A brief introduction to Search Problems
Notes created for MCS 260

Dimitris Diochnos

March 24, 2008

1 What is a search problem?

To put it in a simple and intuitive framework, in a search problem we want to reach a *goal* state given a set of possible initial states. A big class of problems of this kind naturally arises in games. For example, given an initial Sudoku state where few cells are predetermined, find an assignment of numbers to cells so that the Sudoku conditions hold[1]. Another problem with a slightly different flavor (which is still characterized as a search problem) is for example given a position in chess, and allowing legal chess moves, find the *fastest* route to victory for a specific player. Note that the difference here, is that we not only want to achieve victory, but also in the *fastest* possible way.

More formally a search problem is composed by a set S of states, usually called *state space* or *search space*. A subset of those states (many times just a single state) form a set I of the *initial* or *starting* states. There is also another set G called *goal* states; these are the states we want to reach. Moreover, there is a set of rules, or if you prefer a *transition function* r, which gives *legal* transitions (moves) for the player(s) from one state to another. Note that in problems of this kind we are usually interested in just a *single* solution although there may be more solutions. The above information gives rise to a picture like the one on your right. Here we are at state v and there are 5 successor states; namely a, b, c, d, and e. This picture usually represents the structure of the state space at the other states as well; i.e. for example a can take place of v and the successors of a follow on a subsequent *level* (*layer*); until you reach a goal state after which point there is no meaning of branches because you have already accomplished your goal! Of course, you might also encounter states after which there are no states that you can move to, yet these states are neither goal states (dead end; e.g. reach a stalemate position in chess when in fact we have a winning sequence of moves from the starting position).

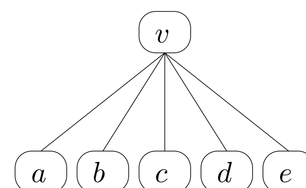


Figure 1: Arbitrary snapshot.

In other problems however, you might not have the tree structure that is shown above, but rather a graph of states, where from one state you have an option (or options) to move back to a state previously encountered. Finally, another important aspect that appears on the figure is the notion of *branching factor*; the number of legal one-step-ahead states that you can reach

[1] Sudoku is a game played usually on an $N \times N$ grid, and initially some of the cells of the grid have some numbers from 1 to N assigned on them. The goal is to fill in the rest of the cells with numbers in $M = \{1, \ldots, N\}$ such that in each row and each column each number of the set M appears exactly once.

given an arbitrary state. If that number varies among different states, we usually talk about the *average* branching factor (of the entire game / problem).

2 How to search?

In order to counter search problems, we typically use a data structure D which holds options (states) that have not been considered so far. Usually this data structure is a *stack* or a *queue* for reasons that will soon become apparent. This way, as long as there are states in D, we pick one of them and check if this is a goal node. If it is, then we are done, and we found a solution; otherwise we generate states that can occur from this position and store them in D so that we can consider them at a later timestep. Of course, some problems might not have even a single solution, and in this case after exhaustively searching the entire state space we can report that there is no satisfying solution. In terms of pseudocode the process can be captured with the following algorithm.

General Search Algorithm(I)

```
1  D ← I
2  while D ≠ ∅
3      do v ← pick a state (and delete) from D
4          if v ∈ G
5              then return v or a path leading to v
6          Insert successors of v into D
7  return ∅
```

3 Basic search techniques

There are two very basic search techniques; *breadth-first* search and *depth-first* search and I will describe them briefly below.

3.1 Breadth-first search (BFS)

In this case our data structure D is a queue, and the algorithm takes the form:

Breadth-First Search(I)

```
1  D ← I
2  while D ≠ ∅
3      do v ← pick first state in D
4          if v ∈ G
5              then return v or a path leading to v
6          Insert successors of v in the end of D
7  return ∅
```

The main idea, is that we examine all states at level l before we examine states at level $l+1$. Hence, a queue is what we need for D, since this will force us visit the states inserted in D in a first-in first-out (FIFO) order.

3.2 Depth-First Search (DFS)

In this case our data structure D which holds states that we want to visit is a *stack*. The pseudocode which represents this way of searching is shown below:

Depth-First Search(I)

1 $D \leftarrow I$
2 **while** $D \neq \emptyset$
3 **do** $v \leftarrow$ pick state *on top* of D
4 **if** $v \in G$
5 **then return** v or a path leading to v
6 Insert successors of v *on top* of D
7 **return** $\emptyset$

The main idea is that we expand states as far away from our starting state as possible. Therefore a stack is ideal for our purpose, since once we visit a node v and figure out that this is not the goal node, then we insert on top of the stack its children (immediate successor states), visit one of them, and keep working in this manner.

3.3 Comparing breadth-first search and depth-first search

Let's see an example of a simple one-player game. The game is played in rounds, and at each round the player has to make a decision as to where to move. Clearly he wants to reach one of the winning states which are indicated in figure 2 with a bold boundary at the specific state. The user starts at the initial state s and can reach states v_1, v_2, and v_3 by making a (different each time) move. Similarly, from v_1 the player can move to the states v_4, v_5, and v_6, as well as from v_2 the player can move to v_7, v_8, and v_9. States v_3, and v_7 are goal states (say winning positions) while states v_4, v_5, v_6, v_8 and v_9 are not winning positions (and the game does not continue beyond these points). Figure 3 indicates the order in which the different states are explored by each method. Hence, in the BFS case, the solution $\langle s, v_3 \rangle$ is returned,

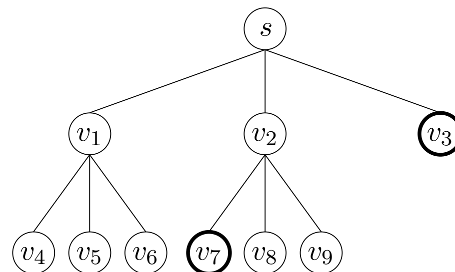


Figure 2: A simple one-player game.

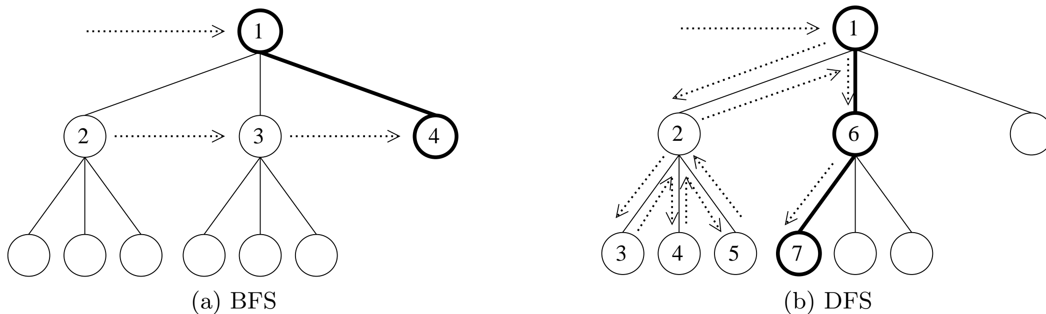


Figure 3: The order in which states are explored via DFS and BFS as well as the solution paths that they return (with bold lines). The dotted arrows indicate the way in which we perform the search.

while in the DFS case the solution $\langle s, v_2, v_7 \rangle$ is returned. As the figure indicates, BFS found a shorter route to a winning state. This is not a coincidence since BFS visits states in a layer-by-layer manner (where each layer indicates the number of moves required from the starting position). This is an important property of the BFS; i.e. *it will always find a solution within a minimal distance (moves) from the starting position*. This is not the case for DFS. Moreover, BFS is quaranteed to find a solution to a given problem assuming there is one just like in the example above. However, this is not true for DFS. For example, assume that in the simple game described above we had instead below the state v_1 an infinite tree which did not contain a single solution; i.e. once the player moved from v_1 to a subsequent state of v_1, no matter which option he chose there was no end in the game and moreover, no winning state appeared below that point. Obviously, DFS would never exhaust the tree below v_1 since the tree is infinite and it does not contain a single winning state!

3.3.1 A note on the running time and the memory requirements of BFS and DFS

Now assume that you have a game just like the one above, where the branching factor is constant $b \geq 2$ at each state and moreover, that the game lasts for no more than d moves. Then clearly, DFS is an option here since we can not run into the trouble described above. A natural question is to ask about the memory requirements and the running time of those methods.

Running time: Both methods in the worst case might have to visit all states and either find the single winning state with their last move or report that there is no winning sequence of moves for the specific game. But the number of all states in such a tree is

a.k.a. *"curse of dimensionality"*

$$1 + b + b^2 + \ldots + b^d = \sum_{i=0}^{d} b^i = \frac{b^{d+1} - 1}{b - 1}.$$

In other words, in the worst case we might require an exponential number of steps (with respect to the amount of moves d) in order to find a solution or report that no solution exists.

Memory requirements:

BFS: In BFS case we store states in a level-by-level manner. Hence, in the worst case we might have to store the states of the last level, which are b^d.

DFS: In DFS case we store at most b states at each level. Hence, in the worst case we have to store no more than $b \cdot d$ states.

Therefore, BFS demands space that grows exponentially with d while DFS demands space that grows only linearly with d. Hence, if we don't care about the shortest solution (in terms of number of moves from the starting position), DFS is the preferred method, because of the small memory requirements compared to BFS.

3.3.2 Getting the best from both worlds

In an attempt to exploit the good parts of both methods, another popular method has emerged; named *iterative deepening*. Briefly, the method *iteratively* performs DFS on subtrees (or sub-

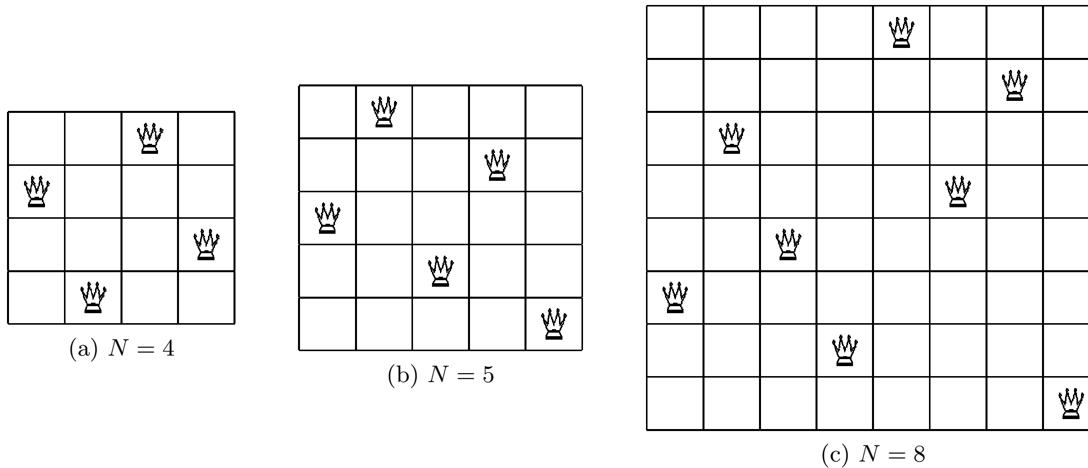
graphs) of the given problem of depth $0, 1, 2, 3, \ldots$ and so on. This way the memory requirements are kept low since at each cut-off level we perform DFS, and moreover, the shortest route to a winning position can be found just like in BFS because we have exhaustively searched all states on earlier levels! On the other hand, the running time is asymptotically the same as in the cases above. Perhaps you'll see more of that method in another course. Finally, it should be mentioned that there are other search methods as well.

4 Application: The N-Queens problem

Let's see an application of the search methods that were described earlier to the N-Queens problem. The problem has its origins in chess, and the goal is to find a way to place 8 queens on the standard 8×8 chessboard in such a way, that no queen attacks any other. The N-Queens problem is the generalization of that problem; more specifically we want to find a way of placing N queens on an $N \times N$ chessboard, in such a way that no queen is attacked by another. Note that a queen can move horizontally, vertically, and diagonally. The squares that are along these ways are *attacked*. Another (obvious?) restriction is that we can not place more than one queen on each square.

From what is given by the problem description, it is obvious that in order to find a solution on an $N \times N$ chessboard one has to place N queens, or in other words find a solution at depth N from the starting state (empty chessboard). Therefore, since there is no way of running into the problem of a non-terminating tree, Depth-First Search is the way to go. Note that there is also no point of saying that we want the *shortest* solution, since all solutions (if there are any) lie at depth N (where at each level we place one queen on the chessboard).

Note that the problem does not have a solution for $N = 2$ or $N = 3$, and it has a trivial solution for $N = 1$. Some solutions for $N \geq 4$ are shown in figure 4.



(a) $N = 4$ (b) $N = 5$ (c) $N = 8$

Figure 4: Sample solutions to the N-Queens problem for $N = 4, 5$, and 8.

4.1 Representation

One very important thing when solving, is the representation that we use for describing various states. The representation I chose is perhaps not the most efficient; however it is pretty simple

and therefore easy to deal with. In particular, we will represent a placement of queens on a chessboard with a list of coordinates where we placed our queens. Hence $[(0,0),(1,2)]$ implies that we have placed a queen on the upper left corner and another queen on the following row and two columns to the right.

In the end, if we find a solution with one of our methods, we will return such a list of length N on an $N \times N$ chessboard, otherwise, we will return the empty list [].

4.2 Direct brute-force approach

The code for this method can be found in file `nqueensbrute.py` under the directory:

`http://www.math.uic.edu/~diochnos/ta/2008/mcs260/code/search/nqueens`

Just like we figured out earlier, we will use DFS. We will name our method `Nqueens` which will be called by our `main` function for the required dimension of the board. The code follows:

```
1 def Nqueens (dimension):
2   stack = [[]]
3   while len(stack) > 0:
4     current_position = stack.pop ()
5     if is_winning_position (current_position, dimension):
6       return current_position
7     successors = get_successors (current_position, dimension)
8     stack = stack + successors
9   return []
```

Some comments might be necessary to justify the code. Note that by our earlier convention, no placement of queens on chessboard is represented by an empty list of coordinates []. Hence, at line 2 we initialize our stack to exactly that placement of queens because this is our starting position. The crucial part on the code of course is implementing the two functions `is_winning_position` and `get_successors` which are needed in lines 5 and 7. Note that at line 8 we insert the successors *on top* of the stack as needed.

Difference between $\emptyset$ and $\{\emptyset\}$ ...

`is_winning_position`: The implementation of the function checks all $\binom{N}{2}$ pairs of placed queens. Note, that if a call to this function is made before we place all N queens, then the function simply returns `False` since this can not be a winning position. If all queens are not attacked by another queen, then the function returns `True`

`get_successors`: Given the `current_position` which is a placement of k queens, this function returns a list of all possible extensions of our k-placement of queens to a $(k+1)$-placement of queens; i.e. we get a list of possible arrangements of $k+1$ queens on the chessboard, but the first k queens are placed on the squares described by our `current_position` (therefore these are naturally the successor states of our current state). Note, that the function does not perform a single check as to whether or not the additional queen is placed on a square that is not attacked by another queen. This condition will be checked by the function `is_winning_position`.

The method solves quickly (less than 1 second) the cases up to $N = 5$. However, for larger values seems to take too long; e.g. for $N = 6$ in the same machine it takes about 10 minutes. Sample executions of the program are shown below:

```
$ python nqueensbrute.py
Give me the dimension of the board: 3

Impossible to place queens!
$ python nqueensbrute.py
Give me the dimension of the board: 4

-  Q  -  -
-  -  -  Q
Q  -  -  -
-  -  Q  -
$
```

4.3 Revising the code - part I

As it was descibed earlier in the `get_successors` function, we might extend arrangements of queens on the board in a way such that two (or more) queens might be in the same row, or column, or even diagonal. This is responsible for the great inefficiency of our previous method. For example, if we generate an (partial) arrangement of queens of the form $[(0,0),(0,1)]$, no matter how we extend this with an extra $N-2$ queens, the function `is_winning_position` will always report that this is not a winning position simply because the first two queens lie on the same row. Hence, we can modify the function `get_successors` and not allow situations like these; i.e. do not generate arrangements of queens such that one queen attacks another. This way our stack of available options will not increase in size so fast as before, or if you prefer we effectively reduce the branching factor of our state space. The file that performs this task is named `nqueens1.py`. The modified version of the function is shown below:

```
 1 def get_successors (arrangement_of_queens, dimension):
 2   if len(arrangement_of_queens) == dimension:
 3     return []    # N queens have been placed. No successors in this case!
 4   attacked_squares = get_squares_attacked (arrangement_of_queens, dimension)
 5   successors = []
 6   for i in range(dimension):
 7     for j in range(dimension):
 8       if attacked_squares [i][j]:
 9         pass
10       else:
11         newcopy = arrangement_of_queens[:]
12         successors.append (newcopy + [(i,j)])
13   return successors
```

where the `get_squares_attacked` function returns an $N \times N$ chessboard with `True`/`False` values for each cell; `True` if the square is attacked by a queen on the current arrangement (or a queen is sitting on it), and `False` otherwise. Hence, with the doubly nested `for` loop,

we extend the current arrangement of queens on the board, only if the square `(i, j)` is not attacked; i.e. we can place a queen there.

This time problems start for $N \geq 12$ where we require more than 1 minute.

4.4 Revising the code - part II

We can further decrease the branching factor in our state-space. The idea comes from the simple observation that we have to place exactly one queen in each row (or column if you prefer). Therefore we now extend an arrangement of queens by considering placing a queen *only* among the cells of the next row; i.e. among the cells of the first empty row. Therefore, we need to modify again the `get_successors` function and search only among the cells of the first empty row. This is shown below and is implemented in file `nqueens2.py`:

```
 1 def get_successors (arrangement_of_queens, dimension):
 2   if len(arrangement_of_queens) == dimension:
 3     return []    # N queens have been placed. No successors in this case!
 4   attacked_squares = get_squares_attacked (arrangement_of_queens, dimension)
 5   successors = []
 6   placed_queens = len(arrangement_of_queens)
 7   for j in range (dimension):
 8     if attacked_squares [placed_queens][j]:
 9       pass
10     else:
11       newcopy = arrangement_of_queens[:]
12       successors.append (newcopy + [(placed_queens,j)])
13   return successors
```

Now we are doing much better. For example:

Note how to redirect input from a text file.

```
$ cat input.txt
12
$ date && python nqueens2.py < input.txt && date
Mon Mar 24 15:31:03 CDT 2008
Give me the dimension of the board:
-  -  -  -  -  -  -  -  -  -  -  Q
-  -  -  -  -  -  -  -  -  Q  -  -
-  -  -  -  -  -  -  Q  -  -  -  -
-  -  -  -  Q  -  -  -  -  -  -  -
-  -  Q  -  -  -  -  -  -  -  -  -
Q  -  -  -  -  -  -  -  -  -  -  -
-  -  -  -  -  -  Q  -  -  -  -  -
-  Q  -  -  -  -  -  -  -  -  -  -
-  -  -  -  -  -  -  -  -  -  Q  -
-  -  -  -  -  Q  -  -  -  -  -  -
-  -  -  Q  -  -  -  -  -  -  -  -
-  -  -  -  -  -  -  -  Q  -  -  -
Mon Mar 24 15:31:04 CDT 2008
$
```

4.5 One last tip

Going back to figure 2 it is obvious that in the DFS case, if we had arranged states v_1, v_2, and v_3 in a different way, then we would have found a solution faster. Such re-arranging of states is usually performed via a *heuristic* function. Essentially the only change that needs to be done is to place successor states in a different order on top of the stack (with the ones you prefer the most closer to the top). Hence, you can define a function `prioritize_successors` which performs such a re-arranging as it is shown below:

```
 1 def Nqueens (dimension):
 2   stack = [[]]
 3   while len(stack) > 0:
 4     current_position = stack.pop ()
 5     if is_winning_position (current_position, dimension):
 6       return current_position
 7     successors = get_successors (current_position, dimension)
 8     prioritize_successors (successors, dimension)
 9     stack = stack + successors
10   return []
```

Sometimes a good idea on guiding our search might be all we need in order to find *one* solution in a chaotic state space.