Traveling Salesman Problem
Parallel Distributed Tree Search

Ned Nedialkov

Dept. of Computing and Software
McMaster University, Canada
nedialk@mcmaster.ca

March 2012

Outline

- Traveling salesman problem (TSP)
- Recursive depth-first search (DFS)
- Iterative DFS
- Parallelizing tree search
- Dynamic mapping
- Some MPI issues

For more details, see Chapter 6, section 2 in the textbook

Traveling Salesman Problem (TSP)

Given a set of cities and the distances between them, determine the shortest path starting from a given city, passing through all the other cities and returning to the first city

Symmetric TSP: the distance between two cities is the same in both directions

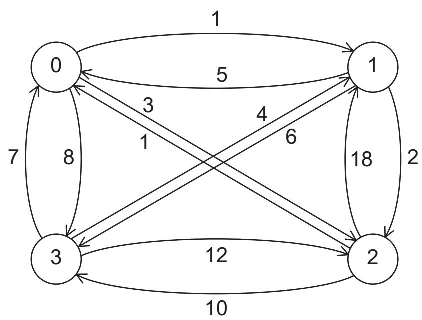
TSP can be modeled as undirected weighted graph

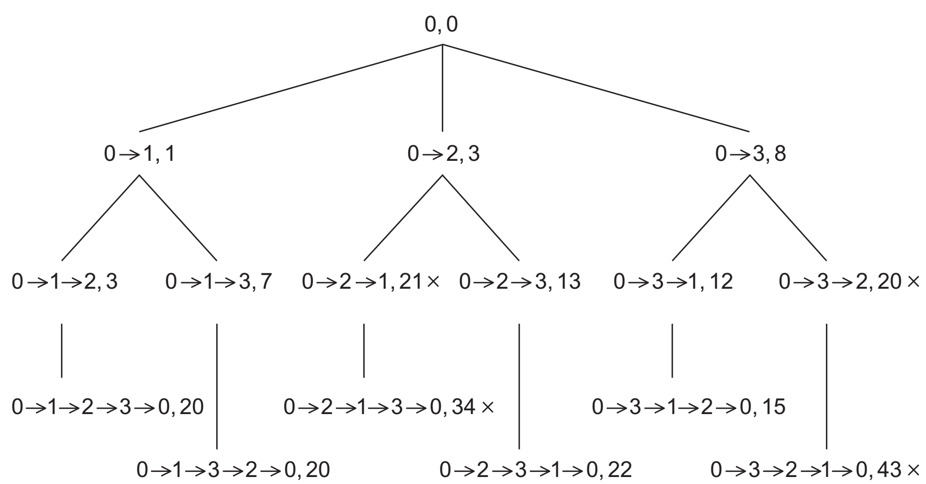
- vertices denote cities
- edges connect them, and the weight of an edge is the distance between two cities

Asymmetric TSP: paths may not exist in both directions, or the distances might be different in opposite directions. We have a directed graph

Can arise e.g. with one-way streets or say air fares that are different in opposite directions

- This is NP-hard problem; no polynomial type algorithm exists
- We study how to do parallel distributed tree search
- Example

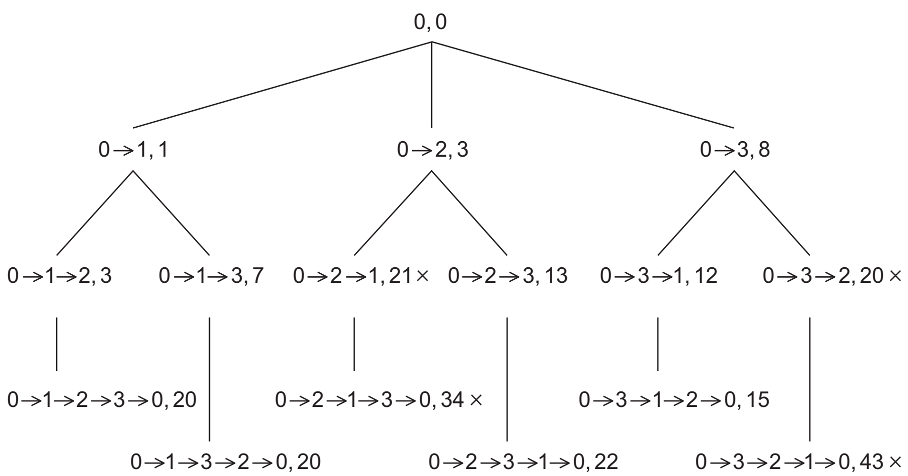




(Figures 6.9 and 6.10 from P. Pacheco, An Introduction to Parallel Programming)

Tree search

- Start at the origin, here city 0
- Do depth-first search
  - maintain the current best tour, that is minimum cost
  - if a node is reached with cost larger than current minimum cost, do not go deeper



For example, going $0 \to 2 \to 1$, we have cost 21 at 1, and we can stop, as $0 \to 1 \to 2 \to 3 \to 0$ has cost 20

Serial depth-first search

- Cities are numbered $0, 1, \ldots, n-1$
- A tour contains number of cities, the cities in the tour, and the cost of it
- number of cities is citycount (tour)
- Initially, tour contains the first city 0 and cost 0
- besttour(tour) checks if this is the best tour so far
- updatebesttour(tour) updates the best tour
- feasible (tour, city) checks if city has been visited, and if not, if it can be added to tour so that
  cost up to city $<$ cost(best_tour)
- add(tour, city) adds city to tour; city must be feasible
- removelast(tour, city) removes last city from tour
- We consider recursive and iterative depth first searches

Recursive DFS

Algorithm (Recursive DFS)

```
DepthFirstSearch(tour)
if citycount(tour) = n
    if besttour(tour)
        updatebesttour(tour)
else
    for each neighboring city of last city in tour
        if feasible (city, tour)
            addcity(tour, city)
            DepthFirstSearch(tour)
            removelast(tour, city)
```

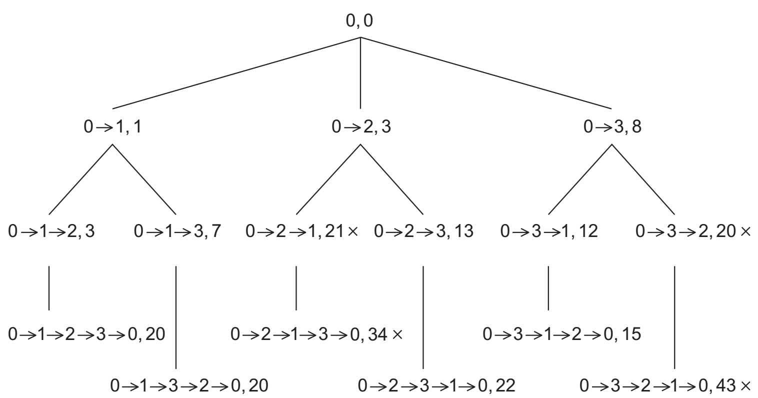
Iterative DFS. Version I

- Use stack to avoid recursive calls
- push(stack,city) pushes city onto stack
- pop(stack) pops city from stack
- Use -1 to recognize when all children of a node are visited
- Stack contains only feasible cities, or -1 to mark the beginning of children nodes

Algorithm (Iterative DFS. Version I)

```
for city ← n − 1 downto 0
    push(stack, city )
while stack is non empty
    city ←pop(stack)
    if city = −1 % no children to visit
        removelast(tour,city)
    else
        add(tour, city )
        if citycount (tour) = n
            if besttour (tour)
                updatebesttour(tour)
                removelast(tour,city)
        else
            push(stack,−1) % mark beginning of children list
            for b = n − 1 downto 1
                if feasible (tour, b)
                    push(stack, b)
```

Example



stack $n-1,\ldots,1,0$, tour empty

After first iteration:

city = 0

tour = 0,0

stack $n-1,\ldots,1,-1,3,2,1$

After second iteration:

city = 1

tour = $0 \to 1, 1$

stack

$n-1,\ldots,1,-1,3,2,-1,3,2$

After third iteration:

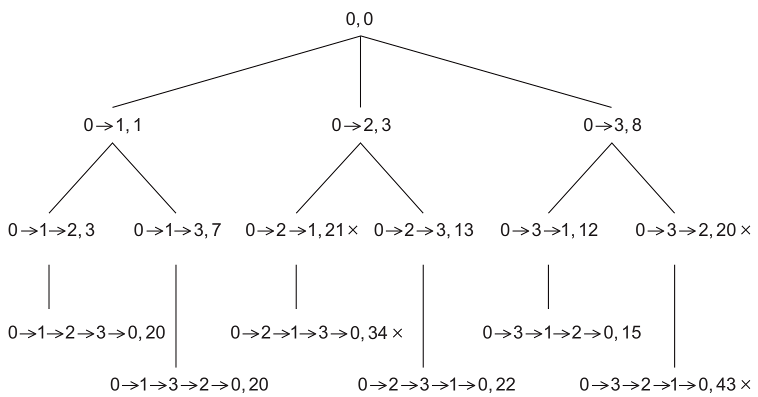
city = 2

tour = $0 \to 1 \to 2, 3$

stack

$n-1,\ldots,1,-1,3,2,-1,3,-1,3$

Example



After third iteration:
city = 2
tour = $0 \to 1 \to 2, 3$
stack
$n-1, \ldots, 1, -1, 3, 2, -1, 3, -1, 3$
After fourth iteration:
city = 3
tour = $0 \to 1 \to 2 \to 3, 20$
best_tour = $0 \to 1 \to 2 \to 3 \to 0, 20$
tour = $0 \to 1 \to 2, 3$
stack
$n-1, \ldots, 1, -1, 3, 2, -1, 3, -1$
After fifth iteration:
city = -1
tour = $0 \to 1, 1$
stack
$n-1, \ldots, 1, -1, 3, 2, -1, 3$

Iterative depth first search. Version II

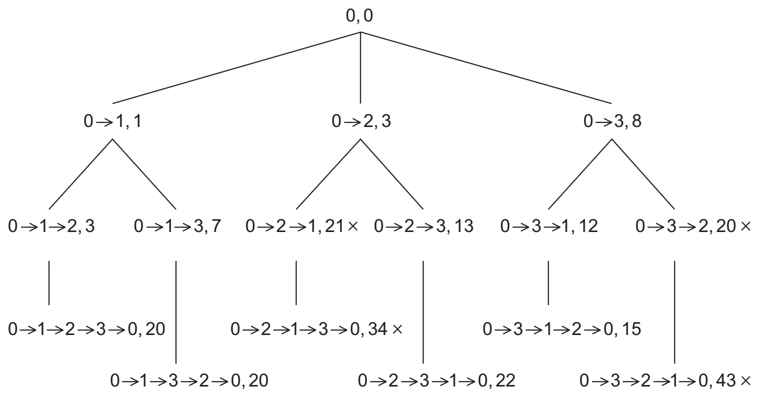
- Push partial tours onto stack
- Code closer to recursive version, but slower than Version I
- Entire (partial) tours are in stack
- Multiple processes can get tours to work on

Algorithm (Iterative depth-first-search. Version II)

push(stack, tour)
while stack is non empty
    tour←pop(stack)
    if citycount(tour) = n
        if besttour(tour)
            updatebesttour(tour)
    else
        for b = $n - 1$ **downto** 1
            if feasible (tour, b)
                add(city, tour)
                push(stack, tour)
                removelast(tour, city)

Example

Initially stack contains 0,0



Iteration 1 of **while** loop

tour = 0,0

$b = 3$, tour $= 0 \rightarrow 3, 8$

stack $0 \rightarrow 3, 8$

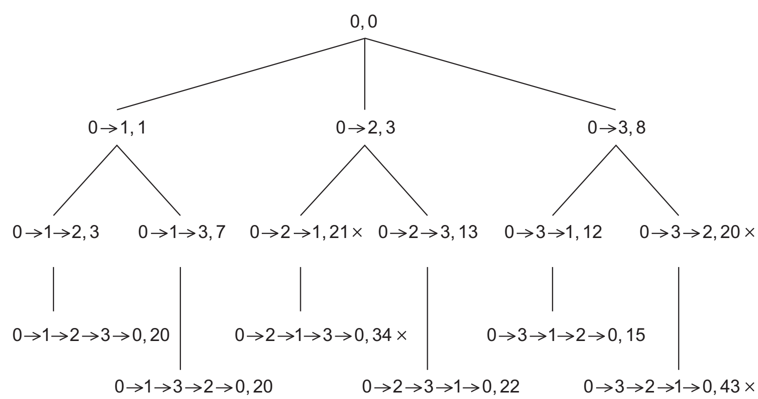
$b = 2$, tour $= 0 \rightarrow 2, 3$

stack $0 \rightarrow 3, 8$
$0 \rightarrow 2, 3$

$b = 1$, tour $= 0 \rightarrow 1, 1$

stack $0 \rightarrow 3, 8$
$0 \rightarrow 2, 3$
$0 \rightarrow 1, 1$

Example



stack $0 \to 3, 8$
$0 \to 2, 3$
$0 \to 1, 1$

Iteration 2 of **while** loop

pop tour $= 0 \to 1, 1$

stack $0 \to 3, 8$
$0 \to 2, 3$

$b = 3$, tour $= 0 \to 1 \to 3, 7$

stack $0 \to 3, 8$
$0 \to 2, 3$
$0 \to 1 \to 3, 7$

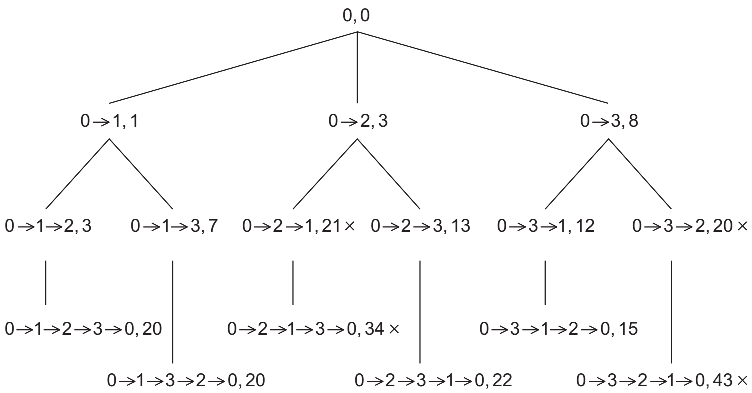
$b = 2$, tour $= 0 \to 1 \to 2, 3$

$b = 1$, not feasible

stack $0 \to 3, 8$
$0 \to 2, 3$
$0 \to 1 \to 3, 7$
$0 \to 1 \to 2, 3$

Parallelizing tree search

Recursive and Version I DFS are difficult (if not impossible) to parallelize. Why?



stack after iteration 2

$0 \to 3, 8$
$0 \to 2, 3$
$0 \to 1 \to 3, 7$
$0 \to 1 \to 2, 3$

A process can get a stack record and work with it independently!

Mapping
Assume p processes
One process could run Version II until there are p tours in the stack
Assign them to processes

Best tour
Processes work independently until each finds it local best tour
Do global reduction on process 0 to find the best tour
Simple, but

- a process may search through partial tours that cannot lead to global best tour

Dynamic mapping

When a process runs out of work, get more work
In version II, each stack entry is partial tour
A process can get a partial tour and work on it
The order in which nodes are visited does not matter

Graph representation
Represent the graph using adjacency matrix
Entry (i, j) is the weight of edge from vertex i to vertex j
Process 0 sends this matrix to each process
Since the adjacency matrix is not large (e.g. 100×100) each process can store it

We have to

- partition the tree
- check and update best tour
- get best tour on process 0

Dynamic mapping

When a process runs out of work, it can busy-wait for

- more work or
- notification that program is terminating

A process with work can send part of it to an idle process
Alternatively, a process without work can request such form other process(es)
How can we do this?

Terminated algorithm

A process checks if it has at least two tours
If it has received request for work, it splits its stack and sends work to the requesting process
If it cannot send, it sends "no work" reply

Setup:

- a process has its own my_stack
- fulfill_request(my_stack) checks if a request for work is received; if so it splits the stack and sends work
- send_rejects checks for work requests and sends "no work" reply

  If no requests, it simply returns

Algorithm (Terminated)

```
if number of tours in my_stack ≥ 2
    fulfill_request(my_stack)
    return false
else
    send_rejects
    if my_stack is non empty % 1 tour
        return false
    else % empty stack
        if only one process left
            return true
        announce "out of work"
        workrequest = false
        while (1)
            if no work left return true
```

Algorithm (Terminated Cont.)

```
    else
        if workrequest = false
            send work request
            workrequest = true
        else
            check for work
            if work available
                receive it in my_stack
                return false
```

MPI issues

Process 0 generates and sends partial tours to *p* processes
Use `MPI_Scaterv`
When a process finds a best tour, it sends its cost to all other processes
MPI_Bcast cannot be used, as processes will block
We can do

```
for (dest =0; dest < p; dest++)
    if (dest!=my_rank)
        MPI_Send(&new_best_cost ,1 ,MPI_INT , dest ,
                                 NEW_COST_TAG, comm ) ;
```

NEW_COST_TAG tells the receiving process the message contains new cost
Destination can check periodically using

```
MPI_Recv(&received_cost ,1 ,MPI_INT ,MPI_ANY_SOURCE,
             NEW_COST_TAG, comm,&status );
```

But receiving process will block

Can use

```
int MPI_Iprobe(int src, int tag, MPI_Comm comm,
                                   int *msg, MPI_Status *status
```

Checks if a message from src with tag in communicator comm is available

If such is available *msg is 1 and status->MPI_SOURCE contains the source; otherwise *msg is 0

To check if there is a message from any source

```
MPI_Iprobe(MPI_ANY_SOURCE,NEW_COST_TAG, &msg, &status)
```

If msg=1, we can receive with

```
MPI_Recv(&received_cost,1,MPI_INT,status.MPI_SOURCE,
                    NEW_COST_TAG,comm,MPI_STATUS_IGNORE);
```