Tree Search

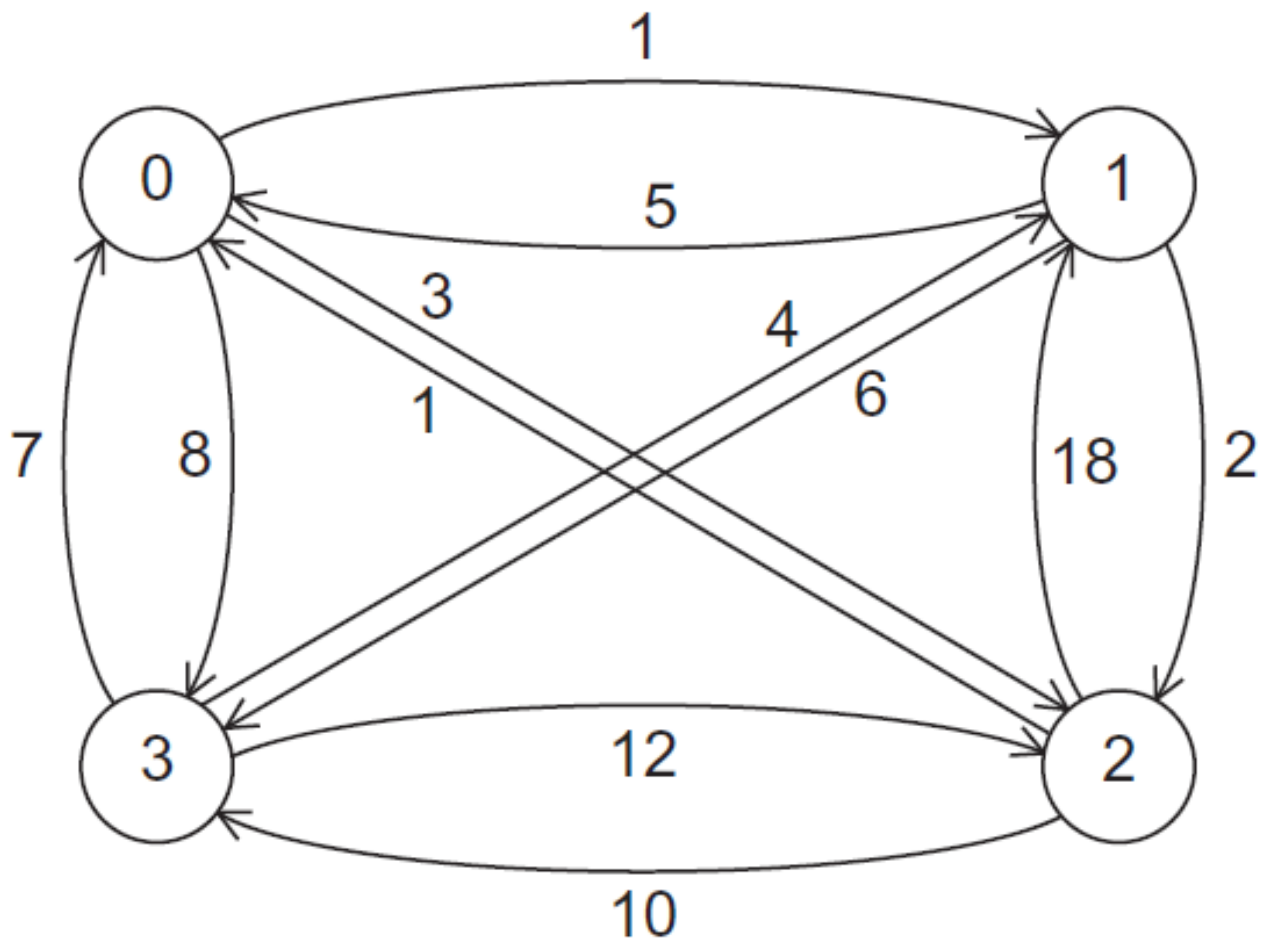
Tree search problem (TSP)

An NP-complete problem.

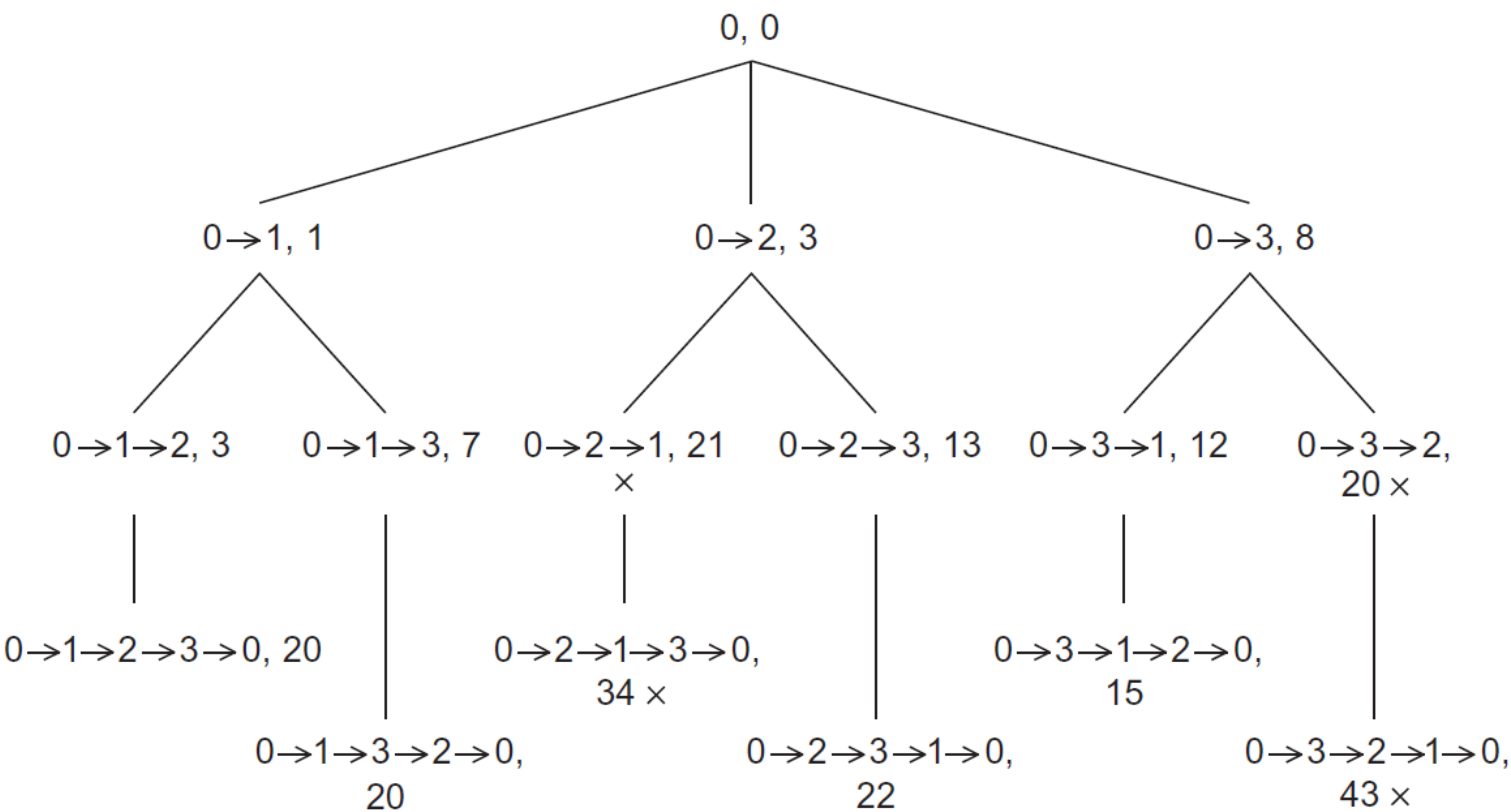
No known solution to TSP that is better in all cases than exhaustive search.

Ex., the travelling salesperson problem, finding a minimum cost tour.

A Four-City TSP



Search Tree for Four-City TSP

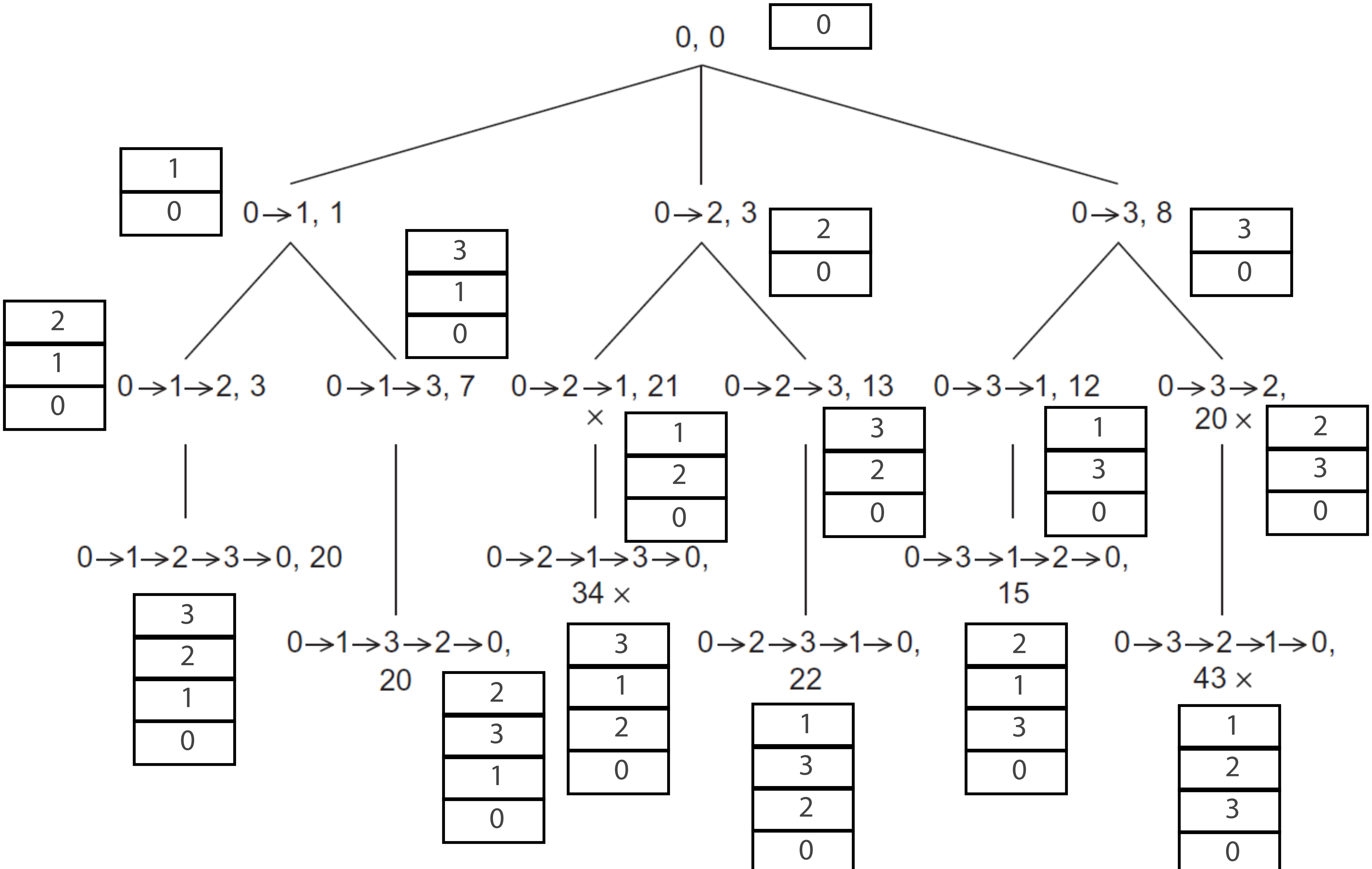


Pseudo-code for a recursive solution to TSP using depth-first search

```
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 */
```

Search Tree for Four-City TSP



Pseudo-code for an implementation of a depth-first solution to TSP without recursion

```
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 */
```

Pseudo-code for a second solution to TSP that doesn't use recursion

```
Push_copy(stack, tour);  // Tour that visits only the hometown
while (!Empty(stack)) {
   curr_tour = Pop(stack);
   if (City_count(curr_tour) == n) {
      if (Best_tour(curr_tour))
         Update_best_tour(curr_tour);
   } else {
      for (nbr = n-1; nbr >= 1; nbr--)
         if (Feasible(curr_tour, nbr)) {
            Add_city(curr_tour, nbr);
            Push_copy(stack, curr_tour);
            Remove_last_city(curr_tour);
         }
   }
   Free_tour(curr_tour);
}
```

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.

Pseudo-code of a statically parallelized solution to TSP

```
Partition_tree(my_rank, my_stack);

while (!Empty(my_stack)) {
   curr_tour = Pop(my_stack);
   if (City_count(curr_tour) == n) {
      if (Best_tour(curr_tour)) Update_best_tour(curr_tour);
   } else {
      for (city = n-1; city >= 1; city--)
         if (Feasible(curr_tour, city)) {
            Add_city(curr_tour, city);
            Push_copy(my_stack, curr_tour);
            Remove_last_city(curr_tour)
         }
   }
   Free_tour(curr_tour);
}
```

Partition the Tree

Process 0 uses breadth-first search to search the tree until there are at least process count partial tours.

- Each process then determines which of these initial partial tours it should get and pushes its tours onto its local stack.

Process 0 will need to send the initial partial tours to the appropriate process.

Sending a different number of objects to each process in the communicator

```
int MPI_Scatterv(
      void*          sendbuf         /* in  */,
      int*           sendcounts      /* in  */,
      int*           displacements   /* in  */,
      MPI_Datatype   sendtype        /* in  */,
      void*          recvbuf         /* out */,
      int            recvcount       /* in  */,
      MPI_Datatype   recvtype        /* in  */,
      int            root            /* in  */,
      MPI_Comm       comm            /* in  */)
```

Maintaining the "best tour"

When a process finishes a tour, it needs to check if it has a better solution than recorded so far.

- having each process use its own best tour is likely to result in a lot of wasted computation

When a process finds a new best tour, it really only needs to send its cost to the other processes.

- it's important to recognize that we can't use MPI_Bcast

Maintaining the "best tour"

Option 1: use MPI_Send to send it to all the other processes

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

Option 2: use asynchronous send or non-blocking send

The destination processes can periodically check for the arrival of new best tour costs.

Checking to see if a message is available

```
int MPI_Iprobe(
        int           source       /* in  */,
        int           tag          /* in  */,
        MPI_Comm      comm         /* in  */,
        int*          msg_avail_p  /* out */,
        MPI_Status*   status_p     /* out */);
```

```
MPI_Iprobe(MPI_ANY_SOURCE, NEW_COST_TAG, comm,
           &msg_avail, &status);
```

Checking to see if a message is available

```
MPI_Iprobe(MPI_ANY_SOURCE, NEW_COST_TAG, comm, &msg_avail,
      &status);
while (msg_avail) {
   MPI_Recv(&received_cost, 1, MPI_INT, status.MPI_SOURCE,
         NEW_COST_TAG, comm, MPI_STATUS_IGNORE);
   if (received_cost < best_tour_cost)
      best_tour_cost = received_cost;
   MPI_Iprobe(MPI_ANY_SOURCE, NEW_COST_TAG, comm, &msg_avail,
     &status);
}  /* while */
```

Printing the best tour

```
struct {
   int cost;
   int rank;
} loc_data, global_data;

loc_data.cost = Tour_cost(loc_best_tour);
loc_data.rank = my_rank;

MPI_Allreduce(&loc_data, &global_data, 1, MPI_2INT, MPI_MINLOC, comm);
if (global_data.rank == 0) return;  /* 0 already has the best tour */
if (my_rank == 0)
   Receive best tour from process global_data.rank;
else if (my_rank == global_data.rank)
   Send best tour to process 0;
```

Dynamic Parallelization of Tree Search

Initial distribution of subtrees doesn't do a good job of distributing the work.

Processes with "small" subtrees will finish early, while the processes with large sub-trees will continue to work.

When a process runs out of work, instead of immediately exiting the while loop, the process waits to see if another process can provide more work.

A process that still has work in its stack finds that there is at least one process without work, and its stack has at least two tours, it can "split" its stack and provide work for one of the processes.

Terminated Function for a Dynamically Partitioned TSP solver with MPI (1)

```
if (My_avail_tour_count(my_stack) >= 2) {
   Fulfill_request(my_stack);
   return false;  /* Still more work */
} else { /* At most 1 available tour */
   Send_rejects();  /* Tell everyone who's requested */
                    /* work that I have none          */
   if (!Empty_stack(my_stack)) {
      return false;  /* Still more work */
   } else {  /* Empty stack */
      if (comm_sz == 1) return true;
      Out_of_work();
      work_request_sent = false;
      while (1) {
         Clear_msgs();  /* Messages unrelated to work, termination */
         if (No_work_left()) {
            return true;  /* No work left.  Quit */
```

Terminated Function for a Dynamically Partitioned TSP solver with MPI (2)

```
      } else if (!work_request_sent) {
         Send_work_request();   /* Request work from someone */
         work_request_sent = true;
      } else {
         Check_for_work(&work_request_sent, &work_avail);
         if (work_avail) {
            Receive_work(my_stack);
            return false;
         }
      }
   }  /* while */
 } /* Empty stack */
} /* At most 1 available tour */
```

Performance of MPI and Pthreads implementations of tree search

Th/Pr	First Problem				Second Problem			
	Static		Dynamic		Static		Dynamic	
	Pth	MPI	Pth	MPI	Pth	MPI	Pth	MPI
1	35.8	40.9	41.9 (0)	56.5 (0)	27.4	31.5	32.3 (0)	43.8 (0)
2	29.9	34.9	34.3 (9)	55.6 (5)	27.4	31.5	22.0 (8)	37.4 (9)
4	27.2	31.7	30.2 (55)	52.6 (85)	27.4	31.5	10.7 (44)	21.8 (76)
8		35.7		45.5 (165)		35.7		16.5 (161)
16		20.1		10.5 (441)		17.8		0.1 (173)

(in seconds)

the total number of times stacks were split