Property	Breadth-First Search (BFS)	Uniform-Cost Search (UCS)	Depth-First Search (DFS)	Depth-Limited Search (DLS)	Iterative-Deepening Search (IDS)
`Completeness`	**YES** (if b is finite)	**YES** (if b is finite, $cost \geq \epsilon > 0$)	NO	NO	NO
`Optimal`	NO	**YES** (Proof below)	NO	NO	NO
`Time`	$\mathcal{O}(b^{d+1})$	$\mathcal{O}(b^{1+\lfloor C^*/\epsilon \rfloor})$	$\mathcal{O}(b^{m+1})$	$\mathcal{O}(b^{\ell+1})$	$\mathcal{O}(b^{d+1})$
`Space`	$\mathcal{O}(b^{d+1})$	$\mathcal{O}(b^{1+\lfloor C^*/\epsilon \rfloor})$	$\mathcal{O}(bm)$	$\mathcal{O}(b\ell)$	$\mathcal{O}(bd)$
Frontier	Queue (FIFO)	Priority Queue	Stack (LIFO)	Stack	-

Notation: b: max num of successor (branching) of any node (maybe ∞); d: depth of (shallowest) goal node; m: max depth of a node from start node; C^*: optimal cost, $cost \geq \epsilon$

Definition: `Complete`: always find a solution; `Optimal`: find a least-cost solution; `Time C.`: number of nodes generated; `Space C.`: max number of nodes in memory

(1) State Space: A state is a representation of a configuration of the problem domain.

(2) Initial State: The starting configuration.

(3) Goal State: The configuration one wants to achieve.

(4) Actions: (or State Space Transitions): Allowed changes to move from one state to another

(1) Costs: Representing the cost of moving from state to state.

(2) Heuristics: Help guide the heuristic process.

Example of Sudoku Representation:

(1) State in this problem is a (partial) valid solution of the Sudoku puzzle. More formally, it would be a matrix $M \in \{0, ..., 9\}^{99}$, with 0 representing a blank square.

(2) Initial state is a completed filled out grid of numbers. All rows, columns, and 3×3 squares should contain all numbers between $1, ..., 9$. Any state in the problem will be a valid goal state.

(3) Action would be removing a number from the grid. More formally, we take as input a matrix M and a matrix $E_{i,j}(a)$, where $E_{i,j}(a) \in \{0, ..., 9\}^{99}$ is a matrix of all zeroes except for a nonzero value $a \in \{1, ..., 9\}$ in coordinate (i, j). An action would be setting $M - E_{i,j}(a)$.

(4) Transition model would be $T(M, E_{i,j}(a)) = M - E_{i,j}(a)$.

Path Checking

A path p_k is represented as a tuple of states $\langle s_0, s_1, ..., s_k \rangle$, where s_k are states, Suppose s_k expanded to obtain a child success state c, $\langle s_0, s_1, ..., s_k, c \rangle$ as $\langle p_k, c \rangle$

In every path $\langle, p_k, c \rangle$, ensure that the final state c is not equal to any ancestors of c along this path: $c \notin \{s_0, s_1, ..., s_k\}$ (make sure does not go back). Does not increase time and space complexity, while does not prune all redundant states

Cycle Checking

Keep track of all nodes previously expanded during the search using a list (close list). When expand n_k to obtain successor c: (1) Ensure c is not equal to any previously expanded node (2) It it is do do not add c to Frontier

BFS

Proof that BFS is Complete: If the shallowest goal node is at some finite depth d, breadth-first search will eventually find it after generating all shallower nodes (provided the branching factor b is finite).

Let path $\langle s_0, s_1, ..., s_k, s_g \rangle$ be the optimal path from s_0 to s_g

Base case: $k + 1 = 0 \implies s_0 = s_g$

Induction Hypothesis: BFS will find a path to all nodes with path length $< k + 1$

Induction Step: assume start at s_k, s_g will be explored as successor of s_k

UCS

Expands the node n with the lowest path cost $g(n)$

Proof that USC is Optimal Finds optimal solution if each transition has $cost \geq \epsilon > 0$

(1) Let $c(n)$ be the cost of path to node n, if n_2 is expanded after n_1 then $c(n_1) \leq c(c_2)$:

a. n_2 was on the frontier when n_1 was expanded, in which case $c(n_2) \geq c(n_1)$ else n_1 would not have been selected for expansion

b. n_2 was added to the frontier when n_1 was expanded, in which case $c(n_2) \geq c(n_1)$ since the path to n_2 extends to the path to n_1

(2) When n is expanded every path with cost strictly less than $c(n)$ has already expand before n

Let $\langle n_0, n_1, ..., n_k \rangle$ be a path with cost less than $c(n)$, let n_i be the last node on this path that has been expanded: $\langle n_0, n_1, ..., n_i, n_{i+1}, n_k \rangle$

So n_{i+1} must still be on the frontier, also $c(n_{i_1}) < c(n)$ since the cost of the entire path to n_k is $< c(n)$

But then, UCS would have expanded n_{i+1}, not n

So every node on this path already be expanded, QED

(3) The first time UCS expand a state, s it has found the minimal cost path to it

No cheaper path exist, else would have been expanded before

No cheaper path will be discovered later, as all those path must be at least as expensive

So when a goal state is expanded, the path to it must be optimal

Uninformed Search vs. Informed Search

Informed search :*domain specific* heuristic function $h(n)$ that guesses the cost of to goal from n

(1) $h(n_1) < h(n_2)$ implies that it is cheaper to get to goal node from n_1 than from n_2

(2) $h(n)$ is a function only of the state of n (use state, ratter than node as argument of h)

(3) h must be defined so that $h(n) = 0$ for every goal node

Greedy First Search: $f(n) = h(n)$

A* Search

Define an evaluation function $f(n)$ such that $f(n) = g(n) + h(n)$

(1) $g(n)$: the cost of the path to n

(2) $h(n)$: the heuristic estimate of the cost of achieving the goal from n

Proof of Completeness:

Theorem 1: A* will always find a solution if one exist as long as (branching is finite AND action has finite $cost \geq \epsilon > 0$):

(1) $h(n)$ is finite for every node n that can be extended to reach a goal node, If a solution node n exist, then all times either:

a. n been expanded by A* or b. An ancestor of n is on the 'Frontier'

(2) Suppose (b) holds and let the ancestor on the 'Frontier' be n_i, then n_i must have a finite f-value

(3) As A* continue to run, the f-value of the nodes on the Frontier eventually increase; thus either A* terminates by finding solution OR n_i become the node on the Frontier with the lowest f-value

(4) If n_i expand, then either $n_i = n$ A* returns as solution OR n_i is replaced by its successors, one of which n_{n+1} is a closer ancestor of n

(5) Apply same argument to n_{n+1}, if A* continues to run, it will eventually expand every ancestor of n, including n itself and so finds and returns a solution

Proof of Optimal with Consistency: $\forall n_1, n_2, \quad h(n_1) \leq h(n_2) + C(n_1, a, n_2)$

WTS: $\hat{f}_{pop}(s_g) = f(s_i)$, when goal node is pooped, have found optimal

Let $\langle s_0, s_1, ..., s_{g-1}, s_g \rangle$ be the path from start node to goal node

Base case: $\hat{f}_{pop}(s_0) = f(s_0) - h(s_0)$

Induction step: Assume $\forall s_0, s_1, ..., s_k, \hat{f}_{pop}(s_i) = f(s_i)$, given

$$\begin{aligned} \hat{f}_{pop}(s_{k+1}) &= \hat{g}_{pop}(s_{k+1}) + h(s_{k+1}) \\ &\geq g(s_{k+1}) + h(s_{k+1}) \\ &= f(s_{k+1}) \end{aligned}$$

To make sure that each s_{k+1} is only explored after pooped s_k, require $f(s_i) \leq f(s_{k+1})$, leading to need of consistency of h, by pooping s_k

$$\begin{aligned} \hat{f}_{pop}(s_{k+1}) &= \min\{\hat{f}(s_{k+1}), \hat{g}_{pop}(s_k) + c(s_k, s_{k+1}) + h(s_{k+1})\} \\ &\leq \hat{g}_{pop}(s_k) + c(s_k, s_{k+1}) + h(s_{k+1}) \\ &= g(s_k) + c(s_k, s_{k+1}) + h(s_{k+1}) && \text{from IH} \\ &= g(s_{k+1}) + h(s_{k+1}) \\ &= f(s_{k+1}) \end{aligned}$$

Proof of Accessibility: $\forall n \quad h(n) \leq h^*(n)$

Proposition 1: $f(n) \leq c^*$: A* with an admissible heuristics never expands a node with f-value greater than the cost of an optimal solution

(1) Let C^* be the cost

(2) Let $p : \langle s_0, s_1, ..., s_k \rangle$ be an optimal solution, so $cost(p) = cost(\langle s_0, s_1, ...s_k \rangle) = C^*$

(3) Let n be a node reachable from the initial state and $n_0, n_2, ..., n_i, ...n$ be ancestors of n, so at least one of $n_0, n_2, ..., n_i, ...n$ is **always** on the 'Frontier'

(4) Want to show that with an admissible heuristic, for every prefix n_i of n, have $f(n_i) \leq C^*$

$$\begin{aligned} C^* &= cost(\langle s_0, s_1, ..., s_k \rangle) \\ &= cost(\langle s_0, s_1, ..., s_i \rangle) + cost(\langle s_i, ..., s_k \rangle) \\ &= g(n_i) + h^*(n_i) && h^*(n_i) \text{ is the c of op, since } p \text{ is optimal} \\ &\geq g(n_i) + h(n_i) && h^*(n_i) \geq h(n_i)\ h \text{ is admissible} \\ &= f(n_i) \end{aligned}$$

(5) Know that A always expands a node on the 'Frontier' that has lowest f-value, so every node A expands has f-value less than or equal to $f(n_i)$, which is less than or equal to C^*

Theorem 2 A* with an admissible heuristic always finds an optimal cost solution, as long as (branching is finite AND action has finite $cost \geq \epsilon > 0$)

(7) Let C^* be the cost of an optimal solution

(8) If a solution exist then by (completeness), A will terminate by expanding some solution node n

(9) According to Proposition 1, $f(n) \leq C^*$

(10) Since n is a goal node, have $h(n) = 0$, so $f(n) = g(n) + 0 = cost(n)$, thus $f(n) = cost(n) \leq C^*$

(11) Since no solution can have lower cost than the optimal: $C^* \leq cost(n) = f(n)$

(12) Therefore $cost(n) = C^*$, thus A returns a cost-optimal solution

Proof of Consistency Implies Admissible:

$(\forall n_1, n_2, a) \quad h(n_1) \leq C(n_1, a, n_2) + h(n_2) \implies (\forall n) \quad h(n) \leq h^*(n)$

Base case: $k = 1$, one step away from s_g, since consistent: $h(s_i) \geq C(s_i, s_g) + h(s_g)$, since $h(s_g) = 0$, $h(s_i) \geq C(s_i, s_g) = h^*(s_i)$, therefore admissible

Induction step: (1) Suppose assumption holds for every node that is $k-1$ action away from s_g, given a node s_i, it is k action away from s_g, thus optimal path has $k > 1$ steps

(2) Since h is consistent, have: $h(s_i) \leq C(s_i, s_{i+1}) + h(s_{i+1})$

(3) Note that s_{k+1} is on a least-cost path from s_i, must have the path s_{i+1} to s_g as well, by induction hypothesis have: $h(s_{i+1}) \leq h^*(s_{i+1})$

(4) Combine inequality: $h(s_i) \leq C(s_i, s_{i+1}) + h^*(s_{i+1})$

<u>A* search</u> $= \min(\hat{g}(u), \hat{g}(v) + C(u,v))$

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

Consistency: $h(S_i) \leq h(S_{i+1}) + C(S_i, S_{i+1})$

Admissibility: $h(S_i) \leq \text{Optimal}(S_i) = h^*(S_i)$

Proof of $C \Rightarrow A$: $h(S_i) \leq h(S_{i+1}) + C(S_i, S_{i+1})$

$(h(S_u) = 0)$ $\leq h(S_{i+2}) + C(S_{i+1}, S_{i+2}) + C(S_i, S_{i+1})$

$\leq h(S_u) + C(S_k, S_u) + \cdots + C(S_i, S_{i+1})$

$= C(S_k, S_u) + \cdots + C(S_i, S_{i+1}) = h^*(S_i)$

Proof: Consistency $\Rightarrow$ Optimality. W.T.P. $\hat{f}_{pop}(S_g) = f(S_g)$

given consistency: $h(S_i) \leq h(S_{i+1}) + C(S_i, S_{i+1})$

$\hookrightarrow$ admissibility: $h(S_i) \leq h^*(S_i)$

let path $s_0, s_1, s_2, \dots, s_{g-1}, s_g$ be the path from s_1 to s_g

B.C.: $\hat{f}_{pop}(S_0) = f(S_0) = h(S_0) = 0 \Rightarrow S_0 = S_u$. B.C. holds

I.H.: $\forall i \in 1, \dots, k$ $\hat{f}_{pop}(S_i) = f(S_i) \Rightarrow \hat{g}_{pop}(S_i) = g(S_i)$

I.S.: $\hat{f}_{pop}(S_{k+1}) \leq \min(\hat{f}(S_{k+1}), \hat{g}_{pop}(S_k) + C(S_k, S_{k+1}) + h(S_{k+1}))$

$= g(S_{k+1})$ $\leq \hat{g}_{pop}(S_k) + C(S_k, S_{k+1}) + h(S_{k+1})$

$= \underline{g(S_k) + C(S_k, S_{k+1})} + h(S_{k+1})$ By I.H. $\Rightarrow f(S_{k+1})$ ②

<u>IDS</u>

Time: $1 + b^1 + b^2 + \dots + b^d = \frac{b^{d+1} - 1}{b - 1} \in O(b^{d+1})$

Space: at worst case, store $b-1$ siblings and d successors $\Rightarrow b - 1 + bd \in O(bd)$

Proof of $C \Rightarrow A$: induction on $k(S_i)$: # of actions to reach S_g

B.C. $k = 1$: $h(S_i) \leq C(S_i, S_g) + h(S_g) = C(S_i, S_g) = h^*$

I.H. assume holds for nodes $\leq k-1$ action away

By def. $h(S_i) \leq C(S_i, S_{i+1}) + h(S_{i+1})$ ①

By I.H. $h(S_{i+1}) \leq h^*(S_{i+1})$ ②

①+②: $h(S_i) \leq C(S_i, S_{i+1}) + h^*(S_{i+1})$

Therefore $S_i, S_{i+1}, \dots, S_g$ is the optimal path from S_i to S_g and h is admissible

By definition: $\hat{f}_{pop}(S_{k+1}) = \hat{g}_{pop}(S_{k+1}) + h(S_{k+1})$

$\geq g(S_{k+1}) + h(S_{k+1})$

$= f(S_{k+1})$ ①

Because ①+②: $\hat{f}_{pop}(S_{k+1}) = f(S_{k+1})$

Constraint Satisfaction Problems

(1) A set of variables $V_1, ..., V_n$;

(2) A (finite) domain of possible values $Dom[V_i]$ for each variable V_i

(3) A set of constraints $C_1, ..., C_m$,

(4) Each variable V_i can be assigned any value from its domain: $Vi = dwhere d \in Dom[Vi]$

(5) Each constraint C Has a set of variables it operates over, called its scope.

Example: The scope of $C(V1, V2, V4) = -V1, V2, V4$

Given an assignment to variables C is True if the assignment satisfies the constraint; False if the assignment falsifies the constraint.

(6) Solution to a CSP: An assignment of a value to all of the variables such that every constraint is satisfied.

A CSP is unsatisfiable if no solution exists.

Back Tracking Search

Searching through the space of partial assignments, rather than paths. Decide on a suitable value for one variable at a time. Order in which we assign the variables does not matter. If a constraint is falsified during the process of partial assignment, immediately reject the current partial assignment.

(1) Root: Empty Assignment.

(2) Children of a node: all possible value assignments for a particular unassigned variable.

(3) The tree stops descending if an assignment violates a constraint.

(4) Goal Node: a. The assignment is complete b. No constraints is violated.

Back Tracking with Inference

(1) Every time we assign a value to a variable V , check all constraints over V and prune values from the current domain of the unassigned variables of the constraints.

Let C be a constraint that includes V in its scope and V_i be a variable (other than V) in the scope of C.

All values in the current domain of Vi must have a supporting assignment. If a value doesn't have any supports, it must be pruned.

(2) A value d in the current domain of V_i has a supporting assignment if there exist at least one value assignment for other variables of C such that C is satisfied under $V_i = d$ and that value assignment.

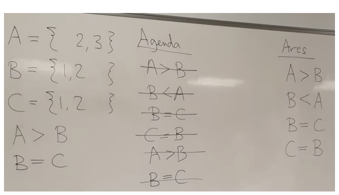
(3) Removing a value from a variable domain may remove a support for other domain values.

(4) Repeat the procedure until all remaining values have a support:

Have a queue of variables that need to be checked.

A variables is added (back) to the queue if its domain is changed.

The procedure stops when the queue is empty.



Consider the *item allocation problem.* We have a group of people $N = \{1, \ldots, n\}$, and a group of items $G = \{g_1, \ldots, g_m\}$. Each person $i \in N$ has a utility function $u_i : G \to \mathbb{R}^+$. The constraint is that every person is assigned *at most one item*, and each item is assigned to *at most one person.* An allocation simply says which person gets which item (if any).

In what follows, you *must* use only the binary variables $x_{i,j} \in \{0, 1\}$, where $x_{i,j} = 1$ if person i receives the good g_j, and is 0 otherwise.

(a) Write out the constraints:

i. each person receives no more than one item

Solution:

$$\forall i \in N : \sum_{g_j \in G} x_{i,j} \leq 1$$

ii. each item goes to at most one person

Solution:

$$\forall g_j \in G : \sum_{i \in N} x_{i,j} \leq 1$$

using only the variables $x_{i,j}$.[1]

(b) Suppose that people were divided into *disjoint types* $N_1, \ldots, N_k$ (e.g. say, genders or ethnicities), whereas items were divided into *disjoint blocks* $G_1, \ldots, G_l$. We further require that each N_p only be allowed to take no more than λ_{pq} items from block G_q.

Write out this constraint using the variables $x_{i,j}$. (Note that each N_i corresponds to the set of people who are of that person-type.)

Solution:

$$\forall p \in \{1, ..., k\}, q \in \{1, ..., l\} : \sum_{i \in N_p} \sum_{g_j \in G_q} x_{i,j} \leq \lambda_{pq}$$

(c) We say that player i *envies* player i' if the utility that player i has from their assigned iter strictly lower than the utility that player i has from the item assigned to player i'.

Write out the constraints that ensure that in the allocation, no player envies any other pla You may assume that the validity constraints from (a) hold.

Solution: Note that for this constraint, the definition requires that the allocation is valid, so will need to add the constraints from (a) to make either of the definitions below meaningful.

$$\forall i, i' \in N, \forall g_j, g_{j'} \in G : (x_{i,j} \wedge x_{i',j'}) \implies u_i(g_j) \geq u_i(g_{j'})$$

Hill-climbing

greedily choose the state with a better value ⇒ local max

Simulated annealing

allowing some suboptimal state to get out of local max, but still not guarantee completeness

CSP

① Variables: $X = \{x_1, x_2, \ldots, x_n\}$

② Domains: $D \in \{D_1, D_2, \ldots, D_n\}$ where D_i = domain (x_i)

③ Constraints

Backtracking

assign each variable to the first possible value in domain and check the violation of constraints from the back

Cons: expensive checking: (64 checks for 4 Queen)

Improved

check the assigned value is consistent before assigning next variable

With inference

After assigning each variable, infer on its constraints and put the affected variable into inference queue

If each variable has empty domain, the variable assigned need to be reassigned

· modified inference:

- forward checking: no inference queue
- only append variable with $|T|=1$: most constrained

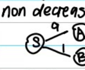
Sample Midterm:

<1> What if $h()$ does not satisfy $h(n) \leq h^*(n)$ for all states?

A: may favor suboptimal nodes

Tutorials:

<2> Why BFS is not optimal with only non decreasing cost in each depth?



A, B both goal states

tie-breaking first A

<3> Proof of UCS optimality with each step cost > ε

Because step cost > ε ⇒ completeness

whenever n is expanded, the optimal path to n has been found.

By contradiction, if the optimal path is not found, there would be n' on the optimal path

By definition, $g(n') < g(n)$, then it should have been selected ⇒ contradicts

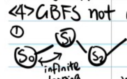
Also, as step cost > ε ⇒ path never gets shorter as nodes are added.

<3> UCS search format

(Node, Path)	Frontier → f-val
	{(S,0)}
(S,S)	{(SP,1), (SD,3), (SE,9)}

<4> GBFS not complete w/out cycle checking

①

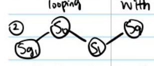


$h(S_0)=3$ $h(S_2)=5$

$h(S_1)=4$ $h(S_g)=0$

with or w/out cycle checking

②



$h(S_0)=9$ $h(S_1)=1$

$h(S_g)=0$ $h(S_{g'})=0$

<5> backtracking format

Current Assignment	Constraint Violation
X=1	
X=1, Y=1	which violates

<6> constraint format

1. $\forall i \in N, \sum_{g_j \in G} x_{i,j} \leq 1$
2. $\forall p \in \{1, \ldots, k\}, q \in \{1, \ldots, l\}: \sum_{i \in N_p} \sum_{g_j \in G} x_{i,j} \leq \lambda_{pq}$
3. $\forall i, i' \in N, \forall g_j, g_{j'} \in G: (x_{i,j} \wedge x_{i',j'}) \Rightarrow u_i(g_j) \geq u_i(g_{j'})$

<7> why using most constrained variable & least constrained value in CSP?

MCV: likely to cause failure → more efficient

LCV: less likely to cause conflicts & backtracking

<8> if $h(n) = h^*(n)$ for all n ⇒ whenever A* expands a node n, n must lie on the optimal path to a goal.

Proof by contradiction:

suppose A* expands a node n' not on the optimal path, then it means that there is another path that is more optimal which contains n', and $f(n') < f(n)$ (by definition).

However it contradicts the fact that n is already on the most optimal path ($h^*(n)$)

Therefore A* always expand optimal node

7. Suppose that we are running a variation of A^* which makes use of an admissible heuristic. Unlike regular A^*, at each iteration in this variation we will select a node from the Frontier at random that has an f-value less than or equal to $(1 + Q) * min_{n \in Frontier} f(n)$ where Q is some positive non-zero constant.

(a) Will this search be **complete**? In 2 sentences or less, explain.

Solution: *Yes, under the same conditions that $A*$ is complete. Finite cost paths will be extracted in order of priority. There are a finite number of paths that can have costs bounded by $(1 + Q) * C^*$.*

(b) Will this search be **optimal**? If C^* is the optimal cost from the start to a goal state, what is the **worst** cost solution this formulation of A^* would yield? Justify your answer in the space below.

Solution: *The search won't be optimal but there is an upper bound on the worst cost solution it will yield. The optimal path to the goal will have an f-value that equal to C^*; we may have to fully exhaust the f-contour containing all paths less than C^* before we find it. If we find a sub-optimal path en route we'll accept it, but this path won't be any higher in cost than $(1 + Q) * C^*$.*

(c) Can you suggest a way to modify this search in order to generate an **optimal** solution every time? In 2 sentences or less, explain your modification(s).

Solution: *There's more than one solution here. You could iterate until you exhaust the f-contour containing the first goal you find. If you find other solutions, the one with the shortest cost will be the optimal one. Or, just subtract Q from every heuristic estimate. It works!*

Consider the following CSP: There are three different musicians: **John, Mark,** and **Sam.** They each come from a different country; one comes from the **United States**, one from **Australia**, and one from **Japan**. They each play a different musical instrument; one plays the **piano**, one the **saxophone**, and one the **violin.** We also have the following information:

- The pianist plays first.
- John plays the saxophone and plays before the Australian.
- Mark comes from the United States and plays before the violinist.

(a) Formulate this problem as a CSP in order the answer the following questions. **1.** What is the order the instruments are played. **2.** Who plays what instrument. **3.** What is the nationality of each player.

Solution: We will use values $\{1, 2, 3\}$ to denote which person/instrument/nationality goes first, second, and third.

- Variables: *john, mark, sam, violin, sax, piano, aust, us, japan.*
- Domains: $D_i = \{1, 2, 3\}$
- Constraints:
  - $john \neq mark$, $john \neq sam$, $sam \neq mark$
  - $violin \neq sax$, $violin \neq piano$, $sax \neq piano$
  - $aust \neq us$, $aust \neq japan$, $us \neq japan$
  - $piano = 1$
  - $john = sax$, $john < aust$
  - $mark = us$, $mark < violin$

(b) Execute plain backtracking to find a solution to this CSP. At each step, you are free to pick any unassigned variable, and you can try values from its domain in any order.

Solution: Given this valid assignment, we see that Mark from the US plays the piano, John from

Current Assignment	Violations
piano = 1,	
piano = 1, sax = 2,	
piano = 1, sax = 2, violin = 3,	
piano = 1, sax= 2, violin = 3, mark = 1	
piano = 1, sax= 2, violin = 3, mark = 1, john = 1	$mark \neq john$
piano = 1, sax= 2, violin = 3, mark = 1, john = 2,	
piano = 1, sax= 2, violin = 3, mark = 1, john = 2, sam = 3,	
piano = 1, sax= 2, violin = 3, mark = 1, john = 2, sam = 3, us = 2	$mark = us$
piano = 1, sax= 2, violin = 3, mark = 1, john = 2, sam = 3, us = 1	
piano = 1, sax= 2, violin = 3, mark = 1, john = 2, sam = 3, us = 1, aust = 1	$us \neq aust$
piano = 1, sax= 2, violin = 3, mark = 1, john = 2, sam = 3, us = 1, aust = 3	
piano = 1, sax= 2, violin = 3, mark = 1, john = 2, sam = 3, us = 1, aust = 3, japan = 2	

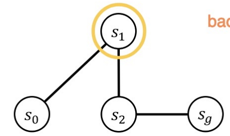
5. (a) Provide a counter-example to show that the **Greedy Best-First Search** algorithm without cycle-checking is incomplete.

Solution: Consider the following search space, where:

$$h(s_0) = 3$$
$$h(s_1) = 4$$
$$h(s_2) = 5$$
$$h(s_g) = 0$$

Greedy best-first:
evaluation f = heuristic h
(ignores weights)

bad heuristic + no cycle checking
= incomplete



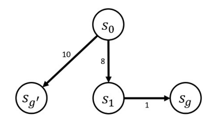
Each time s_0 is explored, we add s_1 to the front of the frontier, and each time s_1 is explored, we add s_0 to the front of the frontier. Notice that s_2 is never at the front of the frontier. This causes the greedy best-first search algorithm to continuously loop over s_0 and s_1.

(b) Provide a counter-example to show that Greedy Best-First Search (with or without cycle-checking) is not optimal.

Solution: Consider the following search space, where:

$$h(s_0) = 9$$
$$h(s_1) = 1$$
$$h(s_g) = 0$$
$$h(s_{g'}) = 0$$

Greedy best-first:
evaluation f = heuristic h
(ignores weights)



With either variant of the greedy best-first search algorithm, when s_0 is explored, $s_{g'}$ would be added to the front of the frontier and then explored next, resulting in tl
non-optimal $s_0 \to s_{g'}$ path.

(only where path ends matters)

2. Prove that the A^* **Search** algorithm with cycle-checking is optimal when a consistent heuristic is utilized.

2

A* with cycle-checking is optimal
1. Optimal path: f^pop(sg) = f(sg)
2. Induction Hypothesis: f^pop(s) = f(s)
3. Induction step: s, sk+1

g = sum c
along optimal path

Solution: The proof uses the following notation: The function $\hat{g}(n)$ denotes the current best known path cost from the initial state to node n. Note that the value of $\hat{g}(n)$ is updated during the execution of the algorithm. The function $g(n)$ denotes the minimum path cost from an initial state to state n.

The function $h(n)$, known as a heuristic, denotes the approximated path cost from state n to the (nearest) goal state.

The functions $f(n)$ and $\hat{f}(n)$ are evaluation functions that are adopted by the informed search algorithm in question. For example, in the case of the greedy best-first search algorithm, $f(n) = \hat{f}(n) = h(n)$; in the case of the A^* search algorithm, $f(n) = g(n) + h(n)$ and $\hat{f}(n) = \hat{g}(n) + h(n)$. $\hat{f}_{\text{pop}}(n)$ denotes the value of $\hat{f}(n)$ when it is popped from the frontier.

^: estimate to be updated
(heuristic is fixed)

Proof: Mathematically, we would want to prove that $\hat{f}_{\text{pop}}(s_g) = f(s_g)$, i.e. when the goal node s_g is popped from the frontier, we would have found the optimal path to it. Let

$$s_0, s_1, \ldots, s_{g-1}, s_g$$

be the path from the start node s_0 leading to the goal node s_g.

Base case: $\hat{f}_{\text{pop}}(s_0) = f(s_0) = h(s_0)$.

Induction step: Assume that for all $s_0, s_1, \ldots, s_k$, $\hat{f}_{\text{pop}}(s_i) = f(s_i)$. We know that

$$\begin{aligned} \hat{f}_{\text{pop}}(s_{k+1}) &= \hat{g}_{\text{pop}}(s_{k+1}) + h(s_{k+1}) \\ &\geq g(s_{k+1}) + h(s_{k+1}) \\ &= f(s_{k+1}) \end{aligned} \tag{1}$$

consistent: h(s) ≤ c(s, next) + h(next)

In order to make sure that each s_{k+1} is only explored after when we pop s_k, the condition of $f(s_i) \leq f(s_{i+1})$ is required, leading to the need for the consistency of h. By popping s_k, we have:

g(next) ≤ g(s) + c(s, next)
g ≤ g^pop ≤ g^
g = g^pop if optimal

$$\begin{aligned} \hat{f}_{\text{pop}}(s_{k+1}) &= \min\{\hat{f}(s_{k+1}), \hat{g}_{\text{pop}}(s_k) + c(s_k, s_{k+1}) + h(s_{k+1})\} \\ &\leq \hat{g}_{\text{pop}}(s_k) + c(s_k, s_{k+1}) + h(s_{k+1}) \\ &= g(s_k) + c(s_k, s_{k+1}) + h(s_{k+1}) \quad \text{from our IH} \\ &= g(s_{k+1}) + h(s_{k+1}) \\ &= f(s_{k+1}) \end{aligned} \tag{2}$$

From equations 1 and 2, we obtain $\hat{f}_{\text{pop}}(s_{k+1}) = f(s_{k+1})$. Hence, by induction, whenever we pop a node from the frontier, the optimal path to the node would have been found.