Tree Search

- Many problems can be solved using a tree search. As a simple example, consider the traveling salesperson problem, or 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 is what’s known as an NP-complete problem. ***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. Thus, although there are only six possible solutions to a four-city problem, there are 4x6 D 24 to a five-city problem, 5x24 D 120 to a six-city problem, 6x120 D 720 to a seven-city problem, and so on. In fact, a 100-city problem has far more possible solutions than the number of atoms in the universe!

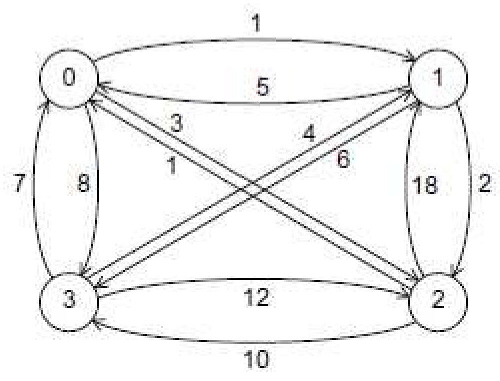


Fig 9. A four-city TSP

- The simple for of solution to TSP is a simple form of tree search. The idea is that in searching for solutions, we build a tree. The leaves of the tree correspond to tours, and the 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. Thus, we want to keep track of the cost of the best tour so far, and, if we find a partial tour or node of the tree that couldn’t possibly lead to a less expensive complete tour, we shouldn’t bother searching the children of that node (see Fig 9 and10).

- We’ve represented a four-city TSP as a labeled, directed graph. A graph (not to be confused with a graph in calculus) 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. 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, and 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.

- 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. Thus, the root of the tree in Figure 6.10 has the partial tour consisting only of the vertex 0 with cost 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. In Fig10 this gives us three children of the root. Continuing, we get six three-city partial tours, and since there are only four cities, once we've chosen three of the cities, we know what the complete tour is.

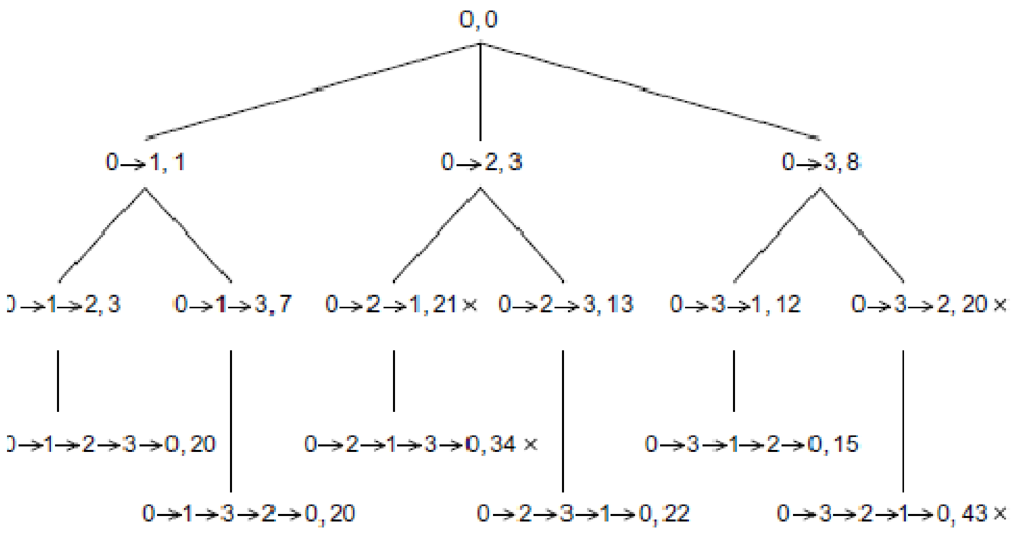


Fig 10:Search tree for four-city TSP

  - To find a least-cost tour, we should search the tree. There are many ways to do this, but one of the most commonly used is called depth-first search. In depth first search, we probe as deeply as we can into the tree. After we've either reached a leaf or found a tree node that can't possibly lead to a least-cost tour, we back up to the deepest "ancestor" tree node with unvisited children, and probe one of its children as deeply as possible.
  - In our example, we'll start at the root, and branch left until we reach the leaf labeled

0➔1➔2➔3➔0, Cost 20.

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

0➔1➔3➔2➔0, Cost 20.

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

0➔2➔1, 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➔2 and branch down to its remaining unvisited child. Continuing in this fashion, we eventually find the least-cost tour

0➔3➔1➔2!0, Cost 15.

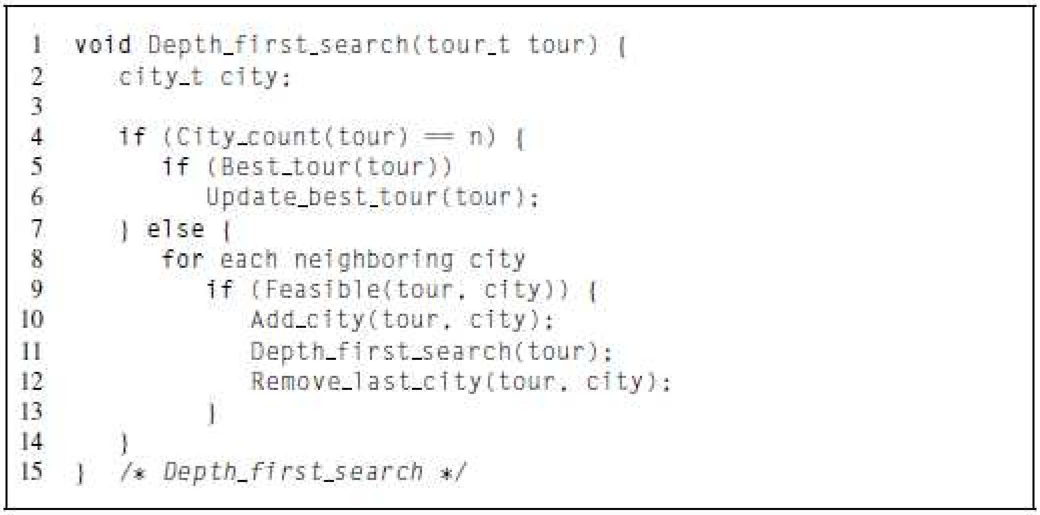
II.1 Recursive depth-first search

- Using depth-first search we can systematically visit each node of the tree that could possibly lead to a least-cost solution. The simplest formulation of depth-first search uses recursion (Program.4). Later on it will be useful to have a definite order in which the cities are visited in the for loop in Lines 8 to 13, so we'll assume that the cities are visited in order of increasing index, from city 1 to city n-1.
- 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.

- The function City count examines the partial tour tour to see if there are ***n*** cities on the partial tour. If there are, we know that we simply need to return to the hometown to complete the tour, and 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**. Note that before the first call to Depth first search, the 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 tour hasn't visited n cities, we can continue branching down in the tree by "expanding the current node," in other words, by trying to visit other
cities from the city last visited in the partial tour. To do this we simply loop through 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. If the city is feasible, we add it to the tour, and recursively call **Depth_ first _search** 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.



```
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, city);
         }
   }
}  /* Depth_first_search */
```

Program 4: Pseudocode for a recursive solution to TSP using depth-first search

Disadvantages :

- Since function calls are expensive, recursion can be slow.
- It also has the disadvantage that 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 the threads or processes.

2 .Non recursive depth-first search

- Recursive function calls can be implemented by pushing the current state of the recursive function onto the run-time stack. Thus, we can try to eliminate recursion by pushing necessary data on our own stack before branching deeper into the tree, and when we need to go back up the tree—either because we've reached a leaf or because we've found a node that can't lead to a better solution—we can pop the stack.
- In this version, a stack record consists of a single city, the city that will be added to the tour when its record is popped. In the recursive version we continue to make recursive calls until we've visited every node of the tree that corresponds to
  a feasible partial tour. At this point, the stack won't have any more activation records for calls to **Depth_first_search**, and we'll return to the function that made the original call to **Depth_first_search**.

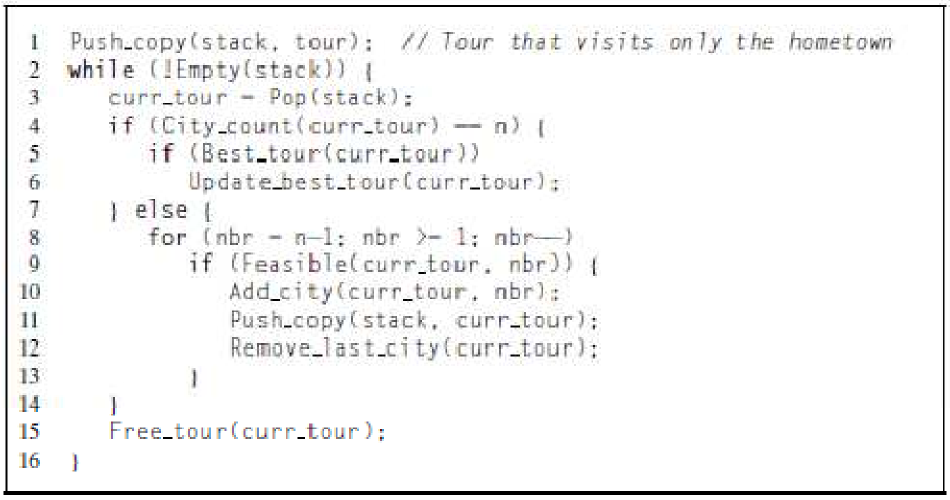
- The main control structure in our iterative version is the while loop extending from Line 3 to Line 20, and the loop termination condition is that our stack is empty. As long as the search needs to continue, we need to make sure the stack is nonempty, and, in the first two lines, we add each of the non-hometown cities. Note that this loop visits the cities in decreasing order, from n-1 down to 1.
- Also notice that in Line 5 we check whether the city we've popped is the constant

NO CITY. This constant is used so that we can tell when we've visited all of the children of a tree node; if we didn't use it, we wouldn't be able to tell when to back up in the tree. Thus, before pushing all of the children of a node (Lines 15–17), we push the NO CITY marker.

```
 1  for (city = n-1; city >= 1; city--)
 2     Push(stack, city);
 3  while (!Empty(stack)) {
 4     city = Pop(stack);
 5     if (city == NO_CITY) // End of child list, back up
 6        Remove_last_city(curr_tour);
 7     else {
 8        Add_city(curr_tour, city);
 9        if (City_count(curr_tour) == n) {
10           if (Best_tour(curr_tour))
11              Update_best_tour(curr_tour);
12           Remove_last_city(curr_tour);
13        } else {
14           Push(stack, NO_CITY);
15           for (nbr = n-1; nbr >= 1; nbr--)
16              if (Feasible(curr_tour, nbr))
17                 Push(stack, nbr);
18        }
19     }  /* if Feasible */
20  }  /* while !Empty */
```

Prog 5: Pseudocode for an implementation of a depth-first solution to TSP that doesn't use recursion

- An alternative to this iterative version uses partial tours as stack records (see Prog.6). This gives code that is closer to the recursive function. However, it also results in a slower version, since it's necessary for the function that pushes onto the stack to create a copy of the tour before actually pushing it on to the stack.
- The function Push copy push a pointer to the current tour onto the stack The extra memory required. Allocating storage for a new tour and copying the existing tour is time-consuming. The stack is more or less independent of the other data structures. Since entire tours are stored, multiple threads or processes can "help themselves" to tours, and, if this is done reasonably carefully.

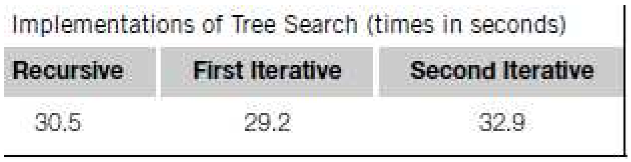


```
Push_copy(stack, tour);  // Tour that visits only the hometown
while (!Empty(stack)) {
   curr_tour = Pop(stack);
   if (City_count(curr_tour) == n) {
      if (Best_tour(curr_tour))
         Update_best_tour(curr_tour);
   } else {
      for (nbr = n-1; nbr >= 1; nbr--)
         if (Feasible(curr_tour,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

Fig 6: Pseudocode for a second solution to TSP that doesn't use recursion

1. Performance of the serial implementations

- The run-times of the three serial implementations are shown in Table.7. The input digraph contained 15 vertices (including the hometown), and all three algorithms visited approximately 95,000,000 tree nodes. The first iterative version is less than 5% faster than the recursive version, and the second iterative version is about 8% slower than the recursive version. As expected, the first iterative solution eliminates some of the overhead due to repeated function calls, while the second iterative solution is slower because of the repeated copying of tour data structures. However, as we'll see, the second iterative solution is relatively easy to parallelize



Implementations of Tree Search (times in seconds)

Recursive	First Iterative	Second Iterative
30.5	29.2	32.9

Table 7. Run-Times of the 3 Serial Implementations of Tree Search (times in sec)