COMP 3760: Algorithm Analysis and Design

Lesson 6: Exhaustive Search Algorithms

Rob Neilson

rneilson@bcit.ca

Homework and Reading

- This stuff is due in lab in the week of Sept29-Oct3…
  - Read chapters 7.1, 7.2, 7.3
- Homework…
  - Chapter 7.1, page 253, questions 2, 3, 4
  - Chapter 7.2, page 264, questions 2
  - Chapter 7.3, page 270, questions 1, 7

Introduction …

- Last week we looked at the linear search algorithm
- Now let's consider its performance on some more difficult problems…
- There are three classic computational problems that for which no known fast algorithm exists
  - assignment problem
  - traveling salesman problem
  - knapsack problem
- One thing that all these problems have in common is that they are *optimization problems* … that is … there is always some *objective function* that we are attempting to optimize (ie: find a max or a min value for)

Optimization vs Decision Problems

- An example of an optimization problem is:

  find the transit route from A to B that minimizes total travel time?

  – the answer would simply be some set of busses, transfer points, subway stops etc that minimizes the travel time

- Each optimization problem with have a corresponding decision problem, for instance:

  is there a transit route from A to B for which the total travel time is less than 30 minutes?

  – the answer would simply be yes or no

- Note that these types of problems can be thought of a search problems, as the solution involves searching through a large set of possibilities to find the best solution or to find out if a solution exists.

Classic Problem: Assignment Problem

- this is a classic optimization problem
- there are *n people* who need to be assigned *n jobs*, and there is a (possibly different) cost for each person to do each job
- the problem is to *find the combination of people and jobs* that has the minimum (or maximum) overall cost

- typically the costs are presented in a tabular form such as

	Job 1	Job 2	Job 3	Job 4
Person 1	9	2	7	8
Person 2	6	4	3	7
Person 3	5	8	1	8
Person 4	7	6	9	4

for example, this means that the cost is "8" for person 3 to do job 2

Assignment Problem (possible solution 1)

- To find a solution by Brute Force, we need to check every combination of assignments, as see what its total cost is, for example we could assign:

	Job 1	Job 2	Job 3	Job 4
Person 1	9	2	7	8
Person 2	6	4	3	7
Person 3	5	8	1	8
Person 4	7	6	9	4

- in this case the total cost is 9+4+1+4 = 18

Assignment Problem (possible solution 2)

- Another possible solution might be …

	Job 1	Job 2	Job 3	Job 4
Person 1	9	2	7	8
Person 2	6	4	3	7
Person 3	5	8	1	8
Person 4	7	6	9	4

in this case the total cost is 6+2+1+4 = 13

… and another possible solution might be …

	Job 1	Job 2	Job 3	Job 4
Person 1	9	2	7	8
Person 2	6	4	3	7
Person 3	5	8	1	8
Person 4	7	6	9	4

in this case the total cost is 5+2+3+4 = 14

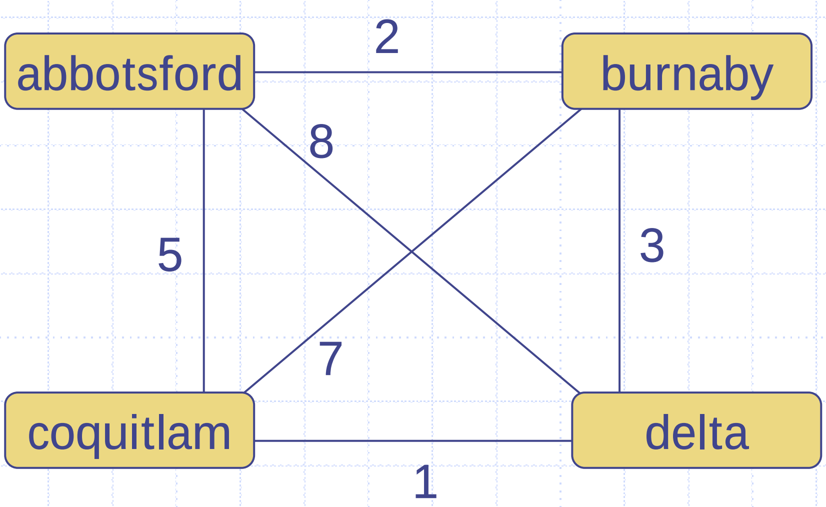
Assignment Problem (algorithm)

- to find the optimal solution, ie, the one that minimizes the objective function, we need to calculate the cost for each and every possible job assignment
- this means that we have to generate all possible permutations of the job assignments, and consider the cost of each one
- so an algorithm to solve this problem might look like this …

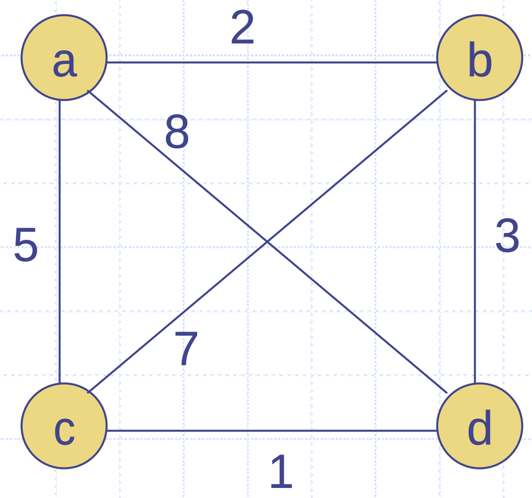
```
for each permutation P of job assignments
    totalcost ← sum of the job costs for P
    if totalcost < mincost
        mincost ← totalcost
        minperm ← P
return P
```

Classic Problem: Traveling Salesman

- A salesman needs to visit n cities. You know the distance between each city. Find the shortest route that visits each city exactly once and returns to the starting city.
- We typically model this problem as an weighted undirected graph, where vertices are cities, edges are roads, and edge weights are road lengths.



Traveling Salesman



- assume our sales dude starts at city a, then one possible solution would be …

  a→b→c→d→a

  … and the length of this route is

  L = 2+8+1+7 = 18

- Just like in the assignment problem, we are going to have to generate all the permutations (this time it is permutations of cities (vertices))

- Other possible solutions include …

a→b→d→c→a	**L = 2+3+1+5 = 11**
a→c→b→d→a	**L = 5+8+3+7 = 23**
a→c→d→b→a	**L = 5+1+3+2 = 11**
a→d→b→c→a	**L = 7+3+8+5 = 23**
a→d→b→c→a	**L = 7+1+8+2 = 18**

Traveling Salesman cont

- how many possible routes?
  - we know that all the permutations of n objects is **n!** – so if there are 4 cities we should get 4!=24 routes … but there clearly are not that many. Where did they go …
- remember that since we are always starting and ending at a specific city (eg: a), we only need to consider routes that start with ‘a’
  - ie: we would consider **a→b→d→c→a** but not **b→d→c→a→b**
  - this means there are only (n-1)! permutations to consider
- but we also notice that there are some duplicate routes, eg: **a→b→d→c→a** is the same as **a→c→d→b→a** (it is just reversed)
  - so we only consider one of them
- therefore the brute force solution requires that we generate and compute the length of (n-1)!/2 routes

Traveling Salesman (solution)

- our brute force solution to this problem will look very similar to the solution for the assignment problem …

```
for each permutation P of cities
        for each city i in P
                distance ← distance + weight(i,i+1)
        if distance < min
                min ← distance
                minroute ← P
return P
```

- We notice that both the assignment problem and TSP have worst case, average case, and best case performance of ***O(n!)***
- This means that only small instances of the problem can actually be solved in reasonable time.

Permutations

- to solve the previous problems we need an algorithm to generate all the permutations of a set of objects ...
- *pen a paper* method for n=3 {1,2,3}

step 1:	let the output set be first item	**1**
step 2:	insert next item into all possible positions of all items in output	**12** **21**
step 3:	repeat step 2 until nothing remains to insert	**123 132 312 213 231 321**

Johnson Trotter Algorithm

- the following algorithm is described on page 179 of your textbook

```
Initialize the first permutation with <1 <2 ... <n
    while there exists a mobile integer
        find the largest mobile integer k
        swap k and the adjacent integer it is looking at
        reverse the direction of all integers larger than k
```

- *a "mobile integer" is one that has a smaller integer adjacent to it in the direction it is moving*

- *we show the direction it is moving with an arrow …*

← ← ← ←
1 2 3 **4**

← ← ← ←
1 2 **4** 3

← ← ← ←
1 **4** 2 3

← ← ← ←
4 1 2 **3**

→ ← ← ←
4 1 3 2

Generating Permutations Exercise 1

use Johnson Trotter to list all the permutations for n=4

Lexicographic order

- Permutation f precedes a permutation g in the lexicographic order if: for the minimum value of k such that $f(k) \neq g(k)$, we have $f(k) < g(k)$.
  - ie: they are in ascending sorted order (alphabetical)

Question: does the Johnson-Trotter algorithm generate permutations in lexicographic order?

NO!

does the pen and paper algorithm generate permutations in lexicographic order?

NO!

- So let's consider a different algorithm that generates them in lexicographic order…

Permute()

```
// input is input set
// perm[] is output array
// L is current level
// N is number elements
```

```
Permute(input, perm, L, N)
   if (L > N)
       process the permutation perm // eg: return perm[]
   else
       for each i in input do
           perm[L]=i;
           Permute(input-{i}, perm, L+1, N)
       end for
   end if
end Permute;
```

```
      Permute({1..n}, perm, 1, n) // initial call
```

eg: if we want to get all permutations of {1,2,3} we call:

```
      Permute({1,2,3}, [-,-,-], 1, 3)
```

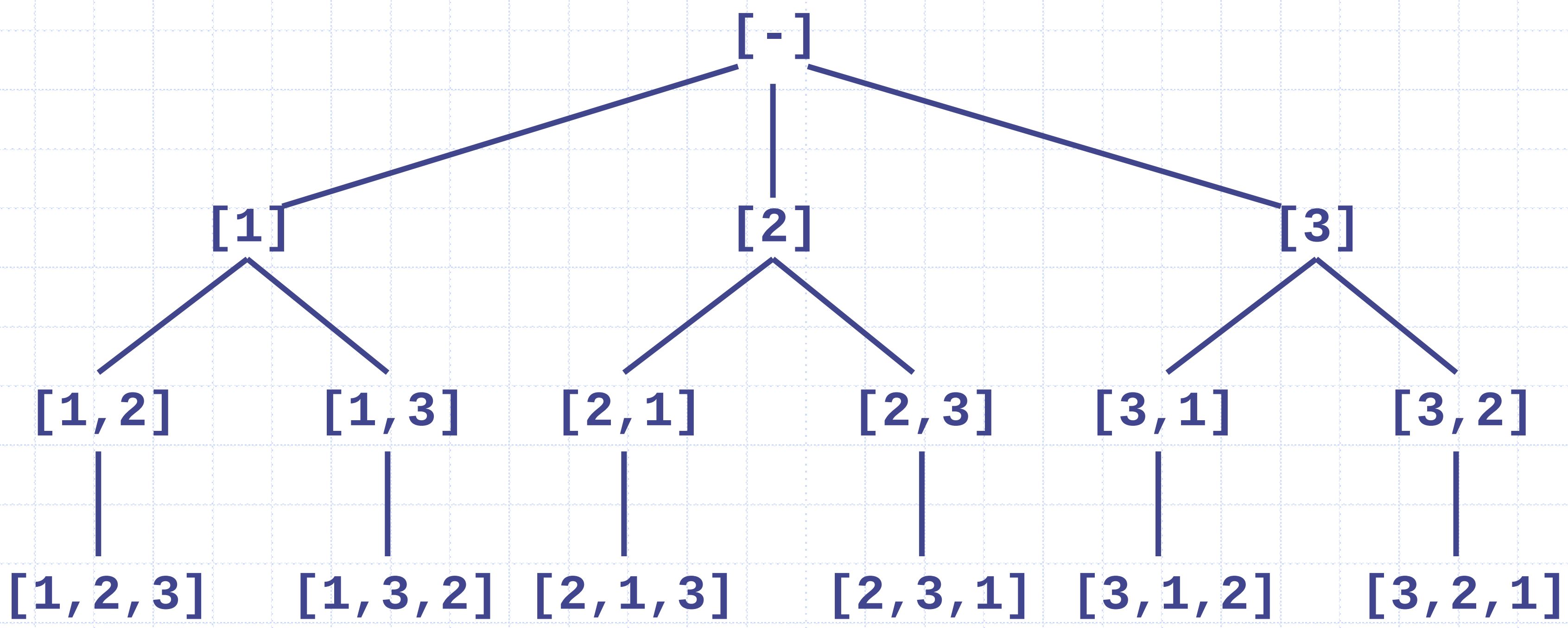
Generating Permutations Exercise 2

- use the preceding algorithm to generate the permutations for n=3
- be sure to show all recursive calls (with parameters) made by the algorithm

Permute Call Trace

```
Permute({1,2,3}, [-,-,-], 1, 3)
   1: Permute({2,3}, [1,-,-], 2, 3)
      1.1: Permute({3}, [1,2,-], 3,3)
           1.1.1: Permute({}, [1,2,3], 4, 3) --> 1,2,3
      1.2: Permute({2}, [1,3,-], 3,3)
           1.2.1: Permute({}, [1,3,2], 4, 3) --> 1,3,2
   2: Permute({1,3}, [2,-,-], 2, 3)
      2.1: Permute({3}, [2,1,-], 3,3)
           2.1.1: Permute({}, [2,1,3], 4, 3) --> 2,1,3
      2.2: Permute({1}, [2,3,-], 3,3)
           2.2.1: Permute({}, [2,3,1], 4, 3) --> 2,3,1
   3: Permute({1,2}, [3,-,-], 2, 3)
      3.1: Permute({2}, [3,1,-], 3,3)
           3.1.1: Permute({}, [3,1,2], 4, 3) --> 3,1,2
      3.2: Permute({1}, [3,2,-], 3,3)
           3.2.1: Permute({}, [3,2,1], 4, 3) --> 3,2,1
```

Tree of Generated Permutations



The End