Branch and Bound

Terminology

Definition 1 ***Live node*** *is a node that has been generated but whose children have not yet been generated.*

Definition 2 ***E-node*** *is a live node whose children are currently being explored. In other words, an E-node is a node currently being expanded.*

Definition 3 ***Dead node*** *is a generated node that is not to be expanded or explored any further. All children of a dead node have already been expanded.*

Definition 4 ***Branch-and-bound*** *refers to all state space search methods in which all children of an E-node are generated before any other live node can become the E-node.*

- Used for state space search
  - In BFS, exploration of a new node cannot begin until the node currently being explored is fully explored

General method

- Both BFS and DFS generalize to branch-and-bound strategies
  - BFS is an FIFO search in terms of live nodes
    - List of live nodes is a queue
  - DFS is an LIFO search in terms of live nodes
    - List of live nodes is a stack
- Just like backtracking, we will use bounding functions to avoid generating subtrees that do not contain an answer node
- Example: 4-queens
  - FIFO branch-and-bound algorithm
    - Initially, there is only one live node; no queen has been placed on the chessboard
    - The only live node becomes E-node
    - Expand and generate all its children; children being a queen in column 1, 2, 3, and 4 of row 1 (only live nodes left)
    - Next E-node is the node with queen in row 1 and column 1
    - Expand this node, and add the possible nodes to the queue of live nodes
    - Bound the nodes that become dead nodes
  - Compare with backtracking algorithm
    - Backtracking is superior method for this search problem
- Least Cost (LC) search
  - Selection rule does not give preference to nodes that will lead to answer quickly but just queues those behind the current live nodes
    - In 4-queen problem, if three queens have been placed on the board, it is obvious that the answer may be reached in one more move
    - The rigid selection rule requires that other live nodes be expanded and then, the current node be tested
  - Rank the live nodes by using a heuristic $\hat{c}(\cdot)$

- The next E-node is selected on the basis of this ranking function
- Heuristic is based on the expected additional computational effort (cost) to reach a solution from the current live node
- For any node x, the cost could be given by
  1. Number of nodes in subtree x that need to be generated before an answer can be reached
     * Search will always generate the minimum number of nodes
  2. Number of levels to the nearest answer node in the subtree x
     * $\hat{c}$(root) for 4-queen problem is 4
     * The only nodes to become E-nodes are the nodes on the path from the root to the nearest answer node
- Problem with the above techniques to compute the cost at node x is that they involve the search of the subtree at x implying the exploration of the subtree
  * By the time the cost of a node is determined, that subtree has been searched and there is no need to explore x again
  * Above can be avoided by using an estimate function $\hat{g}(\cdot)$ instead of actually expanding the nodes
- Let $\hat{g}(x)$ be an estimate of the additional effort needed to reach an answer from node x
  * x is assigned a rank using a function $\hat{c}(\cdot)$ such that

    $$\hat{c}(x) = f(h(x)) + \hat{g}(x)$$

    where $h(x)$ is the cost of reaching x from root and $f(\cdot)$ is any nondecreasing function
  * The effort spent in reaching the live node cannot be reduced and all we are concerned with at this point is to minimize the effort reaching the solution from the live node
    · This is faulty logic
    · Using $f(\cdot) \equiv 0$ biases search algorithm to make deep probes into the search tree
    · Note that we would expect $\hat{g}(y) \leq \hat{g}(x)$ for y which is a child of x
    · Following x, y becomes E node; then a child of y becomes E node, and so on, until a subtree is fully searched
    · Subtrees of nodes in other children of x are not considered until y is fully explored
    · But $\hat{g}(x)$ is only an estimate of the true cost
    · It is possible that for two nodes w and z, $\hat{g}(w) < \hat{g}(z)$ and z is much closer to answer than w
    · By using $f(\cdot) \not\equiv 0$, we can force the search algorithm to favor a node z close to the root over node w, reducing the possiblity of deep and fruitless search
- A search strategy that uses a cost function $\hat{c}(x) = f(h(x)) + \hat{g}(x)$ to select the next E-node would always choose for its next E-node a live node with least $\hat{c}(\cdot)$
  * Such a search strategy is called an LC-search (Least Cost search)
  * Both BFS and DFS are special cases of LC-search
  * In BFS, we use $\hat{g}(x) \equiv 0$ and $f(h(x))$ as the level of node x
    · LC search generates nodes by level
  * In DFS, we use $f(h(x)) \equiv 0$ and $\hat{g}(x) \geq \hat{g}(y)$ whenever y is a child of x
- An LC-search coupled with bounding functions is called an LC branch-and-bound search
- Cost function
  * If x is an answer node, $c(x)$ is the cost of reaching x from the root of state space tree
  * If x is not an answer node, $c(x) = \infty$, provided the subtree x contains no answer node
  * If subtree x contains an answer node, $c(x)$ is the cost of a minimum cost answer node in subtree x
  * $\hat{c}(\cdot)$ with $f(h(x)) = h(x)$ is an approximation to $c(\cdot)$

- The 15-puzzle

- 15 numbered tiles on a square frame with a capacity for 16 tiles
- Given an initial arrangement, transform it to the goal arrangement through a series of legal moves

1	3	4	15
2		5	12
7	6	11	14
8	9	10	13

Initial Arrangement

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	

Goal Arrangement

- Legal move involves moving a tile adjacent to the empty spot E to E
- Four possible moves in the initial state above: tiles 2, 3, 5, 6
- Each move creates a new arrangement of tiles, called *state* of the puzzle
- Initial and goal states
- A state is *reachable* from the initial state iff there is a sequence of legal moves from initial state to this state
- The state space of an initial state is all the states that can be reached from initial state
- Search the state space for the goal state and use the path from initial state to goal state as the answer
- Number of possible arrangments for tiles: $16! \approx 20.9 \times 10^{12}$
  * Only about half of them are reachable from any given initial state
- Check whether the goal state is reachable from initial state
  * Number the frame positions from 1 to 16
  * p_i is the frame position containing tile numbered i
  * p_{16} is the position of empty spot
  * For any state, let l_i be the number of tiles j such that $j < i$ and $p_j > p_i$
  * For the initial arrangement above, $l_1 = 0$, $l_4 = 1$, and $l_{12} = 6$
  * Let $x = 1$ if in the initial state, the empty spot is in one of the following positions: $2, 4, 5, 7, 10, 12, 13, 15$; otherwise $x = 0$

    Theorem 1 *The goal state is reachable from the initial state iff* $\sum_{i=1}^{16} l_i + x$ *is even.*
- Organize state space search as a tree
  * Children of each node x represent the states reachable from x in one legal move
  * Consider the move as move of empty space rather than tile
  * Empty space can have four legal moves: up, down, left, right
  * No node should have a child state that is the same as its parent
  * Let us order the move of empty space as up, right, down, left (clockwise moves)
  * Perform depth-first search and you will notice that successive moves take us away from the goal rather than closer
    · The search of state space tree is blind; taking leftmost path from the root regardless of initial state
    · An answer node may never be found
  * Breadth-first search will always find a goal state nearest to the root
    · It is still blind because no matter what the initial state, the algorithm attempts the same sequence of moves
- Intelligent solution
  * Seeks out an answer node and adapts the path it takes through state space tree
  * Associate a cost $c(x)$ with each node x of state space tree
    · $c(x)$ is the length of path from the root to nearest goal node (if any) in the subtree with root x
  * Begin with root as E-node and generate a child node with $c(\cdot)$-value the same as root
    · May not be always possible to compute $c(\cdot)$
  * Compute an estimate $\hat{c}(x)$ of $c(x)$

    * $\hat{c}(x) = f(x) + \hat{g}(x)$ where $f(x)$ is the length of the path from root to x and $\hat{g}(x)$ is an estimate of the length of a shortest path from x to a goal node in the subtree with root x
    * One possible choice for $\hat{g}(x)$ is the number of nonblank tiles not in their goal position
    * There are at least $\hat{g}(x)$ moves to transform state x to a goal state
      · $\hat{c}(x)$ is a lower bound on $c(x)$
      · $\hat{c}(x)$ in the configuration below is 1 but you need more moves

1	2	3	4
5	6		8
9	10	11	12
13	14	15	7

- Control abstractions for LC-search
  - Search space tree t
  - Cost function for the nodes in t: $c(\cdot)$
  - Let x be any node in t
    * $c(x)$ is the minimum cost of any answer node in the subtree with root x
    * $c(t)$ is the cost of the minimum cost answer node in t
  - Since it is not easy to compute $c()$, we'll substitute it by a heuristic estimate as $\hat{c}()$
  - Algorithm `LCSearch` uses $\hat{c}()$ to find an answer node

```
typedef struct
{
    list_node_t * next;
    list_node_t * parent;
    float         cost;
} list_node_t;

list_node_t list_node;

algorithm LCSearch ( t )
{
    // Search t for an answer node

    if ( *t is an answer node )
    {
        print ( *t );
        return;
    }

    E = t;          // E-node
    Initialize the list of live nodes to be empty;

    while ( true )
    {
        for each child x of E
        {
            if x is an answer node
            {
                print the path from x to t;
                return;
            }
```

```
                Add ( x );          // Add x to list of live nodes;
                x->parent = E;      // Pointer for path to root
            }

            if there are no more live nodes
            {
                print ( "No answer node" );
                return;
            }

            E = Least();      // Find a live node with least estimated cost
                              // The found node is deleted from the list of live nodes
        }
    }
```

- * The algorithm always keeps the list of live nodes in a list
  * When all the children of E have been generated, E becomes a dead node
    - · Happens only if none of E's children is an answer node
  * If there are no live nodes left, the algorithm terminates; otherwise, `Least()` correctly chooses the next E-node and the search continues
- FIFO search
  * Implement list of live nodes as a queue
  * `Least()` removes the head of the queue
  * `Add()` adds the node to the end of the queue
- LIFO search
- The only difference in LC, FIFO, and LIFO is in the implementation of list of live nodes

- Bounding
  - A branch-and-bound method searches a state space tree using any search mechanism in which all children of the E-node are generated before another node becomes the E-node
  - Each answer node x has a cost $c(x)$ and we have to find a minimum-cost answer node
    * Common strategies include LC, FIFO, and LIFO
  - Use a cost function $\hat{c}(\cdot)$ such that $\hat{c}(x) \leq c(x)$ provides lower bound on the solution obtainable from any node x
  - If U is the upper bound on the cost of a minimum-cost solution, then all live nodes x with $\hat{c}(x) > U$ may be killed
    * All answer nodes reachable from x have cost $c(x) \geq \hat{c}(x) > U$
    * Starting value for U can be obtained by some heuristic or set to ∞
    * Each time a new answer node is found, the value of U can be updated
  - Optimization/minimization problem
    * Maximization converted to minimization by changing sign of objective function
    * Formulate the search for an optimal solution as a search for a least-cost answer node in a state space search tree
    * Define the cost function $c(\cdot)$ such that $c(x)$ is minimum for all nodes representing an optimal solution
      - · Easiest done if $c(\cdot)$ is the objective function itself
  - Example: Job sequencing with deadlines
    * Given n jobs and 1 processor
    * Each job j_i has a 3-tuple associated with it represented by (p_i, d_i, t_i)
      - · Job j_i requires t_i units of processing time
      - · If processing is not completed by deadline d_i, a penalty p_i is incurred

- Objective is to select a subset J of n jobs such that all jobs in J can be completed by their deadline
- A penalty will incur only on jobs not in J
- For optimal solution, J should be such that the penalty is minimized among all possible subsets J
- Problem instance: $n = 4, j_1 = (5, 1, 1), j_2 = (10, 3, 2), j_3 = (6, 2, 1), j_4 = (3, 1, 1)$
- Solution space is all possible subsets of (j_1, j_2, j_3, j_4)
- Organize solution space by either of the two formulations of sum-of-subsets problem (variable or fixed tuple size)
- Optimal solution comes as $J = \{j_2, j_3\}$, with a penalty cost of 8
- Cost function $c(x)$ can be defined as the minimum penalty corresponding to any node in the subtree with root x
- $c(x) = \infty$ for any non-feasible node
- Variable tuple size formulation
  - $c(3) = 8$, $c(2) = 9$, and $c(1) = 8$
- Fixed tuple size formulation
  - $c(1) = 8$, $c(2) = 9$, $c(5) = 13$, and $c(6) = 8$
- $c(1)$ is the penalty corresponding to an optimal solution J
- Getting the estimate bound $\hat{c}(x)$
  - $\hat{c}(x) \leq c(x)$ for all x
  - Let S_x be the subset of jobs selected for J at node x
  - If $m = \max\{i | i \in S_x\}$ then $\hat{c}(x) = \sum_{\substack{i < m \\ i \notin S_x}} p_i$ is an estimate for $c(x)$ with the property $\hat{c}(x) \leq c(x)$
  - For a non-feasible node, $\hat{c}(x) = \infty$
  - In the tree with variable tuple formulation, for node 6, $S_6 = \{j_1, j_2\}$ and hence, $m = 2$; Also, $\sum_{\substack{i < 2 \\ i \notin S_2}} p_i = 0$
  - For node 7, $S_7 = \{1, 3\}$ and $m = 3$; therefore, $\sum_{\substack{i < 2 \\ i \notin S_2}} = p_i = p_2 = 10$
  - In the tree with fixed tuple formulation, node 12 corresponds to the omission of job j_1 and hence a penalty of 5
  - Node 13 corresponds to the omission of jobs j_1 and j_3 and hence a penalty of 13
- A simpler upper bound $u(x)$ on the cost of a minimum-cost answer node in the subtree x is $u(x) = \sum_{i \notin S_x} p_i$
  - $u(x)$ is the cost of the solution S_x corresponding to node x

- FIFO branch-and-bound
  - FIFO branch-and-bound for job sequencing problem can begin with $U = \infty$ (or $U = \sum_{1 \leq i \leq n} p_i$) as an upper bound on the cost of a minimum-cost answer node
  - Start with node 1 as the E-node in the variable tuple formulation of the state space tree
    - Nodes 2, 3, 4, and 5 are generated in that order
    - $u(2) = 19$, $u(3) = 14$, $u(4) = 18$, and $u(5) = 21$
  - Variable U is updated to 14 when node 3 is generated
  - Since $\hat{c}(4)$ and $\hat{c}(5)$ are greater than U, nodes 4 and 5 get killed (or bounded)
  - Only nodes 2 and 3 remain alive
  - Node 2 becomes the next E-node; its children are generated as nodes 6, 7, and 8
  - $u(6) = 9$ and so, U is updated to 9
  - $\hat{c}(7) = 10 > U$ and so, node 7 gets killed
  - Next node 3 becomes E-node; generating node 9 and 10
  - $u(9) = 8$ updating U to 8