Exact Solutions to NP-Complete Problems

Ref: - “Computer Algorithms”, Horowitz, Sahni, Rajasekaran (Chapters 7, 8)
- Various texts on Combinatorial Algorithms or on Integer Linear Programming

Backtracking

• An organized exhaustive search which often avoids searching many possibilities
• The desired solution is often expressed as n-tuple, where the \( x_i \)'s are chosen from some finite set \( S_i \) with \( m_i = |S_i| \).
• The problem often requires finding one vector which maximizes, minimizes or satisfies a criterion function \( P(x_1, x_2, \ldots, x_n) \).
• The brute force approach is to evaluate each of \( m = m_1 m_2 \ldots m_n \) n-tuples from \( S_1 \times S_2 \times \ldots \times S_n \) and identify the n-tuple yielding the optimal value.
• The basis idea of backtracking is to build the solution vector using modified criterion function \( P(x_1, x_2, \ldots, x_n) \) to test whether the vector being formed have any chance of success.
• If a partial vector \( (a_1, a_2, \ldots, a_i) \) has no chance of success, we avoid considering all of the \( m_1 m_2 \ldots m_n \) possible test vectors \( (a_1, a_2, \ldots, a_i, x_{i+1}, \ldots, x_n) \)
• Many problems solved by backtracking satisfy a set of constraints which may divided into two categories: explicit and implicit.

Explicit constraints are rules that restrict each \( x_i \) to take on values only from a given set

- examples: \( x_i \geq 0, \ x_i = 0 \ or \ 1, \ 1 \leq x_i \leq u_i \)
- depend on the particular instance \( I \) of the problem being solved
- the n-tuples that satisfy these conditions define a possible solution space for \( I \).

Implicit constraints are rules which the tuples in the solution space for \( I \) must satisfy in order to satisfy the criterion function.

Example (8 Queens Problem)

• A classic problem in combinatorics is to place 8 queens on an 8 by 8 chessboard so that no two can “attack” each other (along a row, column, or diagonal).
• Since each queen (1-8) must be on a different row, we can assume queen \( i \) is on row \( i \).
• All solutions to the 8-queens problem can be represented as an 8-tuple \( (x_1, x_2, \ldots, x_8) \) where queen \( i \) is on column \( x_i \).
• The explicit constraints are \( S_i = \{1, 2, \ldots, 8\}, \ 1 \leq i \leq 8 \). The solution space consists of \( 8^8 \) 8-tuples.
• The implicit constraints are that no two \( x_i \)'s can be the same (as queens must be on different columns) and no two queens can be on the same diagonal.
  • this implies that all solutions are permutations of the 8-tuple \( (1, 2, \ldots, 8) \), and reduces the solution space from \( 8! \) tuples to \( 8! \) tuples.
• Backtracking algorithms determine problem solutions by systematically searching the solution space.
• Search is facilitated using a tree organization for the solution space.
• Many tree organizations may be possible for the same solution space.

• Example (n-Queens): n queens are placed on an n by n chessboard so that no two
  attack (no two queens are on the same row, column, or diagonal).
• Generalizing earlier discussion, solution space contains all n! permutations of (1,2,…,n).
• The tree below shows possible organization for n=4.
• Tree is called a permutation tree (nodes are numbered as in depth first search).
• Edges labeled by possible values of \( x_i \).
• The solution space is all paths from the root node to a leaf node.
• There are 4!=24 leaf nodes in tree.

• Example (Sum of Subsets): Given positive numbers \( w_i \), 1 ≤ i ≤ n, and m, find
  all subsets of \( \{w_1, w_2, \ldots, w_n\} \), whose sum is m.
• If n=4, \( \{w_1, w_2, w_3, w_4\} = \{11,13,24,7\} \) and m=31, the desired solution sets are
  \( (11,13,7) \) and \( (24,7) \).
• If the solution vectors are given using the indices of the \( w_i \) values used, then the
  solution vectors are (1,2,4) and (3,4).
• In general, all solutions are \( k \)-tuples \((x_i, x_j, \ldots, x_k)\) with 1 ≤ k ≤ n and different
  solutions may have different values of k.
• The explicit constraints on the solution space are that each \( x_i \in \{1,2,\ldots,n\} \).
• The implicit constraints are that \( x_i < x_{i+1}, 1 ≤ i < n \), (so each item will occur only once) and that the sum of the corresponding \( w_i \) ’s be m.

• The next figure gives the tree that corresponds to this variable tuple formation.
• An edge from a level i node to a level i+1 node represents a value for \( x_i \).
• The solution space is all paths from the root node to any node in the tree.
• Possible paths include empty path, (1), (1,2), (1,2,3), (1,2,3,4), (1,2,4), (1,3,4), …
• The leftmost subtree gives all subsets containing \( w_1 \), the next subtree gives all
  subsets containing \( w_2 \) but not \( w_1 \), etc.
Example (Sum of Subsets) again: Another formulation of this problem represents each solution by an $n$-tuple $(x_1, x_2, ..., x_n)$ with $x_i \in \{0,1\}, 1 \leq i \leq n$.

- Here $x_i = 0$ if $w_i$ is not chosen and $x_i = 1$ if $w_i$ is chosen.
- Given the earlier instance of $(11,13,24,7)$ and $m=31$, the solutions $(11,13,7)$ and $(24,7)$ are represented by $(1,1,0,1)$ and $(0,0,1,1)$.
- Here, all solutions have a fixed tuple size. The tree below corresponds to this formulation (nodes are numbered as in $D$-search).

- Edge from a level $i$ node to a level $i+1$ node is labeled with the value of $x_i$ (0 or 1)
- All paths from the root to a leaf give solution space.
- The left subtree gives all subsets containing $w_i$ and the right subtree gives all subsets not containing $w_i$.

Generating Problem States

- The two tree organizations for the sum of subsets problem are static trees (tree organization is independent of the problem instance being solved).
- Tree organizations that are problem instance dependent are called dynamic trees and are also used for some problems.
- Once a state space tree organization has been selected for a problem, the problem can be solved by
  - systematically generating the problem states,
  - finding which of these are solution states
  - finding which solution state are answer states
- A live node is a node which has been generated but whose children have not all been generated.
- An $E$-node (i.e., expanding node) is a live node whose children are currently being generated.
- A dead node is a generated node which is not to be expanded further or all of whose children have been generated.
- Two ways to generate problem states:
  - Breadth First Generation (queue of live nodes)
  - Depth First Generation (stack of live nodes)
• **Depth First** Generation (stack of live nodes)
  • When a new child C of the current E-node R is generated, this child becomes the new E-node.
  • Then R will become the new E-node again when the subtree C has been fully explored.
  • Corresponds to a depth first search of the problem states.

• **Breadth First** Generation (queue of live nodes)
  • The E-node remains the E-node until it is dead.

*Bounding functions* are used in both to kill live nodes without generating all of their children.
• At the end of the process, an answer node (or all answer nodes) are generated.
• The depth search generation method with bounding function is called **backtracking**.
• The breadth first generation method is used in the branch-and-bound method.

**Example (backtracking on 4-queens problem)**

• As a bounding function, use criterion that if \((x_1, x_2, ..., x_7)\) is the path to the current E-node, then some continuation \((x_1, x_2, ..., x_7, x_8, ..., x_n)\) exists that represents a chessboard without 2 queens attacking.

• Start with the root as the E-node. Then the path is ()

• The children of the E-nodes are generated in a left to right order.

• Node 2 is generated first and the path becomes (1). This corresponds to placing queen 1 on column 1.

---

**Example (backtracking on 4-queens problem)**

• Node 2 becomes the E-node, as no two queens are attacking.
• Node 3 is generated and is immediately killed, as queens 1 and 2 would be on a diagonal.
• Node 8 is generated and the path (1,3) is ok, so node 8 becomes the next E-node.
• Node 8 gets killed since none of its children can lead to a feasible chessboard.
• Backtrack to node 2 and generate another child, node 13, giving path (1,4) which is ok.
• The first child of node 13 is node 14, which gives path (1,4,2) and the feasible chessboard.
• This process continues, as indicated in the figure. The figure shows the portion of the state space tree we had on slide #3.
• Note that only 16 out of 65 nodes (or 25%) in the solution space are actually generated.
General Backtracking Algorithm

- This algorithm will find all answer nodes.
- If only the first solution is desired, a “flag” parameter can be added to indicate first success.
- Let \((x_1, x_2, ..., x_i)\) be a path from the root to a node in the state space tree.
- Let \(T(x_1, x_2, ..., x_i)\) be the set of all possible values for \(x_{i+1}\) such that \((x_1, x_2, ..., x_i, x_{i+1})\) is also a path to a problem state (i.e., node).
- Let \(B_i\) be a boundary (Boolean) function such that if \(B_i(x_1, x_2, ..., x_i)\) is false, then the path \((x_1, x_2, ..., x_i)\) cannot be extended to reach an answer node.
- Note that if \(B_i(x_1, x_2, ..., x_i) = 1\), this does not guarantee that the path \((x_1, x_2, ..., x_i)\) can be extended to reach an answer node.
- Here is the recursive backtracking algorithm:

```
Algorithm Backtrack(k)
for (each x[k] from T(x[1], x[2], ..., x[k-1]) do
  if (B_i(x[1], x[2], ..., x[k]) != 0) then
    if (x[1], x[2], ..., x[k]) is a path to an answer node
      then write (x[1..k]);
    if (k<n) then Backtrack(k+1);
  
}
```

Comments
- the candidates for \(x[i+1]\) are values generated by \(T(x[1], x[2], ..., x[i])\) that satisfy \(B_{i+1}\)
- \(T()\) gives all candidates for \(x[1]\),
- elements are generated in a depth first manner, creating a preorder traversal (except for eliminated branches) of the state space tree.
- for many problems, the size of the state space tree is too large to permit generation of all nodes.

Efficiency of Backtracking

- The efficiency of a backtracking algorithm depends upon 4 factors
  - the time to generate the next \(x[k]\)
  - the number of \(x[k]\) choices that satisfy the explicit constraints
  - the time required to evaluate the bounding function \(B_i\)
  - the number of \(x[k]\) satisfying the \(B_i\)
- A good boundary function will drastically reduce the number of candidates that have to be considered.
- Often a tradeoff between bounding functions, as one that is good may take more time to evaluate.
- For many problems such as \(n\)-queens, no good bounding function are known.
- Rearrangement:
  - the principle of selecting the set \(S\) with fewest elements each time
  - since these sets can be taken in any order, smaller branching at the higher levels create larger subtrees
  - removal of early nodes cut off larger subtrees (see Fig. 7.7 in HSR)
- The first three factors that effect the time required for backtracking depend primarily on the state space tree organization selected
- Only the fourth factor may vary widely, depending on the problem instances.
• Worst case predictions for backtracking algorithms:
  • If the number of points in the solution space is \(2^n\) or \(n!\) the worst case timing is usually either \(O(p(n)2^n)\) or \(O(p(n)n!)\), where \(p(n)\) is a polynomial.
  • Backtracking can often solve some problem instances with large \(n\) in very small amounts of time. However, may be difficult to predict behavior of algorithm for particular problem instances.

• Estimating Nr. of nodes generated:
  • The number of nodes generated in a particular instance can be estimated using Monte Carlo methods
  • Starting at the top level, a random path is generated, as follows:
    • set \(x\) be a node on this path at level \(i\) of the state space tree; the boundary function \(B_i\) is used to determine the number \(m_i\) of its children which will be generated; one child is randomly selected, and the process continue until the path ends.
    • Then \(m = m_1 + m_1m_2 + m_1m_2m_3 + \ldots\) is an estimate of the nodes that will be generated.
  • Above estimate for \(m\) assumes the bounding functions are static and do not improve with time; It also assumes that the same boundary function is used for all nodes at the same level.
  • The above two assumptions are not true for most backtracking algorithms; e.g., the boundary functions usually get stronger as information is gathered about the search.
  • Consequently, the above value of \(m\) is likely to be high when these two assumptions are false.
  • A better estimate would also result if the value \(m\) is the average returned for several (about 20) random paths.

Backtracking Algorithm for n-Queens problem
• Let \((x_1, x_2, \ldots, x_n)\) represent where the \(i\)th queen is placed (in row \(i\) and column \(x_i\)) on an \(n\) by \(n\) chessboard.
• Observe that two queens on the same diagonal that runs from “upper left” to “lower right” have the same “row-column” value.
• Also two queens on the same diagonal from “upper-right” to “lower left” have the same “row+column” value.
• Then two queens at \((i,j)\) and \((k,l)\) are on the same diagonal if and only if
  \[
  i-j = k-l \text{ or } i+j = k+l
  \]
  iff
  \[
  i-k = j-l \text{ or } j-l = k-i
  \]
  iff
  \[
  |j-l| = |i-k|
  \]
• Algorithm PLACE\((k,i)\) returns true if the \(k\)th queen can be placed in column \(i\) and runs in \(O(k)\) time (see next slide)
• Using PLACE, the recursive version of the general backtracking method can be used to give a precise solution to the n-queens problem.
• Array \(x[i/j]\) is global. Algorithm invoked by NQUEENS\((1,n)\).
bool Place(int k, int i)
// Returns true if a queen can be placed in kth row and ith column. Otherwise it returns false.
// x[j] is a global array whose first (k-1) values have been set.
// abs(r) returns the absolute value of r.
{
    for (int j=1; j<k; j++)
        if ((x[j] == i) || (abs(x[j]-i) == abs(j-k))) // Two in the same column or in the same diagonal
            return(false);
    return(true);
}

void NQueens(int k, int n)
// Using backtracking, this procedure prints all possible placements of n queens on an n x n chessboard so that they are nonattacking.
{
    for (int i=1; i<=n; i++)
        if (Place(k, i)) {
            x[k] = i;
            if (k==n) { for (int j=1;j<=n;j++) cout << x[j] << ' '; cout << endl;}
            else NQueens(k+1, n);
        }
}

---

**Efficiency of n-Queens over Brute Force**

- For an 8x8 chessboard, there are $\left(\begin{array}{c}64\\8\end{array}\right)$ ways to place 8 queens on the chessboard (billions of 8-tuples to examine).

- Requiring placement of queens on distinct rows and columns reduces the number of 8-tuples that must be examined to $8! = 40,320$ 8-tuples.

- Next we estimate the number of nodes that will be generated by NQUEENS. The assumptions needed for this estimate hold for NQueens.
  - Boundary function is static
  - boundary functions does not change as search progress
  - additionally, nodes on the same level of the tree have the same degree

- Five trials using the ESTIMATE function described earlier are given on p.355 of HSR
  - each produces a random path and estimates the total number of nodes generated, based on this path
  - the average of these 5 estimates is 1625
  - the total number of nodes in the 8-queens state space tree is
    $$1 + \sum_{j=0}^{7} \left[ \prod_{i=0}^{j} (8-i) \right] = 69,281$$
  - the estimated number of unbound nodes is only 2.34 % of the total number of nodes in the 8-queens state space tree.
**Sum of Subsets Algorithm Overview**

- **Problem Restated**: Given $n$ distinct positive integers (called weights), find all combinations of these numbers whose sum is $m$.

- We use the state space tree based on the fixed tuple length $(w_1, w_2, \ldots, w_k)$ where $x_i = 0$ if $w_i$ is not included and $x_i = 1$ if $w_i$ is included (see Fig. on slide #5).

- The weights $(w_1, w_2, \ldots, w_k)$ are assumed to initially be sorted into increasing order.

- Note that the tree node corresponding to $(x_1, x_2, \ldots, x_k)$ cannot lead to an answer node unless

\[
\sum_{i=1}^{k} w_i x_i + \sum_{i=k+1}^{n} w_i \geq m
\]

- Also, note that $(x_1, x_2, \ldots, x_k)$ cannot lead to an answer node unless

\[
\sum_{i=1}^{k} w_i x_i + w_{k+1} \leq m
\]

- The boundary function used uses both of the preceding conditions:

\[
B_s (x_1, x_2, \ldots, x_k) = true \iff (1) \text{ and } (2) \text{ hold}
\]

- The algorithm for the sum of subsets problem given in HSR is obtained by using this boundary function in the general recursive backtracking algorithm.

- A couple of implementation simplifications used are explained in HSR (see pp. 358-9).

---

**Hamiltonian Cycles Algorithm**

- Example: Graph $G1$ contains a Hamiltonian cycle 1,2,8,7,6,5,4,3,1 while graph $G2$ contains no Hamiltonian cycle.

- The algorithm given works on both directed and undirected graphs

- All distinct cycles will be found.

- Each $x_i$ in the backtracking solution vector $(x_1, x_2, \ldots, x_n)$ represents the $i$th vertex visited in the proposed cycle

- The vertices of the graph are assumed to be named using the first $n$ positive integers.

- To avoid printing the same cycle $n$ times, we require $x_1 = 1$

- The algorithm *NextValue* determines a possible next vertex for the proposed cycle

  - if $1 < k < n$, then $x_k$ can be any vertex $v$ that is distinct from $x_1, x_2, \ldots, x_{k-1}$ and is connected by an edge to $x_{k-1}$

  - the vertex $x_k$ must be one remaining vertex and must be connected by an edge to both $x_i$ and $x_{k-1}$

- The backtracking algorithm *Hamiltonian(k)* is obtained by using *NextValue* to select a legal vertex to add. No boundary function is used.
The main algorithm starts by

- initializing the adjacency matrix \( G[1:n,1:n] \)
- setting \( x[2:n]=0 \)
- setting \( x[1]=1 \)
- executing \( Hamiltonian(2) \)

Recursive algorithm that finds all Hamiltonian cycles

```cpp
def Hamiltonian(int k):
    # This program uses the recursive formulation of
    # backtracking to find all the Hamiltonian cycles
    # of a graph. The graph is stored as an adjacency
    # matrix \( G[1:n][1:n] \). All cycles begin at node 1.
    #
    # Generate values for \( x[k] \).
    # Assign a legal next value to \( x[k] \).
    # Return if \( x[k] \) is legal.
    # Returns \( 0 \) if \( k == n \) and \( G[x[n]][x[1]] \) is true.
    #
    do { // Generate values for \( x[k] \).
        NextValue(k); // Assign a legal next value to \( x[k] \).
        if (!x[k]) return;
        if (k == n) {
            for (int i=1; i<=n; i++)
                cout << x[i] << ' ';
            cout << 1 << n << '
';
        }
    } while (1);
```

```cpp
def NextValue(int k):
    // \( x[1],...,x[k-1] \) is a path of \( k-1 \) distinct vertices.
    // If \( x[k] = 0 \), then no vertex has as yet
    // been assigned to \( x[k] \). After execution \( x[k] \) is assigned
    // to the lowest numbered vertex which
    // i) does not already appear in \( x[1],x[2],...,x[k-1] \); and
    // ii) is connected by an edge to \( x[k-1] \).
    // Otherwise \( x[k] = 0 \).
    // If \( k == n \), then in addition \( x[k] \) is connected to \( x[1] \).
    #
    do {
        x[k] = (x[k]+1) % (n+1); // Next vertex
        if (!x[k]) return;
        # Is there an edge?
        if (G[x[k-1]][x[k]]) { // Is there an edge?
            for (int j=1; j<k-1; j++)
                if (x[j]==x[k]) break;
                // Check for distinctness.
            if (j==k) // If true, then the vertex is distinct.
                if (((k<n) && ((k==n) & G[x[n]][x[1]]))
                    return;
            }
    } while(1);
```

Homework problem:
Generalize \( Hamiltonian \) so that it processes a graph whose edges have cost associated with them and finds a Hamiltonian cycle with minimum cost. You can assume that all edge costs are positive (Ex. 3, p. 368 of HS).
**Branch and Bound Algorithms**

- Refers to all state space search methods in which all children of the E-nodes are generated before any other live nodes become E-nodes.

- **Breadth Fist Search** (BFS) will be called FIFO (first in, first out)
  - each new node is placed in a queue
  - after all children of current E-node are generated, the node at front of queue become the E-node.

- **D-search** will be called a LIFO (last in, first out)
  - new nodes are placed in stack

As with backtracking, bounding functions will be used to avoid generating trees with no answer node.

- **Example**: 4 Queen FIFO Branch & Bound Algorithm.
  - The state space tree in Fig. on slide #3 is used, so node numbers do not indicate order of generation.
  - Initially, only the root node is alive (no queens placed)
  - Expanding the root E-node generates its children nodes in the order 2,18,34, and 50. These nodes represent a 4x4 chessboard with queen 1 in row 1 and columns 1, 2, 3, and 4 respectively.
  - The only live nodes are now 1, 18, 34, 50, and the next E-node is 2. It is expanded, generating nodes 3, 8, 13.

- Node 3 is killed immediately by the bounding function used in backtracking algorithm and nodes 8, 13 are added to queue of live nodes.
- This process is continued, generating below figure.
- Comparing the two trees of generated nodes, it is clear that backtracking is more efficient on this problem.

**Least Cost (LC) search**

- Both FIFO and LIFO are rigid and blind.
- The search for an answer node can often be speeded up using an “intelligent” ranking function $C(j)$ for the live nodes.
Least Cost (LC) search

- In 4-Queens example, if \( C(\ ) \) had assigned node 30 a better rank than other live nodes, it would have become the E-node following node 29.
- An ideal way to assign rank to each live node \( x \) is the number of levels the nearest answer node (in the subtree with root \( x \)) is from \( x \).
- Using this ranking function on the previous 4-queens example would have assigned rank 1 to both answer nodes 30 and 38.
- Let \( g(x) \) be an estimate of the additional effort needed to reach an answer node from \( x \).
- Node \( x \) is assigned a rank using a function \( C(\ ) \) defined by \( C(x) = f(h(x)) + g(x) \), where \( f \) is a non-decreasing weight function.
- The use of nonzero \( f \) helps prevent the search algorithm from making unnecessarily deep probes into the search tree.
- A nonzero \( f \) is needed, as otherwise a child \( y \) of the current E-node \( x \) will become the next E-node since \( g(y) \leq g(x) \) and \( x \) had the previous lowest rank.
- Use of non-zero \( f \) forces the search algorithm to favor nodes closer to the root, reducing the probability of a deep and fruitless search into the tree.
- A search that uses a cost function \( C(\) to choose the next E-node to be a live node with minimum \( C(\) value is called a LC-search.

Least Cost Search Algorithm

- This algorithm assumes two additional algorithms, \( \text{Least}(\) and \( \text{Add}(\) to manage the list of live nodes.
- \( \text{Least}(\) finds a live node with least \( C(\) value. This node is deleted from the list of live nodes and returned.
- \( \text{Add}(\) adds the new live node \( x \) to the list of live nodes.
- With each node \( x \) that becomes alive, we associate a field \( \text{parent} \) which stores the parent of \( x \).

If \( g = 0, f = 1 \), and \( h(x) \) is the level of node \( x \), this LC-search is a BFS algorithm which generates nodes by levels.

If \( f = 0 \) and \( g(y) < g(x) \) when \( y \) is a child of \( x \), this LC-search is a D-search.

The cost function \( C(\) is defined as follows

- if \( x \) is an answer node, then \( c(x) \) is the cost of reaching \( x \) from the root of the state space tree.
- if \( x \) is not an answer node, but the subtree of \( x \) contains an answer node, then \( c(x) \) is the minimal cost of an answer node in subtree \( x \).
- otherwise, \( c(x) = \infty \)

Then, \( C(\) with \( f = 1 \), that is, \( C(x) = h(x) + g(x) \), is an estimate of \( C(\) .

\( C(\) should be chosen so that it is easy to compute. It will normally have the additional property that if \( x \) is an answer node or leaf node, then \( c(x) = C(x) \).
Least Cost Search Algorithm

• This allows LC-search to output a path from the answer node it finds to the root node.
• LC-search terminates only when either an answer node is found or the entire state space tree has been generated and searched.
• Note that termination is only guaranteed for finite space trees.
• It is advisable to restrict the search in LC-search to find answer nodes with costs not exceeding a given bound \( C \).
• Note: Least and Add can be defined to implement a stack or queue as well, so the algorithms for LC, FIFO, and LIFO search are essentially the same.

```c
struct listnode {struct listnode *next, *parent; float cost;};
LCSearch(struct listnode *t) // Search t for an answer node.
{
    struct listnode *x, *E, *Least();
    if (*t is an answer node) output *t and return;
    E=t; // E-node
    initialize the list of live nodes to be empty;
    do { for (each child x of E) { if (x is an answer node) output the path from x to t and return;
        Add(x); // x is a new live node.
        x->parent = E; // pointer for path to root
    }
    if there are no more live nodes) {count << "No answer node\n"; return; }
    E=Least();
    } while(1);
}
```

Dynamic Programming

• This is another technique for finding exact solutions to NP-Complete problems.
• Examples:
  • 0/1 Knapsack Problem (your old homework problem; \( O(nW) \) time algorithm)
  • Traveling Salesman Problem (HSR p. 298)


Homework 7:
• Problems:
  • no more problems at this moment