Tree search

Sriraghav K

Tree search - TSP

- Many problems can be solved using a tree search = Ex. **TSP**
- In TSP, a salesperson is given a list of cities she needs to visit and a cost for traveling between each pair of cities.
- Her problem is to visit each city once, returning to her hometown, and she must do this with the least possible cost.
- A route that starts in her hometown, visits each city once and returns to her hometown is called a *tour*; thus, the TSP is to find a minimum-cost tour.

TSP

- An **NP-complete problem**.
- No known solution to TSP that is better in all cases than exhaustive search.
- Ex., the travelling salesperson problem, finding a minimum cost tour.

TSP = Exhaustive Search Solution

- **Exhaustive search** means **examining all possible solutions** to the problem and **choosing the best.**
- The **number of possible solutions** to TSP **grows exponentially** as the number of cities is increased.
- For example, if we add one additional city to an **n-city problem**, we'll increase the number of possible solutions by a **factor of n-1.**
  - **5 city problem . Solutions = 4!**

TREE-SEARCH SOLUTIONS FOR TSP

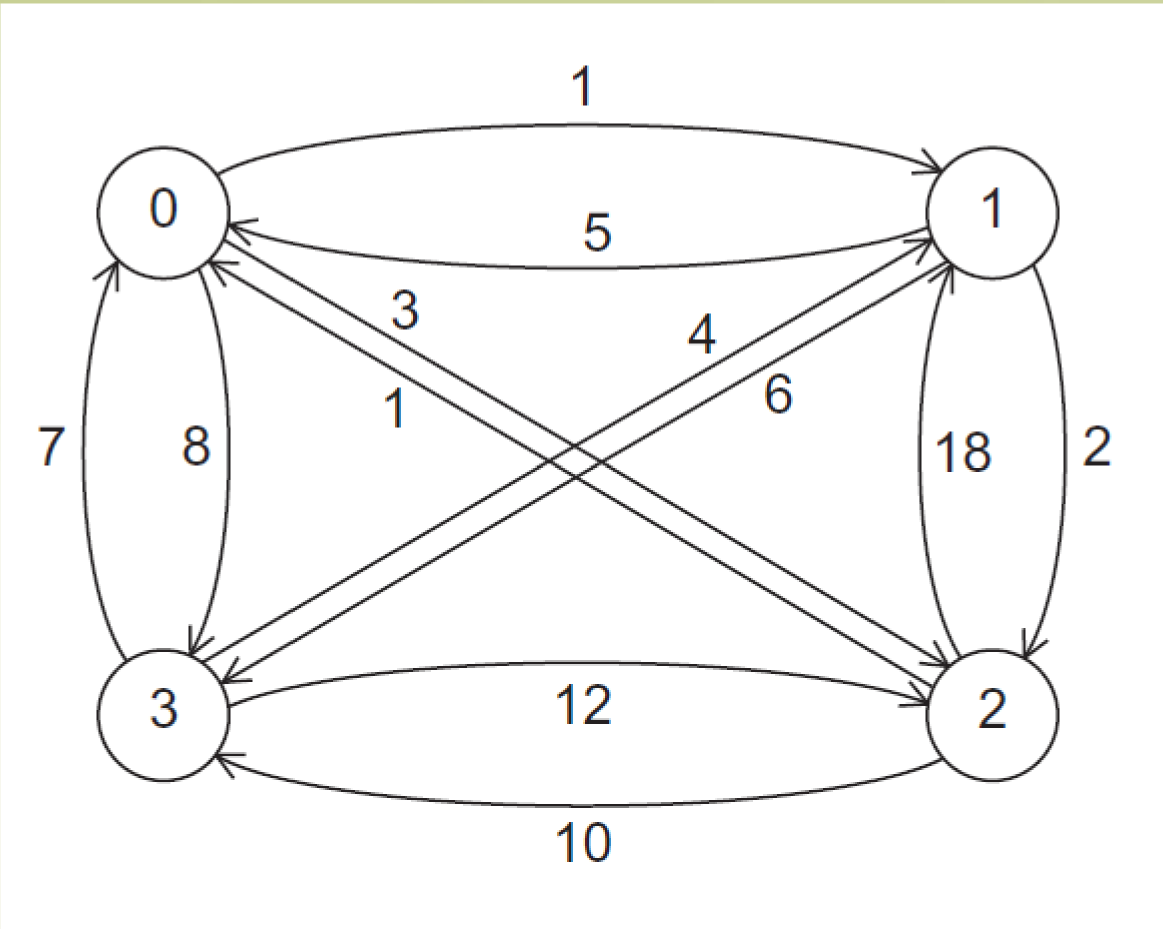
Tree Search Solution to TSP

- The idea is that in searching for solutions, we build a tree.
- The **leaves** of the tree correspond to **tours**.
- **Other tree nodes** correspond to **“partial” tours**—routes that have visited some, but not all, of the cities.
- **Each node** of the tree has an **associated cost**, that is, the cost of the partial tour. We can use this to eliminate some nodes of the tree.

Know Some Graph Basics

- We should represent a **four-city TSP** as a **labeled, directed graph**.
- A **graph** is a **collection of vertices and edges** or line segments joining pairs of vertices.
- In a **directed graph** or digraph, **the edges are oriented**—one end of each edge is the tail, and the other is the head.
- A graph or digraph is labeled if the vertices and/or edges have labels.

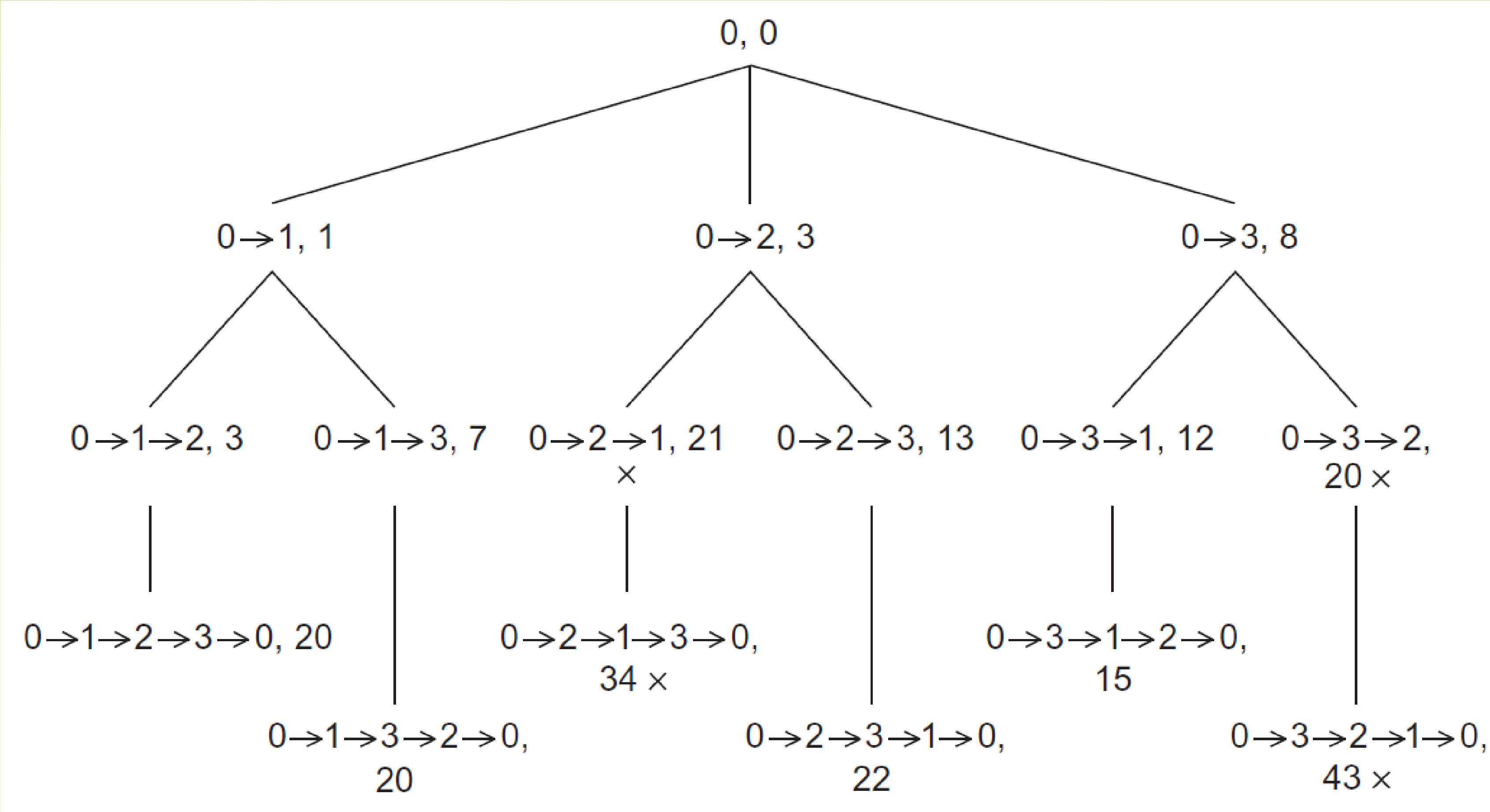
A Four-City TSP



TSP Tree-Search Graph is labelled

- In our example, the **vertices** of the digraph correspond to the **cities** in an instance of the TSP.
- The **edges** correspond to **routes between the cities**.
- The **labels** on the edges correspond to the **costs** of the routes.
- For example, there's a cost of 1 to go from city 0 to city 1 and a cost of 5 to go from city 1 to city 0.

Search Tree for Four-City TSP



Tree-Search Solution – Step 1- **Construct tree**

- If we choose **vertex 0** as the **salesperson's home city**, then the **initial partial tour** consists only of **vertex 0**, and since we've gone nowhere, it's **cost is 0**.
- From 0 we can first visit 1, 2, or 3, giving us three two-city partial tours with costs 1, 3, and 8, respectively.
- **Continue building the tree** until **all** the **cities** have been **traversed**.
- **Sum** up the **cost** of the cities along each branch while traversal.

Tree-Search Solution – Step 2- **Search tree**

- Now, to find a **least-cost tour**, we should **search the tree**.
- In **depth-first search**, we **probe** as **deeply** as we can into the tree.
- If not least cost, we back up to the deepest "ancestor" tree node with unvisited children, and probe one of its children as deeply as possible.

deeply as possible.

In our example, we'll start at the root, and branch left until we reach the leaf labeled

$$0 \to 1 \to 2 \to 3 \to 0, \text{ Cost } 20.$$

Then we back up to the tree node labeled $0 \to 1$, since it is the deepest ancestor node with unvisited children, and we'll branch down to get to the leaf labeled

$$0 \to 1 \to 3 \to 2 \to 0, \text{ Cost } 20.$$

Continuing, we'll back up to the root and branch down to the node labeled $0 \to 2$. When we visit its child, labeled

$$0 \to 2 \to 1, \text{ Cost } 21,$$

we'll go no further in this subtree, since we've already found a complete tour with cost less than 21. We'll back up to $0 \to 2$ and branch down to its remaining unvisited child. Continuing in this fashion, we eventually find the least-cost tour

$$0 \to 3 \to 1 \to 2 \to 0, \text{ Cost } 15.$$

1. Recursive solution to TSP using depth-first search

```
void Depth_first_search(tour_t tour) {
   city_t city;

   if (City_count(tour) == n) {
      if (Best_tour(tour))
         Update_best_tour(tour);
   } else {
      for each neighboring city
         if (Feasible(tour, city)) {
            Add_city(tour, city);
            Depth_first_search(tour);
            Remove_last_city(tour);
         }
   }
}  /* Depth_first_search */
```

The algorithm makes use of several global variables:

- `n`: the total number of cities in the problem
- `digraph`: a data structure representing the input digraph
- `hometown`: a data structure representing vertex or city 0, the salesperson's hometown
- `best_tour`: a data structure representing the best tour so far

1. Recursive solution to TSP using depth-first search

- City_Count(): The function City count **examines the partial tour** to **see if there are *n* cities** on the partial tour.
- If there are, we need to **return to the hometown to complete the tour**
- We can **check** to see **if the complete tour has a lower cost** than the **current** "**best tour**" by calling **Best tour()**.
- **If it does**, we can **replace** the current best tour with this tour by calling the function **Update best tour().**

1. Recursive solution to TSP using depth-first search

- Remember that **best tour variable** should be **initialized** so that its **cost is greater** than the **cost of any possible least-cost tour**.
- If the **partial tour hasn't visited n cities**, we can continue branching down in the tree by "**expanding the current node**," - loop through the cities.
- The function **Feasible** checks to **see if the city** or vertex has **already been visited**, and, if not, whether it can possibly lead to a least-cost tour.

1. Recursive solution to TSP using depth-first search

- **If** the **city is feasible**, we **add** it to the **tour**, and recursively call **Depth first search**.
- When we return from Depth first search, we remove the city from the tour, since it shouldn’t be included in the tour used in subsequent recursive calls.

1. Recursive solution to TSP using depth-first search

- **<u>Disadvantages:</u>**
  - Function calls are expensive.
  - Recursion makes it very slow.
  - At any given instant of time only the current tree node is accessible. This could be a problem when we try to parallelize tree search by dividing tree nodes among processes or threads.

2. Depth-first solution to TSP - Iterative solution

```
for (city = n-1; city >= 1; city--)
   Push(stack, city);
while (!Empty(stack)) {
   city = Pop(stack);
   if (city == NO_CITY) // End of child list, back up
      Remove_last_city(curr_tour);
   else {
      Add_city(curr_tour, city);
      if (City_count(curr_tour) == n) {
         if (Best_tour(curr_tour))
            Update_best_tour(curr_tour);
         Remove_last_city(curr_tour);
      } else {
         Push(stack, NO_CITY);
         for (nbr = n-1; nbr >= 1; nbr--)
            if (Feasible(curr_tour, nbr))
               Push(stack, nbr);
      }
   } /* if Feasible */
} /* while !Empty */
```

2. Depth-first solution to TSP - Iterative solution

- **Recursive function** works with the **run-time stack**.
- Thus, in this iterative version, we can try to **eliminate recursion** by **pushing necessary data** on our own stack before branching deeper into the tree.
- When we need to go back up the tree—
  - Either because we've reached a leaf
  - Or we found a node that can't lead to a better solution—we can pop the stack.

2. Depth-first solution to TSP - Iterative solution

- The **loop termination condition** is that our **stack is empty**.
- As long as the search needs to continue, we need to make sure the stack is nonempty.
- **NO_CITY:** This constant is used so that we can tell when we've visited all of the children of a tree node.

Data structures for the serial implementations

Data Structures for tree search

- Our principal data structures are - **the tour**, **the digraph**, and, in the iterative implementations, **the stack**.
- The **tour and the stack** are essentially **list** structures.
- As an improvement, we use a **struct** with three members:
  - The array storing the cities
  - The number of cities
  - The cost of the partial tour.
- **Digraphs** represented using **adjacency matrices**.

Using pre-processor macros

```
/* Find the ith city on the partial tour */
int Tour_city(tour_t tour, int i) {
   return tour->cities[i];
}  /* Tour_city */
```

```
/* Find the ith city on the partial tour */
#define Tour_city(tour, i) (tour->cities[i])
```

Run-Times of the Three Serial Implementations of Tree Search

Recursive	First Iterative	Second Iterative
30.5	29.2	32.9

(in seconds)

The digraph contains 15 cities. All three versions visited approximately 95,000,000 tree nodes.

Parallelizing tree search

Communication between nodes and their functions

- The **tree structure** suggests that **we identify tasks with tree nodes**.
- A **parent** will **communicate** a **new partial tour** to **a child**.
- A child, except for terminating, doesn't communicate directly with a parent.

Functions :

- **Inner Nodes**: examines the best tour to determine whether the current partial tour is feasible or the current complete tour has lower cost.
- **Leaf task**: determines if its tour is a better tour and update the best tour.

Need for parallelizing

- The Best tour data structure “sends” data to every tree node task, and receives data from some of the leaves.
- This latter view is convenient for shared-memory, but not so convenient for distributed-memory.
- A natural way to agglomerate and map the tasks is to **assign a sub tree to each thread** or process, and **have each thread/process carry out all the tasks in its sub tree.**
- Refer the tree diagram.

Parallelizing method - 1

- **Processes operate independently** of each other **until** they **have completed searching their sub trees**.
- **Each process** would store its **own local best tour**. This local best tour would be used by the process in **Feasible** and updated by the process each time it calls **Update best tour**.
- When all the processes have finished searching, they can perform a global reduction to find the tour with the global least cost.
- **Adv :** Simplicity
- **Disadv:** 1. Wasting time on partial tours that may not lead to best soln

  2. Load imbalance

Parallelizing method 2 – **Dynamic mapping of tasks**

- **In a dynamic scheme, if one thread/process runs out of useful work, it can obtain additional work from another thread/process.**
- Each stack record contains a partial tour.
- With this data structure, a thread or process can give additional work to another thread/process by dividing the contents of its stack.
- **Second solution** for load imbalance = **Shared stack**.

Static parallelization of tree search using pThreads

Idea of Implementation

- In this parallel version, we need to generate at least **thread_count partial tours** to **distribute** among the threads.
- As we discussed earlier, we can use **breadth-first search** to generate a list of at least thread_count tours .
- A single thread search the tree until it reaches a level with at least thread_count tours.

Problem → RACE CONDITION

- To implement the Best_tour function, **a thread** should **compare** the **cost of its current tour** with the **cost of the global best tour**.
- Since **multiple threads** may be **simultaneously accessing** the **global best cost**, it might at first **seem** that there will be a **race condition**. **-- Actually no race.**
- However, the Best_tour function **only reads the global best cost**, so there **won’t be any conflict** with threads that are also checking the best cost.

Problem → RACE CONDITION

- On the other hand, we call Update_best_tour with the intention of ***writing*** to the best_tour structure
- This clearly can **cause a race condition if two threads call it simultaneously**.
- To **avoid this problem**, we can **protect** the body of the Update_best_tour function with a **mutex**.

```
pthread_mutex_lock(best_tour_mutex);
/* We've already checked Best_tour, but we need to check it
    again */
if (Best_tour(tour))
   Replace old best tour with tour;
pthread_mutex_unlock(best_tour_mutex).
```

Dynamic parallelization of tree search using pThreads

Dynamic = Share your work to other free threads!!!!!!

Pthreads condition variables provide a natural way to implement this.

a) When a **thread** runs **out of work,** it can **call** **pthread_cond_wait** and **go to sleep**.

b) When a **thread** with work **finds** that **there is at least one thread waiting for work**,
   a) Splitting its stack
   b) Call **pthread_cond_signal**.

c) When a **thread** is **awakened,** it can **take one of the halves** of the split stack and return to **work**.

Termination Conditions

- checks that it has **at least two tours** in its stack
- checks that there are **threads waiting**
- checks whether the **new_stack variable** is **NULL**.

Tree Search – OpenMP implementations

1. Master-Slave Model – Slave Code

When a single thread executes some code in the PThreads version, the test

if (my rank == whatever)

can be replaced by the **OpenMP directive**

 # pragma omp single

This will insure that the **following structured block** of code will be **executed by one thread in the team**, and the **other threads** in the team **will wait in an implicit barrier** at the end of the block until the executing thread is finished.

1. Master-Slave Model – Master Code

When **whatever is 0** (as it is in each test in the Pthreads program), the test can also be replaced by the OpenMP directive

 # pragma omp master

This will insure that **thread 0 executes the following structured block of code.**

However, the **master directive doesn't put an implicit barrier** at the end of the block, so it may be **necessary to also add a barrier directive** after a structured block that has been modified by a master directive.

2. OpenMP → Working with locks

- The dynamically load-balanced PThreads implementation depends heavily on PThreads condition variables.
- OpenMP provides a lock object **omp_lock_t** and the following functions for acquiring and relinquishing the lock, respectively:

```
void omp_set_lock(omp_lock_t*    lock_p   /* in/out */);
void omp_unset_lock(omp_lock_t*  lock_p   /* in/out */);
```

It also provides the function

```
int omp_test_lock(omp_lock_t*    lock_p   /* in/out */);
```

3. Emulating a condition wait in OpenMP

Usually, a thread starts to wait because:

- Another thread has split its stack and created work for the waiting thread.
- All of the threads have run out of work.

OpenMP implementation : Since there are two conditions a waiting thread should test for, we can use two different variables in the **busy-wait loop**:

```
/* Global variables */
int awakened_thread = -1;
int work_remains = 1;  /* true */
. . .
while (awakened_thread != my_rank && work_remains);
```

4. Splitting my stack !!!!!

- When a thread runs out of work, it enqueues its rank before entering the busy-wait loop.
- When a thread splits its stack, it can choose the thread to awaken by dequeuing the queue of waiting threads

```
got_lock = omp_test_lock(&term_lock);
if (got_lock != 0) {
   if (waiting_threads > 0 && new_stack == NULL) {
      Split my_stack creating new_stack;
      awakened_thread = Dequeue(term_queue);
   }
   omp_unset_lock(&term_lock);
}
```

Performance Study : OpenMP implementations

Table 6.9 Performance of OpenMP and Pthreads Implementations of Tree Search (times in seconds)

	First Problem						**Second Problem**					
	Static		*Dynamic*				*Static*		*Dynamic*			
Th	*OMP*	*Pth*	*OMP*		*Pth*		*OMP*	*Pth*	*OMP*		*Pth*	
1	32.5	32.7	33.7	(0)	34.7	(0)	25.6	25.8	26.6	(0)	27.5	(0)
2	27.7	27.9	28.0	(6)	28.9	(7)	25.6	25.8	18.8	(9)	19.2	(6)
4	25.4	25.7	33.1	(75)	25.9	(47)	25.6	25.8	9.8	(52)	9.3	(49)
8	28.0	23.8	19.2	(134)	22.4	(180)	23.8	24.0	6.3	(163)	5.7	(256)