Artificial Intelligence

Augustin Cosse.

Fall 2020

September 27, 2021

Problem solving performance

- We usually evaluate the performance of an algorithm according to four criteria:
  - **Completeness**: Is the algorithm garanteed to find a solution when there is one ?
  - **Optimality**: Does the algorithm find the optimal solution ?
  - **Time complexity**: How long does it take to find a solution?
  - **Space complexity**: How much memeory is needed for perform the search?

Problem solving performance

Criterion	Breadth First	Uniform Cost	Depth - First	Depth - Limited	Iterative Deepening	Bidirectional
Complete?	Yes	Yes	No	No	Yes	Yes
Time	$O(b^d)$	$O(b^{1+\lfloor C^*/\varepsilon \rfloor})$	$O(b^m)$	$O(b^\ell)$	$O(b^d)$	$O(b^{d/2})$
Space	$O(b^d)$	$O(b^{1+\lfloor C^*/\varepsilon \rfloor})$	$O(bm)$	$O(b\ell)$	$O(bd)$	$O(d^{d/2})$
Optimal?	Yes	Yes	No	No	Yes	Yes

- Recall that b is the branching factor (number of children per node), m is the max depth of the tree and d is the depth of the first solution.
- The difference between BFS and DFS lies in the loopy paths that can appear in the tree search version of the algorithm. if there is a solution at finite depth d, BFS will ultimately find it because it escapes loops in its tree search version. the tree version of DFS on the other hand might get stuck in a loop until it reaches the max depth of the tree.

Optimality of A^*

- Recall that we call a heuristic admissible if it never overestimates the cost of reaching the goal (i.e. $h(n) \leq h^*(n)$) and we call it consistent if for every node n and every successor n' generated by any action a, we have $h(n) \leq c(n, a, n') + h(n')$

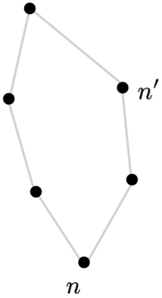
- We will show that the graph search version of A^* is optimal if $h(n)$ is consistent. To see this, note that :
  - For any successor n' of n, we have $g(n') = g(n) + c(n, a, n')$ for some action a
  - From the definiition of $f(n)$, we also have

$$f(n') = g(n') + h(n') = g(n) + c(n, a, n') + h(n') \geq g(n) + h(n)$$

- From this we see that consistency implies non decreasing value of f along a path followed by A^*.

Optimality of A^*

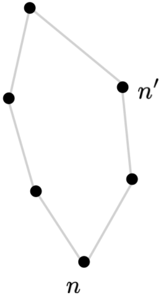
- Then note that whenever A^* selects a node for expansion, the optimal path to that node has been found. if this was not the case, that means there is another node n' in the frontier, that lies on the optimal path. However, for this node we must have $f(n') \leq f(n)$ since f is non decreasing. But this is impossible as it would mean that n' should have been expanded before n.



Optimality of A^*

- From those properties we see that the first goal node that is reached by A^* must necessarily be optimal as all other goal nodes will have a value of $f(n)$ that is at least as large.

- Note that for the goal nodes, $f(n) = g(n)$ (which as we saw is the cost of the optimal path to node n)



Optimality of A^*

- if C^* denotes the cost of the optimal solution path, then we can say
  - A^* expands all nodes with $f(n) < C^*$
  - A^* might then expand some of the nodes right on the "goal contour" (where $f(n) = C^*$) before selecting a goal node

Uniform cost search

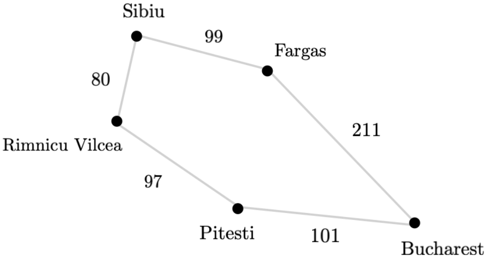
- When all step costs are equal, BFS is optimal because it always expands the shallowest unexpanded node.
- By a simple extension, we can design an algorithm that remains optimal for any step-cost function.
- Instead of expanding the shallowest node, **uniform-cost search** expands the node n with the lowest path cost $g(n)$. This is done by storing the frontier as a priority queue ordered by $g(n)$
- In our version of Uniform cost search, we add two components on top of BFS. The first difference is that the goal test is applied to a node when is selected for expansion and not when it is first generated (this is because the first goal node generated may be on a suboptimal path). The second difference is that a test is added to discard a node in frontier in case a better path is found to that node.

Uniform-Cost-Search

```
Function Uniform-Cost-Search(problem) returns  solution or failure ;
node  ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0;
frontier ← a priority queue ordered by PATH-COST, with node as the only element;
explored ← an empty set;
while EMPTY?(frontier) is false do
    node ← POP(frontier) /* choose lowest cost node in frontier */;
    if problem.GOAL-TEST(node.STATE) then
        return SOLUTION(node)
    end
    ;
    add node.STATE to explored;
    for  each  action  problem.ACTIONS(node.STATE) do
        child ← CHILD-NODE(problem, node, action);
        if child.STATE is not in explored or frontier then
            frontier ← INSERT(child, frontier)
        end
        else if child.STATE is in frontier with higher PATH-COST then
            replace that frontier node with child
        end
    end
end
```

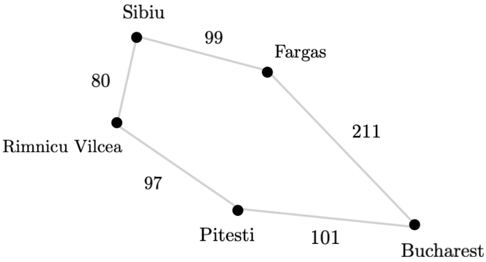
Uniform cost search

- Consider the simple illustration below. In this case, our problem is to go from Sibiu to Bucharest. The successors of Sibiu are Rimnicu Vilcea and Fagaras with costs 80 and 99 respectively. The least cost node, Rimnicu Vilcea is expanded next, adding Pitesti to the queue with total cost $80 + 97 = 177$. The least cost node is now Fagaras, so it is expanded adding Bucharest with total cost $99 + 211 = 310$.



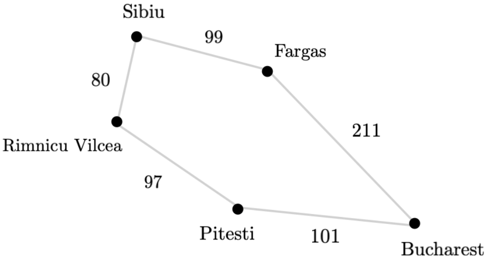
Uniform cost search

- A goal node has been generated but not yet expanded. UNIFORM COST SEARCH continues to search for better path costs and gets back to Pitesti for expansion, adding a second path to Bucharest with total cost $80 + 97 + 101 = 278$.



Uniform cost search

- The algorithm then checks to see if this new path is better than the old one. Since it is, the old path is discarded.



Constraint Satisfaction Problems (CSPs)

- So far we have explored the idea that our problems could be solved by searching in a space of states.
- We will now study how how to solve problems more efficiently using a **factored representation** for each state (that is a set of variables, each of which has an associated value)
- A problem is then considered to be solved when each variable has a value that satisfies all the constraints on the variables
- A problem described this way is called a **constraint satisfaction problem** or CSP

Constraint Satisfaction Problems (CSPs)

- A constraint satisfaction problem consists of three components, $\boldsymbol{X}$, $\boldsymbol{D}$ and $\boldsymbol{C}$ where
  - $\boldsymbol{X}$ is a set of variables $\{X_1, x_2, \ldots, X_n\}$
  - $\boldsymbol{D}$ is a set of domains $\{D_1, \ldots, D_n\}$, one for each variable
  - $\boldsymbol{C}$ is a set of constraints that specify allowable sets of values
- Each domain D_i consists of a set of allowable values $\{v_1, \ldots, v_k\}$ for variable X_i and each constraint C_i consists of a pair $\langle \text{scope}, \text{rel} \rangle$ where scope is a tuple of variables that participate in the constraint and rel is a relation that defines the values that those variables can take on.

Constraint Satisfaction Problems (CSPs)

- relation can be represented as an explicit list of all tuples of values that satisfy the constraints, or as an abstract operation that supports two operations: (1) testing if a tuple is a member of the relation and (2) enumerating all members of the relation.
- E.g. if X_1 and X_2 both have the domain $\{A, B\}$, then the constraintsaying the two variables must have different values can read either as $\langle (X_1, X_2), [(A, B), (B, A)] \rangle$ or as $\langle \langle (X_1, X_2), X_1 \neq X_2 \rangle \rangle$
- To solve a CSP, we need to define a state space and a notion of solution

Constraint Satisfaction Problems (CSPs)

- Each state in a CSP is defined by an assignment of values to some or all the variables, i.e. $\{X_i = v_i, X_j = v_j, \ldots\}$
- An assignment that does not violate any constraint is called a consistent or legal assignment
- A complete assignment is one in which every variable is assigned, and a solution to a CSP is a consistent, complete assignment
- A partial assignment is one that assigns values to only some of the variables.

Constraint Satisfaction Problems (CSPs)

- An example of constraint satsfaction problem is the Map coloring problem, an instance of which is represented below.
- In this case, we are given the task of coloring each region either in red, green or blue, such that no neighboring regions have the same color.

Constraint Satisfaction Problems (CSPs)

- To formulate this problem as a CSP, we can define the variables to represent the regions

$$\boldsymbol{X} = \{WA, NT, Q, NSW, V, SA, T\}$$

- The domain of each variable is the set $D_i = \{\text{red, green, blue}\}$

Constraint Satisfaction Problems (CSPs)

- The constraints require neighboring regions to have distinct colors. Since there are nine places where regions border, there are nine constraints:

$$C = \{SA \neq WA, SA \neq NT, SA \neq Q, SA \neq NSW, SA \neq V, \\ WA \neq NT, NT \neq Q, Q \neq NSW, NSW \neq V\}$$

Constraint Satisfaction Problems (CSPs)

- Note that here, we use $SA \neq WA$ as a shortcut for $\langle (SA, WA), SA \neq WA \rangle$ where $SA \neq WA$ can be fully enumerated in turn as

$$\{(red, green), (red, blue), (green, red),$$
$$(green, blue), (blue, red), (blue, green)\}$$

Constraint Satisfaction Problems (CSPs)

- An we have several solutions for this constraint given by

$$\{WA = red, NT = green, Q = red, NSW = green, \\ V = red, SA = blue, T = red\}$$

- The constraint graph representation (nodes = variables and edges represent the existence of a constraint involving the two nodes) of the problem is given below.

Constraint Satisfaction Problems (CSPs)

- CSP yield a natural representation for a wide variety of problems.
- Moreover, CSP solvers can be faster than state space searchers because the CSP solver can quickly eliminate large swatches of the search space
- As an example, if we have chosen $\{SA = blue\}$, we know that none of the 5 neighboring variables can take the value *blue*.

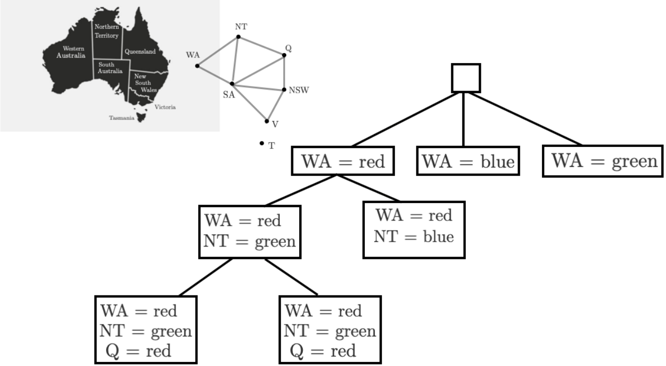
Constraint Satisfaction Problems (CSPs)

- The simplest kind of CSPs involve variables that have discrete, finite domains (Map coloring problems and scheduling with time limits are both of this kind)
-

Backtracking search for CSPs

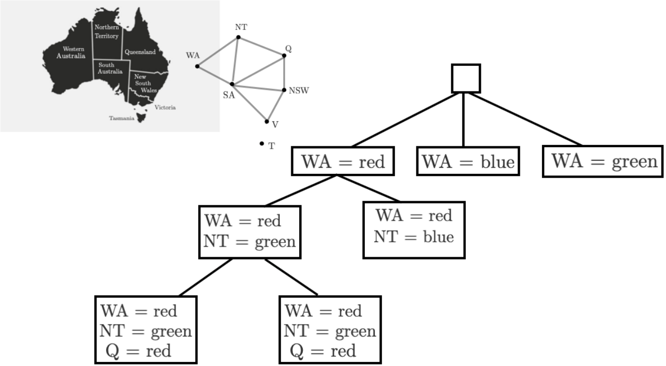
- A crucial property common to all CSPs is **commutativity** i.e. the fact that the order of application of any given set of actions has no effect on the outcome.

- CSPs are commutative because when assigning values to variables, we reach the same partial assignment regardless of the order.



Backtracking search for CSPs

- Following from those ideas, we can just consider a single variable at each node as shown below



WA = red

WA = blue

WA = green

WA = red
NT = green

WA = red
NT = blue

WA = red
NT = green
Q = red

WA = red
NT = green
Q = red

Backtracking search for CSPs

- The CSP can then be solved through backtracking search. Recall that the name backtracking was used to denote a depth first search that uses one variable at a time and backtracks when a variable has no legal value left to assign.
- For CSP, backtracking repeatedly chooses an unassigned variable, then tries all values in the domain of that variable in turn, trying to find a solution
- If an inconsistency is detected, it returns a failure
- Note that BACKTRACKING SEARCH keeps only a single representation of a state and alters that representation rather than creating new ones.

Backtracking search for CSPs

```
function BACKTRACKING-SEARCH(csp) returns a solution, or failure ;
return BACKTRACK({} , csp)
```

```
function BACKTRACK(assignment, csp) returns a solution, or failure ;
if assignment is complete then
  |  return assignment
end
var ← SELECT-UNASSIGNED-VARIABLE(csp, assignment)
/*see next slide*/
```

Backtracking search for CSPs

```
function BackTrack(assignment, csp) returns  a solution, or failure ;
/*see previous slide*/
for each value in
ORDER-DOMAIN-VALUES(var, assignment, csp) do
    if value is consistent with assignment then
        add $\{var = value\}$ to assignment
        inferences $\leftarrow$ INFERENCE(csp, var, assignment)
        if  inferences $\neq$ failure then
            add inferences to assignment
            result $\leftarrow$ BACKTRACK(assignment, csp)
            if result $\neq$ failure then
                return result
            end
        end
    end
    remove $\{var = value\}$ and inferences from assignment
end
return failure
```

Inference in CSPs

- In CSPs there is a choice. an algorithm can either search (choose a new variable assignment from seevral possibilities) or do a specific type of **inference** called **constraint propagation**
- The idea of **constraint propagation** is to use the constraints to reduce the number of legal values for a variable which in turn can reduce the legal values for another variable ans so on.
- Sometimes this preprocessing can solve the whole problem so that no search is needed at all

Inference in CSPs

- The key idea behind constraint propagation is the notion of local consistency. There are different types of local consistency
  - Node consistency
  - Arc consistency
  - Path consistency
  - K-consistency

Node Consistency

- A single variable is Node consistent if all the values in the variable's domain satisfy the variable unary constraints

- As an example, in the map coloring problem, if we enforced the constraint that **South Australia** could not be colored in green and started with the domain $\{\text{red}, \text{green}, \text{blue}\}$

- It could then be made node consistent by eliminating green from the domain, thus leaving **South Australia** with the reduced domain $\{\text{red}, \text{blue}\}$

- It is always possible to eliminate all the unary constraints in a CSP by running node consistency

- Note that it is also possible to transform all *n*-ary constraints into binary ones. For this reason, it is common to define CSP solvers as solvers working with binary constraints only.

Arc Consistency

- A variable in CSP is arc-consistent if every value in its domain satisfies the variable's binary constraints.

- More formally, X_i is arc-consistent with respect to another variable X_j if for every value in the current domain D_i, there is some value in the domain D_j that satisfies the binary constraint on the arc (X_i, X_j)

- A network is arc-consistent if every variable is arc-consistent with every other variable.

- As an example, consider the constraint $Y = X^2$ where the domain of both X and Y is the set of digits. This constraint can be written explicitely as

  $$\langle (X, Y), \{(0,0), (1,1), (2,4), (3,9)\} \rangle$$

  To make X arc-consistent with Y, we reduced X's domain to $\{0, 1, 2, 3\}$