Constraint Systems

About Search

Our Search Strategy

This is the search strategy that we are still using:

function $DFS(CSP)$:
  if a solution has been found: **return** true
  if the CSP is infeasible: **return** false
  for d in $decisions(CSP)$:
    if $DFS(apply(d, CSP))$: **return** true
  return false

- Let x_i be the first unbound variable x_i
- Let v its minimum value v
- Then the decisions are:
  - $x_i = v$ (left branch)
  - $x_i \neq v$ (on backtracking)

Our Search Strategy

This is the search strategy that we are still using:

function $DFS(CSP)$:
  if a solution has been found: **return** true
  if the CSP is infeasible: **return** false
  for d in $decisions(CSP)$:
    if $DFS(apply(d, CSP))$: **return** true
  return false

- It gets the job done
- but there's lots of room for improvement!

What can we change?

Restriction: let's stick to Depth First Search

Variable/Value Selection Heuristics

Let's start easy: we can change the criteria for picking:

- The branching variable (variable selection heuristic)
- The branching value (value selection heuristic)

Variable/Value Selection Heuristics

Let's start easy: we can change the criteria for picking:

- The branching variable (variable selection heuristic)
- The branching value (value selection heuristic)

Examples of variable selection heuristics:

- Use the input order
- Smallest domain minimum
- Largest domain maximum
- Custom heuristic!
- ...

Variable/Value Selection Heuristics

Let's start easy: we can change the criteria for picking:

- The branching variable (variable selection heuristic)
- The branching value (value selection heuristic)

Examples of value selection heuristic:

- Minimum value
- Maximum value
- Median value
- Custom heuristic!
- …

A First Design Principle

How to choose our var/val heuristics? One basic suggestion:

Choose the heuristics that are best for solving the problem

Which may look like a dumb advice :-)

- But it's not!
- In fact, it tells us many things...

For Starters

- For a generic problem...
- ...there is not easy way to choose the "right" heuristics!

A theoretical result:

Choosing the perfect var/val heuristics for a problem is as complex as solving the problem itself

- So, if we want to provide non-trivial advice...
- ...we need to be more specific

Back To Our Warehouses

Let's consider a simplification of our warehouse problem:

- Assign customers to warehouses
- Respect capacity constraints
- No travel cost

Back To Our Warehouses

Let's consider a simplification of our warehouse problem:

$$\sum_{i=0..m-1} d_{i,j}\,(x_i = j) \leq c_j \qquad \forall j = 0..n-1 \qquad (capacity)$$

$$x_i \in \{0..n-1\} \qquad \forall i = 0..m-1 \qquad (variables)$$

This is the core of many combinatorial problems

Back To Our Warehouses

Let's consider a simplification of our warehouse problem:

$$\sum_{i=0..m-1} d_{i,j}\,(x_i = j) \leq c_j \qquad \forall j = 0..n-1 \qquad (capacity)$$

$$x_i \in \{0..n-1\} \qquad \forall i = 0..m-1 \qquad (variables)$$

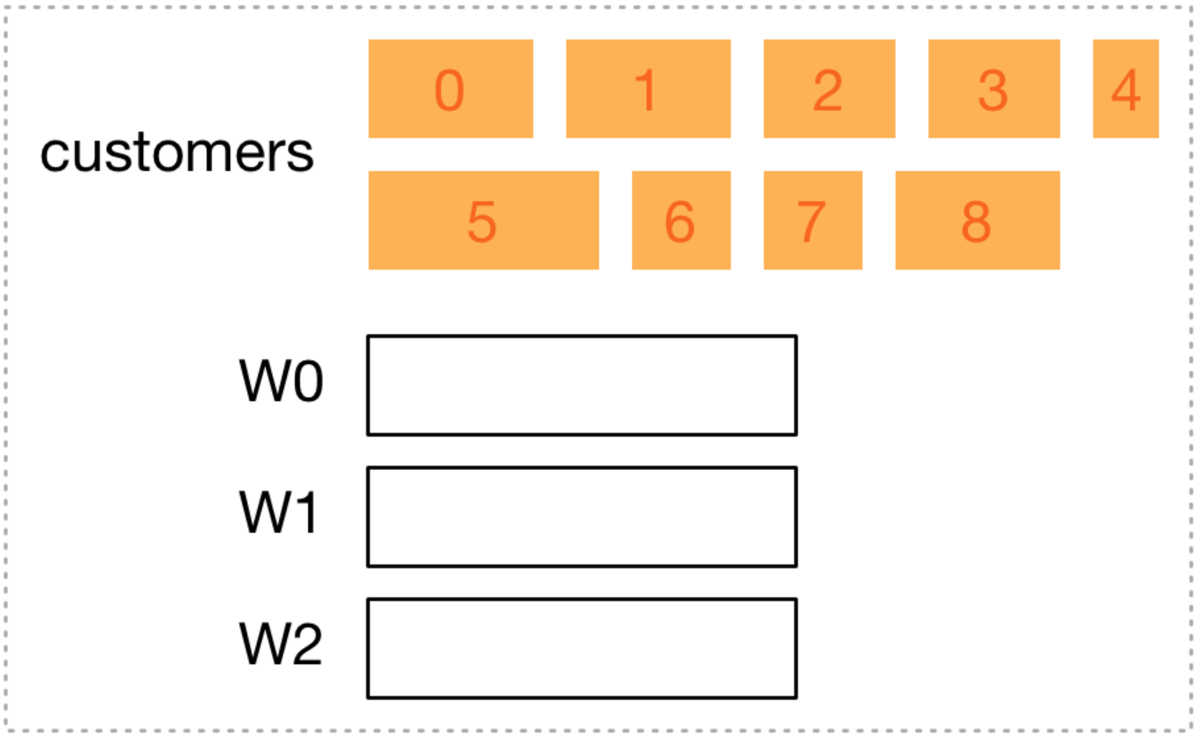
This is the core of many combinatorial problems

Our situation:

- We wan to find a feasible solution
- Only one constraint in our way: warehouse capacity

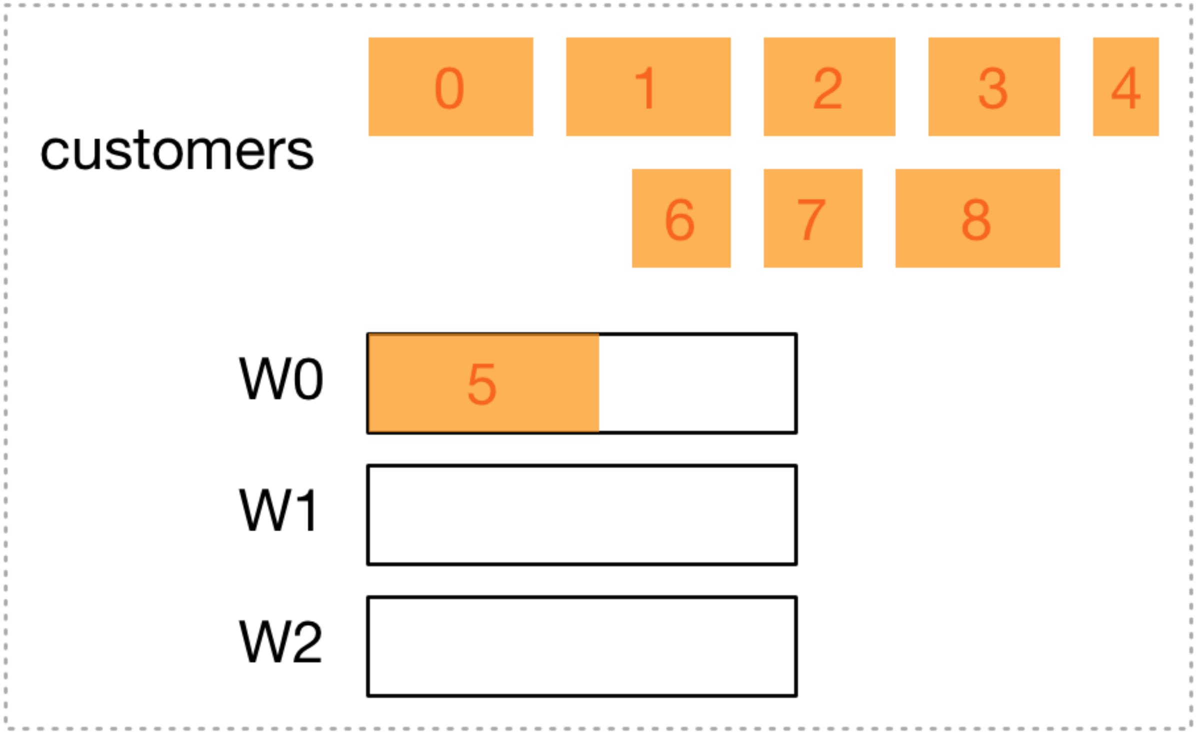
A possible approach: maintain balanced demands

Heuristics for the Warehouses Problem



- Pick customer with the largest demand
- Assign to the least loaded warehouse

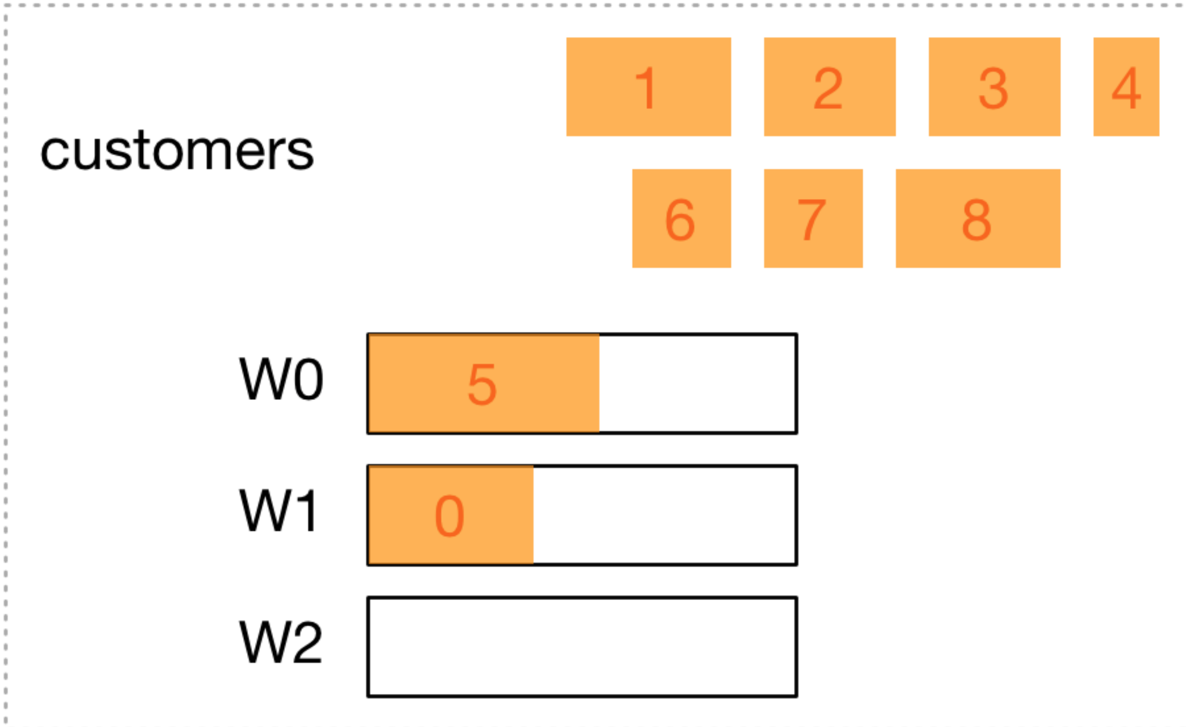
Heuristics for the Warehouses Problem



- Pick customer with the largest demand
- Assign to the least loaded warehouse
- Break ties using indices

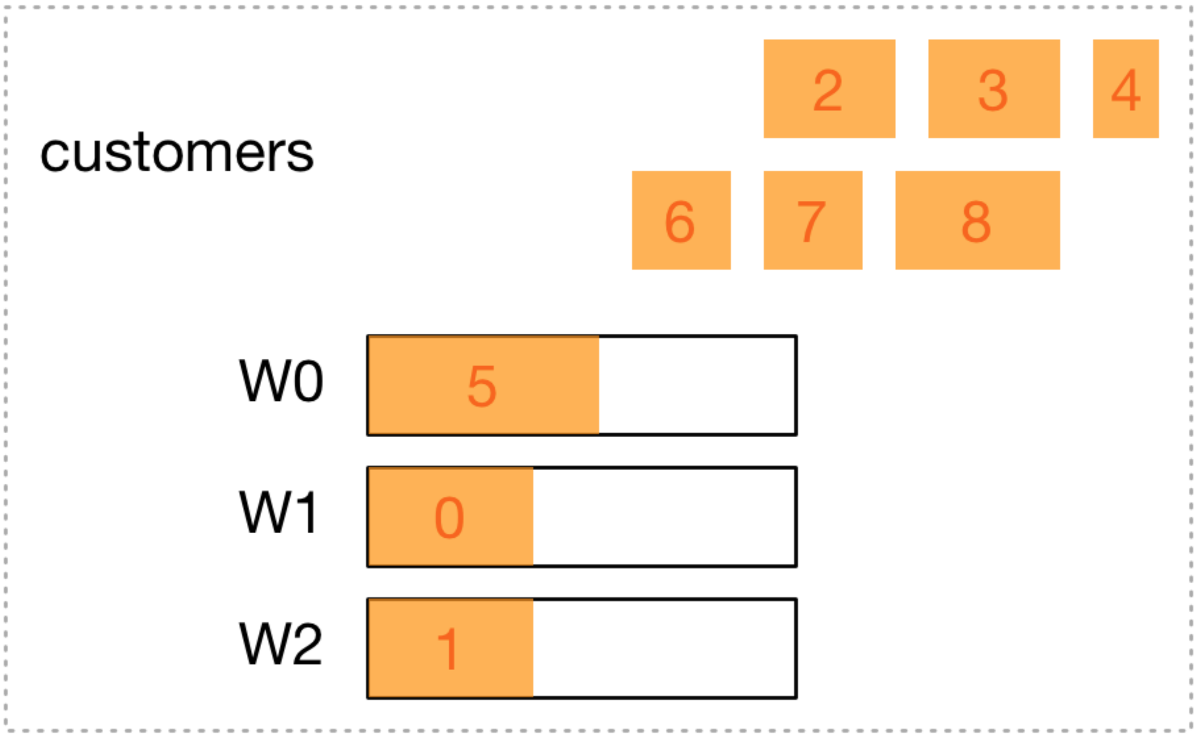
The tie-breaking rule can sometimes be very important

Heuristics for the Warehouses Problem



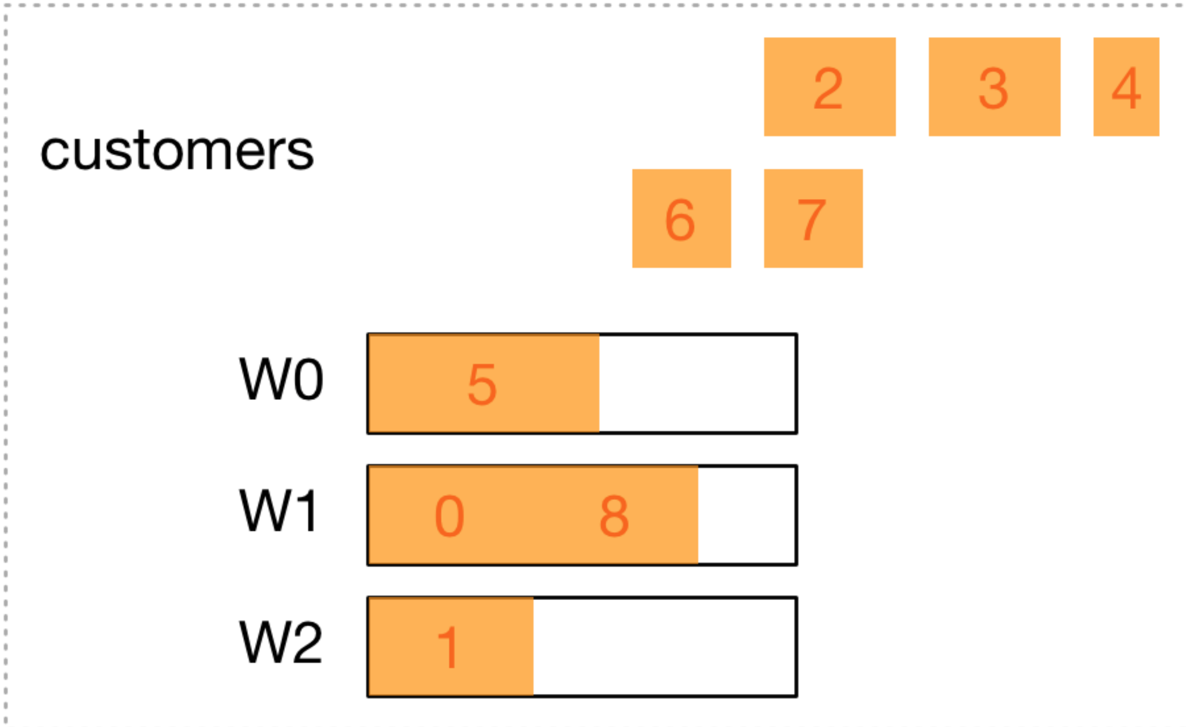
- Pick customer with the largest demand
- Assign to the least loaded warehouse
- Break ties using indices

Heuristics for the Warehouses Problem



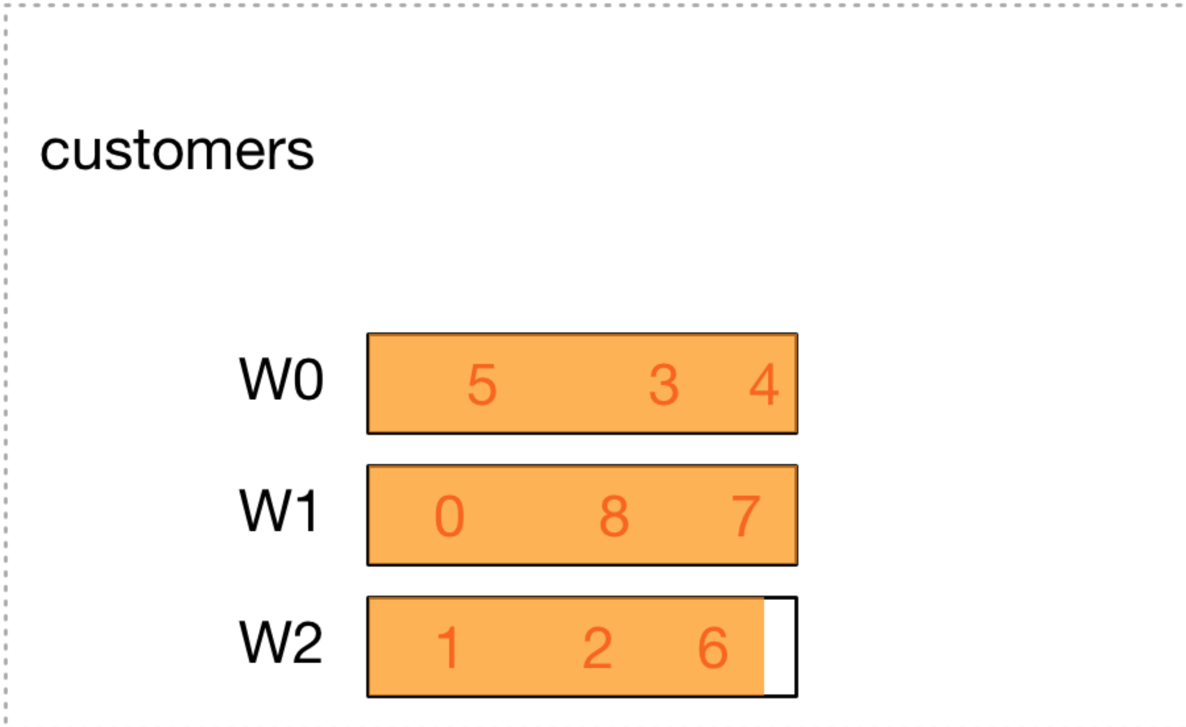
- Pick customer with the largest demand
- Assign to the least loaded warehouse
- Break ties using indices

Heuristics for the Warehouses Problem



- Pick customer with the largest demand
- Assign to the least loaded warehouse
- Break ties using indices

Heuristics for the Warehouses Problem



- Pick customer with the largest demand
- Assign to the least loaded warehouse
- Break ties using indices

Generalizing the Main Idea

Another possibility:

- Pick customer with lowest demand
- Assign to the least loaded warehouse

Main underlying idea: maximize the chance of feasibility

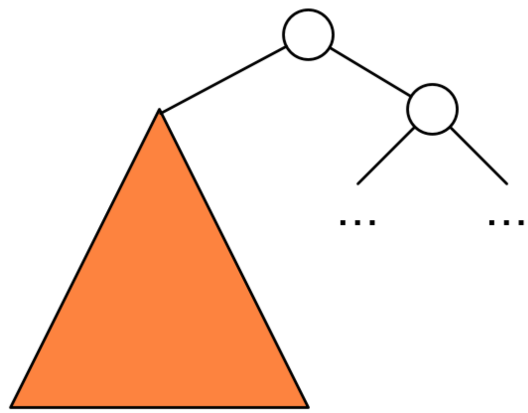
A first general rule, for feasible problems:

Choose variables and values that are likely to yield a feasible solution

Because once we have a feasible solution we are done!

Mistakes Happen

What if we make a mistake?



- We have an infeasible sub-problem
- We need to explore the whole sub-tree before backtracking
- This behavior is called trashing

Our goal: explore the sub-tree as quickly as possible

Heuristics for Infeasible Problems

Say we have:

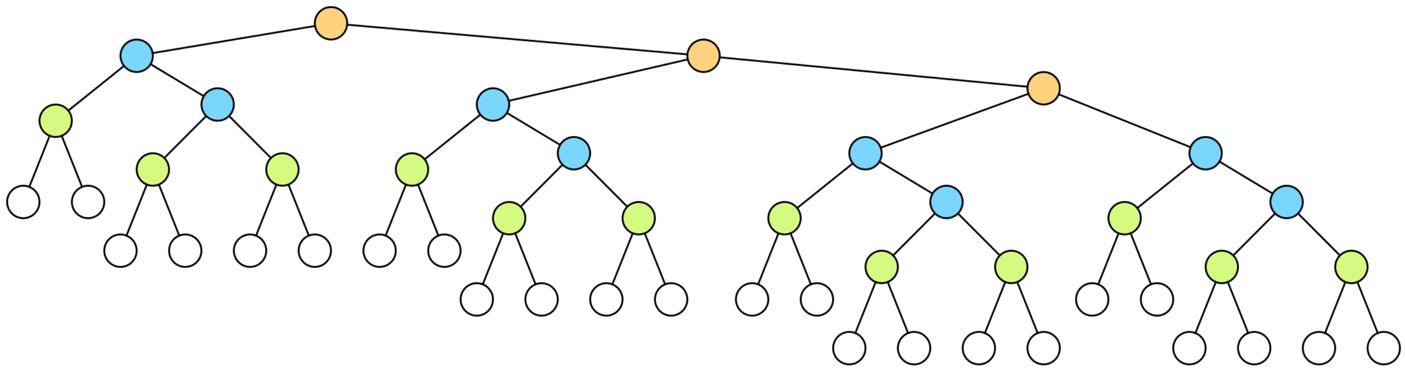
$$x_0 \in \{0, 1, 2, 3\}, x_1 \in \{0, 1, 2\}, x_2 \in \{0, 1\}$$

...and some unspecified constraints

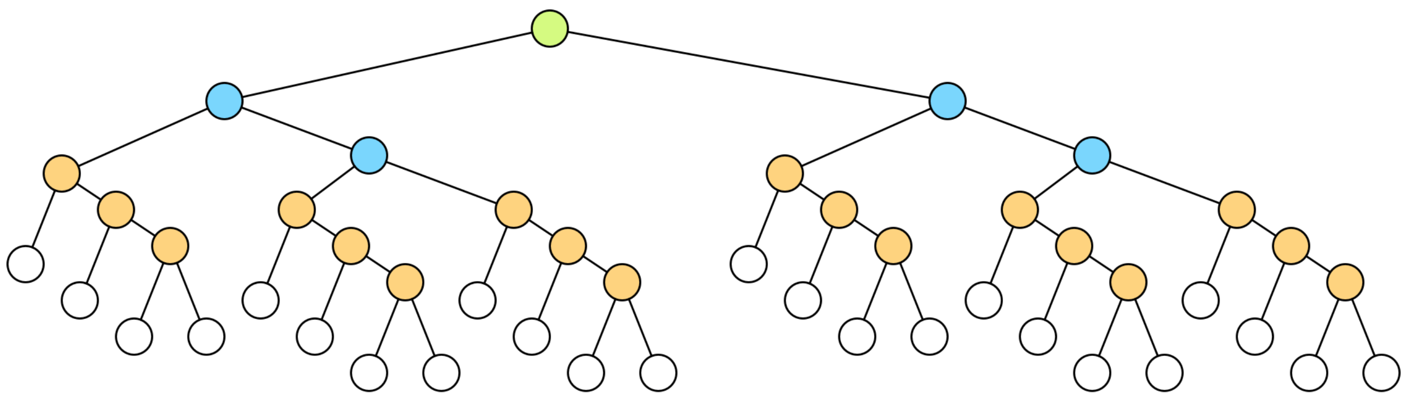
- The raw search space contains $4 \times 3 \times 2$ assignments
- This number is independent on the variable order...
- ...But the shape of the search tree is not!

Heuristics for Infeasible Problems

- Order x_0, x_1, x_2:

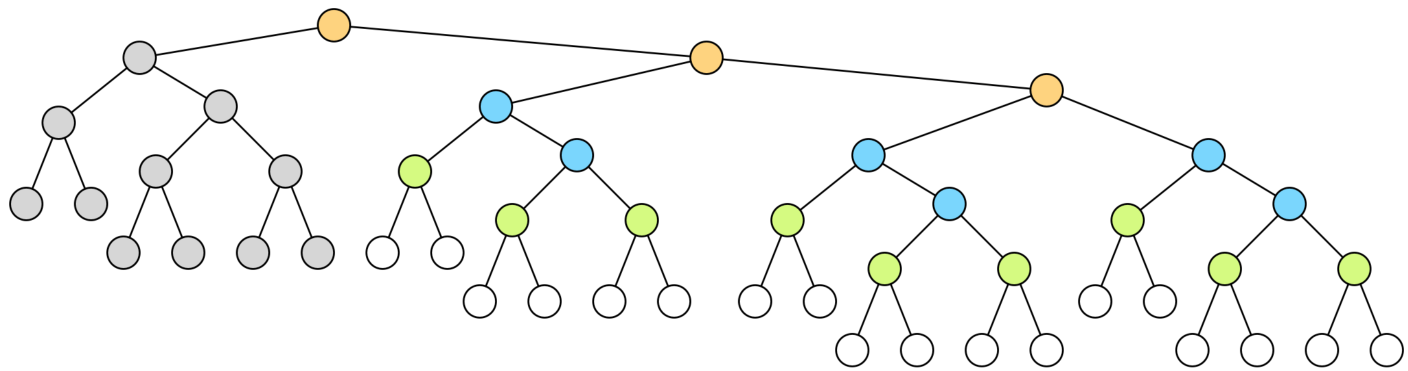


- Order x_2, x_1, x_0:

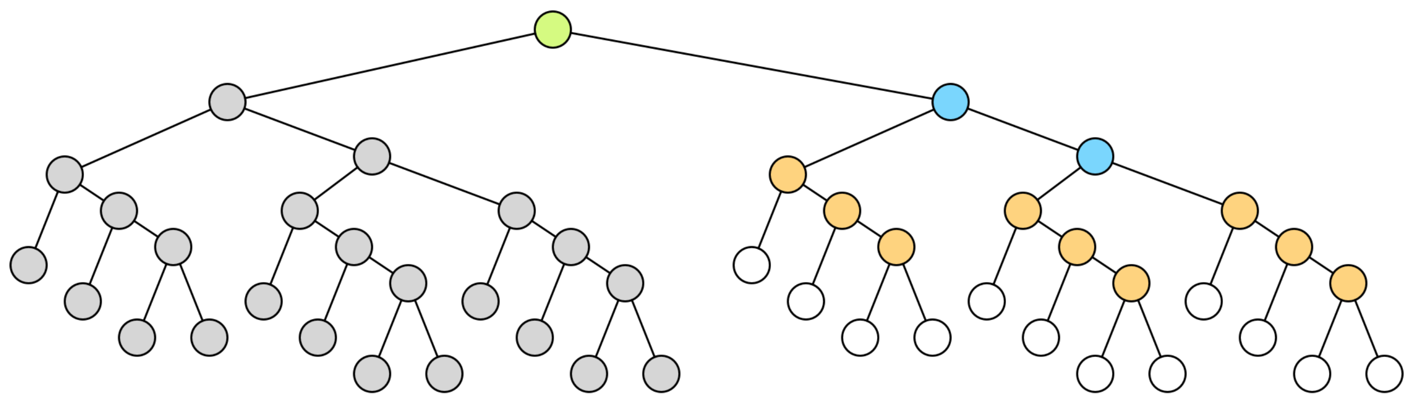


Heuristics for Infeasible Problems

- If propagation prunes a value at depth 1...

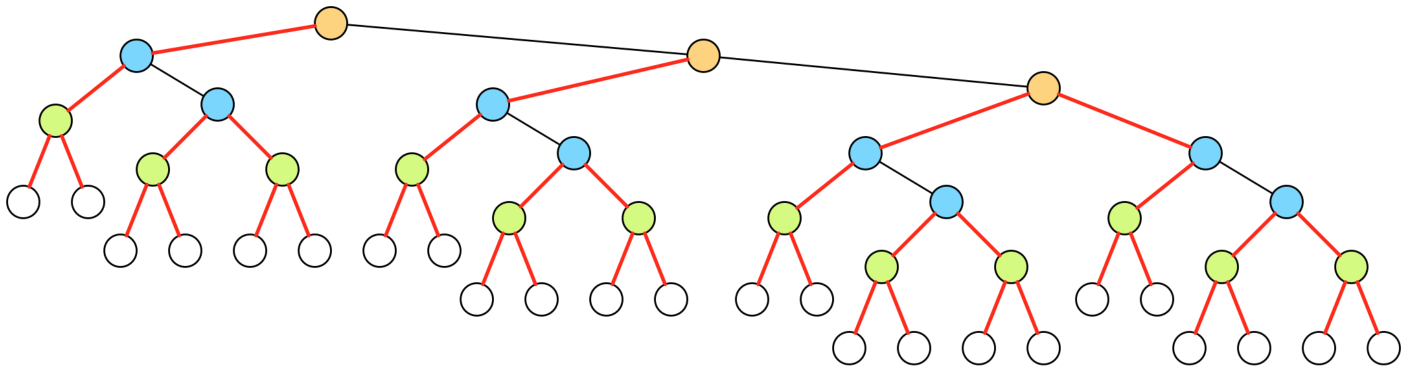


- ...The effect is much stronger with the second ordering!

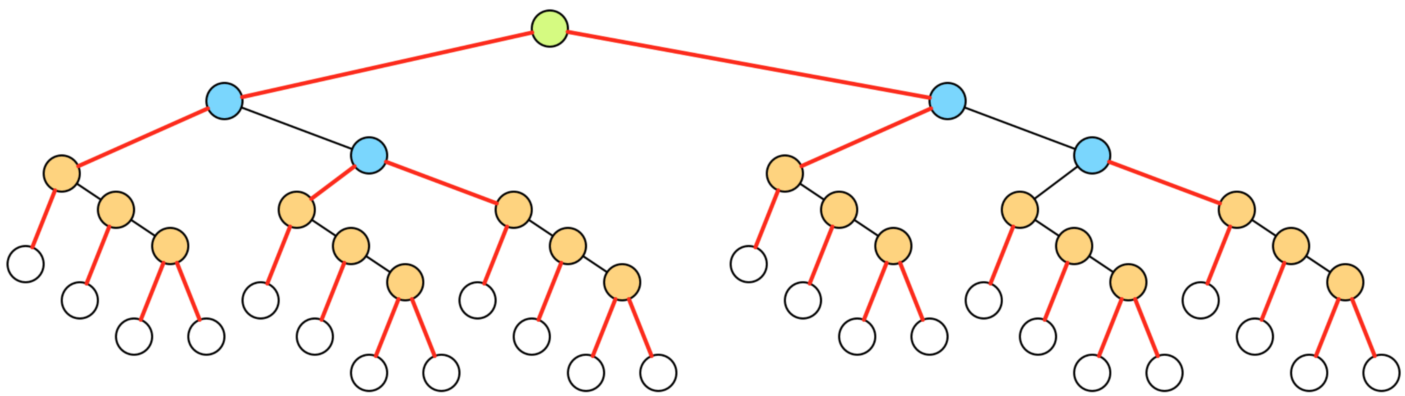


Heuristics for Infeasible Problems

- The number of bound variables (by branching) at low depth…



- …Is greater with the second ordering!



Heuristics for Infeasible Problems

- By branching first on the variable with min size domain
- Propagation is stronger and more likely to occur

The more we prune, the quicker we explore the sub-tree

So, for infeasible (sub) problems we should:

Try to maximize propagation by choosing variables and values that are likely to cause a fail

This is called the first-fail principle

A Compromise

- Usually, we don't know whether a CSP is feasible or not
- Hence, we may want a trade-off:

Choose a variable that is likely to cause a fail,
choose a value that is likely to be feasible

A very frequent setup in practice:

- Variable selection: min size domain
- Value selection: some problem-dependent rule

Optimization Problems

Optimization problems are an interesting case:

- We search for feasible solutions
- We want to prove optimality (requires complete exploration)

Optimization Problems

Optimization problems are an interesting case:

- We search for feasible solutions
- We want to prove optimality (requires complete exploration)

Moreover, with all the optimization methods:

- Whenever we find a new solution, we get a new constraint
- Higher-quality solutions yield tighter constraints
- Tighter constraints prune more, and speed-up the search process

Optimization Problems

A general rule for optimization problems:

Choose variables and values that are likely to yield high-quality solutions

Optimization Problems

A general rule for optimization problems:

Choose variables and values that are likely to yield high-quality solutions

An example for our simple production scheduling problem:

- Choose the start time var s_i with the smallest domain element
- Choose the minimum value in $D(s_i)$

This is likely to lead to small makespan values!

Example: the VM Placement Problem

Let's consider our (improved) VM problem:

$$\min\ z = 1 + \max_{i=0..n_vm-1}(x_i)$$

$$\begin{aligned}
\text{subject to: } & u_j = \sum_{i=0}^{n_{vm}-1} r_i(x_i = j) && \forall j = 0..n_s - 1 \\
& u_j \leq n_c && \forall j = 0..n_s - 1 \\
& u_j \geq u_{j+1} && \forall j = 0..n_s - 2 \\
& x_i < x_j && \forall i, j = 0..n_{vm} - 1 \ :\ i < j, s_i = s_j \\
& x_i \in \{0..n_s - 1\} && \forall i = 0..n_{vm} - 1 \\
& u_j \in \{0..n_c\} && \forall j = 0..n_s - 1
\end{aligned}$$

- The lower bounds on z have been omitted for sake of simplicity

Which var/value heuristics should be choose?

Example: the VM Placement Problem

Starting point: what are our difficulties?

- Finding feasible solutions is easy
- Finding the optimal solution quickly is important
- Proving optimality is difficult

A possible choice:

- Var selection: min size domani (to ease the optimality proof)
- Value selection: min value (quickly find high-quality solutions)

On my laptop, on instance `data-vm-hard/data-vam-7.json`

- Basic search: > 7 sec
- New search < 1 sec

Alternative Branching Schemes

One more major consideration

Let's look again at our DFS algorithm:

function $DFS(CSP)$:
    if a solution has been found: **return** true
    if the CSP is infeasible: **return** false
    for d in $decisions(CSP)$:
        if $DFS(apply(d, CSP))$: **return** true
    return false

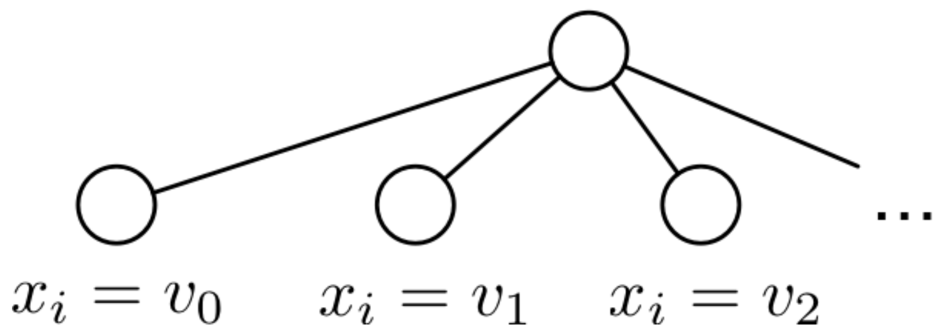
- Binary choice points (i.e. $x_i = v$ or $x_i \neq v$) are a good idea
- But we can use other branching schemes, too

Let's see a few common cases...

Labeling

We can have more than two branches

...And branch over all values of the selected variable



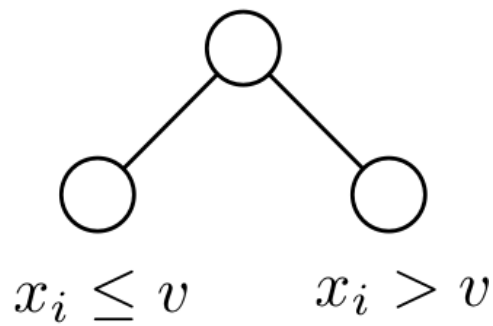
This branching scheme is sometimes called labeling

- Historically important
- Useful in some cases
- ...We will see an example later

Domain Split

We can partition the domain in two halves

...Based on a threshold value



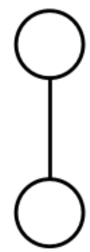
This branching scheme is sometimes called domain splitting

- Useful for variables representing quantities
- Useful for variables with large domains

Probing/Diving

We can try only a single assignment

...With no backtracking



$x_i = v$

This (strange) branching scheme is sometimes called probing or diving

- No domain partitioning, no backtracking
- ...Which means that search will usually be incomplete!
- But also very efficient, when used wisely

We will see an example quite soon

Branching Schemes, an Example

Let's consider our Job Shop Scheduling model:

$$
\begin{aligned}
&\min z = \max_{i=0..n-1} \left(s_{i,m-1} + d_{i,m-1} \right) \\
&\text{subject to: } s_{i,j} + d_{i,j} \leq s_{s,j+1} && \forall i = 0..n-1,\ j = 0..m-2 \\
&(s_{i,j} + d_{i,j} \leq s_{h,k}) \vee (s_{h,k} + d_{h,k} \leq s_{i,j}) && \begin{array}{c}\forall i, j, h, k : i < h \\ m(i,j) = m(h,k)\end{array} \\
&s_{i,j} \in \{0..eoh\} && \forall i = 0..n-1,\ j = 0..m-1
\end{aligned}
$$

- Main variables: start times s_i
- Disjunctions of reified constraints to model resources

Which kind of search strategy shall we use?

Let's see two reasonable alternatives (both with PROs and CONs)

Branching Schemes, an Example

Alternative 1: binary choice points ($s_i = v \vee s_i \neq v$)

Choose s_i with minimum domain minimum, choose minimum v

- PRO: likely to yield to good solutions
- CON: likely poor performance when proving optimality
  - If the durations d_i are large, eoh will be large…
  - …Posting $s_i \neq v$ on backtrack prunes almost nothing!

Alternative 2: domain splitting ($s_i \leq v \vee s_i > v$)

Choose s_i with minimum domain minimum, choose middle v

- PRO: both constraints prune a lot
- CON: more branches needed
  - No constraint actually fixes the s_i variables

What to Branch Upon

In a model, there are often different groups of variables

Typically, we have:

- Main decision variables
- Dependent variables (fixed once the main variables are assigned)
- Cost variable (a special dependent variable)

Let's see one example...

What to Branch Upon

In the VM placement problem:

$$\min\ z = 1 + \max_{i=0..n_vm-1} (x_i)$$

$$\begin{aligned}
\text{subject to: } u_j &= \sum_{i=0}^{n_{vm}-1} r_i(x_i = j) && \forall j = 0..n_s - 1 \\
u_j &\leq n_c && \forall j = 0..n_s - 1 \\
u_j &\geq u_{j+1} && \forall j = 0..n_s - 2 \\
x_i &< x_j && \forall i, j = 0..n_{vm} - 1\ :\ i < j, s_i = s_j \\
x_i &\in \{0..n_s - 1\} && \forall i = 0..n_{vm} - 1 \\
u_j &\in \{0..n_c\} && \forall j = 0..n_s - 1
\end{aligned}$$

- Main variables: x_i
- Dependent variables: u_j
- Cost variable: z

What to Branch Upon

In a model, there are often different groups of variables

Typically, we have:

- Main decision variables
- Dependent variables (fixed once the main variables are assigned)
- Cost variable (a special dependent variable)

In most cases, we want to branch on the main variables

- The dependent variables will be fixed as a consequence

But sometimes it helps to branch on dependent variables!

- For example, it may help propagation...
- ...Or it may simplify the problem

Drawback: afterwards, we will still need to branch on the main vars

What to Branch Upon: an Example

Let's consider our two alternatives for Job Shop Scheduling:

- Binary choice points ($s_i = v \lor s_i \neq v$)
  - s_i with minimum domain minimum, choose minimum v
  - Finds good first solution, bad for trashing/optimality proof
- Domain splitting ($s_i \leq v \lor s_i > v$)
  - s_i with minimum domain minimum, choose middle v
  - Many branches necessary, perhaps sub-par initial solution

What to Branch Upon: an Example

Let's consider our two alternatives for Job Shop Scheduling:

- Binary choice points ($s_i = v \vee s_i \neq v$)
  - s_i with minimum domain minimum, choose minimum v
  - Finds good first solution, bad for trashing/optimality proof
- Domain splitting ($s_i \leq v \vee s_i > v$)
  - s_i with minimum domain minimum, choose middle v
  - Many branches necessary, perhaps sub-par initial solution

Let's make an important consideration:

- We do not really need to choose exactly when to start the activities
- We just need to order them on each resource

At that point, all resource constraints will be satisfied

- We can just start every activity ASAP, and we will get a solution

What to Branch Upon: an Example

How can we turn this idea into a search strategy?

Consider the constraints used to model the disjunctive resources:

$$(s_{i,j} + d_{i,j} \leq s_{h,k}) \vee (s_{h,k} + d_{h,k} \leq s_{i,j})$$

What to Branch Upon: an Example

How can we turn this idea into a search strategy?

Consider the constraints used to model the disjunctive resources:

$$(s_{i,j} + d_{i,j} \leq s_{h,k}) \vee (s_{h,k} + d_{h,k} \leq s_{i,j})$$

And the reformulation:

$$y_{(i,j),(h,k)} = (s_{i,j} + d_{i,j} \leq s_{h,k})$$
$$y_{(h,k),(i,j)} = (s_{h,k} + d_{h,k} \leq s_{i,j})$$

- Where $y_{(i,j),(h,k)}, y_{(h,k),(i,j)}$ are new, dependent, binary variables...
- ...And they are subject to:

$$y_{(i,j),(h,k)} + y_{(h,k),(i,j)} = 1$$

- I.e. only one of the two can be equal to 1

What to Branch Upon: an Example

We can now branch over the y variables!

By doing so, we take ordering decisions

- Posting $y_{(i,j),(h,k)} = 1$ activates:

$$s_{i,j} + d_{i,j} \leq s_{h,k}$$

- Posting $y_{(i,j),(h,k)} \neq 1$ on triggers $y_{(h,k),(i,j)} = 1$
- Which in turn activates:

$$s_{h,k} + d_{h,k} \leq s_{i,j}$$

Both the precedence constraints propagate quite effectively

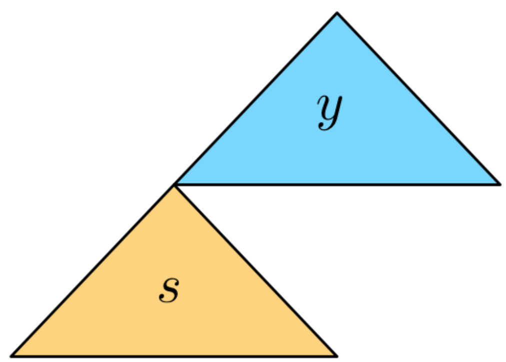
- With a proper choice of a heuristic on the y vars…
- …This search strategy can be very effective

What to Branch Upon

Once all y variables are bound:

- All machine capacity constraints are resolved
- We just have a collection of end-to-start precedence constraints
- But the start time variables are not (yet) assigned!

So, we need a second search phase over the s vars:



What to Branch Upon

We can use the original strategy:

"Pick s_i with minimum $\min(D(s_i))$, assign $\min(D(s_i))$"

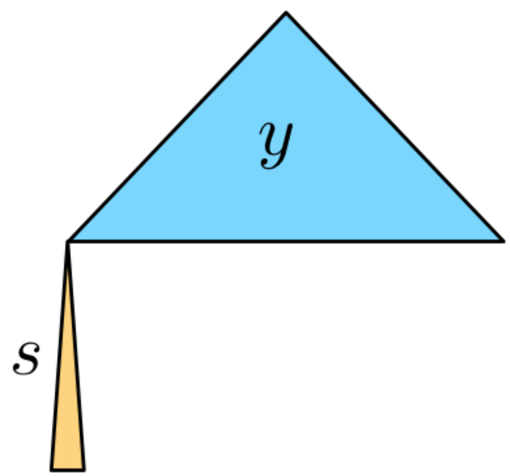
What to Branch Upon

We can use the original strategy:

"Pick s_i with minimum $\min(D(s_i))$, assign $\min(D(s_i))$"

- But if the we have only end-to-start constraints...
- ...This strategy always finds the best makespan

We can use probing for this second phase!



What to Branch Upon: An Interesting Case

We can even choose to branch on the cost variable

That changes radically how the solver operates

- We will still need to branch on the main variables afterwards…
- …But we may get some advantages

A first interesting case (HP: we want to minimize z):

Branching on z, assign min value, then branch on the main vars

- After each $z = v$ branch, either we have the optimal solution…
- …Or we prove that the best solution is larger than v

This is a CP implementation of destructive lower bounding

- Typically less efficient than branch & bound…
- …But we iteratively compute a valid lower bound on z

What to Branch Upon: An Interesting Case

We can even choose to branch on the cost variable

That changes radically how the solver operates

- We will still need to branch on the main variables afterwards...
- ...But we may get some advantages

A second interesting case (HP: we want to minimize z):

Branching on z, domain split on the middle value v

- After each $z \leq v$ branch, we know that the optimal solution...
- ...Is either lower than v or higher than v

This is a CP implementation of the bisection method/binary search

- Typically less efficient than branch & bound...
- ...But we always have a valid lower and upper bound on z

Value Symmetries

We have define the concept of variable symmetry:

A problem has a variable symmetry iff:

- there exists a permutation π of the variable indices
- s.t. for each feasible solution…
- …we can re-arrange the variables according to π…
- and obtain another feasible solution

- Swapping variables = re-assigning the values
- The permutation π identifies a specific symmetry

Value Symmetries

Similarly, we can say that

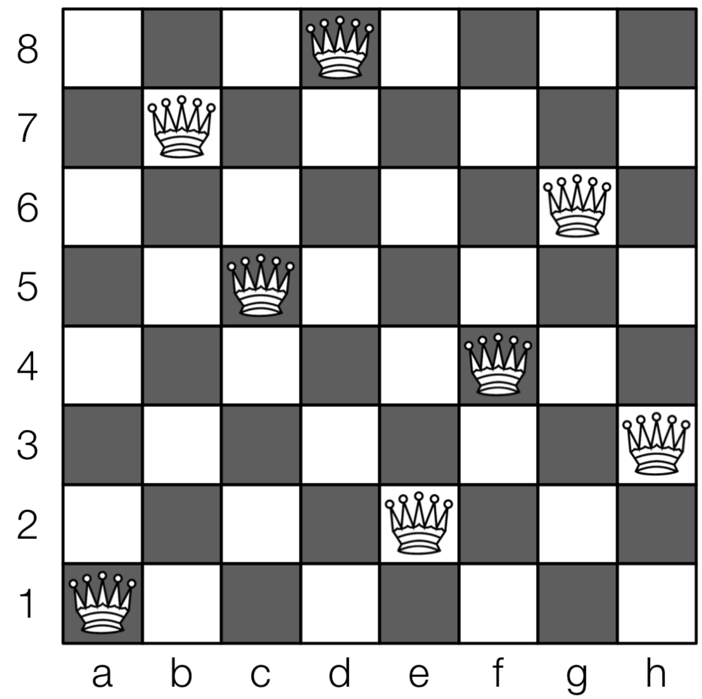
A problem has a value symmetry iff:

- there exists a permutation π of the values
- s.t. for each feasible solution
- we can replace the variable values according to π
- and obtain another feasible solution

- Intuitively: replacing the values = renaming the values
- The permutation π identifies a specific symmetry

Value Symmetries - Example

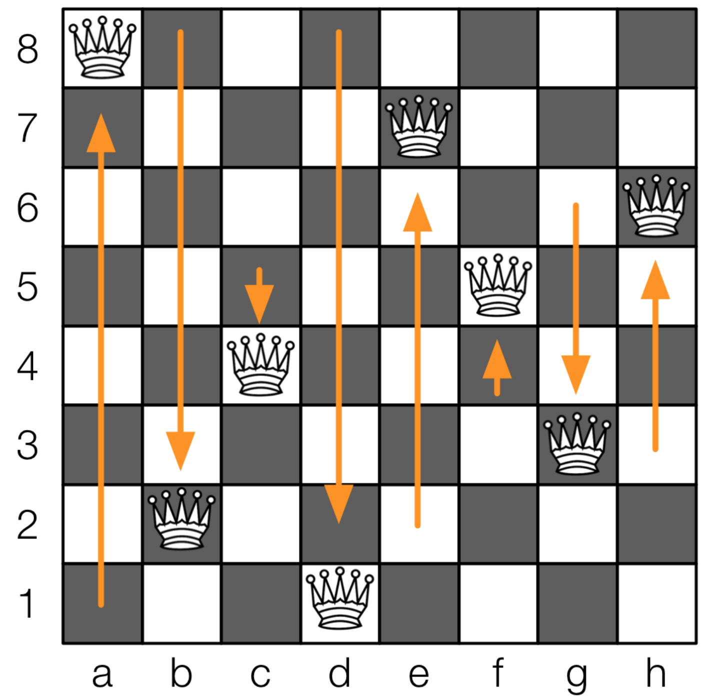
E.g. given a solution for our nqueens model:



- We can swap value 0 with n-1, 1 with n-2...

Value Symmetries - Example

E.g. given a solution for our nqueens model:



- We can swap value 0 with n-1, 1 with n-2...
- ...and obtain a new feasible solution

Intuitively: we flip the chessboard on the x-axis

Breaking (Value) Symmetries: an Example

In our VM placement problem all servers were equivalent

- That's a value symmetry! What can we do about it?

We can add a symmetry breaking constraint

- For example, in our improved model we added:

$$u_j \geq u_{j+1} \quad \forall j = 0..n_s - 2$$

- Another possibility: force the assignment for a single VM

$$x_0 = 0$$

Devising breaking constraints for value symmetries may be tricky:

- There exists a general method, but it is complex...
- ...And it may generate a huge number of constraints

Dynamic Symmetry Breaking

Moreover, as we have mentioned:

Symmetry breaking constraint may antagonize search

E.g. they may forbid the solution on the left-branch

- In the VM placement problem, using...

$$u_j < u_{j+1} \quad \forall j = 0..n_s - 2$$

- ...Causes a huge drop in performance

A possible solution:

**Dynamic symmetry breaking =
break symmetries at search time**

Dynamic Symmetry Breaking

Can this be done in an automated fashion? Kind of:

- Symmetry Breaking During Seach (SBDS)
  - Automatic, given a list of symmetries/permutations
- Symmetry Breaking with Dominance Check (SBDD)
  - Automatic, given a symmetry checking function

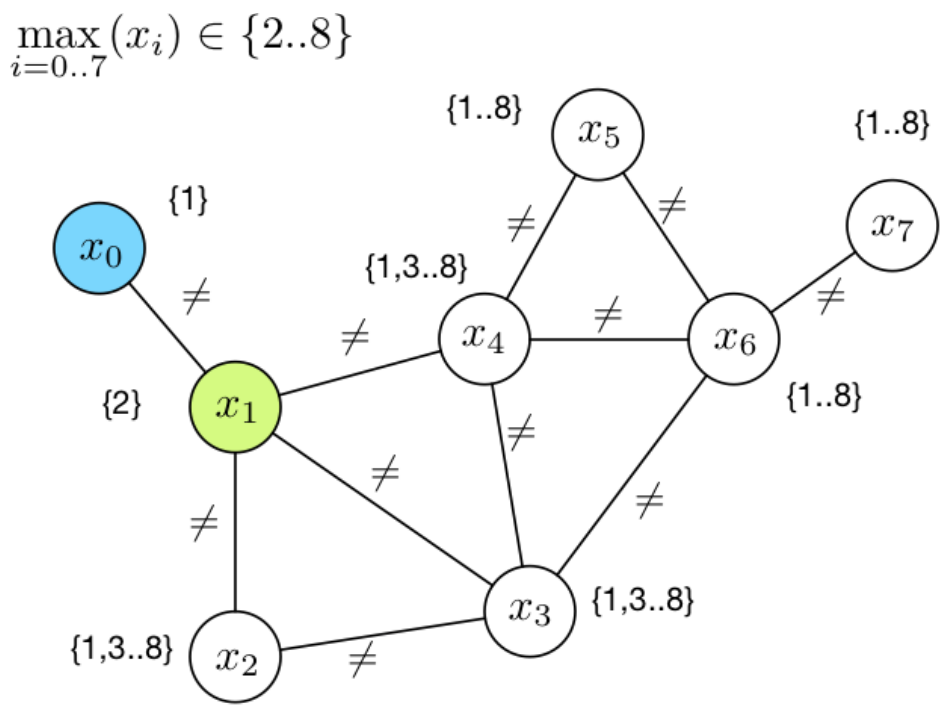
If you are interested, check the papers on the course web site

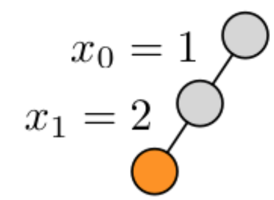
Main idea, in both cases: avoid symmetries on backtrack

- We can use it as a rule-of-thumb to design custom heuristics
- Let's see an example...

Dynamic Symmetry Breaking

Consider this B&B state for our map coloring example:



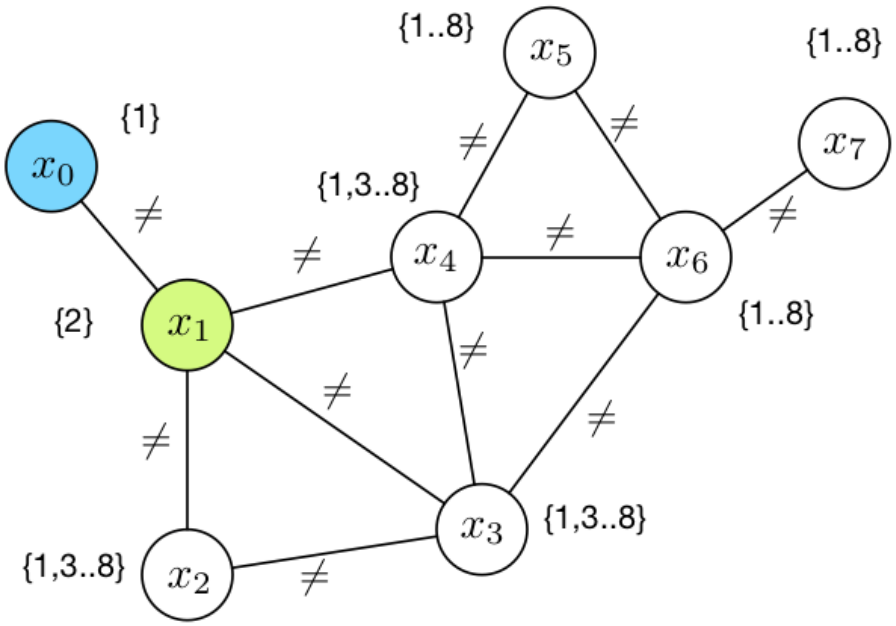


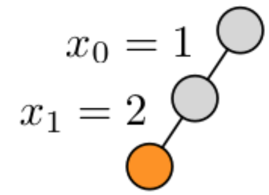
- We need to assign x_2

Dynamic Symmetry Breaking

Consider this B&B state for our map coloring example:

$$\max_{i=0..7}(x_i) \in \{2..8\}$$

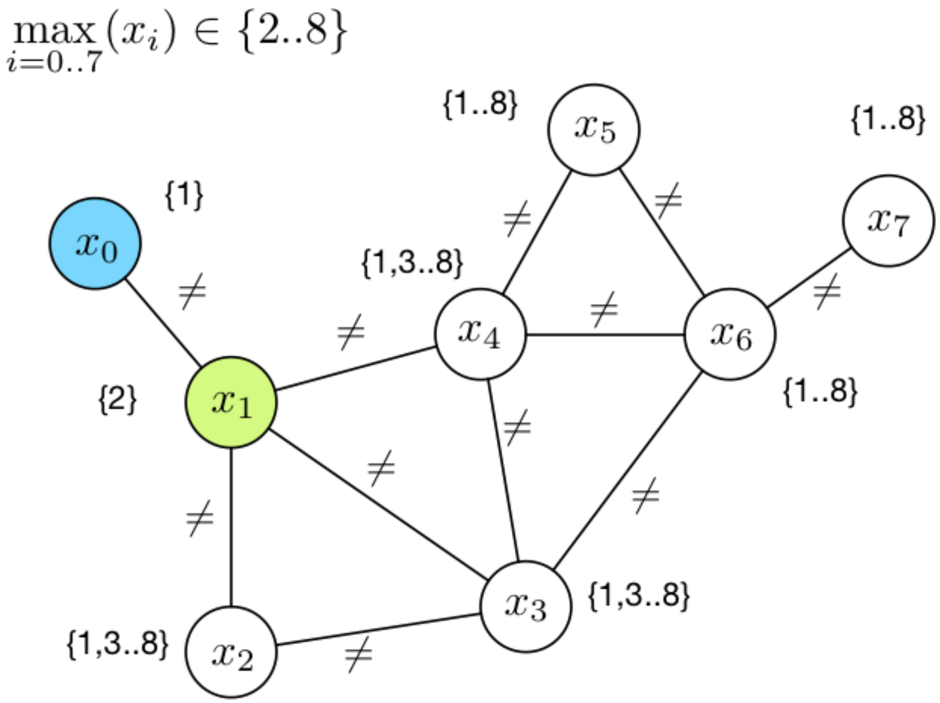


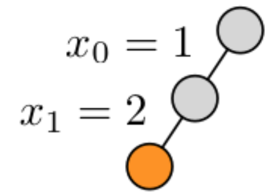


- Colors $3..8$ are all unused: they are symmetric values!

Dynamic Symmetry Breaking

Consider this B&B state for our map coloring example:



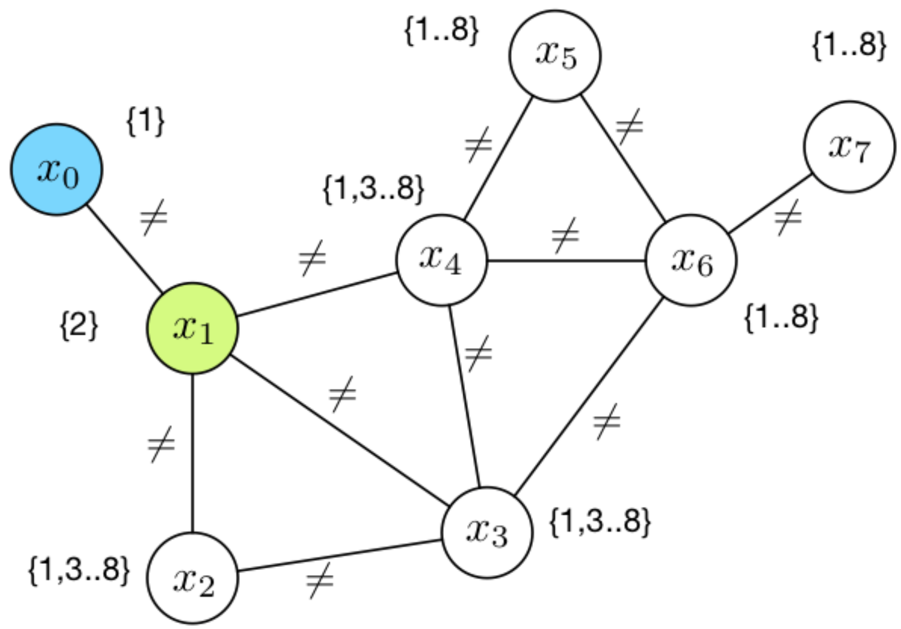


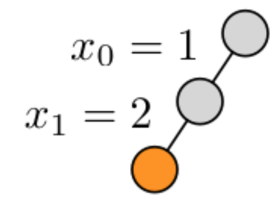
- If we try value 3 and we fail…

Dynamic Symmetry Breaking

Consider this B&B state for our map coloring example:

$$\max_{i=0..7}(x_i) \in \{2..8\}$$





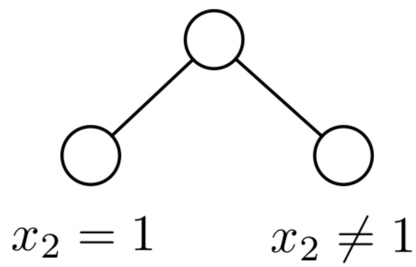
- ...there is no need to try $4..8$!

Dynamic Symmetry Breaking

How can we take advantage of this information?

A simple approach:

- Instead of opening this choice point…

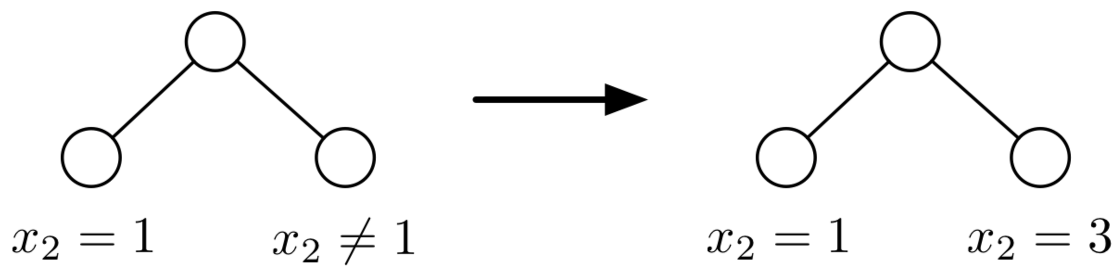


Dynamic Symmetry Breaking

How can we take advantage of this information?

A simple approach:

- Instead of opening this choice point... we open this one



- A bit dirty (we had to switch to labeling)
- But does the job (symmetric values are not considered)

Search: Wrap Up

Search in CP is extremely flexible

- var/val selection heuristics
- branching schemes
- branching variables

Search is so flexible that:

- We can use CP to implement any constructive heuristic
- And get propagation for free!

Search: Wrap Up

Too much flexibility?

- Allows for powerful custom approaches
- Customization is often needed for best results
- This may be difficult for non-experts
- Several (very interesting) attempts to devise robust, general, strategies
- ...But this is an important open research question

The best search strategy depends on the problem

In practice, we usually employ:

- Some rule of thumbs (feasible, infeasible, optimization)
- Nothing is written in stone!
- Use your intuition... and experiment a lot