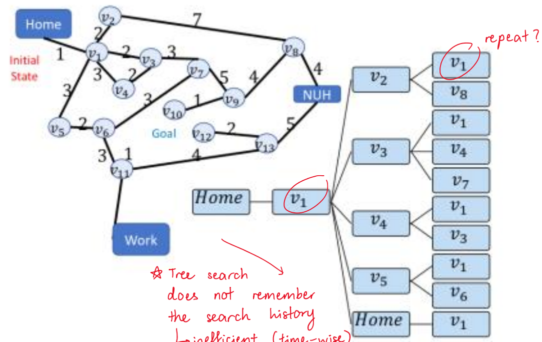
Midterm

Tuesday, 28 September 2021 10:56 PM

Tree Search



☆ Tree search does not remember the search history
↳ inefficient (time-wise)
↳ saves space

16

BFS

Algorithm 2 Breadth First Search: FindPathToGoal(u)

```
F(Frontier) ← Queue(u)
E(Explored) ← {u}
while F is not empty do
    u ← F.pop()
    for all children v of u do
        if GoalTest(v) then return path(v)
        else
            if v not in E then
                E.add(v)
                F.push(v)
return Failure
```

43

Properties of BFS

Property	
Complete?	Yes
Time	$O(b) + O(b^2) + \cdots + O(b^d) = O(b^d)$
Space	$O(b^d)$
Optimal	

46

Depth-First Search

Algorithm 5 Depth First Search(DFS): FindPathToGoal(u)

```
F(Frontier) ← Stack(u)
E(Explored) ← {}
while F is not empty do
    u ← F.peek()
    if GoalTest(u) then
        return path(u)
    if HasUnvisitedChildren(u) then
        for all children v of u do
            if v not in E then
                F.push(v)
                E.add(v)
    else
        F.pop()
        E.add(u)
return Failure
```

84

Depth-First Search

Property	
Complete?	No on infinite depth graphs
Optimal	No
Time	$O(b^m)$
Space	$O(bm)$

When checking a node v, we push at most b descendants to stack.

We do so at most m times $\Rightarrow O(bm)$ space.

85

UCS vs Dijstrak's

1 source,
1 destination,
1 shortest path

1 source,
multiple destinations,
shortest path to all.

Uniform Cost Search

Based on path cost

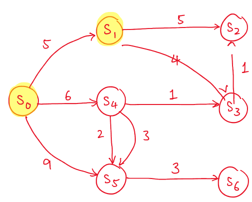
$\hookrightarrow g(u)$ = path cost to get to u from source

Algorithm 4 Uniform Cost Search(UCS): FindPathToGoal(u)

```
F(Frontier) ← PriorityQueue(u)
E(Explored) ← {u}
ĝ[u] ← 0
while F is not empty do
    u ← F.pop()
    if GoalTest(u) then
        return path(u)
    E.add(u)
    for all children v of u do
        if v not in E then
            if v in F then
                ĝ[v] = min(ĝ[v], ĝ[u] + c(u, v))
            else
                F.push(v)
                ĝ[v] = ĝ[u] + c(u, v)
return Failure
```

51

1)

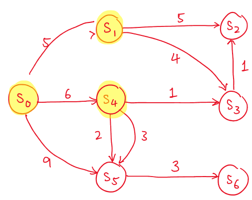


g: | 0 | 5 | | | 6 | 9 | |

F: 1 4 5 3 2

pop

2)



7

g: | 0 | 5 | 10 | ~~9~~ | 6 | 9 | |

F: ~~1~~ 4 5 3 4

pop

move up priority

☆ NOT the same as Dijstra

Proof of Optimality

Theorem: When we pop u from F, we have found optimal path to u from the start node (say, S_0)

Notations:

- $g(u)$: Minimum distance from S_0 to u
- $\hat{g}_{pop}(u)$: The value of $\hat{g}$ when u is popped

Formally, we want to prove $\hat{g}_{pop}(u) = g(u)$

☆ NOT the same as Dijstra

↳ Dijstra is too ALL nodes from root

↳ Uniform cost is only start to goal

Proof by induction:

① Assume optimal path to u:
$S_0, S_1, \dots S_K, u$

② Base case:
$g_{pop}(S_0) = g(S_0) = 0$

③ Assume for S_0 to S_K
$g_{pop}(S_K) = g(S_K)$

④ $g(s_1) \le g(s_2) \le \dots \le g(u)$ → ☆ $\varepsilon \ge 0$

⑤ $g_{pop}(u) \ge g(u)$

↳ cannot be less

After popping S_K,

$$g(u) = \min(\hat{g}(u), g_{pop}(S_K) + c(S_i, u))$$
$$= g(S_K) + c(S_i, u)$$
$$= g_{pop}(u)$$

no other path will produce a better g(u)

↳ u eventually popped

Uniform Cost Search

Property	
Complete?	Yes (if all step costs are $\ge \varepsilon$)
Optimal	Yes (shortest path nodes expanded first)
Time	$O(b^{1+\lfloor \frac{C^*}{\varepsilon} \rfloor})$ where C^* is the optimal cost.
Space	$O(b^{1+\lfloor \frac{C^*}{\varepsilon} \rfloor})$

Summary

Property	BFS	UCS	DFS
Complete	Yes[1]	Yes[2]	No
Optimal	No[3]	Yes	No
Time	$O(b^d)$	$O(b^{1+\lfloor \frac{C^*}{\varepsilon} \rfloor})$	$O(b^m)$
Space	$O(b^d)$	$O(b^{1+\lfloor \frac{C^*}{\varepsilon} \rfloor})$	$O(bm)$

1. if b is finite.
2. f b is finite and step cost $\ge \varepsilon$

A* Search

Algorithm 6 A* Algorithm: FindPathToGoal(u)

```
F(Frontier) ← PriorityQueue(u)          ▷ it should be implement with f̂ minimum
E(Explored) ← {u}
ĝ[u] ← 0
while F is not empty do
    u ← F.pop()
    if GoalTest(u) then
        return path(u)
    E.add(u)
    for all children v of u do
        if v not in E then
            if v in F then
                ĝ[v] = min(ĝ[v], ĝ[u] + c(u, v))
                f̂[v] = h[v] + ĝ[v]
```

using $\hat{f}(u)$

→ ☆optimal

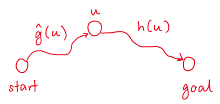
$g(u)$ → min path cost

$\hat{g}(u)$ → path cost so far

$h(u)$ → estimated cost from u to goal (heuristic function)

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

$f(u)$ → optimal cost $= g(u) + h(u)$



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

```
        E.add(u)
        for all children v of u do
            if v not in E then
                if v in F then
                    ĝ[v] = min(ĝ[v], ĝ[u] + c(u, v))
                    f̂[v] = h[v] + ĝ[v]
                else
                    F.push(v)
                    ĝ[v] = ĝ[u] + c(u, v)
                    f̂[v] = h[v] + ĝ[v]
    return Failure
```

$\hat{f}(u)$ must be updated when $\hat{g}(u)$ is updated

A* Search

What property of h would ensure that A* is optimal?

Thm A* with graph-search and consistent h is optimal.

25

☆ What property of $h(u)$, and hence, $\hat{f}(u)$ would make the algorithm optimal?

Refer to UCS → nodes along optimal path get popped in the right order

↳ $g_{pop}(s_0) \leq g_{pop}(s_1) \leq \ldots \leq g_{pop}(u)$

↳ $\hat{g}_{pop}(s_i) = g(s_i)$

$g(s_0) \leq g(s_1) \leq \ldots \leq g(u)$

A* uses $\hat{f}(u)$ → prove $\hat{f}_{pop}(s_i) = f(s_i)$ Q2

Q1 : $\hat{f}_{pop}(s_0) \leq \hat{f}_{pop}(s_1) \leq \ldots \hat{f}_{pop}(u)$

↳ bc of priority queue

$f(s_i) \leq f(s_{i+1})$ → if true, proves by Induction Q3

$\Leftrightarrow g(s_i) + h(s_i) \leq g(s_{i+1}) + h(s_{i+1})$

$\Leftrightarrow h(s_i) \leq \underbrace{g(s_{i+1}) - g(s_i)}_{c(s_i,\, s_{i+1})} + h(s_{i+1})$ Q4

logically equivalent

↳ ☆ Consistency

Q4:

$h(goal) \geq 0$

$h(s_0) \leq c(s_0, s_1) + h(s_1)$

$\leq c(s_0, s_1) + c(s_1, s_2) + \ldots + c(s_k, u) + \cancel{h(u)}^{0}$

$\leq$ path cost s_0 to u

↳ ☆ Admissibility

☆ Consistency & $h(goal) \geq 0 \Rightarrow$ Admissibility → $A \not\Rightarrow C$

↳ A* OPTIMAL UNDER THESE SPECIFIC CONDITIONS

Tree search varient
no record of explored
↳ still optimal with ADMISSIBILITY (Tutorial 4)

Heuristic Functions

Consistent

$\forall n, n'$ $\boxed{h(n) \leq h(n') + c(n, n')}$ → Triangle Inequality

where, n' is successor of n

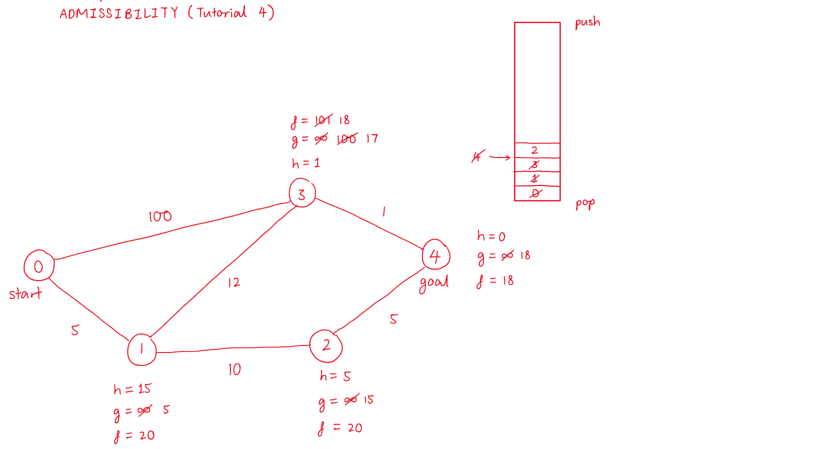
Admissible

$\forall n$ $h(n) \leq h^*(n)$

where $h^*(n) = \underline{g(goal)} - g(n)$:= True cost to reach the goal.

↳ path cost from n to goal

26



The Power of Admissibility

Observation:

if h(goal) ≥ 0, then consistency => admissibility

Theorem:

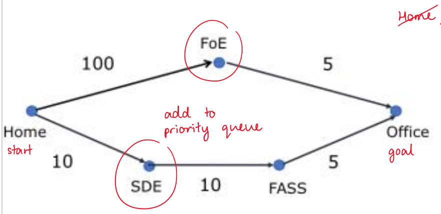
A* search with Tree-Search is optimal for admissible heuristics.

27

Greedy Best First Search

- What if we have priority queue based on h instead of $\hat{f}$
- Would this be optimal ? No. → not consistent

~~Home~~, ~~FoE~~, SDE
~~office~~ ← popped early



h (FoE) = 5
h(SDE) = 15
h(FASS) = 5
h (Office) = 0

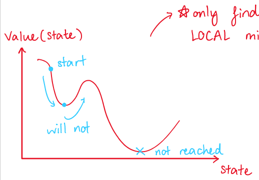
Hill Climbing

Algorithm 1 HillClimbStep(s)

$minVal \leftarrow \mathrm{val}(s)$
$minState \leftarrow \{s\}$
for each u in $N(s)$ **do**
  if $\mathrm{val}(u) < minVal$ **then**
    $minVal = \mathrm{val}(u)$
    $minState = u$
return $minState$

☆ keep moving to a state where Val(next) < Val(current)

☆ only finds LOCAL minimum



Simulated Annealing

- Condensed matter physics
  - Study of materials at low temperatures.
  - Spin Glass models
    - Atoms : have spin ± 1
    - μ_i : spin of atom i
    - E(c): Energy of configuration c
      - E(c) = $e^{\frac{-\sum_{(i,j)} J \mu_i \mu_j}{k_B T}}$
      - k_B : *Boltzman constant*
      - T : temperature
    - Probability of going from state c_1 to c_2
      - $\Pr[c_1 \to c_2] \propto e^{\frac{E(c_1)-E(c_2)}{k_B T}}$ → $T \to \infty$, $e^{\frac{E(c_1)-E(c_2)}{k_B T}} \to e^0 = 1$

$\Pr(c_1 \to c_2) \approx \frac{1}{n}$

- How do we reach the lowest energy state?
  - Material scientists : have a "cooling schedule"
  - Cooling schedule : "first have the high temperature and then slowly decrease the temperature"

T = 1, probability depends on $E(c_1) \to E(c_2)$

☆ How to ensure reaching to lowest energy state?

☆ When $E(c_1) > E(c_2)$, probability ↑. } generally goes to a better state

When $E(c_1) < E(c_2)$, probability ↓ but never 0 } but small chance of making mistakes

Simulated Annealing

How do we implement this algorithmically?
Kirkpatrick, Gelatt and Veechi (1983)
Replace $E(c) \to val(c)$

SimulatedAnnealing($initialState$)

$C \leftarrow initialState$
for $t = 0$ to ∞ **do**
  $C' \leftarrow$ PickRandomNeighbour(C) → what are the neighbours? → valid actions
  $T \leftarrow$ Schedule(t)
  if $val(C') = 0$ **then**
    return C' ☆ lowest value = solution
  if $val(C') < val(C)$ **then**
    $C \leftarrow C'$
  else → mistake
    $C \leftarrow C'$ with Probability $\propto \exp\left\{-\frac{val(C')-val(C)}{k_B T}\right\}$ → can change according to the algo

- We terminate when $val(c) = 0$, but for some cases, the $val(goal)$ may not be defined.
  - Optimization problems seek to minimize $val(c)$
  - Schedule can be modified according to application

Simulated Annealing

How do we implement this algorithmically?
Kirkpatrick, Gelatt and Veechi (1983)
Replace $E(c) \to val(c)$

SimulatedAnnealing($initialState$)

$C \leftarrow initialState$
for $t = 0$ to ∞ **do**
  $C' \leftarrow$ PickRandomNeighbour(C) what are the neighbours? ↳ valid actions
  $T \leftarrow$ Schedule(t)
  if $val(C') = 0$ **then**
    return C' ☆ lowest value = solution
  if $val(C') < val(C)$ **then**
    $C \leftarrow C'$
  else mistake
    $C \leftarrow C'$ with Probability $\propto \exp\left\{-\frac{val(C') - val(C)}{K_B T}\right\}$ can change according to the algo

- We terminate when $val(c) = 0$, but for some cases, the $val(goal)$ may not be defined.
  - Optimization problems seek to minimize $val(c)$
  - Schedule can be modified according to application

Constraint Satisfaction Problem (CSP)

- A CSP comprises of three components:
  1. Set of Variables X = { $x_1, x_2, \ldots, x_n$ } psn of queen in that col
  2. Set of domains corresponding to each variable:
     (1 to 4)
     D : { $Dx_1, Dx_2, \ldots, Dx_n$ } where $Dx_i = domain(x_i)$
  3. Set of constraints

	x_1	x_2	x_3	x_4
1				
2				
3				
4				

- Find the value for each of the variable in its domain that satisfies all the constraints.

Backtracking Algorithm (Attempt II)

Before assigning value, checks if it is consistent with the previous assignments.

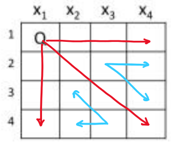
BacktrackingSearch($prob, assign$)

if AllVarAssigned($prob, assign$) **then return** $assign$
$var \leftarrow$ PickUnassignedVar($prob, assign$)
for $value$ in OrderDomainValue($var, prob, assign$) **do**
  if ValIsConsistentWithAssignment($value, assign$) **then**
    $assign \leftarrow assign \cup (var = value)$
    $result \leftarrow$ BacktrackingSearch($prob, assign$)
    if $result != failure$ **then return** $result$
    $assign \leftarrow assign \setminus (var = value)$
return $failure$

set minus

Backtracking Algorithm with Inference

- Assign value to x_i and ***infer*** the restrictions on rest of the variables



$x_2 \neq 1, 2$

$x_3 \neq 1, 3$

$x_4 \neq 1, 4$

If $x_3 = 2$,
$x_4 \neq 2, 3$ ← no solution!

If $x_3 = 4$,
$x_2 \neq 3, 4$ ← no solution!

} $x_3 \neq 1, 2, 3, 4$
↳ $x_1 \neq 1$

✩ inference takes a long time
↳ balance assign & check w/ inferring

62

Backtracking Algorithm with Inference

BacktrackingSearch_with_Inference(*prob*, *assign*)

```
if ALLVARIABLESASSIGNED(prob, assign) then return assign
var ← PICKUNASSIGNEDVAR(prob, assign)   x3
for value in ORDERDOMAINVALUE(var, prob, assign) do   {1, 2, 3, 4}
    if VALISCONSISTENTWITHASSIGNMENT(value, assign) then   {2, 4}
        assign ← assign ∪ (var = value)
        inference ← INFER(prob, var, assign)
        assign ← assign ∪ inference
        if inference != failure then
            result ← BACKTRACKINGSEARCH.(prob, assign)
            if result != failure then return result
        assign ← assign \ {(var = value) ∪ inference}
return failure
```

63

How to Implement INFER

Data structure for Inference:

(Unordered) list of tuples of the form: $(x \notin S)$

Data structure for assign:

variable = value

(Unordered) list of tuples, where every tuples is of the form (x=v) or $(x \notin S)$

↳ variable not part of S

ComputeDomain(x,assign,inference): $\xrightarrow{e.g.}$ $\{x_2 \notin \{2, 3\}$, $x_3 \notin \{1, 2, 3, 4\}\}$

$\xrightarrow{e.g.}$ $\{x_1 = 1\}$

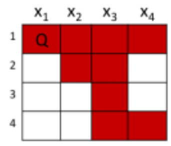
Returns S such that the effective domain of x is S

↳ the spaces that x can take given the current assignment & inference

$CD(x_1, \dots, \dots) = \{1\}$

$CD(x_2, \dots, \dots) = \{1, 4\}$

6



x_1
NoAttack(x_1,x_2):
$x_2 \notin \{1,2\}$
NoAttack(x_1,x_3):
$x_3 \notin \{1,3\}$
NoAttack(x_1,x_4):
$x_4 \notin \{1,4\}$

x_2
NoAttack(x_2,x_4):
....
NoAttack(x_2,x_3):
$x_3 \notin \{3\}$

x_4
NoAttack(x_3,x_4):
$x_3 \notin \{2\}$

Ideas

1. Iterate over variables: Which variables?

   For which we infer *new information*

2. Iterate over constraints where the variable appears

3. We can infer $x \neq v$ from constraint C(x,y,z) if there is no value that y and z can take such that C(v,y,z) returns True → from the effective domain

Returns True = Not falsified = Not violated

Returns False = falsified = Violated

INFER

x_1

```
INFER(prob, var, assign)
1: inference ← ∅
2: varQueue ← [var]   x1
3: while varQueue is not empty do
4:     y ← varQueue.pop()   x1
5:     for each constraint C in prob where y ∈ Vars(C) do   ← all the constraints that y appears in
6:         for all x ∈ Vars(C) \ y do   {x2, x3, x4}
7:             S ← COMPUTEDOMAIN(x, assign, inference)   {1, 2, 3, 4}
8:             for each value v in S do
9:                 if no valid value exists for all var ∈ Var(C) \ x s.t. C[x ⊢ v] is satisfied then
10:                    inference ← inference ∪ (x ∉ {v})   x2 ∉ {1, 2}
11:            T ← COMPUTEDOMAIN(x, assign, inference)
12:            if T = ∅ then return failure
13:            if S ≠ T then
14:                varQueue.add(x)
15: return inference
```

↳ returns failure no matter what the other vars are

} new information
↳ domain smaller
↳ recompute

The above algorithm is conceptually identical to AC-3 algorithm

The order of constraints in line 5 only impacts runtime efficiency but not the final answer

Forward Checking

only infer 1 var

```
INFER(prob, var, assign)
1: inference ← ∅
2: varQueue ← [var]
3: while varQueue is not empty do
4:     y ← varQueue.pop()
5:     for each constraint C in prob where y ∈ Vars(C) do
6:         for all x ∈ Vars(C) \ y do
7:             S ← COMPUTEDOMAIN(x, assign, inference)
8:             for each value v in S do
9:                 if no valid value exists for all var ∈ Var(C) \ x s.t. C[x ⊢ v] is satisfied then
10:                    inference ← inference ∪ (x ∉ {v})
11:            T ← COMPUTEDOMAIN(x, assign, inference)
12:            if T = ∅ then return failure
13:            if S ≠ T then
14:                varQueue.add(x)   ] remove
15: return inference
```

Forward Checking: Don't add variable to varQueue at each iteration. Remove lines 13 and 14.

INFER: Different Heuristics

```
INFER(prob, var, assign)
inference ← ∅
varQueue ← [var]
while varQueue is not empty do
    y ← varQueue.pop()
    for each constraint C in prob where y ∈ Vars(C) do
        for all x ∈ Vars(C) \ y do
            S ← COMPUTEDOMAIN(x, assign, inference)
            for each value v in S do
                if no valid value exists for all var ∈ Var(C) \ x s.t. C[x ↦ v] is satisfied then
                    inference ← inference ∪ (x ∉ {v})
            T ← COMPUTEDOMAIN(x, assign, inference)
            if T = ∅ then return failure
            if |T| = 1 then  → and |S| ≠ |T|
                varQueue.add(x)
return inference
```

more constraints for other vars

- Find inference for variables that have only ONE valid value in the domain, i.e., |T| = 1
- Can be set to any value, depending on the complexity of the application:
  |T| < 2 , |T| <= 3 ....

19

Backtracking Algorithm with Inference

```
BacktrackingSearch_with_Inference(prob, assign)
if ALLVARIABLESASSIGNED(prob, assign) then return assign
var ← PICKUNASSIGNEDVAR(prob, assign)  → what order to pick
for value in ORDERDOMAINVALUE(var, prob, assign) do  → what order to try
    if VALISCONSISTENTWITHASSIGNMENT(value, assign) then
        assign ← assign ∪ (var = value)
        inference ← INFER(prob, var, assign)
        assign ← assign ∪ inference
        if inference != failure then
            result ← BACKTRACKINGSEARCH.(prob, assign)
            if result != failure then return result
        assign ← assign \ {(var = value) ∪ inference}
return failure
```

20

Backtracking Algorithm with Inference: Different Heuristics

- PickUnassignedVar
  - Minimum Remaining Value Heuristic: Choose the next unassigned variable with the smallest effective domain size
    - Can allow us to backtrack quickly
      ↳ exhaust the possible values more easily
- OrderDomainValue
  - Least Constraining Value Heuristic: Pick a value for a variable that rules out the least domain values for other variables.
    - Can allow us to find a solution faster
      ↳ more choices

21