Intelligent Systems Programming

Lecture 8: **Constraint Programming II**

1. COP and Symmetries
2. Global Constraints
3. Constraint Propagation Systems
4. Examples: Sudoku and Container Stowage problem

IT University of Copenhagen

Based partly on Christian Schulte's slides

Today's Program

- [10:00-10:50]
  - Bounds Consistency
  - Constraint Optimization Problem (COP)
  - Symmetries
  - Global Constraints
  - Example 1: Sudoku
- [11:00-11:50]
  - Constraint propagation systems (Gecode)
  - Example 2: Container stowage problem

Bounds Consistency

- For every variable *X*, there is a support value for its *lower* and *upper* bound in any variable *Y* related through a constraint with *X*.

  e.g.

  $X_1 \in \{0, \ldots, 165\}, \quad X_2 \in \{0, \ldots., 385\}$

  $X_1 + X_2 = 420$

  $X_1 \in \{35, \ldots, 165\},$

  $X_2 \in \{255, \ldots, 385\}$

Optimality

- **C**onstraint **O**ptimization **P**roblem (COP) :
  CSP + Objective to maximize.

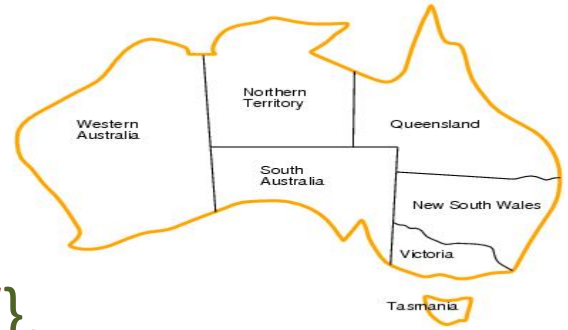
  e.g. $X_1, X_2 \in \{1, 2, 3, 4\}$

  $X_1 > X_2$

  Max $X_1 + X_2$

- Solve with Branch and Bound (Backtracking Based)
  1. Run usual backtracking
  2. When a solution with value v is found, add constraint: $obj > v$

Symmetries



- Multiple equivalent solutions:

  e.g. {*NT* = *blue*, *SA* = *green*, *WA* = *red*},

  {*NT* = *green*, *SA* = *blue*, *WA* = *red*}, ...

- Multiple equivalent inconsistent assignments:

  e.g. {*NT* = *blue*, *SA* = *green*, *WA* = *green*},

  {*NT* = *green*, *SA* = *blue*, *WA* = *blue*}, ...

Symmetries

- Map coloring with *n* colors has *n!* permutations for every solution.
- Value symmetry.
- Add symmetry-breaking constraint e.g., $NT < SA < WA$.
- In general breaking all symetries is NP-hard.

Global Constraints

Definition

- Depends on more than 2 variables

Generalized Arc Consistency (GAC)

- For all v in X_i of constraint c, there exist a valid assignment for all the remaining variables

Ex

- $x, y, z \in \{1,2,3\},\ \ x > y = z\ \ : \{(3,2,2),(3,1,1),(2,1,1)\}$
  $\text{GAC} : x \in \{2,3\}, y \in \{1,2\}, z \in \{1,2\}$

Alldifferent

- Alldifferent: all $X_i \in$ scope of constraint are assigned to different values.

  e.g. $X_1, \ldots, X_4 \in \{0, 1, 2, 3, 4\}$ ***Alldiff***(X_1, X_2, X_3, X_4)

  $X_1=0, X_2=1, X_3=2, X_4=3$ ✔

  $X_1=0, X_2=1, X_3=1, X_4=2$ ✖

- How to represent Alldifferent with binary ≠ constraints?

Simple approximation to AllDiff GAC

1. If #values < #vars then return failure

   $X_1=0, X_2 \in \{1, 2\}, X_3 \in \{1, 2\}, X_4=3$ ✔

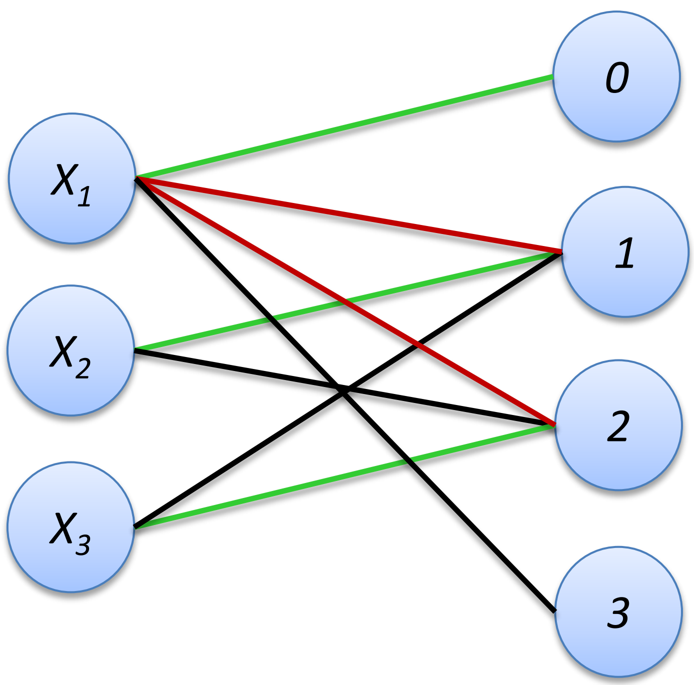
   $X_1 \in \{1, 2\}, X_2 \in \{1, 2\}, X_3 \in \{1, 2\}, X_4=3$ ✖

2. Remove all vars with singleton domains
3. Remove singleton values from remaining domains
4. Goto 1

Fast Computation of AllDiff GAC

– GAC = Find all max matchings in bipartite graph

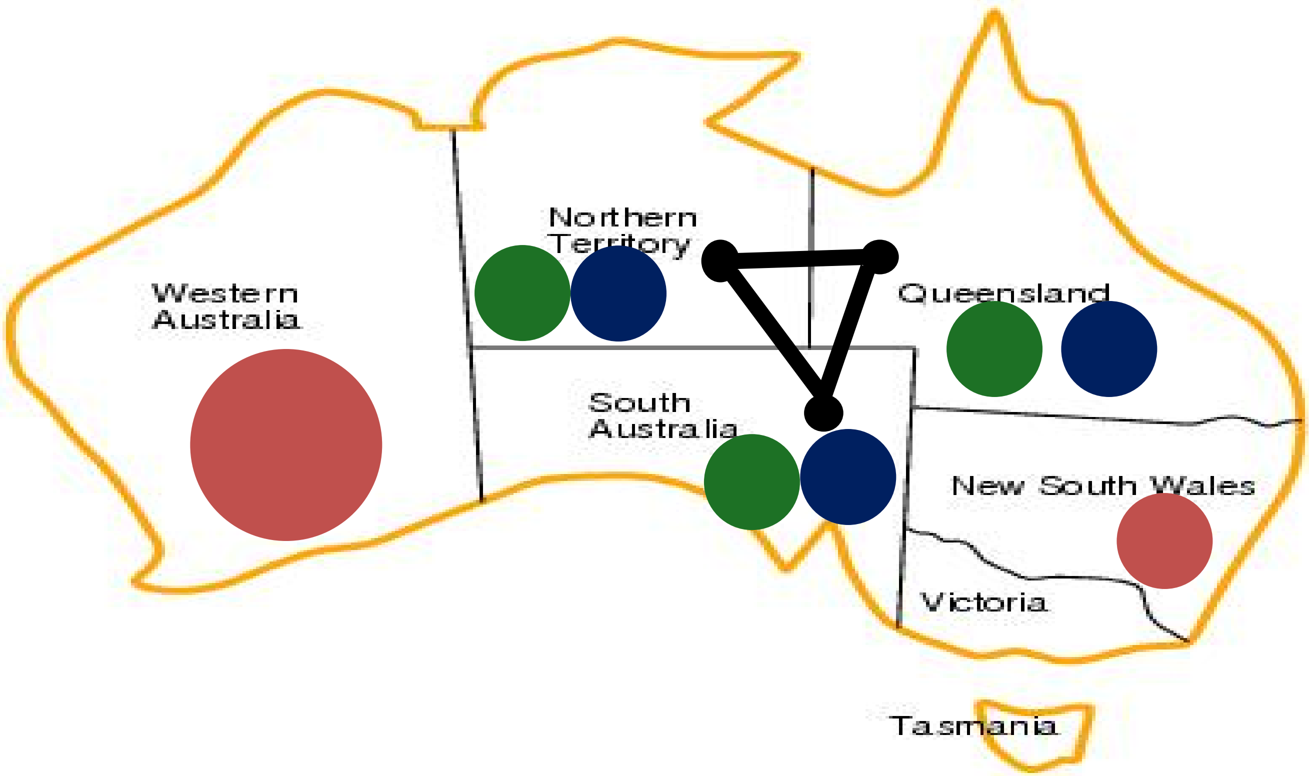
e.g. $X_1 \in \{0, 1, 2, 3\}$, $X_2 \in \{1, 2\}$, $X_3 \in \{1, 2\}$



Can be shown to be possible in $O(d\sqrt{n})$

Where AC-3 on $O(d^2)$ not equal constraints takes $O(d^5)$

AllDiff Stronger than binary constraints



Inconsistency detected by AllDiff GAC, but not by AC-3!

Example 1: Sudoku

			2		5			
	9					7	3	
		2			9		6	
2						4		9
				7				
6		9						1
	8		4			1		
	6	3					8	
			6		8			

- Assign blank fields digits such that:

  digits distinct per rows, columns, blocks

Example 1: Sudoku

- Variables:

  $X=\{X_{ij} \mid i \in rows,\ \ j \in columns\}$

- Domains:

  $X_{ij} \in \{1, 2, 3, \ldots, 9\}$, for all $i \in rows,\ \ j \in columns$

- Constraints:

  $Alldiff(X_{1j}, \ldots, X_{9j})$, for all $j \in\ rows$

  $Alldiff(X_{i1}, \ldots, X_{i9})$, for all $i \in\ columns$

  $Alldiff($ getVarsBlock(i) $)$, for all $i \in blocks$

Linear Expressions

- Linear expressions:
  - Specific case:

$$\sum X_i \leq v$$

e.g. $X_1, \ldots, X_4 \in \{1, \ldots, 10\}$

$$\sum_{i=1}^{4} X_i \leq 10$$

$X_1=1, X_2=2, X_3=3, X_4=4$ ✔

$X_1=6, X_2=6, X_3=1, X_4=1$ ✖

Linear Expressions

- Detecting inconsistencies in linear constraints ($\sum X_i \leq v$)
  - Basic consistency rule: $\sum D_i^- \leq v$, where D_i^- is the smallest value in D_i (Lower bound).

    e.g.

$X_1 \in \{0, 1, 2, 3\}, X_2 \in \{1, 2\}, X_3 \in \{1, 2\}, X_4 \in \{1, 2\}$ ✔

$X_1 \in \{3, 4, 5, 6\}, X_2 \in \{3, 4, 5, 6\}, X_3 \in \{3, 4, 5, 6\}, X_4 \in \{3, 4, 5, 6\}$ ✖

Break

Constraint propagation systems

World's best **Gecode** (link on ISP page)

- Constraint Store
- Propagators
- Propagator Loop
- Search

Constraint store

$x \in \{3,4,5\}$ $y \in \{3,4,5\}$

- Maps variables to possible values

Constraint store

finite domains

$x \in \{3,4,5\}$ $y \in \{3,4,5\}$

- Maps variables to possible values
- Others: finite sets, continuous domains, trees, ...

Constraints and Propagators

- Constraints state relations among variables

  $x + y + z + w \geq k, \quad x = 2*z, \; w \in \{3, 4, 5\}$

- Propagators implement constraints
  - prune values in conflict with constraint

- Constraint propagation drives propagators for several constraints

Propagators

$x \geq y$ $y > 3$

$x \in \{3,4,5\}$ $y \in \{3,4,5\}$

- Amplify store by constraint propagation

Propagators

$x \geq y$ $\quad$ $y > 3$

$x \in \{3,4,5\}$ $\quad$ $y \in \{3,4,5\}$

- Amplify store by constraint propagation

Propagators

$x \geq y$ $\quad$ $y > 3$

$x \in \{3,4,5\}$ $\quad$ $y \in \{4,5\}$

- Amplify store by constraint propagation

Propagators

$x \geq y$ $y > 3$

$x \in \{3,4,5\}$ $\mathbf{y \in \{4,5\}}$

- Amplify store by constraint propagation

Propagators

$x \geq y$ $\qquad\qquad$ $y > 3$

$x \in \{4,5\}$ $\;$ $y \in \{4,5\}$

- Amplify store by constraint propagation
- Disappear when done (subsumed, entailed)
    - no more propagation possible

Propagators

$x \geq y$

$x \in \{4,5\}$ $y \in \{4,5\}$

- Amplify store by constraint propagation
- Disappear when done (subsumed, entailed)
    - no more propagation possible

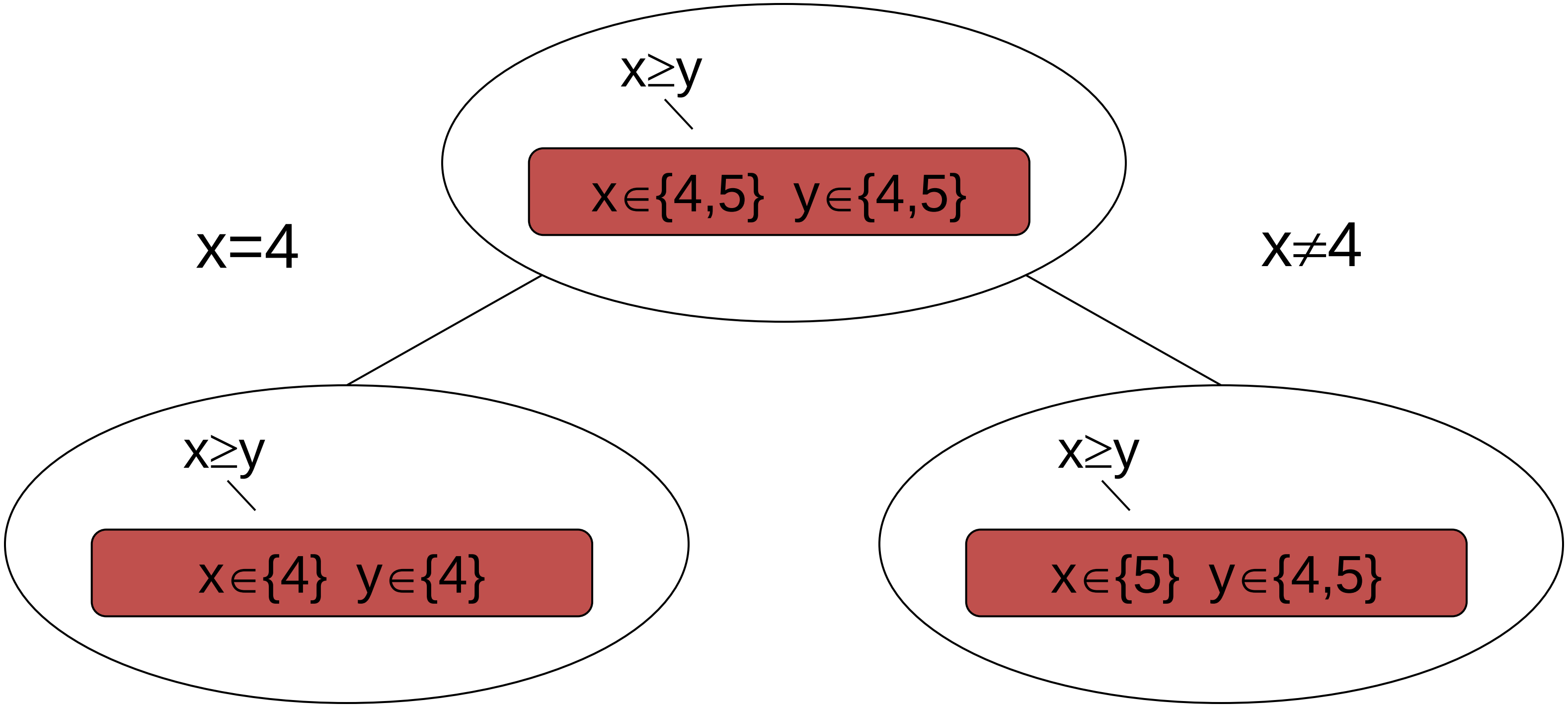
Propagation loop

function PROPAGATE(*csp*)
  $Q \leftarrow \{c_1, c_{2,} \ldots, c_n\}$ // Queue of propagators from *csp*
  while $Q \neq \{\}$ **do**
    $c \leftarrow$ REMOVE-FIRST(Q)
    EXECUTE(c) // run propagator, prune domains of variables
    if any domain in *csp* is empty **then**
      return *false*
    for each $X_i \in$ SCOPE(c) s.t. D_i was narrowed **do**
      ADD(GET-PROPS(X_i) - c) to Q //Add new propagators
    end for
  end while // Fixed point reached!!
return *true*

Search

- Propagation **alone** is not enough!
- Search: Explore search tree for solutions (Backtracking)
- Branching: Select variable and value (define search tree)
  - Minimum remaining values
  - Degree
  - Least constraining value

Search: Branching



- Create subproblems with additional information
- Enable further constraint propagation

Constraint propagation systems

- Each node in the search tree has different information.
- Constraint store and propagators are encapsulated in a Space.
- Manipulate spaces: Copy , add new constraints, ask for solutions, etc.

Search algorithm

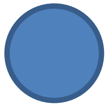
function SEARCH(*csp*) **returns** solution or failure
  csp ← PROPAGATE(*csp*)
  if *csp* is *failure* **then return** *failure*
  if *csp* is solved **then return** SOL(*csp*)
  csp' ← *csp*
  ADD(*csp'*, BRANCH(*csp'*,*1*))
  csp'' ← *csp*
  ADD(*csp''*, BRANCH(*csp''*,*2*))
  solution ← SEARCH(*csp'*)
  if *solution* is valid **then return** *solution*
  else return SEARCH(*csp''*)

function BRANCH(*csp*, *nbranch*)
  return the constraint representing the nbranch according to variable
    and value heuristic selection.

Search

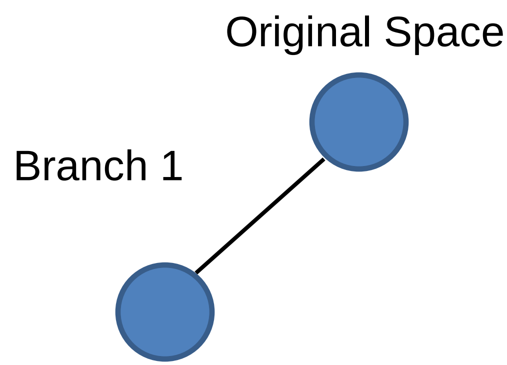
- Let’s try to implement the search

Original Space



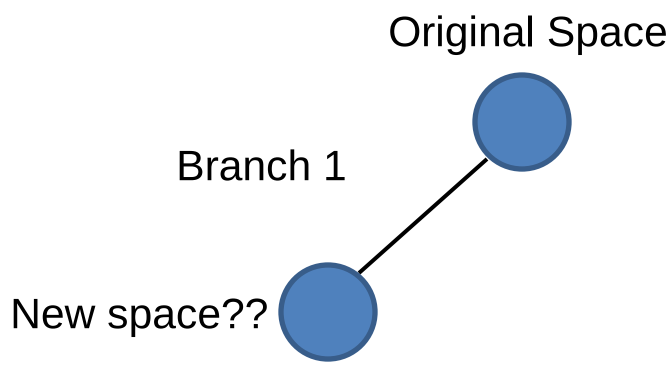
Search

- Let’s try to implement the search



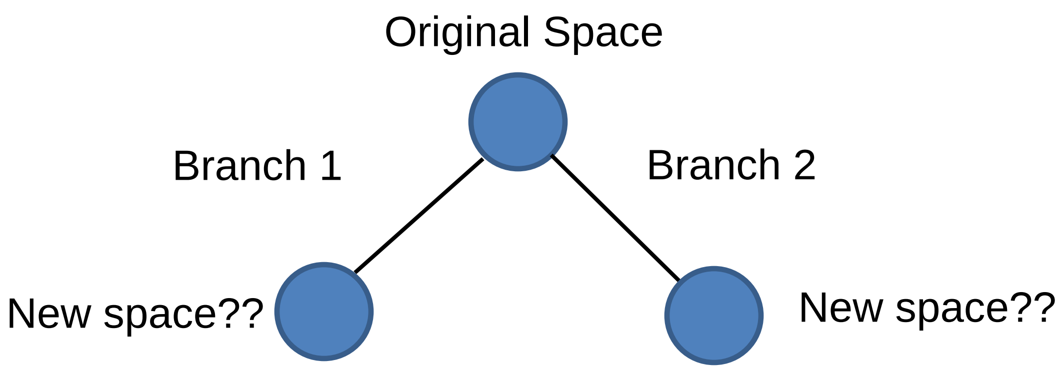
Search

- Let's try to implement the search

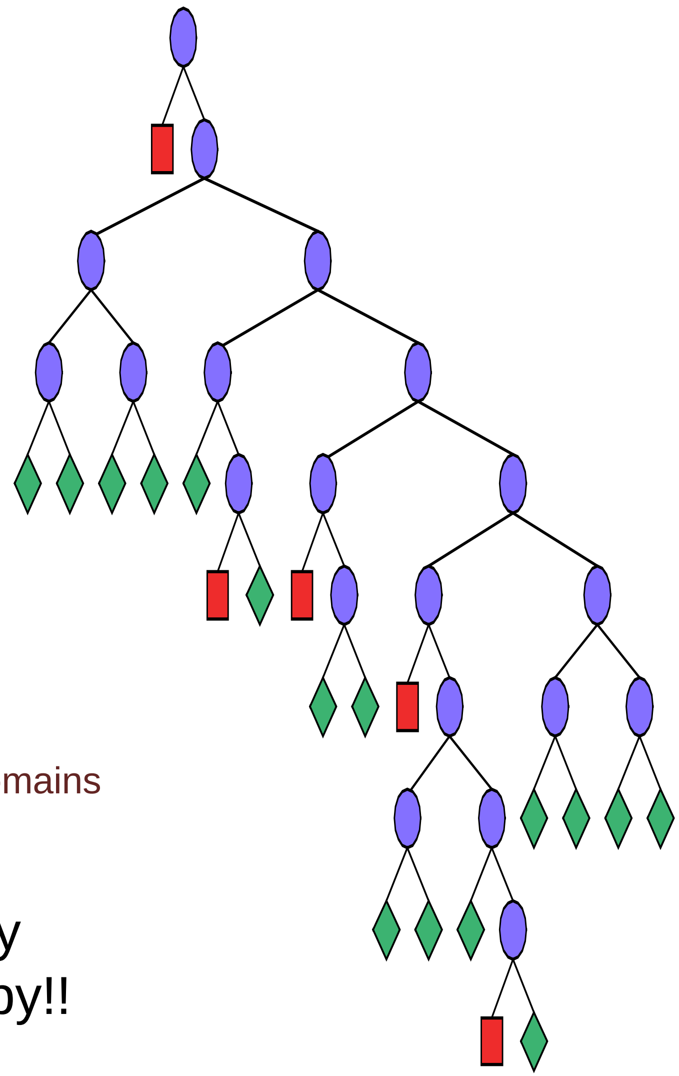


Search

- Let’s try to implement the search



Search



All new spaces??

Worst case scenario:

2^{nd} spaces!!!

n = # of variables

d = Average size of domains

A lot of memory
and time to copy!!

Search

Possible optimizations:

- Delete all failed nodes and branches.

  Reduction in memory but not in time

- Copy one space per node, not two.

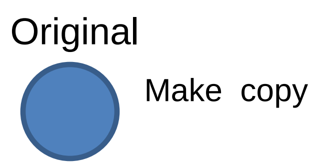
Search

Possible optimizations:

- Delete all failed nodes and branches.

  Reduction in memory but not in time

- Copy one space per node, not two.



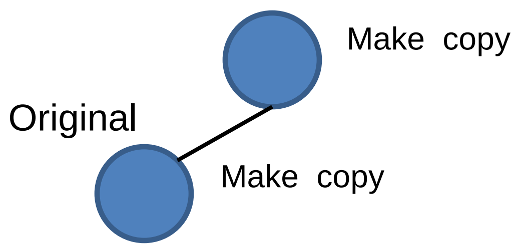
Search

Possible optimizations:

- Delete all failed nodes and branches.

  Reduction in memory but not in time

- Copy one space per node, not two.



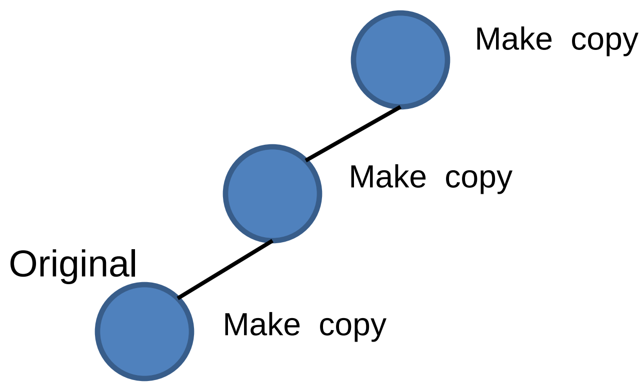
Search

Possible optimizations:

- Delete all failed nodes and branches.

  Reduction in memory but not in time

- Copy one space per node, not two.



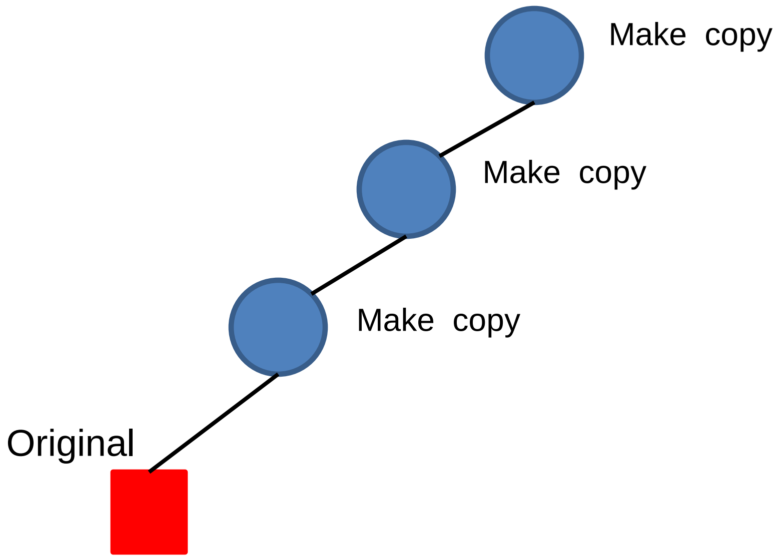
Search

Possible optimizations:

- Delete all failed nodes and branches.

  Reduction in memory but not in time

- Copy one space per node, not two.



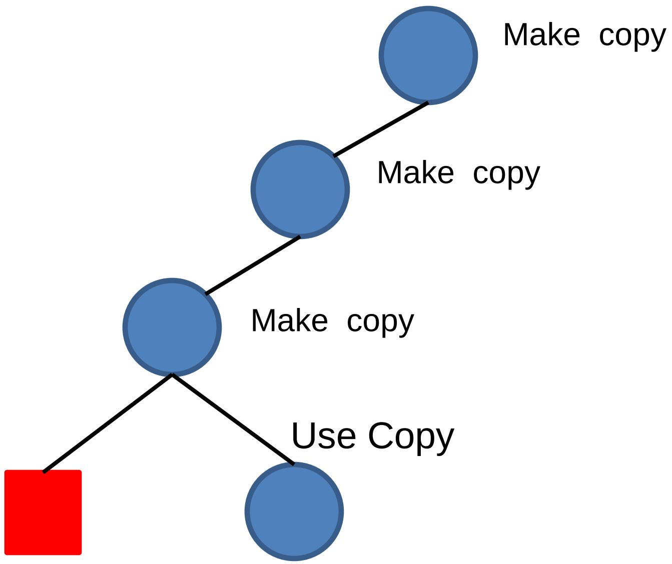
Search

Possible optimizations:

- Delete all failed nodes and branches.

  Reduction in memory but not in time

- Copy one space per node, not two.



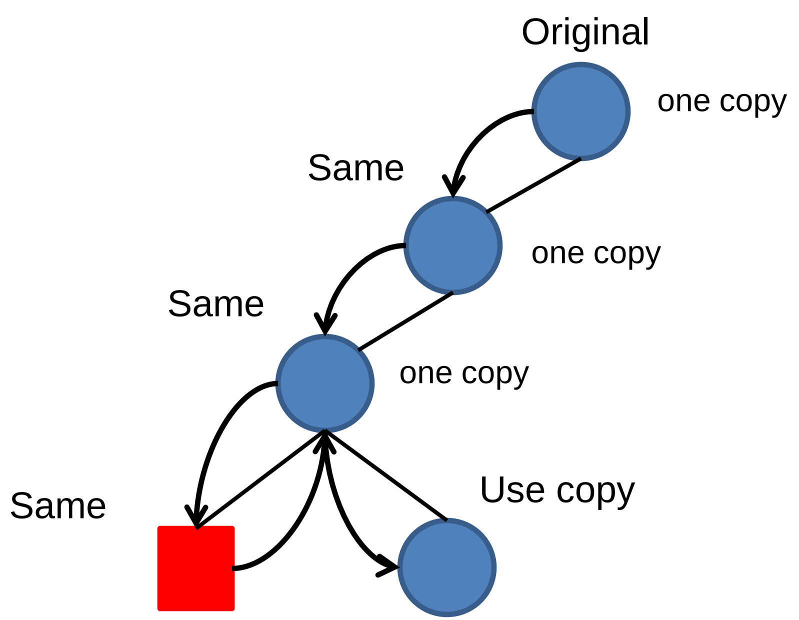
Search

Possible optimizations:

- Delete all failed nodes and branches.

  Reduction in memory but not in time

- Copy one space per node, not two.



Kind of clever, it cuts copy time in half !!!!

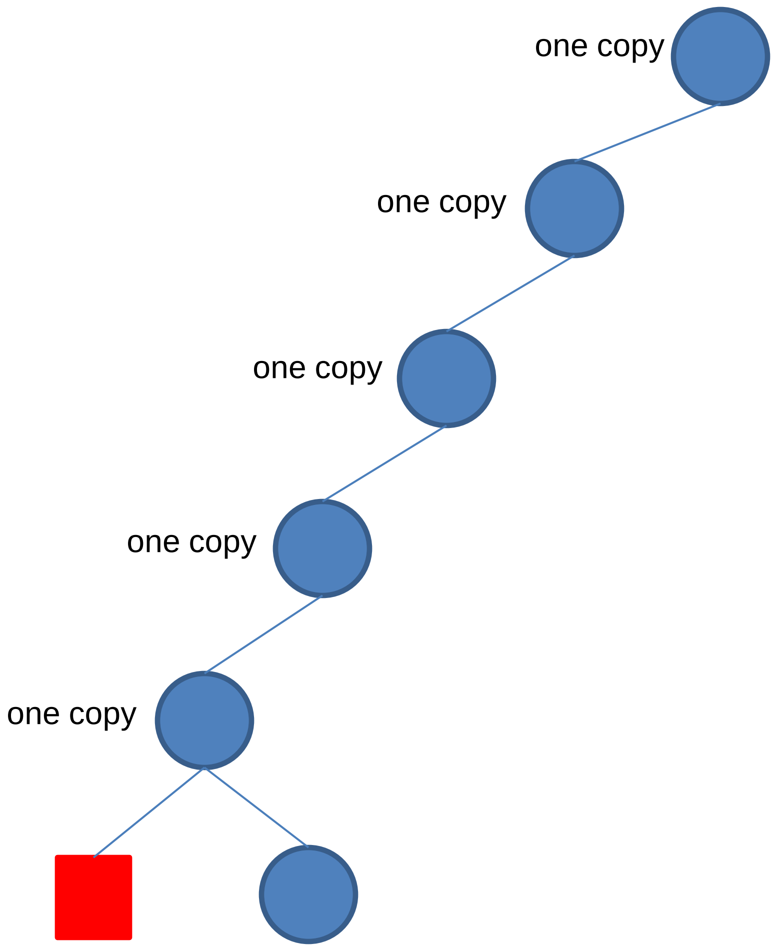
Search algorithm modified

function SEARCH(*csp*) return solution or failure
  csp ← PROPAGATE(*csp*)
  if *csp* is *failure* **then return** *failure*
  if *csp* is solved **then return sol**(*csp*)
  csp' ← *csp*
  ADD**(***csp*, BRANCH(*csp*,*1*)**)** **//add branching constraint to csp instead of csp'**
  //csp'' ← csp
  ADD**(***csp'*, BRANCH(*csp'*,*2*)**)** **//add second branch constraint to csp'**
  solution ← SEARCH(*csp*) **//call SEARCH with csp instead of csp'**
  if *solution* is valid **then return** *solution*
  else return SEARCH(*csp')* **//call SEARCH with csp instead of csp'**

function BRANCH(*csp*, *nbranch*)
  return the constraint representing the nbranch according to variable
    and value heuristic selection.

Search

- So far so good

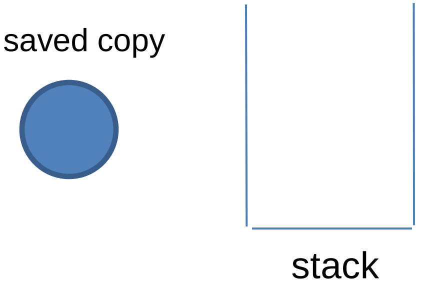


Five spaces created, can we do better??

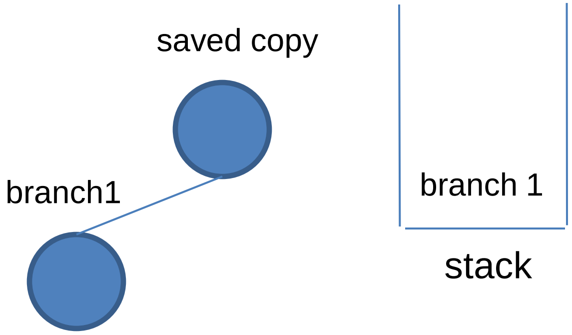
Recomputation

- Save spaces time to time (parameter)
- Use a stack to store the branching constraints
- Recreate spaces from the latest one saved

Recomputation

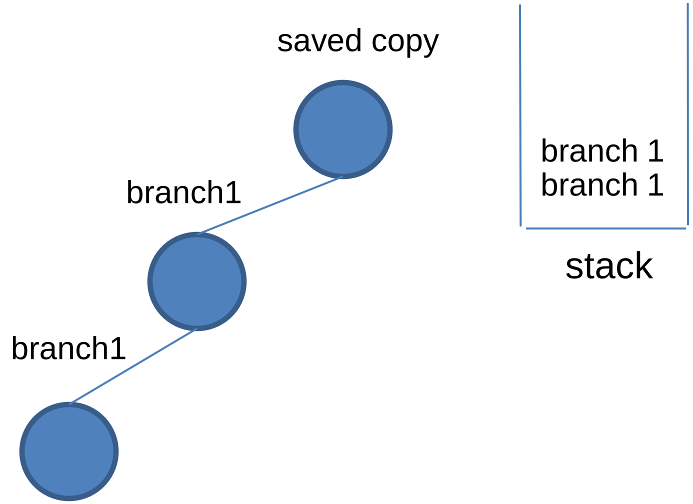


Recomputation



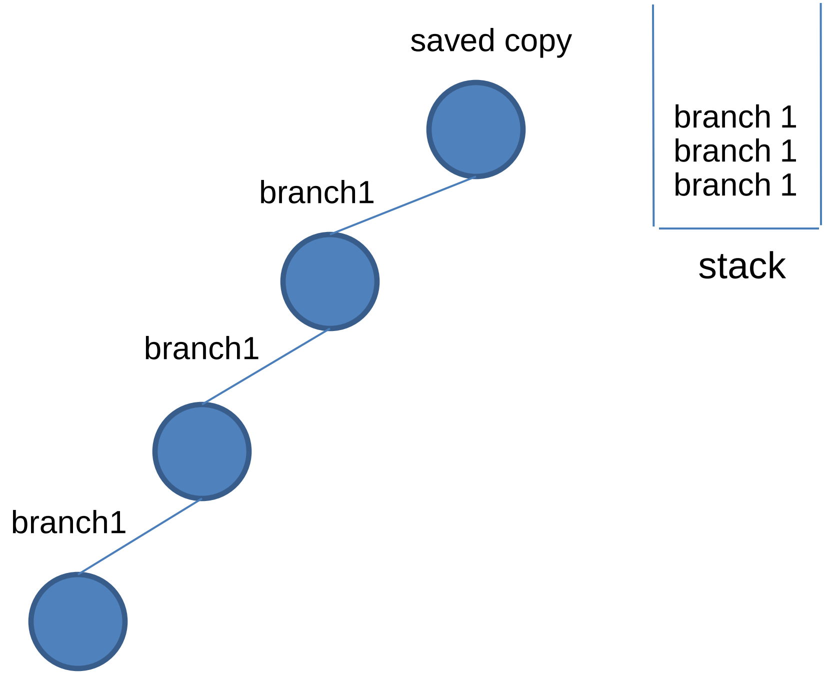
Push alternative

Recomputation



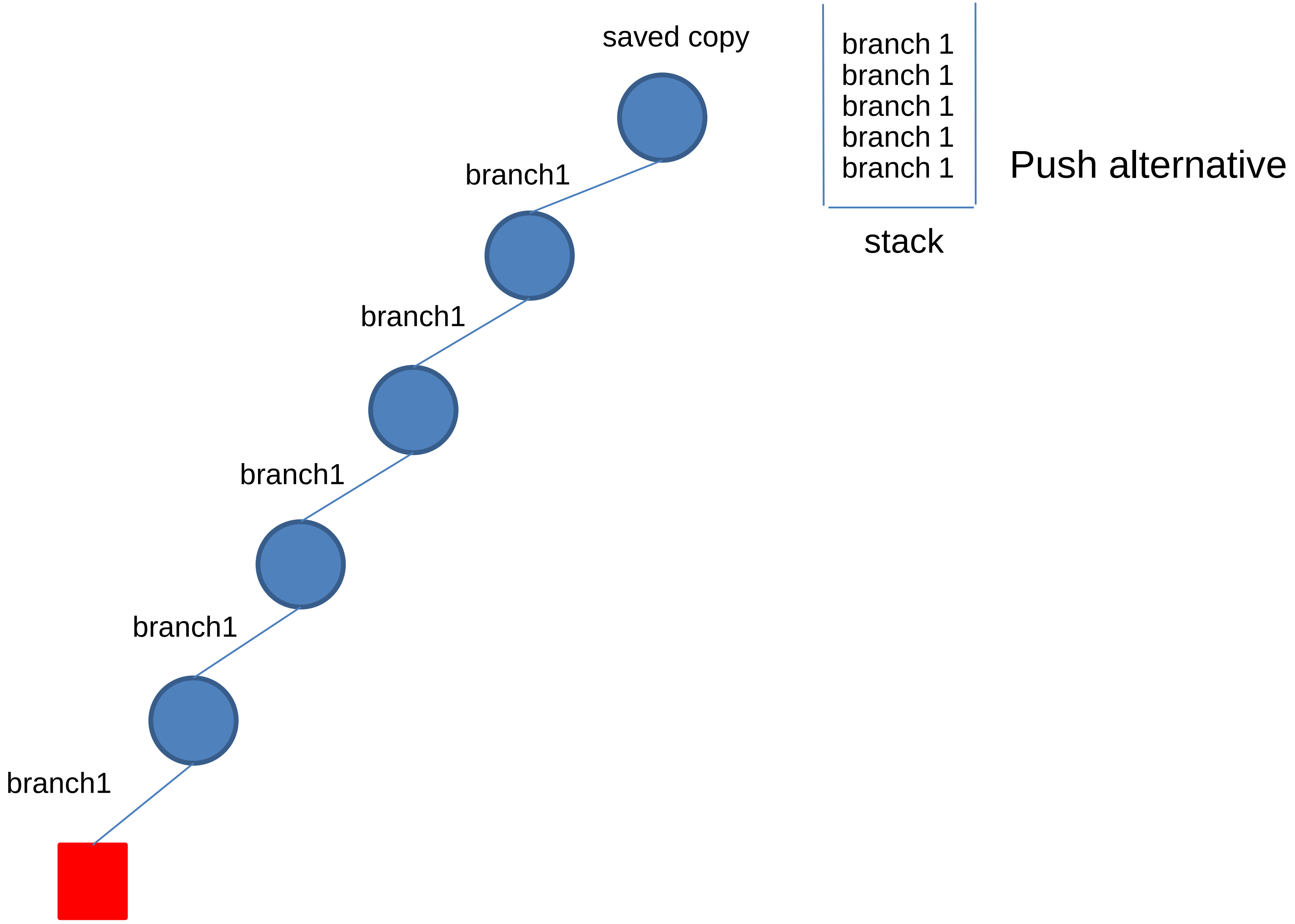
Push alternative

Recomputation



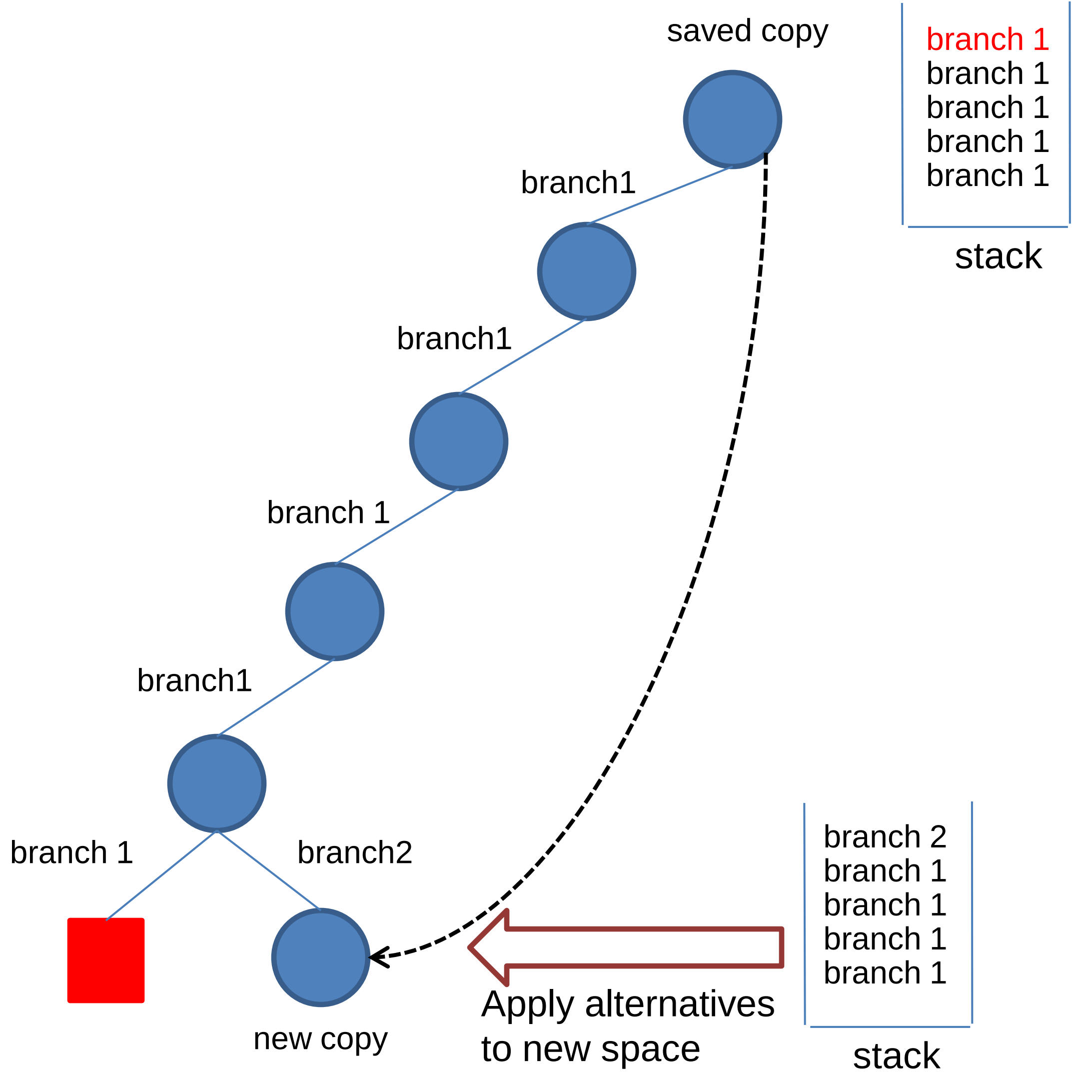
Push alternative

Recomputation



Recomputation

Pop latest alternative:
Failed node



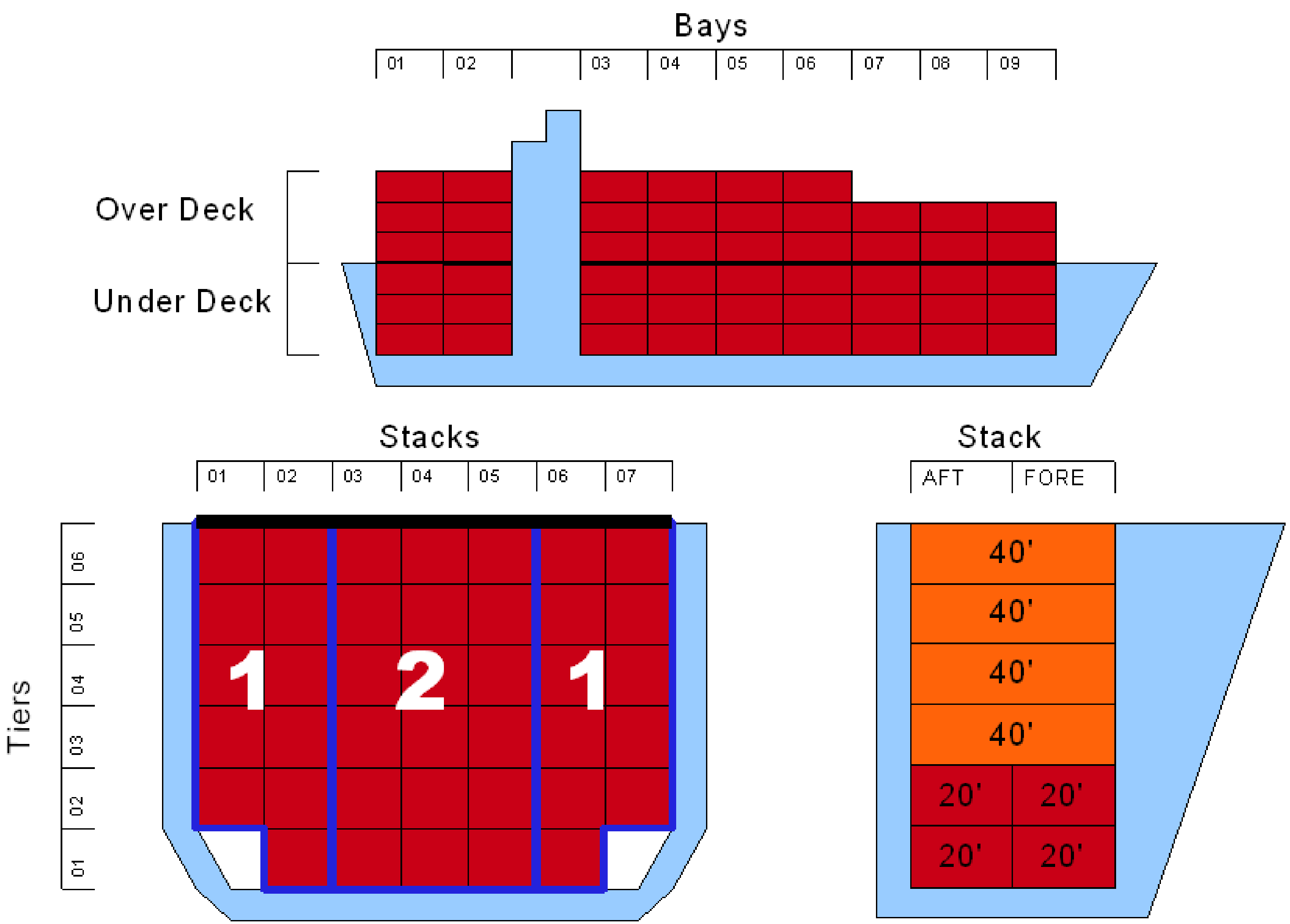
Research Impact: Container Stowage

Example 2: Container stowage problem

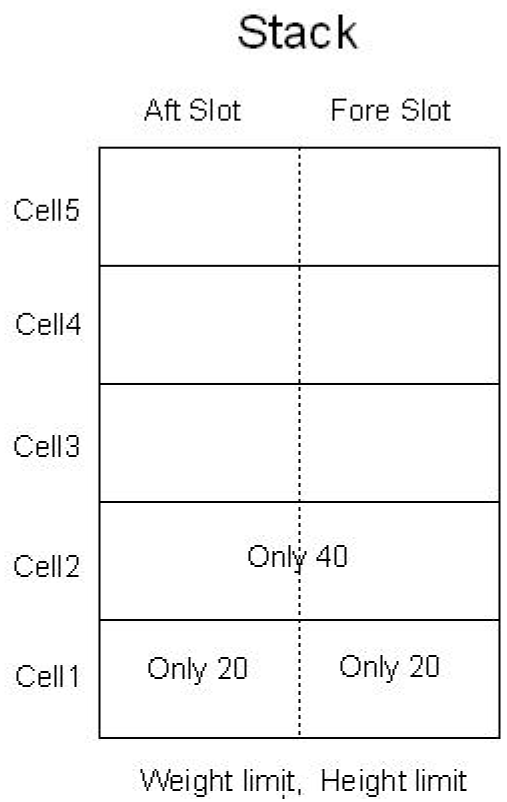
- Definition of the problem
- Defining variables
- Defining constraints
- Variable and value heuristic selection

Overall Problem

Layout of the vessel



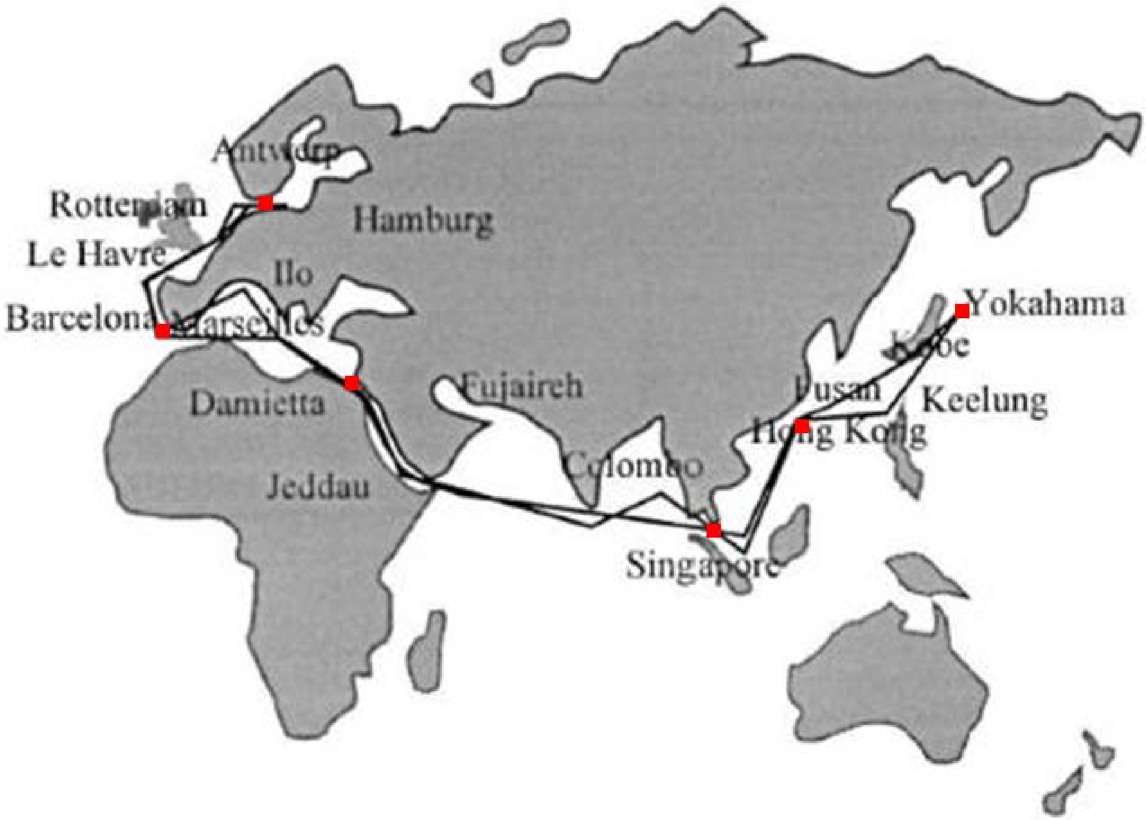
Stack Info



Container Info

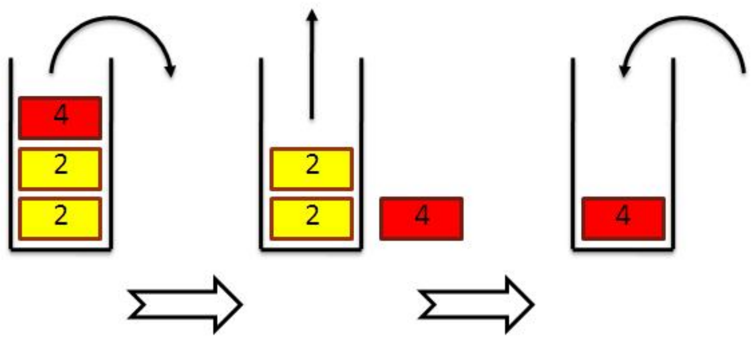
- Length (20/40-foot)
- Height (Normal/High-cube)
- Weight (Kg)
- DIPO(discharge port)

What is overstowage?



Several ports in the route of a ship

What is overstowage?



Overstowage = Extra time in port and cranes use

Problem definition

- Constraints
  - Weight and height capacity for each stack
  - Containers must form stacks
  - No 20-foot on top of 40-foot containers.
  - Cells capacity
  - Load all containers
- Objectives
  - Minimize overstowage

Model of the problem

- Defining our variables:
  - ***Variables***: $S = \{ s_1, \ldots, s_{\#Slots}\}$
  - ***Domains***: *{0, .., #Conts}*, for each variable
- Constraints:
  - **Weight and Height capacity**: Not evident how to model these constraints with the variables and the info we have, at least not with constraints provided by standard solvers.

    You can implement your own constraints!!!

Model of the problem

- Solution: Define a new set of auxiliary variables
  - ***Variables***: $H = \{ h_1, \ldots, h_{\#Slots} \}$
  - ***Domains***: {{*0*} U All possible heights of containers}, for each variable
  - ***Constraint:***

$$\sum_{i \in \mathrm{Stack}_j} h_i \leq \mathrm{stackLimit}_j \quad , \quad \text{for all } \ j \in \mathrm{Stacks}$$

And we add a constraints to connect both set of variables:

$$\mathrm{heightCont}[s_i] = h_i \ , \quad \text{for all } \ i \in \mathrm{Slots}$$

- And it goes the same for the weight constraint

Model of the problem

– ***Containers must form a stack and 20 foot cont cannot be on top of 40 foot cont.***
  - New set of auxiliary variables *L* for length of cont in each slot
  - Define regular expression for valid patterns: *R = 20*40*0**
  - Constrain each stack to follow the valid patterns

$$\text{regular}(L_i, R), \quad i \in \text{Stacks}$$

  - Where L_i represents all variables from *L* bound to slots in stack *i*

Model of the problem

– ***Overstowage:***

- New set of variables *P* for the ports
- Set of boolean variables *Ov* to indicate overstowage

$$\left(\sum_{k \in under(j)} (p_k < p_j)\right) > 0 \Leftrightarrow Ov_{ij} = 1, \quad \text{for } j \in Stack_i, \text{ for } i \in Stacks$$

$$TotalOv = \sum_{i \in Stacks} \sum_{j \in Stack_i} Ov_{ij}$$

Branching

- Example of bay :

20 foot containers	10
40 foot containers	81
Discharge ports	3
Slots available	180
Stacks	10

Branching

- First try:
  - Branching over slots, selecting variables bottom-up

    Still waiting!!!
- Second try:
  - Branching over height variables
  - Branching over slot variables

    Around 5 minutes, very bad with overstowage

Branching

- Third try:
  - Specific branching over port variables, focus on avoid overstowage
  - Branching over height variables
  - Branching over slot variables

    Around 1 second, first solution is the best solution