National University of Singapore
School of Computing
CS3243 Introduction to AI

Tutorial 2: Uninformed Search

Issued: February 5, 2020 Due: Week 4 in the tutorial class

Important Instructions:

- *Your solutions for this tutorial must be TYPE-WRITTEN.*
- *Make TWO copies of your solutions: one for you and one to be SUBMITTED TO THE TUTOR IN CLASS. Your submission in your respective tutorial class will be used to indicate your CLASS ATTENDANCE. Late submission will NOT be entertained.*
- *YOUR SOLUTION TO QUESTION* 3 *will be GRADED for this tutorial.*
- *You may discuss the content of the questions with your classmates, but you should work out and write up ALL the solutions by yourself.*

1. Sudoku is a popular number puzzle that works as follows: we are given a 9×9 square grid; some squares have numbers, while some are blank. Our objective is to fill in the blanks with numbers from $1 - 9$ such that each row, column and the highlighted 3×3 squares contain no duplicate entries (see Figure 1). Solving Sudoku puzzles is often modeled as a CSP (which we will cover later in the course). Consider, however, the problem of *generating* Sudoku puzzles. Consider the following procedure: start with a completely full number grid (see Figure 2), and iteratively make some squares blank. We continue blanking out squares as long as the resulting puzzle can be completed in only one way. Questions:
   - Give the representation of a state in this problem;
   - Using the state representation defined above, specify the initial state and goal state.
   - Define its actions; and
   - Using the state representation and actions defined above, specify the transition model/function T (i.e., when each of the actions defined above is applied to a current state, what is the resulting next state?).

2	5			3		9		1
	1				4			
4		7				2		8
		5	2					
				9	8	1		
	4				3			
			3	6			7	2
	7							3
9		3				6		4

Figure 1: A simple Sudoku Puzzle

2	5	8	7	3	6	9	4	1
6	1	9	8	2	4	3	5	7
4	3	7	9	1	5	2	6	8
3	9	5	2	7	1	4	8	6
7	6	2	4	9	8	1	3	5
8	4	1	6	5	3	7	2	9
1	8	4	3	6	9	5	7	2
5	7	6	1	4	2	8	9	3
9	2	3	5	8	7	6	1	4

Figure 2: Solution to the Sudoku puzzle in Figure 1.

Solution:

- A state in this problem is a (partial) valid solution of the sudoku puzzle. More formally, it would be a matrix $A \in \{0, \ldots, 9\}^{9\times 9}$ with 0 representing a blank square.
- The initial state is a completely filled out grid of numbers, where the grid is valid: all rows, columns and 3×3 squares contain all numbers between $1, \ldots, 9$.
- An action would be removing a number from the grid. More formally, we take as input a matrix A and a matrix $E_{i,j}(a)$ where $E_{i,j}(a) \in \{0, \ldots, 9\}^{9\times 9}$ is a matrix of all zeros except for a non-zero value $a \in \{1, \ldots, 9\}$ in coordinate (i, j). An action would be setting $A - E_{i,j}(a)$.
- The transition model would be $T(A, E_{i,j}(a)) = A - E_{i,j}(a)$. It is important to note that as we start removing numbers it would be important to start testing for a goal node - this would be a matrix A^* such that for any $E_{i,j}(a)$, $A^* - E_{i,j}(a)$ *cannot* be uniquely completed. Think - how would you ensure this? It is not a trivial check to implement efficiently. In addition, it is worthwhile to think about your search strategy in this case. How would you make a *good* puzzle generator? It would probably make sense to randomly remove numbers rather than deterministically doing so. There are many goal nodes here, but it would be boring to use your sudoku generator if it always outputted the same puzzle on the same input.

2. Consider a class managed by an instructor. The instructor allocates one day to meet with students to discuss a final project; the day has t time slots in it denoted $S = \{s_1, \ldots, s_t\}$. Every student i indicates their availability denoted by a subset $A_i \subseteq S$. Our objective is to schedule all students for a meeting, while respecting their availability constraints[1]. Your goal is to model this as a search problem, with the state space defined as a partial (valid) allocation of some of the students into the time slots.

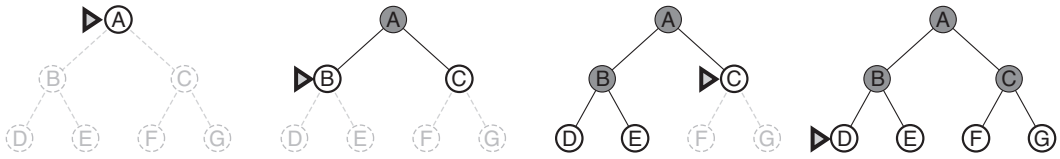

Figure 3: BFS on a simple binary tree, At each stage, the node to be expanded next is indicated by a marker.

Suppose there are three time-slots, and three students whose availability is given below. Assume that whenever you need to choose a student to add to a schedule, you do so in the

[1]This problem can also be modeled as a CSP, which we will cover in later classes.

order (A)lice (B)ob and then (C)laire. In addition, when choosing a time slot, ties are always broken in order of $s_1 \succ s_2 \succ s_3$.

	s_1	s_2	s_3
Alice	✓	✓	✓
Bob	✓		
Claire		✓	

(a) Give a trace of the BFS algorithm in the style of Figure 3.
That is, show the search tree at each stage (all repeated states are eliminated).

(b) Give a similar trace of the DFS search algorithm.

(c) Which of these two search algorithms is better for this problem? Why? Is one search strategy always better than the other in general?

Solution: Traces for BFS and DFS are given in Figures 4 and 5, respectively (I have omitted nodes that are added to the frontier but not explored). I have written next to each node the order in which it was explored. BFS takes much longer in this case; while this is not a general truth, it is true in this case since all goal states are in the same depth, and the depth is rather low.

3. We have seen various search strategies in class, and analyzed their worst-case running time. Prove that *any deterministic search algorithm* will, in the worst case, search the entire state space. More formally, prove the following theorem

Theorem 1. *Let $\mathcal{A}$ be some complete, deterministic search algorithm. Then for any search problem defined by a finite connected graph $G = \langle V, E \rangle$ (where V is the set of possible states and E are the transition edges between them), there exists a choice of start node s_0 and goal node g so that $\mathcal{A}$ searches through the entire graph G.*

Solution: Let us begin by running $\mathcal{A}$ on the graph G, without setting any goal node at all: that is, there are no goal nodes at all in G. In this case, the algorithm $\mathcal{A}$ will return "False" when it explores the entire set V. Let $H_t(\mathcal{A}, s_0) \subseteq V$ be the set of nodes that $\mathcal{A}$ explores if it starts at s_0, and does not encounter a goal node at steps $1, \ldots, t$ (at $t = 1$ we have $H_1(\mathcal{A}, s_0) = \{s_0\}$). We also let v_t be the node that $\mathcal{A}$ selects at time t given that it has observed the set $H_{t-1}(\mathcal{A}, s_0)$ so far. We note that it is entirely possible that $\mathcal{A}$ selects $v_t \in H_{t-1}(\mathcal{A}, s_0)$; however, we make a simple observation: the sequence $(H_t(\mathcal{A}, s_0))_{t=1}^{\infty}$ is weakly increasing in size, and there exists some time t^* such that for all $t > t^*$, $H_t(\mathcal{A}, s_0) = V$; in other words, since $\mathcal{A}$ is a complete search algorithm, it will continue exploring the

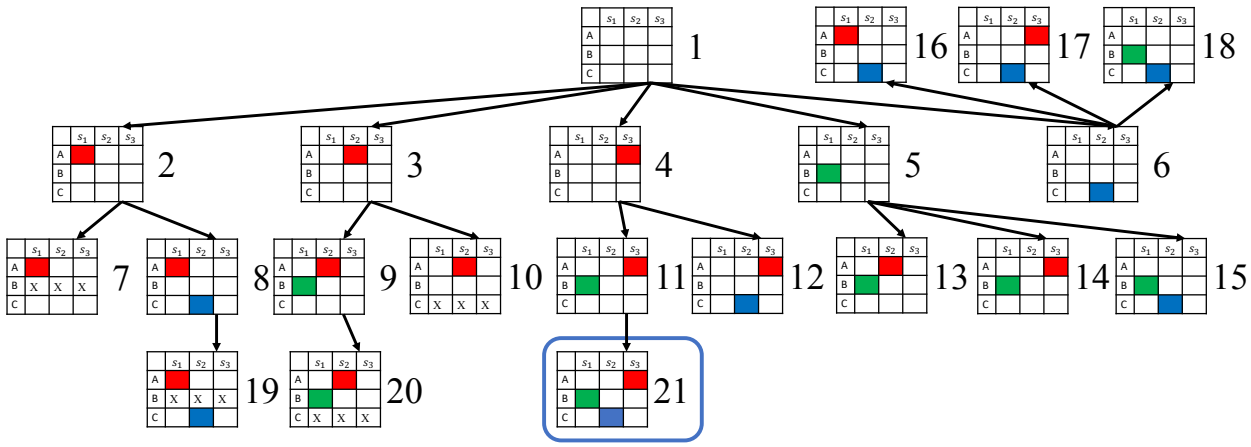

Figure 4: Search trace under BFS.

nodes in G until all nodes have been exhausted. Let us assume that t^* is the first time step for which $H_t(\mathcal{A}, s_0) = V$. In other words, at time $t^* - 1$, $|H_{t^*-1}(\mathcal{A}, s_0)| = |V| - 1$.

We now set the goal node to be v_{t^*}. From our previous argument, we know that when $\mathcal{A}$ starts at s_0 it will explore a set of size $|V| - 1$ before reaching v_{t^*}, realizing that it is a goal node and terminating. In other words, for any node s_0, if we select a goal node according to the above procedure, the algorithm $\mathcal{A}$ will exhaustively search through the entire graph before reaching a goal node.

Here is another, inductive proof. let us set the goal node to some arbitrary node g_1. If $\mathcal{A}$ searches through the entire graph G when g_1 is the goal, we are done; otherwise, let U_1 be the set of unsearched node when g_1 is the goal node. We take an arbitrary node g_2 in U_1 to be the goal; since $\mathcal{A}$ is deterministic and complete it will run the same search order that it did when g_1 was the goal, and then search through the nodes in U_1 until it reaches g_2. If it searched through all the nodes in U_1 as well, we are done, otherwise repeat. In general, suppose that we have set g_t to be the goal node and that $\mathcal{A}$ did not search through the entire graph until it reached g_t; let U_t be the set of unsearched nodes when g_t is the goal node. We set g_{t+1} to be some arbitrary node in U_t and rerun $\mathcal{A}$; since $\mathcal{A}$ is deterministic we know that when g_{t+1} is the goal we have $U_{t+1} \subset U_t$. Since $U_1 \supset U_2 \supset \cdots \supset U_t$ and the number of nodes in G is finite, there exists some iteration t^* such that $U_{t^*} = \emptyset$; thus g_{t^*} is a goal node

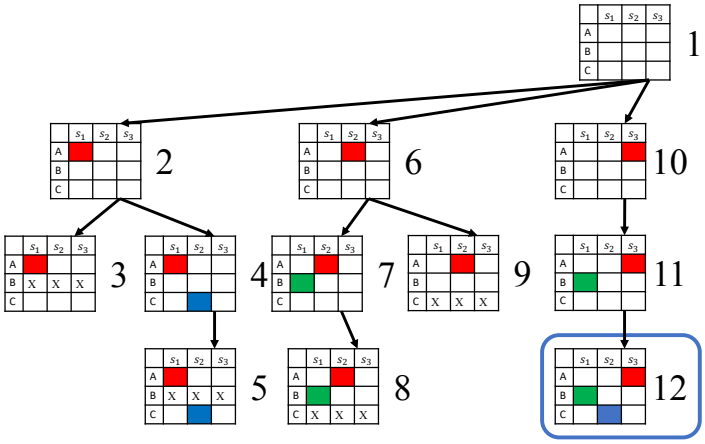

Figure 5: Search trace under DFS.

for which $\mathcal{A}$ searches through the entire graph.

4. Consider the graph shown in Figure 6. Let S be the initial state and G be the goal state. The cost of each action is as indicated.

   (a) Give a trace of uniform-cost search.

   (b) When A generates G which is the goal with a path cost of 11, why doesn't the algorithm halt and return the search result since the goal has been found? With your observation, discuss how uniform-cost search ensures that the shortest path solution is selected.

Solution: If we apply the `TREE-SEARCH` algorithm, the following are the nodes in frontier at the beginning of the loop and at the end of each iteration of the loop (A node n is followed by its path cost in parenthesis):

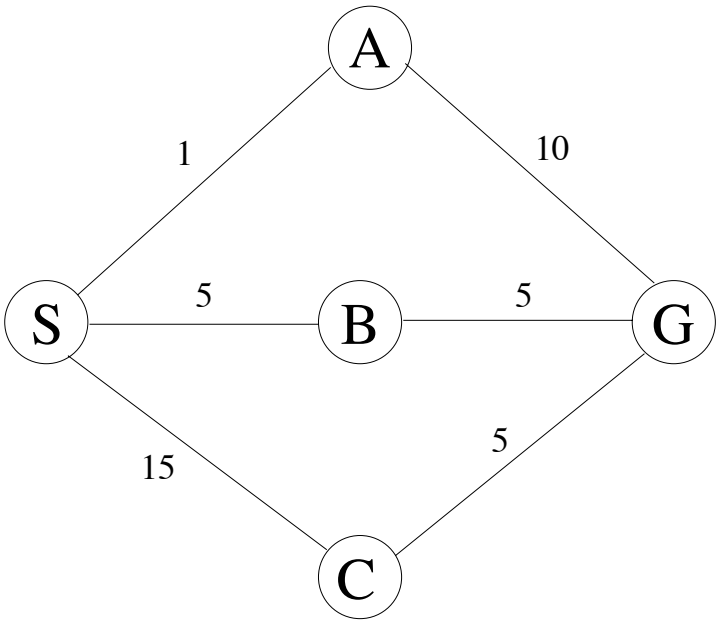

Figure 6: Graph of routes between S and G.

frontier
S(0)
A(1) B(5) C(15)
S(2) B(5) G(11) C(15)
A(3) B(5) B(7) G(11) C(15) C(17)
S(4) B(5) B(7) G(11) G(13) C(15) C(17)
B(5) A(5) B(7) B(9) G(11) G(13) C(15) C(17) C(19)
A(5) B(7) B(9) G(10) S(10) G(11) G(13) C(15) C(17) C(19)
⋮

which is somewhat cumbersome. So, instead, we try the `GRAPH-SEARCH` algorithm, which is modified to include an extra check in case a shorter path to a frontier state is discovered (see Fig. 3.14 on page 84 of AIMA 3rd edition). We obtain the following, which shows the nodes in frontier and explored at the beginning of the loop and at the end of each iteration of the loop (A node n is followed by its path cost in parenthesis):

frontier	explored
S(0)	
A(1) B(5) C(15)	S
B(5) G(11) C(15)	S, A
G(10) C(15)	S, A, B

After G is generated, it will be sorted together with the other nodes in frontier. Since node B has lower path cost than G, it is then selected for expansion. Node B then expands to G with a path cost of 10, which gives the solution. By always selecting the node with the least path cost for expansion (i.e., nodes are expanded in increasing order of path cost), uniform-cost search ensures that the first goal node selected for expansion is an optimal (i.e., shortest path) solution. A more detailed proof is provided on page 85 of AIMA 3rd edition; we discuss a more general case in class.