

Lecture 3

Constraint satisfaction and Local Search

In this lecture we examine another type of state-space search problems, where path cost doesn't matter.

This type of problems cannot be improved using the approach (heuristic function) we have learnt in the last lecture.

In this lecture we will see other ways to deal with them.

Reference:

- Section 4.3, Chapter 5.

AI(0270)

Local search

So far we see algorithms which **systematically** search the state space, to make sure a solution close to the initial state will be found.

- When the **objective** is a **small path cost**, it is important to remember a **lot of short-cost alternatives**, and **completely** exhaust them.
- When path cost doesn't matter, searching this way will **not keep us from testing a lot of states**.

So we consider **changing our strategy**:

- Don't store all the states we tried. Instead, just store one **current state**, and try finding the best state from there. We call it **local search**.
- How to **guide our search**, then? Given a state, we need to know **how good is it**, so that we know which direction is more promising.

This works with **optimization function**, trying to maximize an **objective function**, or to minimize a **cost function**.

AI(0270)-3.2

8-queen problem

Let's see a formulation of the 8-queen problem using local search.

- **States**: 8 numbers from 1 to 8 specifying the location of the queen in each column.
Not allowing unassigned queen location!
- **Successor function**: the resulting states modifying one of the 8 numbers to a new value.
There are $7 \times 8 = 56$ of them.
- **Initial state**: arbitrary. In general we will create a random initial state.
So it depends on a bit of luck.
- **Cost function**: number of **pairs** of queens that are **conflicting** each other, i.e., on the same row or diagonal line.
The goal is understood to be the state with minimum cost, presumably 0.

AI(0270)-3.4

Why A* doesn't solve all search problems?

The heuristic function in the last lecture has an **important assumption**:

We want to look for an optimal solution, in the sense that the optimal solution means the solution with the **minimum path cost**.

For many state-space, this leaves no room for informed search:

- 8-queen problem...
- Crypto-arithmetic...
- map coloring problem... (Given a geographic map and a number of colors, make sure that two adjacent regions won't share the same color.)

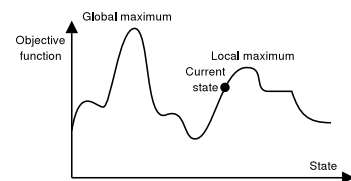
For these algorithm, **we cannot express our domain knowledge as a heuristic function**, since "adding up path costs" makes no sense.

For all the above problems, the path cost is always 0, so the poor 0-function is the only reasonable heuristic function.

AI(0270)-3.1

State space landscape

One can **visualize** a local search by a picture like this:



In other words, we **look around the landscape** and find a promising way to **reach a place with highest evaluation function**.

But in general the state space is not really like an "axis". For some problems (like 8-queen in our formulation) it looks like an n -dimensional space, for other it is just an arbitrary graph.

AI(0270)-3.3

Hill climbing

- Repeatedly pick the choice of the successor function that **increases the evaluation function** by the largest amount.
"Finding the top of Mount Everest in a thick fog while suffering amnesia"?
- The code is more or less trivial:

```
def HillClimb():
    state = make_random_state()
    while True:
        oldstate = state
        for s in Successor(oldstate): # find best successor
            if cost(s) < cost(state):
                state = s
        if oldstate == state: # no more improvement
            return state
```

AI(0270)-3.5

Example: 8-Queen

E.g., in 8-Queen, always **select the one which reduce the number of conflicts the most**.

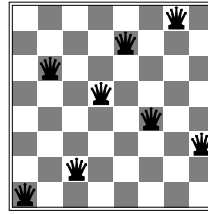
18	12	14	13	13	12	14	14
14	16	13	15	12	14	12	16
14	12	18	13	15	12	14	14
15	14	14	13	16	13	16	16
17	14	17	15	14	16	16	16
17	16	18	15	14	15	16	16
18	14	15	15	14	14	16	16
14	14	13	17	12	14	12	18

The current cost is 17 (conflicts), and the best (e.g., moving king of 2nd column to row 1) of the 56 moves yield cost of 12.

AI(0270)-3.6

Difficulties

- **Plateau:** if we have a large area where the **evaluation function is essentially the same**, the search becomes a *random walk*. In fact the code would stop immediately rather than doing a random walk. One solution is to allow a certain number of **side-way** moves before stopping.
- **Local maxima:** the search may get stuck at a place where the evaluation function is **largest in the short range**, but is not the global optimal.



Current cost is 1 (the only conflict is between queens at column 4 and column 7).

Any single-step move creates more conflict, so it is a local maximum.

If the allowable operations are “swapping the row numbers of 2 columns”, none reduce the number of conflicts, although some (e.g., column 4 and 8) doesn't increase it—plateau.

AI(0270)-3.7

Random restart

- So the algorithm repeat the steps **until the evaluating function is no longer increasing**, and stops there.
- What to do then? **We might be in a local maxima, plateau or ridge**. We might be at an optimum. We just can't tell.
- One possibility: **restart from another random initial position!**
- We will **remember the best solution so far**. After a certain number of tries, we **report the best solution we recorded**.
- This is called **Hill climbing with random restart**. It is very effective for certain problems.
E.g., solving million-queen problem is feasible.
- Of course, if enough tries are made, we will eventually be “lucky enough” to hit somewhere that leads to the optimal solution.

AI(0270)-3.8

Simulated annealing

- For other problems, **forgetting everything done so far is too expensive** to deal with small local maxima.
- Don't use the “best” next-state: that end up in local optimal too easily.
- Instead: just choose a **random next-state**. Move to it with a **probability** depending on **how much it worsen the solution**.
If worsen more, move to it at a lower probability.
- But we can't keep moving around randomly forever! At later stage we want **less randomness** (hopefully we are already at a good place).
- This is captured by a “temperature”, high at the beginning and 0 at end: **simulated annealing**. High temperature means more random.
Anneal: To subject, e.g., glass, metal, etc., to great heat, and then cool slowly, for the purpose of rendering it less brittle, to temper, to toughen.
- The **schedule** of temperature reduction is **predetermined**.

AI(0270)-3.9

Code

Code makes it much clearer...

```
def SimulatedAnnealing(schedule): # schedule is an array of real nos
    current = make_random_state()
    step = 1
    while True:
        temperature = schedule[step]
        if temperature == 0:
            return current
        next = random_choice_of_successor(state)
        improvement = value(next) - value(state)
        with probability of prob(improvement, temperature):
            current = next
        step += 1
```

prob(x, T) is usually set to $e^{x/T}$ to mimic physical temperature.

If the probability is more than 1 (i.e., improvement > 0), the next line is always executed.

AI(0270)-3.10

Local beam search

Local beam search is a way to improve the chance to reach the global maximum: **keep a few (k) current states** rather than just 1.

- It looks a bit like running *k* concurrent copies of hill climbing, but **with communication**.
- If one of the state has many good successors, those without good successor can “move over” to the more promising state.
- There is a **problem**, though: **all states might moves over** to the same region, making it essentially hill climbing, except more costly.
- The idea of simulated annealing can be used to **increase diversity**...
- **Stochastic beam search:** don't use the *k* **best** states. Instead choose *k* random successors, with better states having higher probability.

AI(0270)-3.11

Genetic algorithm

- Genetic algorithm is one **variant of stochastic beam search** that use **2 states** to create each new state, randomly.
Hopefully, by chance the good characteristics of 2 states are combined and results in a very good state quickly. It also helps to maintain diversity.
- It is easy to imagine choosing 2 states from the k state **population** to create new state, with probability proportional to the **value** of the state. But **how to combine** them?
- For this to work, the state must be **represented** as a **sequence**, of bits or numbers or whatever.
- Each time a **crossover** is done: randomly choose a cut-off point to **cut** both genes into 2. So each part has 2 choices, to be done randomly.
- After crossover, **mutation** is done: with some probability, one of the bit or number or whatever is changed to another value.
- Repeat k times and replace the whole pool by the k new states.

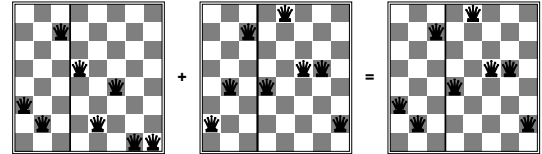
AI(0270)-3.12

Illustration: 8-queen problem

The whole process: fitness evaluation, selection, crossover and mutation. The representation is the 8 row-numbers for queen at each column.



The interpretation of the topmost crossover:



AI(0270)-3.13

Constraint Satisfaction Problem (CSP)

A class of problem, called CSP, worth special attention:

- There are a number of **variables**.
- Each variable has **some values** that it can take.
- Our target is to find some way to set all the variables so that **a set of constraints** about the variables are all satisfied.

In our standard state space form:

- A **state** is any **partial assignments** to the variables.
- Operator**: **adding an assignment** to one of the variables.
- Initial state**: **no assignment** is made. **Goal state**: any state with **all constraints satisfied**.
- All goal states are **equally good**. In other words, path cost is the same for all goal states.

AI(0270)-3.14

Example

- 8-queen problem**: we can have **one variable per row**, each may take a values from 1 to 8 (the column number of the queen in that row).

Constraint: all variables are different; no two queens are on the same diagonal.

- Map coloring**: each region is a variable, the value being one of the available colors.

Constraint: variables corresponding to adjacent regions are given different values. (See next slide.)

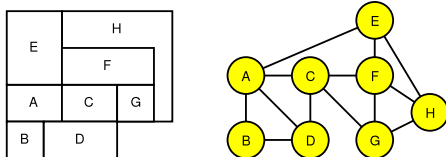
- Crypto-arithmetic**: E.g., TWO + TWO = FOUR. This is a bit tricky. Each letter is a variable, of course, but each "carry" is also a variable. (See the slide after the next.)

- Many real world problem, e.g., scheduling, resources allocation, etc.

AI(0270)-3.15

Graph coloring

E.g., to color the following map on the left (or equivalently, the graph on the right) using 3 colors: Red, Green and Blue.



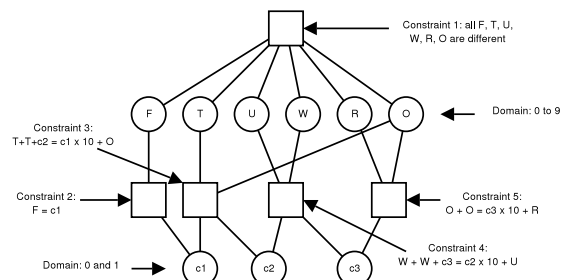
Variables: color of each node.

Constraints: adjacent node can't use the same color. E.g., if A is colored Red, then B, C, D and E cannot be Red.

AI(0270)-3.16

Crypto-arithmetic as CSP

Once we add the carries as CSP variables, we can represent the formula as **constraints to the assignments of the letters**.



Note that constraints involves many variables, not just 2. Such constraints can be converted to those with 2 variables ("binary constraints"), by adding more variables.

AI(0270)-3.17

Local search on CSP

- It is possible to **apply local search strategies** on CSP (start with wrong solution, patch it up as far as possible, and use random restart, simulated annealing, etc).
- This **works excellently** if solutions are **densely located** in the state space (so we can start at many place and end up with solutions).
- It also has the merit that if a new constraint suddenly pop up, we can **start a local search from the previous solution**, hopefully finding a solution similar to the old one.
If we start from scratch, we are likely to find something completely different, which can be annoying for many real world problem like scheduling.
- On the other hand, it provides no **guarantee** to always find a solution, especially if **there are only a few of them** and there are a lot of local maximum.
- Another strategy: searching systematically, just like last lecture.

AI(0270)-3.18

Plain searching and Commutativity

- We can apply normal searching, at cost of b^d .
- In CSP, the "actions" are **choose a variable** and **choose a value** in such a way that **no constraint is violated**. If we can assign all variables we get a solution (goal).
- What is b ? If we have n variables with m values, then first step has nm choices, second $(n - 1)m$ choices, etc.
- Depth is n . Time cost: $n! \times m^n$. But there are only m^n assignments!
Note: depth is known exactly, so we use DFS, or better, backtracking search.
- What excessive work we have done? Suppose at a step we need to assign value to a variable a , and all values are tried without success.
- Plain DFS would **try another variable!** This is stupid: the order of variable assignment doesn't affect the search result.

AI(0270)-3.19

Actual algorithm

Thus at each node we **pick only one variable** and try different values for it. This brings the search tree size down to m^n . The code:

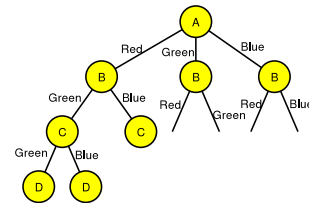
```
def BacktrackingSearch(csp):
    return RecursiveBacktrackingSearch({}, csp)

def RecursiveBacktrackingSearch(assignment, csp):
    var = SelectUnassignedVariable(csp, assignment)
    if var == NoMoreVariable:
        return assignment
    for value in DomainValues(csp, assignment, var):
        if value doesnt violate constraints:
            assignment[var] = value
            result = RecursiveBacktrackingSearch(assignment, csp)
            if result != failure:
                return result
            del(assignment[var])
    return failure
```

AI(0270)-3.20

Example

We will use the map as example throughout. The first part of search if we use the **fixed** ordering of nodes A, B, C, ...; and **fixed** ordering of 3 available colors red, green and blue:



Note that assignments that causes immediate conflicts are not searched. E.g., after A=red, B cannot be assigned red again, or a conflict with occurs. This is immediately detected, and not searched. So the search tree is much smaller than m^n . We will see more such optimizations.

AI(0270)-3.21

Doing better

In general, CSP is NP-hard, so in the worst case exponential time cannot be avoided. But in many real world problems, much better performance can be achieved.

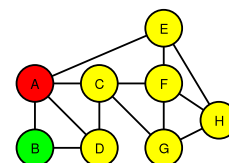
The remainder of the lecture is about **how to do better**, in 4 directions:

1. How to pick a good variable and what is a good order of assigning values? (*SelectUnassignedVariable, OrderDomainValues.*)
2. How to **know early that a branch is doomed**? We can't always do it, but if we can detect a lot of such cases the cost of search can be reduced very significantly.
3. Suppose we search in a doomed subtree for some time and eventually **fail and backtrack**. How to avoid **repeating the same mistake** by repeating the search after modifying an **irrelevant** variable?
4. How to **decompose** a large problem to more managable ones?

AI(0270)-3.22

Better ordering

- While any ordering of variable and values can find solutions can result in solutions, using the right ordering **dramatically improves efficiency**.
- Why? Consider the case when A and B are assigned red and green:



- If we try C now, we get a **subtree** of green C and another with blue C (which eventually fail). If we try D now, we **only** get a subtree of blue D and **one** subtree for green C.

AI(0270)-3.23

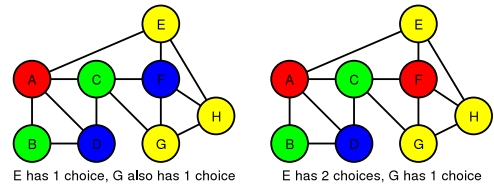
Ordering of variables: MRV and degree heuristic

- Why D is better? **D only has 1 remaining value.** If C is selected and conflict with that remaining value, the search is doomed.
- Generalization: it is better to **try variables with minimum remaining values** first: the **MRV heuristic**.
- How about the beginning? All have 3 possible values, so all are the same?
- We probably want to **try A first anyway** (or C) rather than B or D: it is adjacent with 4 other variables, so it reduces the size of the search tree the most.
- Generalization: when there are the same number of values for some variables, try the one **conflicting with the largest number of nodes**: "**degree heuristic**".
Degree = number of edges from a node.

AI(0270)-3.24

Ordering of values: LCV heuristic

So we go on, coloring A=red, C=green, D=blue, B=green. By degree heuristic we have to choose a color for F. We have 2 choices: red or blue.



Which is better? **Red.** (Choosing blue leads to failure.)

We say F=red is **less constraining** than F=blue, and we want to try the **least constraining value (LCV)** first.

Question: Why this time we are not inclined to reduce search tree size?

AI(0270)-3.25

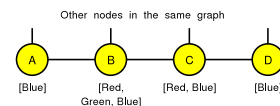
Forward checking: incremental updates for MRV

- When MRV is applied, we will continuously need to know **how many possible values remain** for each variable. It's slow if it needs to be calculated every time.
- How to improve? We can, for each variable, **maintain a set** of values that is still **valid**, called the **current domain**. Initially the current domain of each variable is set to its domain.
- To implement MRV, each time a variable is assigned, **remove conflicted values** from the current domain of its **adjacent** variables.
- On backtracking, these removal are **undone**.
So at each level of the search, a list of removed variable-values is remembered.
- We call it **forward checking**: update the domains for use in the future.

AI(0270)-3.26

More checking

- Forward checking can be viewed as **early finding of what values become impossible** due to previous assignments.
- If we can quickly strike out more values, **the branching factor of search tree becomes smaller**, and we can thus search much more quickly.
- Given the large payoff of this, we usually want such search to do done **more extensively**. E.g., suppose the current domain of allowable colors of some nodes of a graph looks like this:

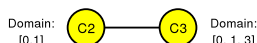


- By just looking at the values (without performing any search), C can only be Red, and B can only be Green!

AI(0270)-3.27

Arc consistency

- It might seem useless: MRV does all these anyway. But if you think about other constraint satisfaction problem, that's a different picture. E.g., 8-queen:



- It seems C3 is not urgent to choose a value. No! C3 must choose 3 now: if it chooses 0 or 1, there is **no way to choose a value for C2!**
- Let's focus on a **directed arc** (edge), say from C2 and C3. We say it is **consistent if for all values of C2**, there is a value of C3 that **does not conflict** with it.
- Here the arc from C2 to C3 is arc consistent (C3 can take 3), while the arc from C3 to C2 is not (because choosing 0 or 1 cause C2 to fail). It can be made consistent by deleting 0 and 1 from domain of C3.
After that MRV correctly chooses value for C3 immediately.

AI(0270)-3.28

Maintaining Arc Consistency (MAC)

After removing a value in one domain, its neighbours might become inconsistent. So we can do something like this to restore arc consistency:

```
def AC3(csp):
    # Reduce domain of variables of csp, i.e., csp.domain
    schedule = queue() # schedule of arc checking
    for arc in csp.arcset:
        schedule.enqueue(arc) # arc in form (X, Y)
    while not schedule.empty():
        (X, Y) = schedule.dequeue()
        if RemoveInconsistentVals(X, Y): # i.e., domain of X is reduced
            for Z in neighbour(X):
                schedule.enqueue((Z, X))
```

How fast is it? Each run of *RemoveInconsistent* can take $O(m^2)$ time if there are m possible values in the domain of each side. Each arc can be enqueued only m times, so the total complexity is $O(e \times m^3)$.

MAC: do this **everytime a value is chosen**.

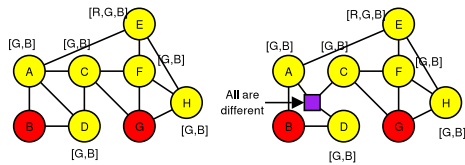
Expensive? But if this reduce the branching factor, the extra effort pays off.

AI(0270)-3.29

Special constraints 1: all-different

Sometimes **special constraints** of many variables can be handled **more efficiently** than its corresponding binary constraints.

E.g., if we want 10 variables to be all different, but the union of their domain has only 6 values, we know the search is doomed. Compare these:



Even arc consistency can't detect failure on the left. On the other hand, the configuration of the right is clearly infeasible (3 states need 3 different colors, but we only have green and blue for coloring A, C and D).

AI(0270)-3.30

Special constraints 2: resources constraints

Another type of constraint commonly found in resources allocation problems, called **"at most"**, also has special purpose checkers.

E.g., if we have 3 variables a1, a2, a3:

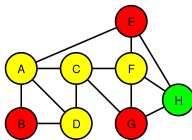
- Domain of a1: [3, 4, 5]
- Domain of a2: [2, 3, 4, 5]
- Domain of a3: [1, 2, 3, 4]
- $a1+a2+a3$ is at most 5

Then we know that the branch failed, by summing the minimum values of the domain.

AI(0270)-3.31

Problem of simple backtrack

So far, if we fail, we backtrack to the **last level**. This is sometimes not the best. E.g., suppose we color our map in the order [B, G, E, H, A, C, D, F]: For simplicity, assume no MRV, LCV and MAC.



Now we try all possibilities of A, C and D, all fails. What to do? In **simple backtracking**: we choose H to be blue and try again... and later recolor E to green to retry.

This is stupid! The failure has **nothing to do with E and H**, so if we just recolor E or H we of course fail again!

AI(0270)-3.32

Backtracking more intelligently: Back jumping

What we want: skip E and H, and backtrack to G directly. But how?

- Suppose we assign A=green, C=blue, and D fails.
- Why D fails? It is due to the **domain reductions when assigning** { B=red, A=green, C=blue }. We call { A, B, C } the **conflict set** of the failure. This can be returned alongside with the failure indication. So we keep conflict sets when performing forward checking.
- When backtracking, we can **immediately declare failure** if the conflict set does **not include the current variable**.
- E.g., after D fails, C receive control. C is in conflict set, so it will try other values rather than return immediately.
- When C runs out of value, it **merges** the conflict set returned with its own conflict set, remove C, and return { A, B, G }. When A also runs out of value and return { A, G }, both E and H are skipped.

AI(0270)-3.33

Decomposing

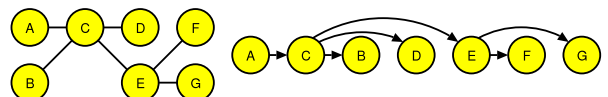
No heuristic change the fact that larger maps are much more difficult than smaller ones, since complexity is exponential.

- Decomposition: **split a large problem to many small ones** (here, to 2, one containing T itself, the other containing the remaining 6 states).
- Nice for **all** algorithms that has exponential complexity.
- E.g., if we have a problem with 80 variables each having 2 choices, the time cost can be $2^{80} \approx 10^{24}$, more than the lifetime of the universe even if each step takes only 10^{-9} second.
- If we can break it into 4 problems of 20 variables, the complexity becomes $4 \times 2^{20} = 4 \times 10^6$, which can be well within 1 second.
- If our graph contains many **connected components**, we can break it up. But what if it contains only 1, like our graph?

AI(0270)-3.34

Tree CSP

First let's examine a special case that can be done efficiently (linear time): **Trees** (or **forest**, i.e., many trees). E.g., diagram on the left...



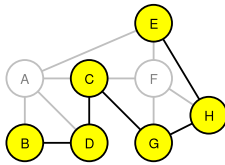
Why we can deal with it efficiently? Because we can arrange into a list on the right, so that each node has 1 arc pointing to it ("parent"). Then...

- For each node X in reverse list ordering (G, F, ..., C, A), make the arc Parent(X) → X consistent. If empty domain results for any variable, then the CSP has no solution.
- For each node X in list ordering (A, C, ..., F, G), choose a value from its domain that is consistent to Parent(X). Always succeed, by definition of arc consistency.

AI(0270)-3.35

Reduction to tree 1: cycle cutset

If we can somehow turn a CSP to a tree CSP, then we are done. But how? Method 1: **delete some variables**. E.g.: if we delete A and F from our map:

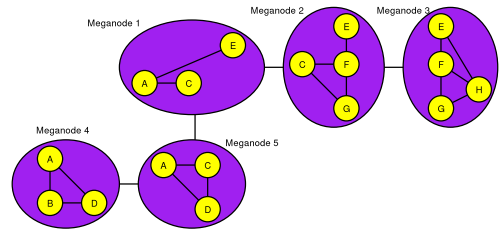


We thus end up with a tree. How to delete A and F? Simple: choose a value for each of them, and perform forward checking. If a value fails, we just try another.

Such a set of nodes is called a **cycle cutset**. Time cost: m^k , where k is size of cycle cutset, and m is number of values for each variable.

Reduction to tree 2: tree decomposition

- Another method: **break up the problem into subproblems**. The exact definition is complicated without an example, so let's see it first:



- The meganodes form a "tree". Each meganode is a subproblems, with possible values being **all** solutions of the subproblems. Two meganode values conflict if their common variables are assigned differently.

Criteria for decomposition

- Every variable and edge of the original problem appears at least in 1 meganode. In reverse, all variables and edges within a meganode come from the original problem.
- If a variable appears in 2 meganodes, then it appears in **all meganodes along the path between them**.
E.g., A appears in meganode 1 and 4, so it must appear in 5. Otherwise meganode 1 must connect to 4, making up a non-tree edge.

How fast is it? It depends of the size of the largest subproblem (called **its tree-width**), denoted $1 + w$. Here $w = 3$.

Maximum number of solution of subproblem is m^{w+1} , and there are $O(n)$ subproblems, so time cost is $n \times m^{w+1}$.

But for our graph, largest subproblem only have 12 solutions.

How to find minimum cycle cutset or narrowest tree decomposition? They are NP-hard, but has reasonable **approximation algorithms**.