Search Algorithms

IE 496 Lecture 17

Reading for This Lecture

- Primary
  - Horowitz and Sahni, Chapter 8

Basic Search Algorithms

Search Algorithms

- *Search algorithms* are fundamental techniques applied to solve a wide range of optimization problems.
- Generally speaking, search algorithms explore a graph in order to find a set of nodes or edges satisfying a particular property.
- Simple examples
  - Find nodes that are reachable by directed paths from a source node.
  - Find nodes that can reach a specific node along directed paths
  - Identify the connected components of a network
  - Identify directed cycles in network

Basic Graph Search Algorithm

- Node s is a given initial node
- $Q \leftarrow \{s\}$
- While $Q \neq \emptyset$
  - Let v be any member of Q
  - Remove v from Q
  - Mark v
  - For v' in $A(v)$
    - If v' is not marked
      - $Q \leftarrow Q \cup \{v\}$

Search Algorithms

- The search proceeds depending on how element *v* is selected in each iteration.
- *Q* is usually ordered in some way by storing it in an appropriate data structure.
  - If *Q* is a queue, we get FIFO ordering (traversal order?).
  - If *Q* is a stack, we get LIFO ordering (traversal order?).
- The efficiency of the algorithm can be affected by the ordering and the data structure used to maintain *Q*

Search Tree

- Associated with each search ordering is a *search tree* that can be used to visualize the algorithm.
- At the time a node v' is added to Q, we record v as its *predecesor*.
- The set of arcs formed by each node and its predecessor forms a tree rooted at s.

Complexity of Search Algorithms

- Initialization
- Maintaining the set *Q*.
  - What is the maximum number of additions and removals?
  - How many operations are required for each?
- Searching the adjacency lists.
  - How many times do we touch each element of each list?
  - How much work do we do each time we touch an element?
- In some cases, the adjacency lists are constructed dynamically and this step may also be expensive.
- The size of the graph may not be polynomial in the size of the input.

The Graph Search Paradigm

- The basic algorithm is a template for a whole class of algorithms.
- How can we use it to determine whether a graph is connected?
- What other algorithms that we've seen can be viewed as graph search algorithms?

Topological Ordering

- In a directed graph, the arcs can be thought of as representing *precedence constraints*.
- In other words, an arc (i,j) represents the constraint that node i must come before node j.
- Given a graph $G=(N,A)$ with the nodes labeled with 1 through n, let $order(i)$ be the label of node i.
- Then, this labeling is a *topological ordering* of the nodes if for every arc (i,j) in A, $order(i) < order(j)$.
- Can all graphs be topologically ordered?

Topological Ordering Algorithm

Analysis

- Correctness
  - If G has a cycle...
  - If G has no cycle.
- Running time

Advanced Search Algorithms

The Bin Packing Problem

- We are given a set of n items, each with a size/weight w_i
- We are also given a set of bins of capacity C.
- Bin Packing Problem: Pack the items into the smallest number of bins possible.
- The total size/weight of items assigned to each bin must not exceed the capacity C.
- This problem is NP-hard.

Heuristic Methods

- Heuristic methods derive an approximate solution quickly (usually polynomial time).
- Heuristics for the Bin Packing Problem.
- Performance guarantees.

Integer Knapsack Problem

- We are given n objects.
- Each object has a weight w_i and a profit p_i.
- We also have a knapsack with capacity M.
- Objective: Fill the knapsack as profitably as possible.
- We do not allow fractional objects.
- This is an *NP*-hard problem.

Exact Solution Method

- We cannot hope for a polynomial-time algorithm for this problem.
- How do we solve it?

- What is the complexity?

Heuristic Methods

- Heuristic methods derive an approximate solution quickly (usually polynomial time).
- Heuristic for the Knapsack Problem.
- Performance guarantees.

Branch and Bound Methods

- *Branch and Bound* is a general method that can be used to solve many NP-complete problems.
- Components of Branch and Bound Algorithms
  - Definition of the state space.
  - Branching operation.
  - Feasibility checking operation.
  - Bounding operation.
  - Search order.

Definition of the State Space

- To apply branch and bound, the solution must be expressible as an *n-tuple* $(x_1, x_2, \ldots, x_n)$ where x_i is chosen from a finite set S_i.
- A set of all possible *n-tuples* is the state space S.
- Knapsack Problem
- Bin Packing Problem

Decisions, Feasibility, Optimization

- Feasibility problems
  - A defined subset of the state space contains the "feasible" elements.
  - There are various ways to define "feasibility".
  - The goal is to find one feasible element of the state space.
- Optimization problems
  - We are also given an *objective function* f which assigns a cost to each element of the state space.
  - We would like to find a feasible state with the lowest cost.
- Decision problems

Branching Operation

- Operation by which the original state space is partitioned into at least two non-empty *subproblems*.
- Typical branching operation
  - Pick an element i of the n-tuple.
  - Generate $|S_i|$ subproblems by setting x_i to each of its possible values in succession.
- Knapsack
- Bin Packing

Feasibility Checking Operation

- Given a subproblem, we need to check whether it contains any feasible solutions.
- This may or may not be possible for partially defined states.
- It must be possible if the state is fully defined.
- Knapsack Problem
- Bin Packing Problem

Bounding Operation

- If applicable, we want upper and lower bounds on the optimal value of the current subproblem.
- This may not be possible.
- *Upper bounds* generally come from finding a feasible solution.
- Upper bounds are global
- *Lower bounds* can come from a number of sources.
- Knapsack
- Bin Packing

Basic Branch and Bound Algorithm

BBound (S, U)

S = {s: s is a feasible state}, U = current upper bound

```
if (FEASIBLE(S) == FALSE) return(∞);
if (LBOUND(S) >= U)  return(∞);
if (UBOUND(S) < U) U = UBOUND(S);
if (LBOUND(S) < U)
    BRANCH(S) -> S_1, . . . , S_k;
    for (i = 0; i < k; i++)
      if (BB(U, S_i) < U) U = BB(S_i);
return(U);
```

More Generally

- Associate branch and bound with a search tree.
- Maintain a priority queue of candidate subproblems.
- Iterate
  - Pick a subproblem from the queue and process it.
    - Check feasibility.
    - Perform upper and lower bound.
  - Prune if infeasible or lower bound greater than or equal to upper bound.
  - Branch.
  - Add new subproblems to the queue.

Search Strategies

- Depth First
- Breadth First
- Highest Lower Bound
- Lowest Lower Bound

The Traveling Salesman Problem