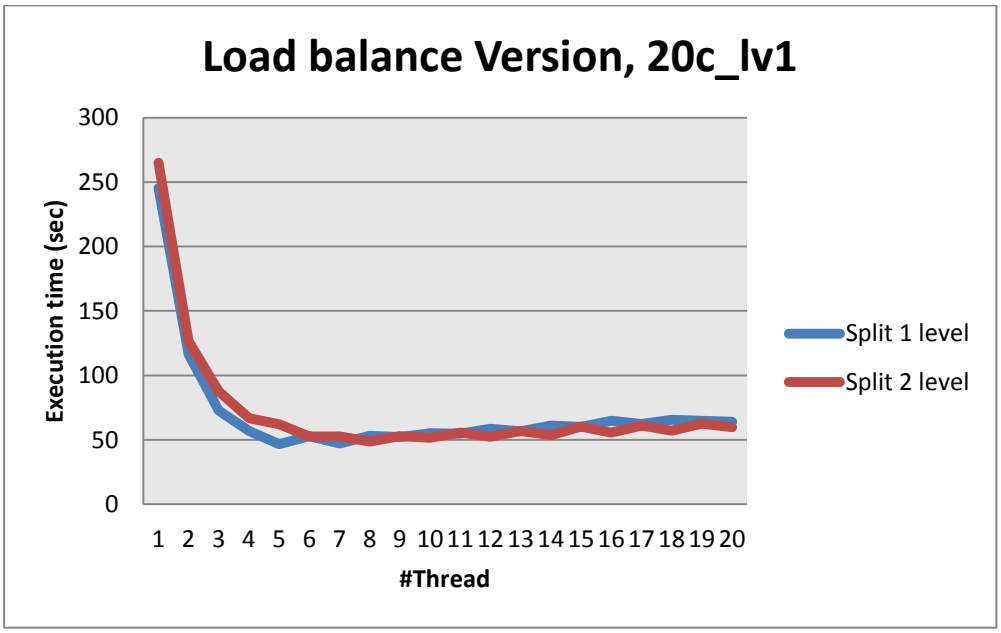
相對來說，在 split 2 level 切割 Job 比較細的情況下，這種現象就會出現的比較少。(如下圖)

如果 Problem size 更大而且將 Job 切割得更細的話，Load balance 的好處應該會更明顯的顯示出來！



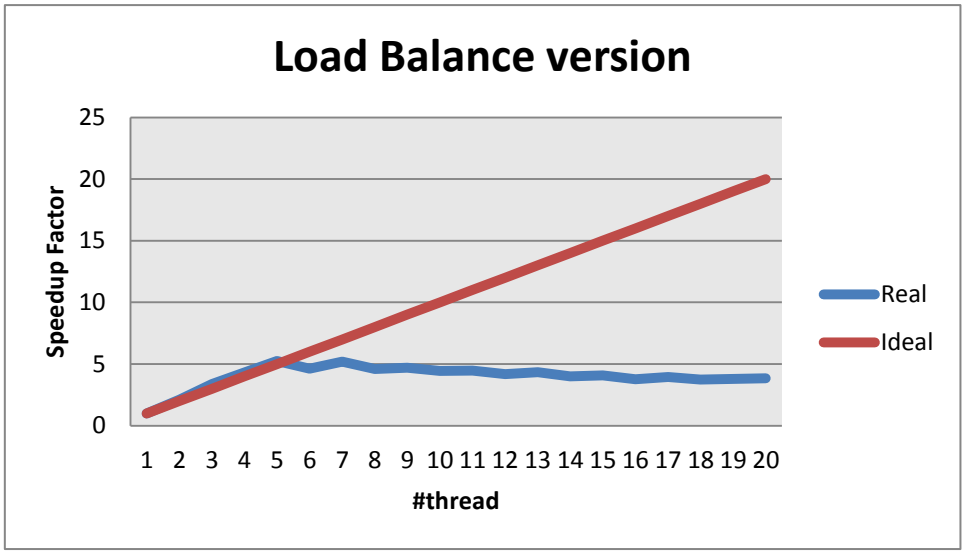
(4) #Job effect

這筆實驗主要測試到底 Job 切割的越細，Load 是否會更 balance? 更進一步對於效能的推進有多大?



由數據結果來看並沒有明顯的差異，猜想也是發生了上面討論 Balance effect 所產生的 case，造成那種 Unbalance 會比較好的原因如之前所討論的：切的不夠細！因為我在切割 Job 的時候並不知道每個 Job 的 Loading 為何(因為有做 cut)，在每個 Job 的 loading 不一樣的時候，其實做 Load balance 就沒甚麼多大意義！如果能將 Job 切割小到都差不多 Loading 時，應該就真能顯示出 Load balance 機制的好處！

(5) Scalability



從結果發現在 #thread <= 5 時幾乎貼近理想曲線，但在之後就幾乎維持定值，對於這個奇怪的現象我猜想應該是在以 city = 4 為起點的時候會產生一條路徑進行強力的剪枝，但在多開 thread 平行從不同的城市出發時，也是要等到 city 4 為開頭的那個 thread 更新完那個強力剪枝的答案後，才能進行剪枝以免走冤枉路。而 thread 開越多加速越慢的原因也是因為如此，因為在 thread = 4 的時候會先跑 city = 1~4 為起點的情況，之後就會因為剪枝的關係少很多計算，但如果 thread 數量一多，反而會走多餘的路徑，造成效能減低。

所以為了驗證這個猜想是否正確，我將每次更新最佳解的內容給輸出，發現到最後更新的 5 條路徑分別為

start = 3, update ans = 505
start = 2, update ans = 502
start = 4, update ans = 490
start = 4, update ans = 460
start = 20, update ans = 454

的確說明了我所猜想，許多強力的剪枝都集中在前面的 case，後半段 thread 開越多越糟糕的現象。

參、Best execution time

	5c_lv1	5c_lv2	5c_lv3	10c_lv1	10c_lv2	10c_lv3	20c_lv1	20c_lv2	20c_lv3
Exec time (sec)	0.00054	0.00067	0.00032	0.0012	0.0014	0.00068	27.82	2.18	1.11
#thread	2	3	2	3	7	2	10	11	10

肆、Conclusion

根據實驗結果來回應先前的問題，並做出以下幾點結論

(1) 因為進行暴搜的搜索樹切割成數個子樹後，彼此的相依姓相當的低，所以非常容易去做平行化，而且會有很明顯的加速！

(2) 為了加速搜索所做的剪枝，其產生的效益相當的高，所以就算在平行處理要多花一點 mutex 的 overhead 去保護資料的 Synchronization 也是相當值得的！

(3) 由於 Job 在經由剪枝優化過後 Load 都不盡相同，所以除非能夠將 Job 切的非常的細，細到 Loading 差不多，那才會有做 Load balance 的價值。