Algorithms and Data Structures

13th Lecture: Heuristic Search

Yutaka Watanobe, Jie Huang, Yan Pei, Wenxi Chen,
S. Semba, Deepika Saxena, Yinghu Zhou, Akila Siriweera

University of Aizu

Last Updated: 2025/01/26

Outline

- Backtracking
- Depth First Search
- Breadth First Search
- Iterative Deepening
- IDA*
- A*

Searching (1)

- Some problems involved searching through a vast number of potential solutions to find an answer.
- An algorithm starts from the initial point and searches forward on certain paths to find the goal (solution).
- For some of the problems, we know there is a *success search path* that definitely leads to the goal.
- For such kind of problems, we can design algorithms which search the solutions on the success paths.

Searching (2)

- On the other hand, there are many problems for which we do not know which paths lead to the solutions.
- One way to solve these problems is to exhaustively search every possible paths.
- However, there may be too many possible paths to search.
- Exhaustive search is related to a great number of operations. So, we need some techniques to bound the number of possible paths to increase the search efficiency.

Backtracking (1)

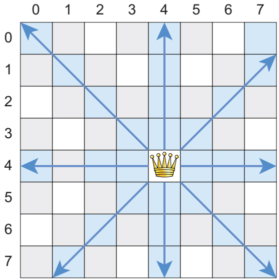
- **Backtracking** is a systematic way to go through all the possible configurations of a search space (all the possible paths).
- In backtracking search, when we know we can not go forward anymore on some possible path, we go backward to find another path.
- Depth-first-search is an example of backtracking algorithms.

Backtracking (2)

- Backtracking is quite widely applicable as general problem-solving techniques.
- For example, they form the basis for many programs that play games such as Chess.
- In this case, a partial solution is some legal positioning of all the pieces on the board, and the descendant of a node in the exhaustive search tree is a position that can be the result of some legal move.
- A backtracking search is typically done with quite sophisticated pruning rules so that only "interesting" positions are examined.
- Exhaustive search techniques are also used for other applications in artificial intelligence.

8 Queens Problem (1)

- Backtracking technique can be used to solve the classic eight queens problem:
  - put eight queens on a chess-board such that none of them threatens any of others (a queen threatens the squares in the same row, in the same column, or on the same diagonals).



8 Queens Problem (2)

- An obvious way to solve this problem consists of trying all the ways of placing eight queens on a chess-board, checking each time to see whether a solution has been obtained.
- This approach is of no practical use, since the number of possible positions we have to check would be:

$$\binom{64}{8} = 4,426,165,368$$

- Can we do it better?

8 Queens Problem (3)

- First, we know that two queens can not be in the same row. So, eight queens should be put in eight rows, one queen in one row.
- Since each queen has 8 positions to put in the row, there are $8^8 = 16,777,216$ positions. Similarly, two queens can not be in the same column.
- Thus, if the queen in the first row has been put in the *i*th column, the other queens can not be in the *i*th column.
- From this, we can reduce the possible positions to $8! = 40,320$.

Backtracking for 8 Queens Problem (1)

- Backtracking allows us to do much better than the above. Using a recursive call, we can realize a backtracking algorithm for eight queens problem as follows:
  - We put the queen of the first row at any position of the row.
  - Then we put the queen of the second row to a position of the row that is not threatened by the queen of the first row.
  - ...

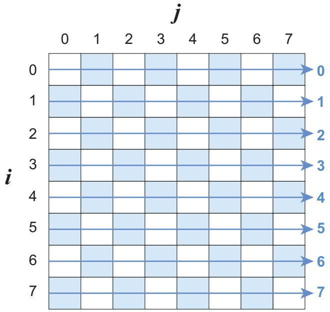
Backtracking for 8 Queens Problem (2)

- Assume, we have put i queens in the first i rows such that none of them threatens any of others. We put the queen of the $(i + 1)$th row to a position of the row that is not threatened by any of the previous i queens.
- If we can not find such a position for the queen of the $(i + 1)$th row, we go back to the ith row to find another non-threatened position for the queen of the ith row (if no such position exists we go back further to $(i - 1)$th row) and then try again.

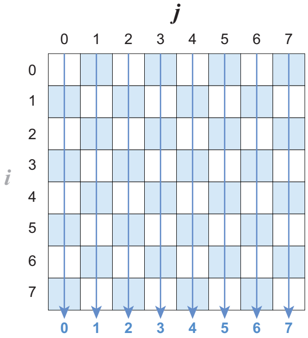
Implementation (1)

- Assume the rows, columns, and diagonals of chess-board is defined as in the next Figure:

`row: x = i`

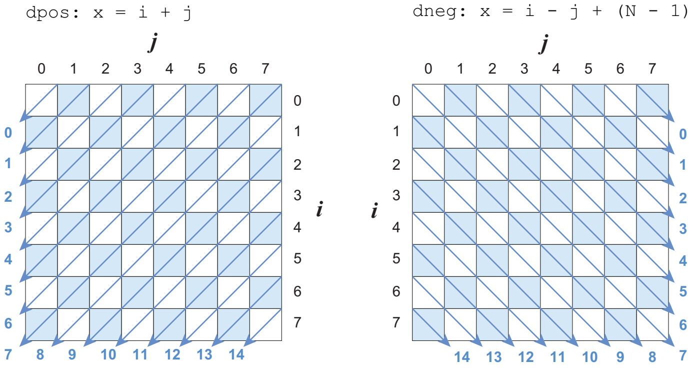


`col: x = j`



Implementation (2)

- Assume the rows, columns, and diagonals of chess-board is defined as in the next Figure:



Implementation (3)

- We use a 1×8 integer array row[8] to express the positions of the queens at each row. That is, row[0], row[1], ..., row[7] are used to keep the positions (column) of queens in the row 0, 1, ..., 7, respectively.
- Next, we use a 1×8 integer array col[8] to show if each position of a given row is threatened by a queen in the same column.
- For a given row *i*, col[*j*] == *free* means there is no queen in the *j*th column, otherwise a queen has been put in the same column and we can not put the queen of the given row at the *j*th column.

Implementation (4)

- We also have to consider the threatens from the queens on the same diagonals.
- As shown in the Figure there are 15 positive diagonals (at 45 degrees) and 15 negative diagonals (at 135 degrees).
- We use two other arrays dpos[15] and dneg[15] to denote if a position is threatened by a queen on the same positive diagonal and negative diagonal, respectively.
- dpos[x] == *free* means that there is no queen on the diagonal with number x, otherwise there is a queen on diagonal x. Similarly define dneg[x].

Implementation (5)

- When we have a queen at the position p[i][j] (the ith row and the jth column), we know that the positions on the positive diagonal $i + j$ and the positions on the negative diagonal $i - j + (N - 1)$, where N is the size of chess-board (8 in 8-queens problem), are threatened by this queen.
- So, when we have put a queen at the position p[i][j], we set dpos[$i + j$] and dneg[$i - j + n - 1$] to not-free.

Implementation (6)

```
putQueen(i)
  if i == N
    printBoard()
    return

  for j = 0 to N-1
    if col[j] == NOT_FREE ||
       dpos[i+j] == NOT_FREE || dneg[i-j+N-1] == NOT_FREE
      continue
    // put a queen at (i, j)
    row[i] = j
    col[j] = dpos[i+j] = dneg[i-j+N-1] = NOT_FREE
    // try the next row
    putQueen(i+1)
    // remove the queen at (i, j) for backtracking
    col[j] = dpos[i+j] = dneg[i-j+N-1] = FREE
```

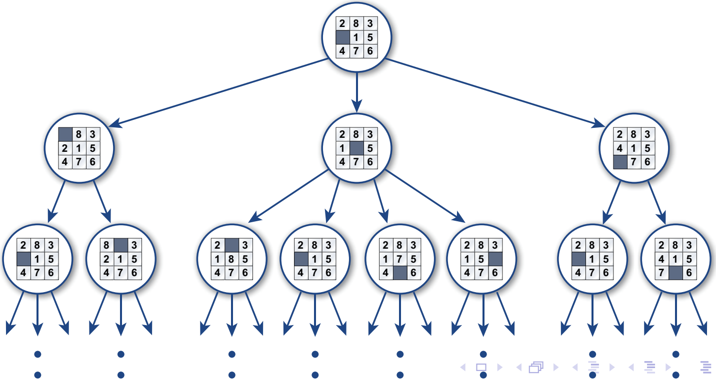
8 Puzzle Problem

- The goal of the 8 puzzle problem is to complete pieces on 3×3 cells where one of the cells is empty space.
- You can move a piece to the empty space at one step.
- Your goal is to solve an 8 puzzle problem in the shortest move (fewest steps).

```
1 3 0          1 2 3
4 2 5   -->    4 5 6
7 8 6          7 8 0
```

State Transition (1)

- Such kinds of puzzle can be solved by repetitive state transitions in the search space.
- Generally, a search algorithm generates a sequence (or set) of the states by the transitions.



State Transition (2)

- Important thing is that we should not create the same state during the state transitions. So, we generate a tree structure as the search space where nodes and edges represents the states and the transitions respectively.
- For the 8 puzzle problem, a state (node) corresponds to an alignment sequence (permutation) of the pieces (including the empty space) and a transition corresponds to the movement of a piece.
- Generally, you can solve the problem by depth-first search and breadth-first search.
- To manage the states, you can use data structures related to hash or binary search trees.

Depth First Search (1)

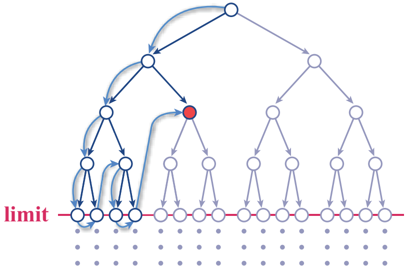
- The depth-first search is based on the DFS algorithm on graphs.
- The depth-first search starts with the initial state of the given puzzle and repeats the state transitions until the algorithm find the goal state by visiting candidate notes recursively.
- The depth-first search uses the following pruning techniques:
    - Abandon the search and backtrack when you can not create the new state in the search space.
    - Abandon the search and backtrack when you create the same state which is in the sequence of the state transitions.
    - Abandon the search and backtrack when you can determine that you do not need to create new states any more.

Depth First Search (2)

- The depth-first search has the following features:
  - It's not always true that the depth-first search finds the shortest path.
  - It can be an exhaustive search when the pruning does not work well.

Depth Limit Search (1)

- The depth-limit search applies the depth limit during the depth-first search.
- The depth-limit search abandons its search when the depth of the search reaches the specified limit.

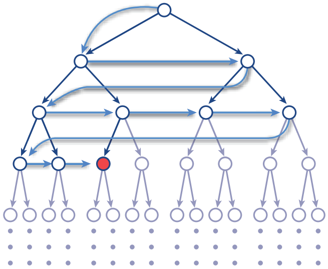


Depth Limit Search (2)

- The depth-limit search has the following features:
  - We do not need to memorize the search states.
  - It can be a basis for the Iterative Deepening algorithm to find the shortest path.

Breadth First Search (1)

- The breadth-first search is based on the BFS algorithm on graphs.
- First of all, the BFS generates an initial (start) state and puts it into a queue. Then, the BFS algorithm gets a state from the queue and generate the next states based on that state, and so on.



Breadth First Search (2)

- The generated states should be memorized by hash or other data structures (binary search trees, etc.).
- The breadth-first search has the following features:
  - It can find the shortest path from the initial state.
  - It can consume excessive amounts of memory to maintain the state transitions.

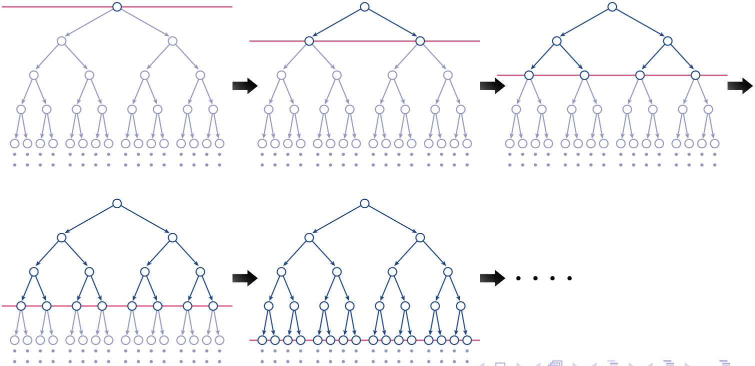
15 Puzzle Problem

- The goal of the 15 puzzle problem is to complete pieces on 4×4 cells where one of the cells is empty space.
- You can move a piece to the empty space at one step.
- Your goal is to solve an 15 puzzle problem in the shortest move (fewest steps).
- Can you solve it by BFS or DFS?

```
1  2  3  4         1  2  3  4
6  7  8  0   -->   5  6  7  8
5 10 11 12         9 10 11 12
9 13 14 15        13 14 15  0
```

Iterative Deepening

- The iterative deepening algorithm repeats the depth-limit search by incrementing the limit until the algorithm find the goal.
- Generally, we do not need to memorize the state transitions (but avoid back tracking to the previous state).

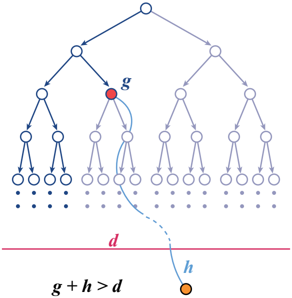


IDA* (1)

- The iterative deepening can be extended as the IDA* algorithm by pruning based on estimate values so called *heuristic*.
- The heuristic estimated is the lower limit of steps to the goal.
- For the 15 puzzle problem, we can prune the search by using the shortest cost h from the current state to the goal state as the heuristic.

IDA* (2)

- So, if we can find a heuristic h such that "we need at least h steps from the current state to the goal state", we can assert that if $g + h$ (where g is the current depth) exceeds the limit d (of depth-limit search) we do not need to search any more.

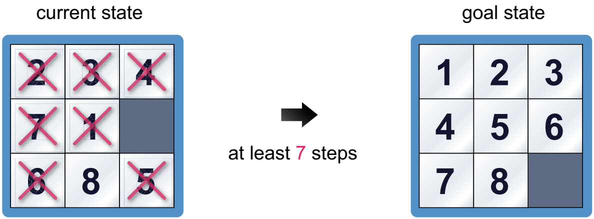


IDA* (3)

- The heuristic value *h* can be an estimation. It doesn't need to be exact value.
- If we can estimate higher value as heuristic, the search algorithm will be faster.
- On the other hand, if you estimate too much, you will miss the solution.

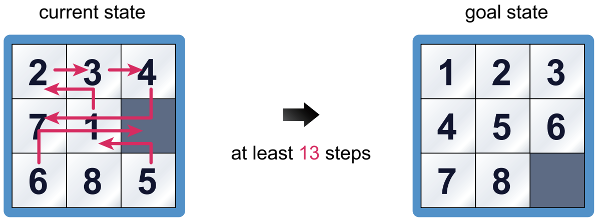
Possible Heuristic for 8 Puzzle Problem (1)

- $H1$: The number of pieces which are on incorrect position.



Possible Heuristic for 8 Puzzle Problem (2)

- *H*2: Sum of manhattan distance between the initial position to the goal position for each piece.
- The manhattan distance is the distance between two points in a grid based on a strictly horizontal and/or vertical path.



A*

- The estimate values (heuristic) can be used for Dijkstra's algorithm (or BFS) based on a priority queue.
- This A* algorithm manages the state transitions by a priority queue.
- The algorithm select the current node (extracting the next element from the priority queue) in such a way that $g + h$ is minimized where g is the cost from the initial state to the current state and h is the estimated value to the goal state.
- In this way, we can speed up the BFS based search algorithm.