Restrições - Métodos de Resolução

- Resolução Simbólica /Algébrica
  - Domínios Herbrand – Unificação
  - Boleanos – Unificação Boleana
  - Programação Linear Inteira (Simplex)
- Retrocesso
  - Percorre todo o espaço de pesquisa
    - Complexidade k^n
  - Redução do espaço de pesquisa
    - Propagação de restrições
    - Cortes - Programação inteira
    - Complexidade k_r^n $(k_r < k)$
- Pesquisa Local
  - Reparação de “soluções”
  - Optimização
  - Método incompleto
  - Metaheurísticas – óptimos locais
- Métodos Mistos

Heuristic Search

- Algorithms that maintain some form of consistency, remove (many?) redundant values but, not being complete, do not eliminate the need for search.
- Even when a constraint network is consistent, enumeration is subject to failure.
- In fact, a consistent constraint network may not even be satisfiable.
- All that is guaranteed by maintaining some type of consistency is that the networks are equivalent.
- Solutions are not “lost” in the reduced network, that despite having less redundant values, has all the solutions of the former.

Heuristic Search

- Hence, the domain pruning does not eliminate in general the need for search. The search space is usually organised as a tree, and the search becomes some form of tree search.

- As usual, the various branches down from one node of the search tree correspond to the assignment of the different values in the domain of a variable.

- As such, a tree leaf corresponds to a complete compound label (including all the problem variables).

- A depth first search in the tree, resorting to backtracking when a node corresponds to a dead end (unsatisfiability), corresponds to an incremental completion of partial solutions until a complete one is found.

Heuristic Search

- Given the execution model of constraint logic programming (or any algorithm that interleaves search with constraint propagation)

```
Problem(Vars):-
        Declaration of Variables and Domains,
        Specification of Constraints,
        Labelling of the Variables.
```

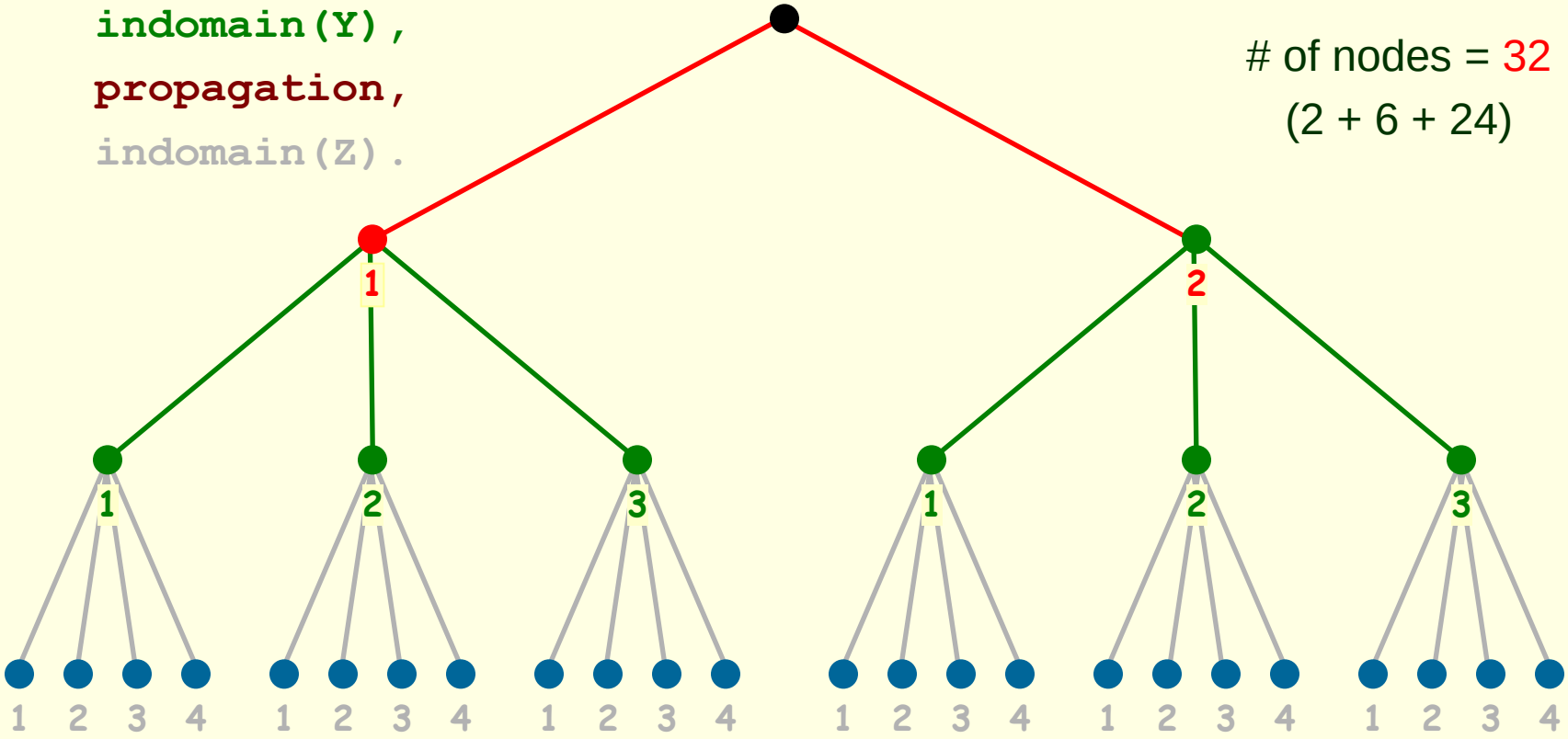
  the enumeration of the variables determines the shape of the search tree, since the nodes that are reached depend on the order in which variables are enumerated.

- Take for example two distinct enumerations of variables whose domains have different cardinality, e.g. **`X in 1..2, Y in 1..3`** and **`Z in 1..4.`**

Heuristic Search

```
enum([X,Y,Z]):-
    indomain(X)
    propagation
    indomain(Y),
    propagation,
    indomain(Z).
```

(X in 1..2, Y in 1..3, Z in 1..4)



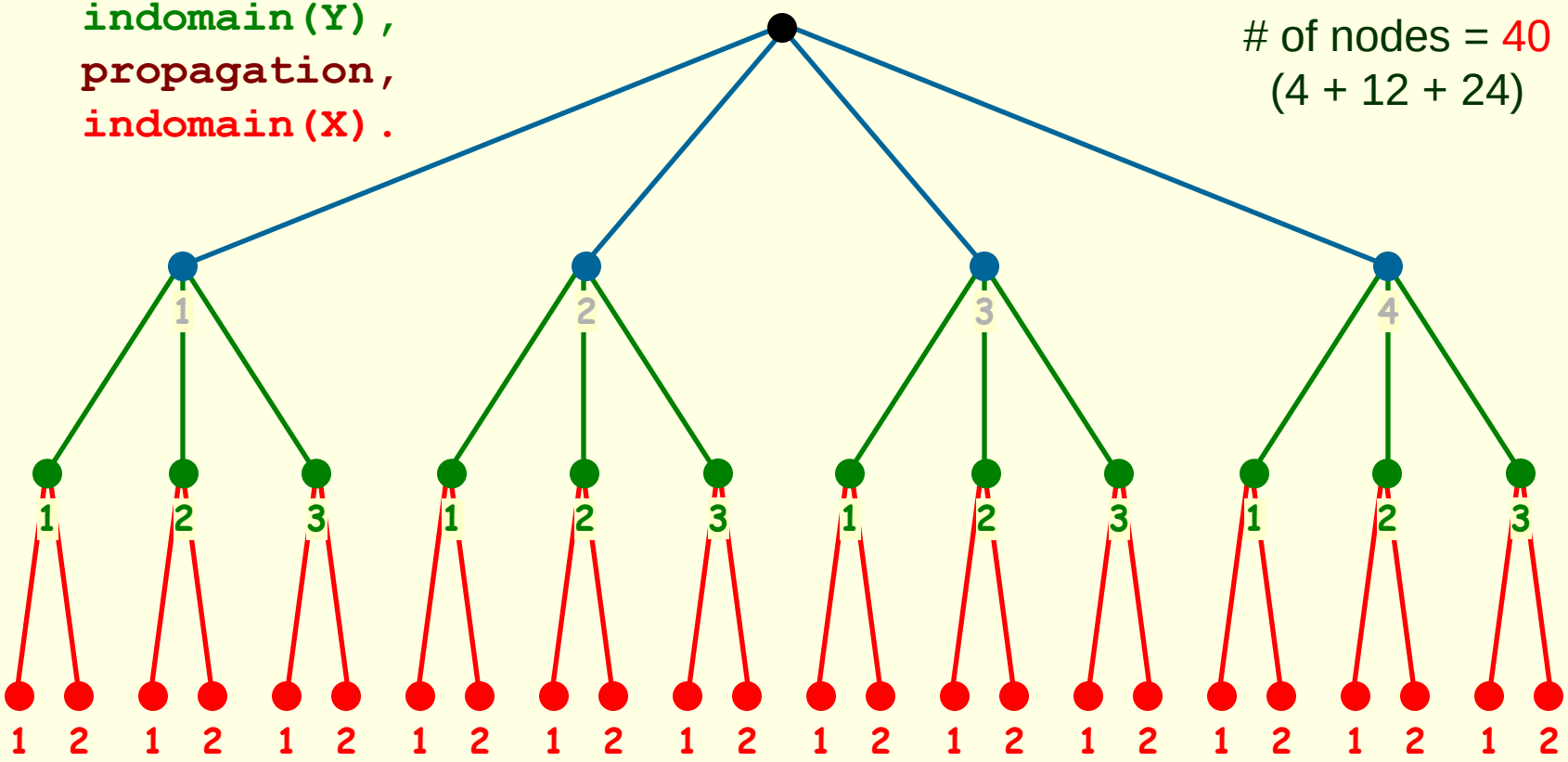
# of nodes = 32
(2 + 6 + 24)

Heuristic Search

```
enum([X,Y,Z]):-
    indomain(Z),
    propagation
    indomain(Y),
    propagation,
    indomain(X).
```

(X in 1..2, Y in 1..3, Z in 1..4)

# of nodes = 40
(4 + 12 + 24)



Heuristic Search

- The order in which variables are enumerated may have an **important** impact on the efficiency of the tree search, since
  - The number of internal nodes is different, despite the same number of leaves, or potential solutions, Π **#D$_i$**.
  - Failures can be detected differently, favouring some orderings of the enumeration.
  - Depending on the propagation used, different orderings may lead to different prunings of the tree.
- The **ordering of the domains** has no direct influence on the search space, although it may have great importance in finding the first solution.

Heuristic Search

- To control the efficiency of tree search one should in principle adopt appropriate heuristics to select
    - The next **variable** to label
    - The **value** to assign to the selected variable

- Since heuristics for **value choice** will not affect the size of the search tree to be explored, particular attention will be paid to the heuristics for **variable selection**.

Variable Selection Heuristics

- There are two types of heuristics that can be considered for variable selection.
  - **Static** - the ordering of the variables is set up before starting the enumeration, not taking into account the possible effects of propagation.
  - **Dynamic** - the selection of the variable is determined after analysis of the problem that resulted from previous enumerations (and propagation).

Static Heuristics

- Static heuristics are based on some properties of the underlying constraint graphs.

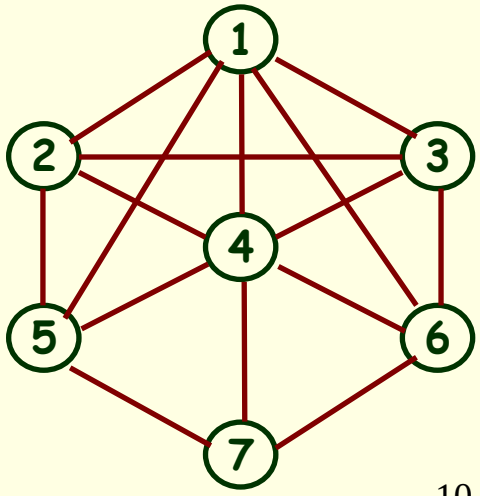
- A typical example is the Maximum Degree Ordering heuristics:

MDO Heuristics (*Maximum Degree Ordering*):

With a *Maximum Degree Ordering* heuristics, the variables of a constraint problem are enumerated by decreasing order of their degree in the constraint graph.

- Example:

Heuristics MDO would use an ordering starting in nodes 4 (d=6) and 1 (d=5) and ending in node 7 (d=3). Nodes 2, 3, 5 and 6 would be sorted arbitrarily.



Dynamic Heuristics

- In contrast to static heuristics, variable selection may be determined dynamically. Instead of being fixed before enumeration starts, the variable is selected taking into account the propagation of previous variable selections (and labellings).

- In addition to problem specific heuristics, there is a general principle that has shown great potential, the *first-fail* principle.

- The principle is simple: when a problem includes many interdependent “tasks”, start solving those that are most difficult. It is not worth wasting time with the easiest ones, since they may turn to be incompatible with the results of the difficult ones.

First-Fail Heuristics

- There are many ways of interpreting and implementing this generic *first-fail* principle.
- Firstly, the tasks to perform to solve a constraint satisfaction problem may be considered the assignment of values to the problem variables. How to measure their difficulty?
- Enumerating by itself is easy (a simple assignment). What turns the tasks difficult is to assess whether the choice is viable, after constraint propagation. This assessment is hard to make in general, so we may consider features that are easy to measure, such as
  - The domain of the variables
  - The number of constraints (degree) they participate in.

First-Fail Heuristics

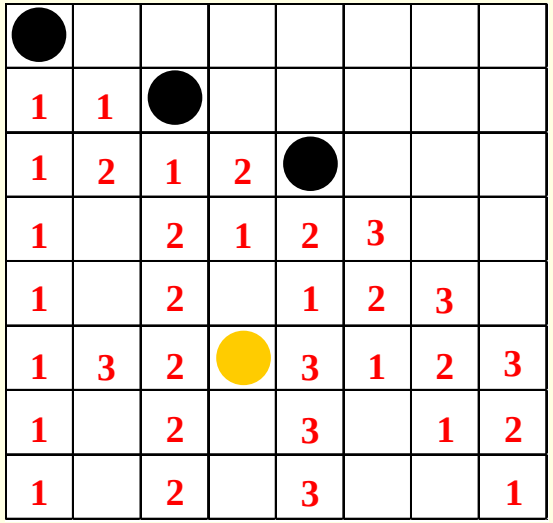
The domain of the variables

- Intuitively, if variables X_1 / X_2 have m_1 / m_2 values in their domains, and $m_2 > m_1$, it is preferable to assign values to X_1, because there is less choice available !
- In the limit, if variable X_1 has only one value in its domain, ($m_1 = 1$), there is no possible choice and the best thing to do is to immediately assign the value to the variable.
- Another way of seeing the issue is the following:
  - On the one hand, the “chance” to assign a good value to X_1 is higher than that for X_2.
  - **On the other hand, if that value proves to be a bad one, a larger proportion of the search space is eliminated.**

First-Fail Heuristics: Example

Example:

In the 8 queens problem, where queens Q1, Q2 e Q3, were already enumerated, we have the following domains for the other queens



- Q4 in {2,7,8}, Q5 in {2,4,8}, Q6 in {4}, Q7 in {2,4,8}, Q8 in {2,4,6,7}.
- Hence, the best variable to enumerate next should be Q6, not Q4 that would follow in the “natural” order.
- In this extreme case of singleton domains, node-consistency achieves pruning similar to arc-consistency with less computational costs!

First-Fail Heuristics

The number of constraints (degree) of the variables

- This heuristics is basically the Maximum Degree Ordering (MDO) heuristics, but now the degree of the variables is assessed dynamically, after each variable enumeration.

- Clearly, the more constraints a variable is involved in, the more difficult it is to assign a good value to it, since it has to satisfy a larger number of constraints.

- Of course, and like in the case of the domain size, this decision is purely heuristic. The effect of the constraints depends greatly on their propagation, which depends in turn on the problem in hand, which is hard to antecipate.

Problem Dependent Heuristics

- In certain types of problems, there might be heuristics specially adapted for the problems being solved.
- For example, in scheduling problems, where ‘tasks’ should not overlap but have to take place in a certain period of time, it is usually a good heuristic to “scatter” them as much as possible within the allowed period.
- This suggests that one should start by enumerating first the variables corresponding to tasks that may be performed in the beginning and in the end of the allowed period, thus allowing “space” for the others to execute.
- In such case, the dynamic choice of the variable would take into account the values in its domain, namely the minimum and maximum values.

Value Choice Heuristics

- Once a variable to label is selected, a value within its domain has to be chosen.
- There are not many generic methods to handle value choice. The only one widely used is the principle of choosing the value with higher “likelihood” of success!
- The reason for this is obvious. In contrast with variable selection, value choice will not determine the size of the search space, so one should be interested in finding as quickly as possible the path to a solution.
- Of course, the application of this principle is highly dependent on the problem (or even the instance of the problem) being solved.

Value Choice Heuristics

- Some forms of assigning likelihood are the following:

Ad hoc choice:

Again in scheduling problems, once the variable with lowest/highest value in its domain is selected, the natural choice for the value will be the lowest/highest, which somehow “optimises” the likelihood of success.

Lookahed:

One may try to anticipate, for each of the possible values, the likelihood of success by evaluating (after its propagation) the effect on an aggregated indicator on the size of the domains not yet assigned, choosing the one that maximises such indicator.

Value Choice Heuristics

Optimisation:

In optimisation problems, where there is some function to maximise/minimise, one may get *bounds* for that function when the alternative values are chosen for the variable, or check how they change with the selected value.

- Of course, the heuristic will choose the value that either optimises the bounds in consideration, or that improves them the most.

- Notice that in this case, the computation of the bounds may be performed either before propagation takes place (less computation, but also less information) or after such propagation.

Heuristics in SICStus

- A program based on the Constraint Logic Programming paradigm has the structure already described:

```
Problem(Variables):-
    Declaration of Variables and Domains,
    Specification of Constraints,
    Labelling of the Variables.
```

- In the labelling of the variables X_1, X_2, ..., X_n, of some list Lx, one should specify the intended heuristics.

- Although these heuristics may be programmed explicitly, there are some facilities that SICStus provides, both for variable selection and value choice.

Heuristics in SICStus

- The simplest form to specify enumeration is through a buit-in predicate, labeling/2, where
  - the 1st argument is a list of options, possibly empty
  - the 2nd argument is a list Lx = [X_1, X_2, ..., X_n] of variables to enumerate

- By default, labeling([ ], Lx) selects variables X_1, X_2, ..., X_n, from list Lx, according to their position, “from left to right”. The value chosen for the variable is the least value in the domain.

- This predicate can be used with no options for static heuristics, provided that the variables are sorted in the list Lx according to the intended ordering.

Heuristics in SICStus

- With an empty list of options, predicate **labeling([ ],L)** is in fact equivalent to predicate enumerating(L) below

```
enumerating([]).
enumerating([Xi|T]):-
    indomain(Xi),
    enumerating(T).
```

where the built-in predicate, **indomain(Xi)**, chooses values for variable Xi in increasing order.

- There are other possibilities for user control of value choice. The current domain of a variable, may be obtained with built-in *clpfd* predicate **fd_dom/2**. For example

```
?- X in 1..5, X #\=3, fd_dom(X,D).
     D = (1..2)\/(4..5),
     X in(1..2)\/(4..5) ?
```

Heuristics in SICStus

- Usually, it is not necessary to reach this low level of programming, and a number of predefined options for predicate **labeling/2** can be used.

- The options of interest for value choice for the selected variable are **up** and **down**, with the obvious meaning of chosing the values from the domain in increasing and decreasing order, respectively.

- Hence, to guarantee that the value of some variable is chosen in decreasing order without resorting to lower-level *clpfd* predicates, it is sufficient to call predicate **labeling/2** with option **down**

```
labeling([down],[Xi])
```

Heuristics in SICStus

The options of interest for variable selection are **leftmost**, **min**, **max**, **ff**, **ffc** and **variable(Sel)**

- **leftmost** - is the default mode.
  - Variables are simply selected by their order in the list.
- **min**, **max** - the variable with the lowest/highest value in its domain is selected.
  - Useful, for example, in many applications of scheduling, as discussed.
- **ff**, **ffc** - implements the *first-fail* heuristics, selecting the variable with a domain of smallest size, breaking ties with the number of constraints, in which the variable is involved.

Heuristics in SICStus

– **variable(Sel)**

- This is the most general possibility. **Sel** must be defined in the program as a predicate, whose last 3 parameters are **Vars, Selected, Rest**. Given the list of **Vars** to enumerate, the predicate should return **Selected** as the variable to select, **Rest** being the list with the remaining variables.

- Other parameters may be used before the last 3. For example, if option **variable(includes(5))** is used, then some predicate *includes/4* must be specified, such as

  includes(V, Vars, Selected, Rest)

  which should choose, from the **Vars** list, a variable, **Selected**, that includes **V** in its domain.

Heuristics in SICStus

- Notice that all these options of predicate **labeling/2** may be programmed at a lower level, using the adequate primitives available from SICStus for inspection of the domains. These Reflexive Predicates, named fd_predicates include

```
fd_min(?X, ?Min)
fd_max(?X, ?Max)
fd_size(?X, ?Size)
fd_degree(?X, ?Degree)
```

with the obvious meaning. For example,

```
?- X in 3..8, Y in 1..5, X #< Y,
   fd_size(X,S), fd_max(X,M), fd_degree(Y,D).
D = 1,  M = 4, S = 2,
X in 3..4, Y in 4..5 ?
```

Search

- Constraint Logic Programming uses, by default, depth first search with backtracking in the labelling phase.
- Despite being “interleaved” with constraint propagation, and the use of heuristics, the efficiency of search depends critically of the first choices done, namely the values assigned to the first variables selected.
- Backtracking “chronologically”, these values may only change when the values of the remaining k variables are fully considered (after some $O(2^k)$ time in the worst case). Hence, alternatives have been proposed to pure depth first search with chronological backtracking, namely
    - Intelligent backtracking,
    - Iterative broadening,
    - Limited discrepancy; and
    - Incremental time-bounded search.

Search

- In chronological backtracking, when the enumeration of a variable fails, backtracking is performed on the variable that immediately preceded it, *even if this variable is not to blame for the failure*.

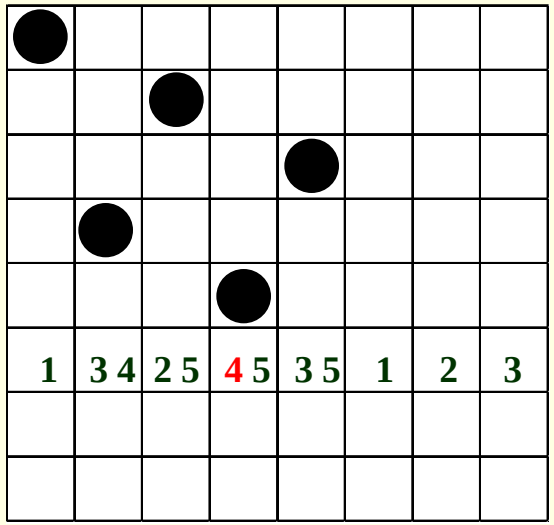
- Various techniques for intelligent backtracking, or dependency directed search, aim at identifying the causes of the failure and backtrack directly to the first variable that participates in the failure.

- Some variants of intelligent backtracking are:
    - Backjumping ;
    - Backchecking ; and
    - Backmarking .

Intelligent Backtracking Example

Backjumping

- Failing the labeling of a variable, all variables that cause the failure of each of the values are analysed, and the “highest” of the “least” variables is backtracked



In the example, variable Q_6, could not be labeled, and backtracking is performed on Q_4, the “highest of the least“ variables involved in the failure of Q_6.

All other positions of Q_6 are, in fact, incompatible with the value of some variable lower than Q_4.

Intelligent Backtracking

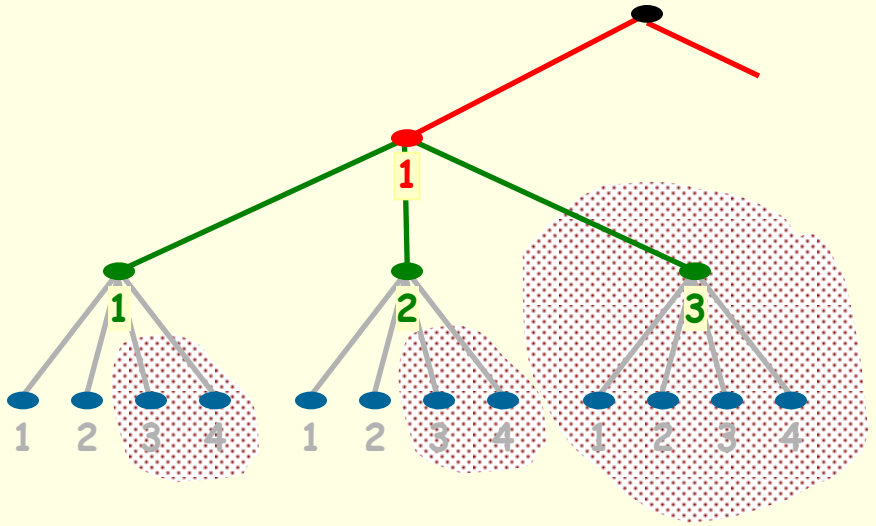
Backchecking and Backmarking

- These techniques may be useful when the testing of constraints on different variables is very costly. The key idea is to memorise previous conflicts, in order to avoid repeating them.
  - In backchecking, only the assignments that caused conflicts are memorised.
  - In backmarking, the assignments that did not cause conflicts are also memorised.
- The use of these techniques with constraint propagation is usually not very effective (with a possible exception of SAT solvers, with nogood clause learning), since propagation anticipates the conflicts, somehow avoiding irrelevant backtracking.

Iterative Broadening

- In iterative broadening it is assigned a limit ***b***, to the number of times that a node is visited (both the initial visit and those by backtracking), i.e. the number of values that may be chosen for a variable. If this value is exceeded, the node and its successors are not explored any further.

- In the example, assuming that b=2, the search space prunned is shadowed.

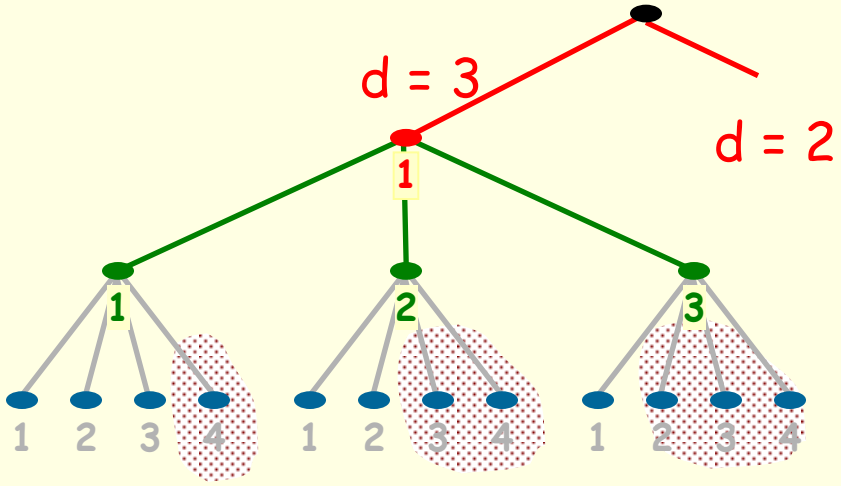


- Of course, if the search fails for a given *b*, this value is iteratively increased, hence the iterative broadening qualification.

Limited Discrepancy

- Limited discrepancy assumes that the value choice heuristic may only fail a (small) number of times. It directs the search for the regions where solutions more likely lie, by limiting to some number ***d*** the number of times that the suggestion made by the heuristic is not taken further.

- In the example, assuming heuristic options at the left and d=3 the search space prunned is shadowed.



- Again, if the search fails, *d* may be incremented and the search space is increasingly incremented.

Incremental Time Bounded Search

- In ITBS, the goal is similar to iterative broadening or limited discrepancy, but implemented differently. For each choice made (e.g. variable to label and/or value assigned), search is allowed for a given time ***T***.

- If no solution is found, another choice is tested. Of course, if the search fails for a certain value of *T*, this may be increased incrementally in the next iterations, guaranteeing that the search space is also increasing iteratively.

- In all these algorithms (iterative broadening, limited discrepancy and incremental duration) parts of the search space may be revisited. Nevertheless, the worst-case time complexity of the algorithms is not worsened.

Incremental Time Bounded Search

- For example, in the case of incremental time-bounded search if the successive and failed iterations increase the time limit by some factor $\alpha \geq 2$, i.e. $T_{j+1} = \alpha T_j$, the iterations will last

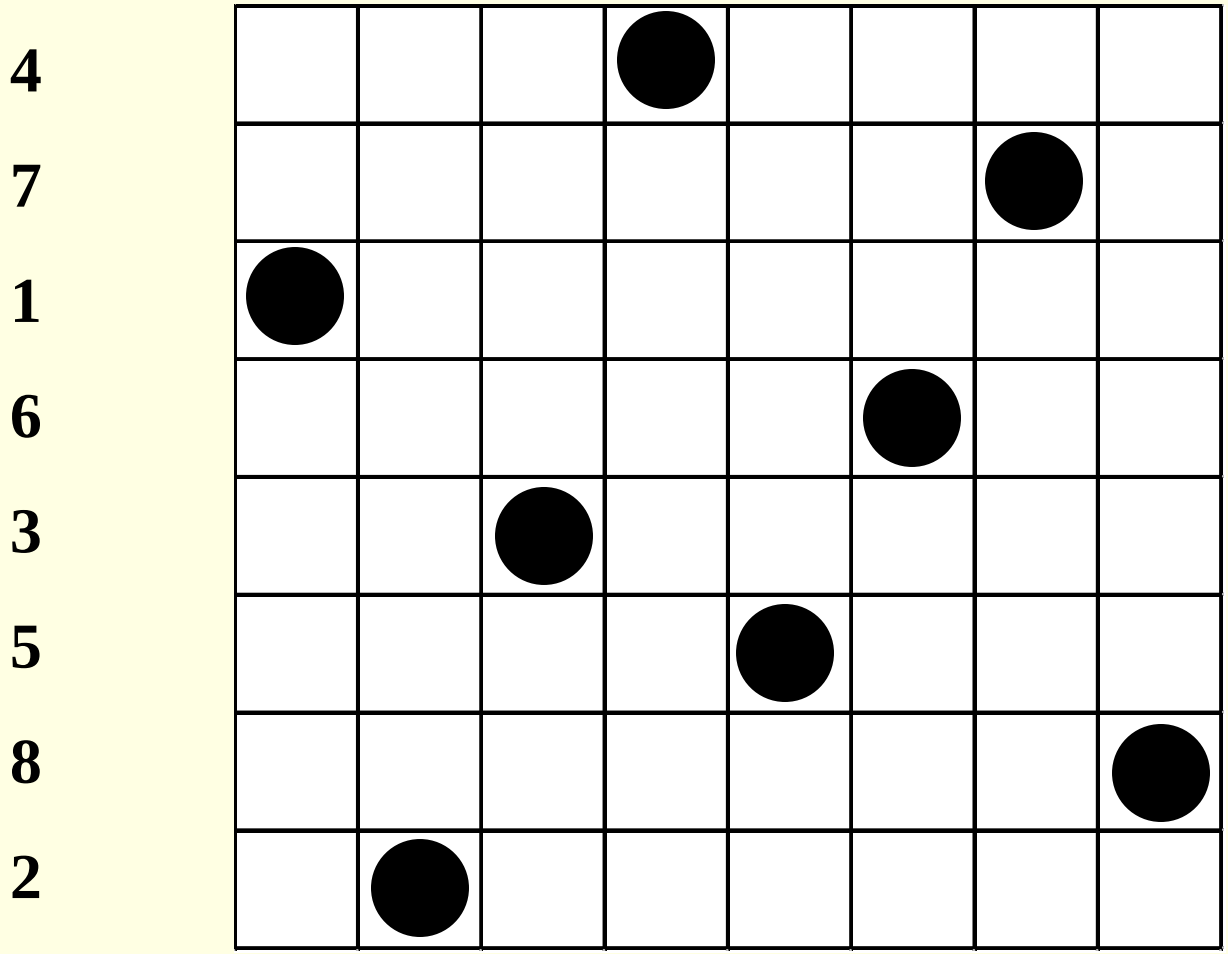
$$T_0 + T_1 + \ldots + T_j$$
$$= T (1 + \alpha + \alpha^2 + \ldots + \alpha^j) < T \alpha^{j+1}$$

- If a solution is found in iteration j, then
  - the time spent in the previous iterations is $T\alpha^{j-1}$.
  - iteration j may last for $T\alpha^j$. In average, the solution is found in half this time.
- Hence, the "wasted" time is of the same magnitude of the "useful" time spent in the search (in iteration j+1).

Pesquisa Local

- Reparação de “soluções”
  - A partir duma etiqueta (conjunto de atribuições de valores a variáveis) completa (ou mesmo parcial), que viola algumas restrições, tentar repará-la por sucessivas alterações de algumas variáveis.
  - Noção de “vizinhança”
  - Ao contrário do retrocesso, não há “variável mais recente”. Todas são, à partida, candidatas a serem alteradas no próximo passo.
  - Método, em geral, não é completo. Está confinado a uma “zona” do espaço de pesquisa.

Pesquisa Local - Reparação



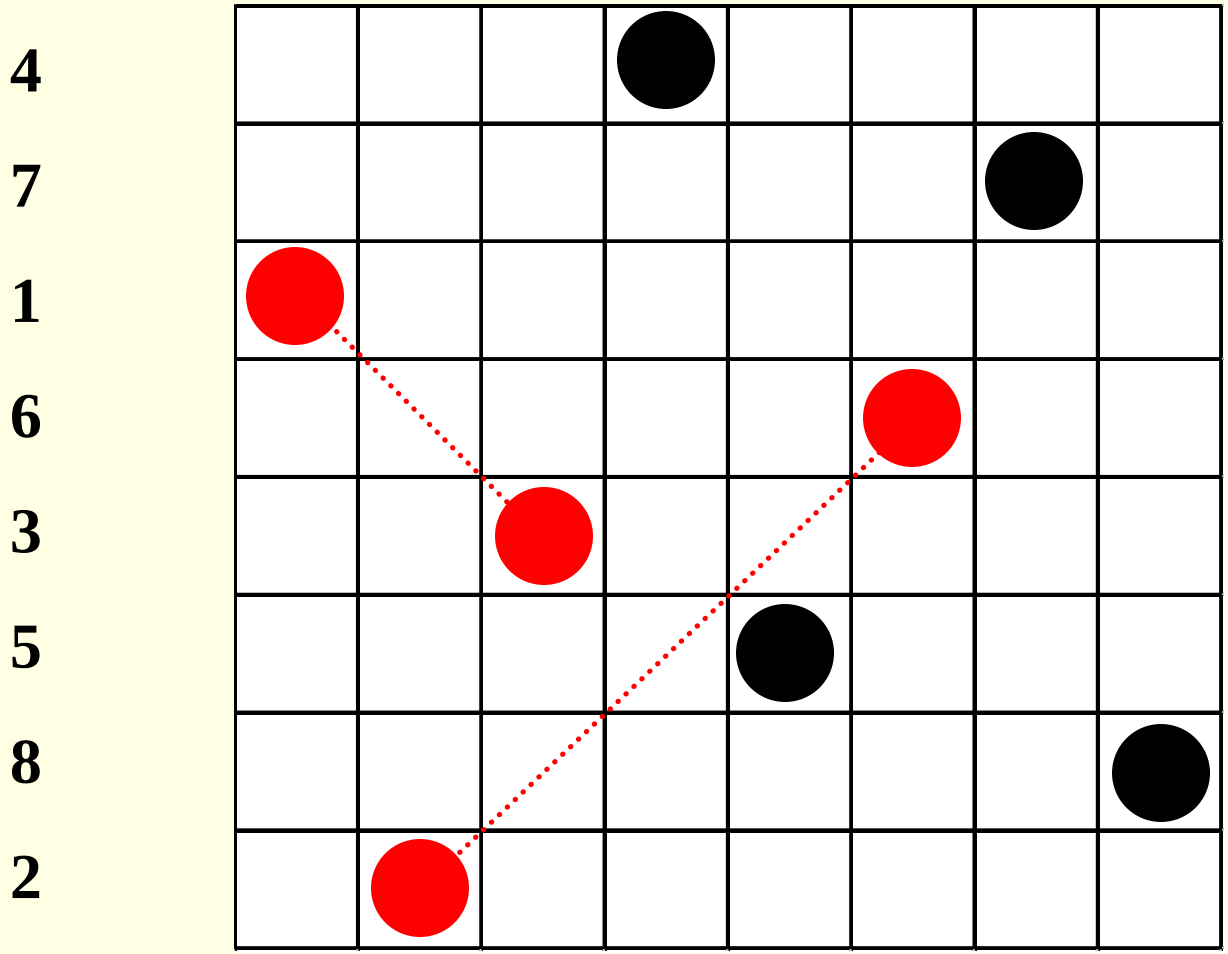
Solução

Permutação

Vizinhança

Troca

Pesquisa Local - Reparação

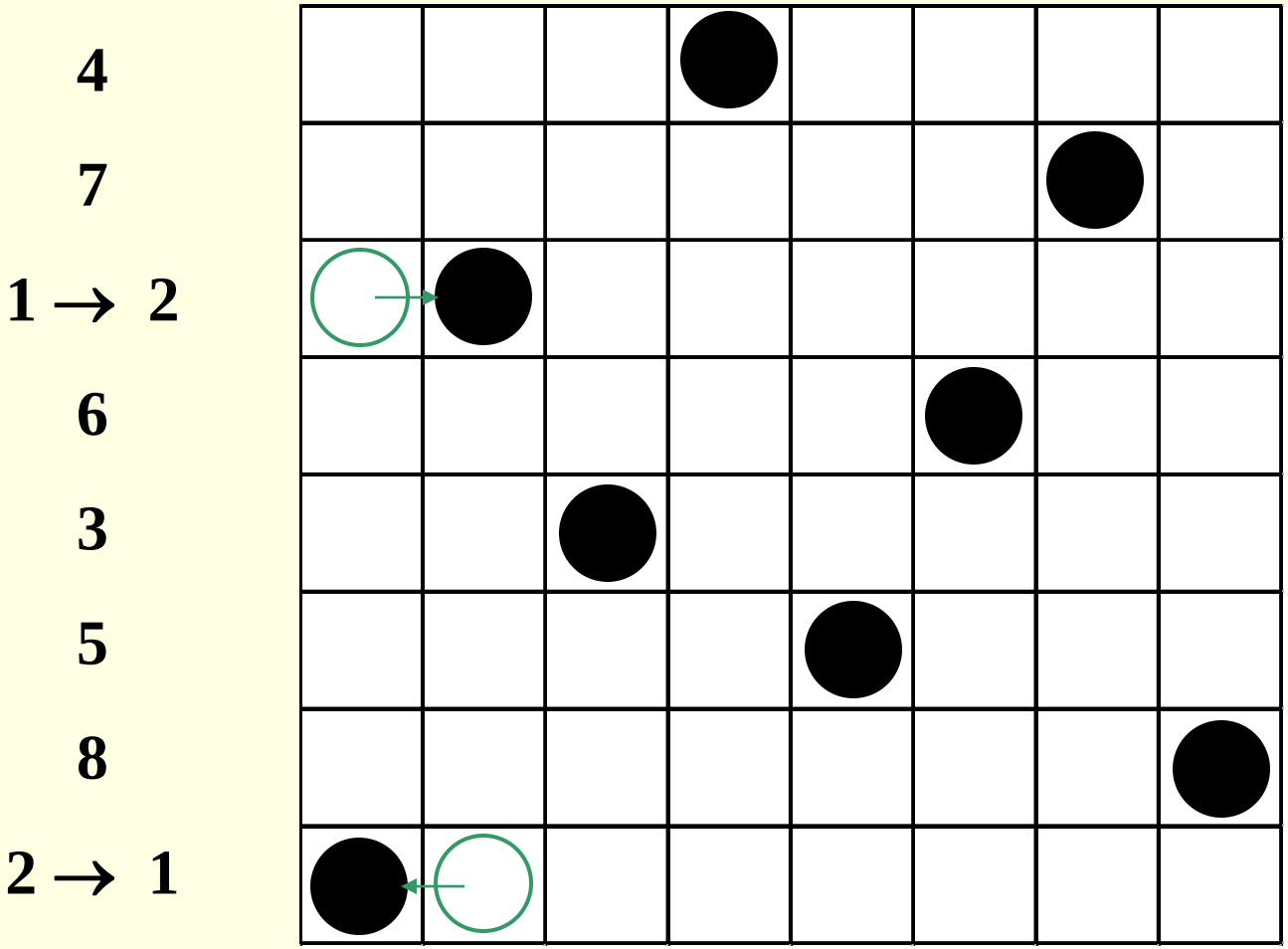


2 Ataques

3 - 5

4 - **8**

Pesquisa Local - Reparação

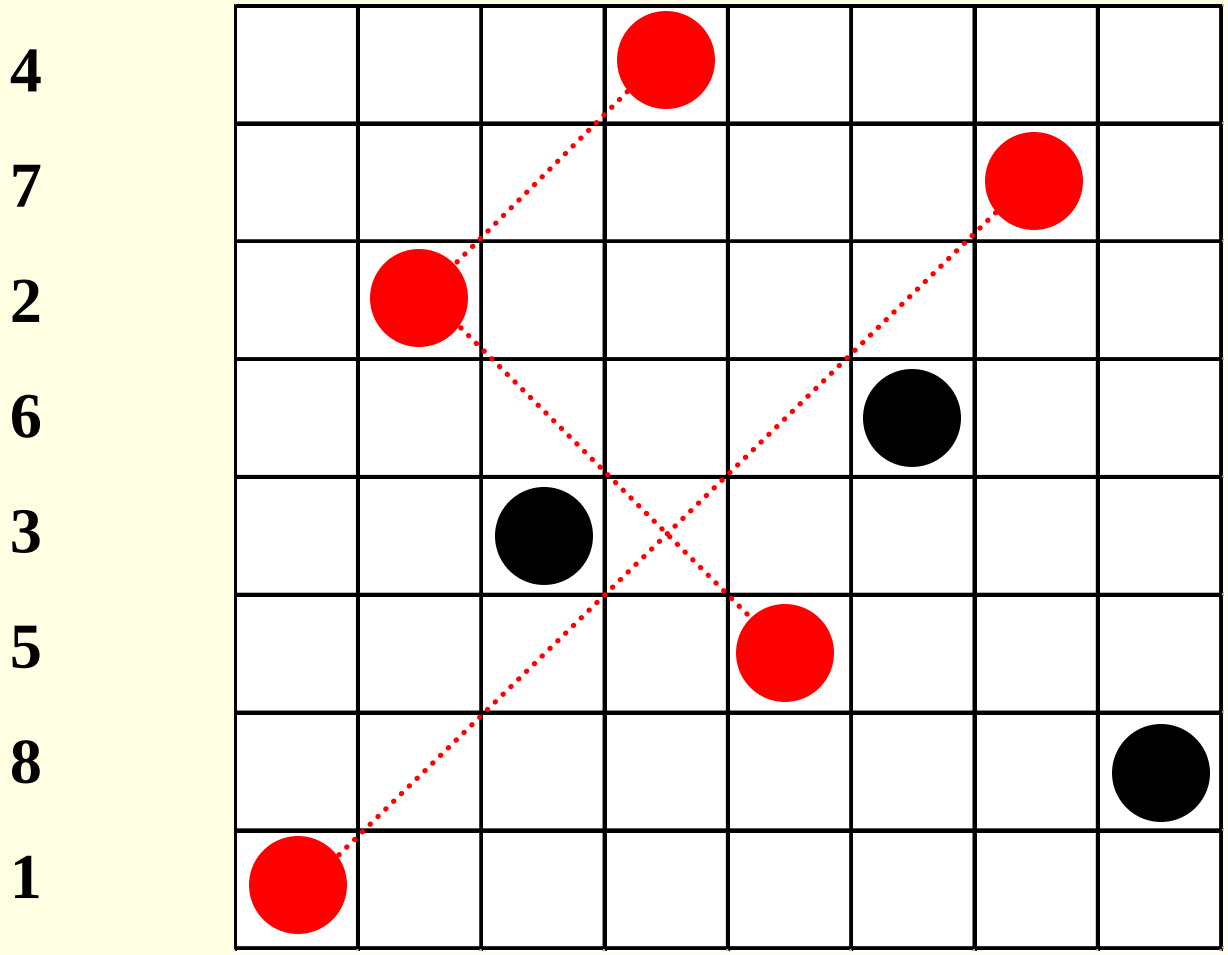


2 Ataques

3 - 5

4 - **8**

Pesquisa Local - Reparação



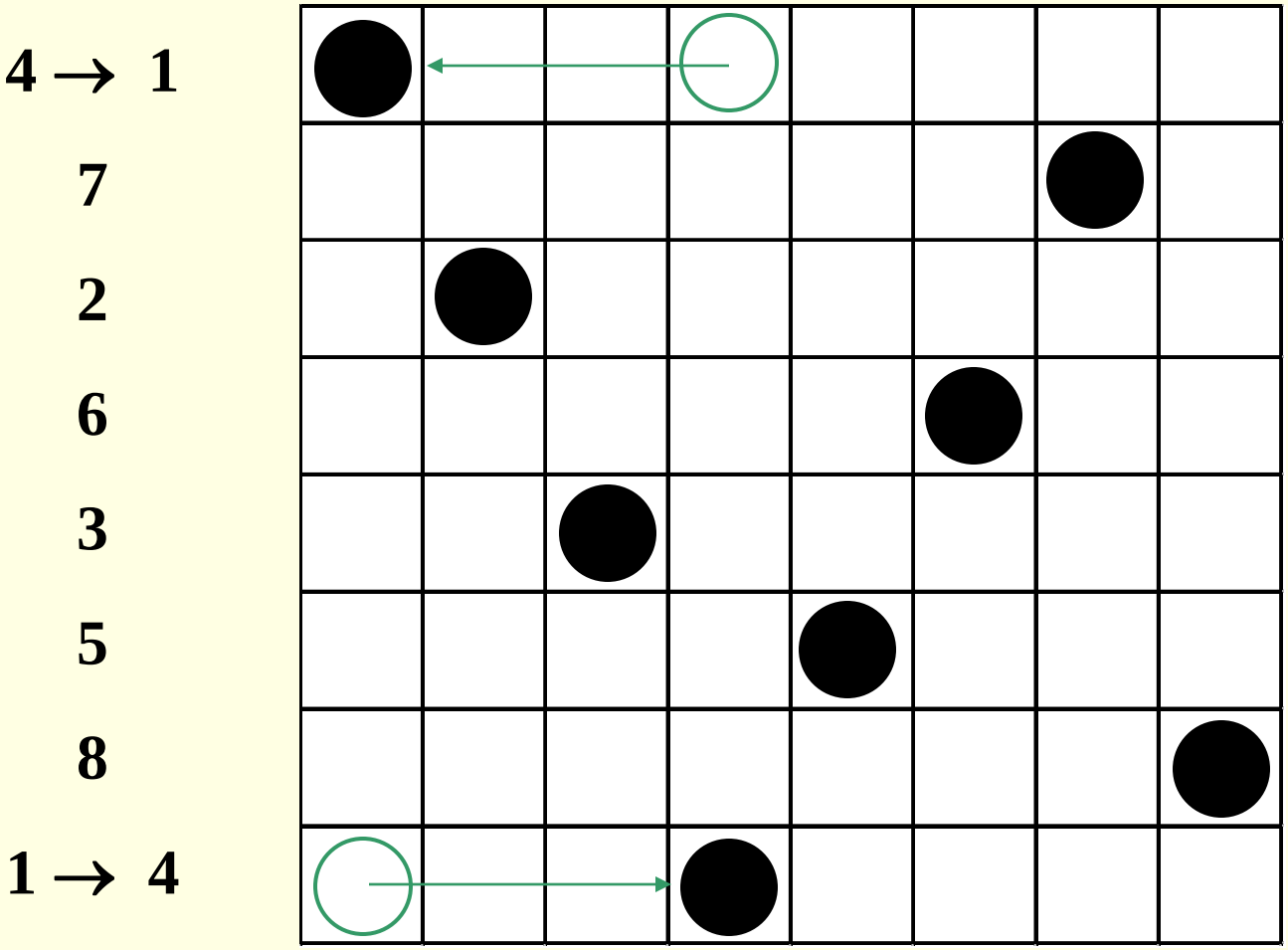
3 Ataques

1 - 3

2 - **8**

3 - 6

Pesquisa Local - Reparação



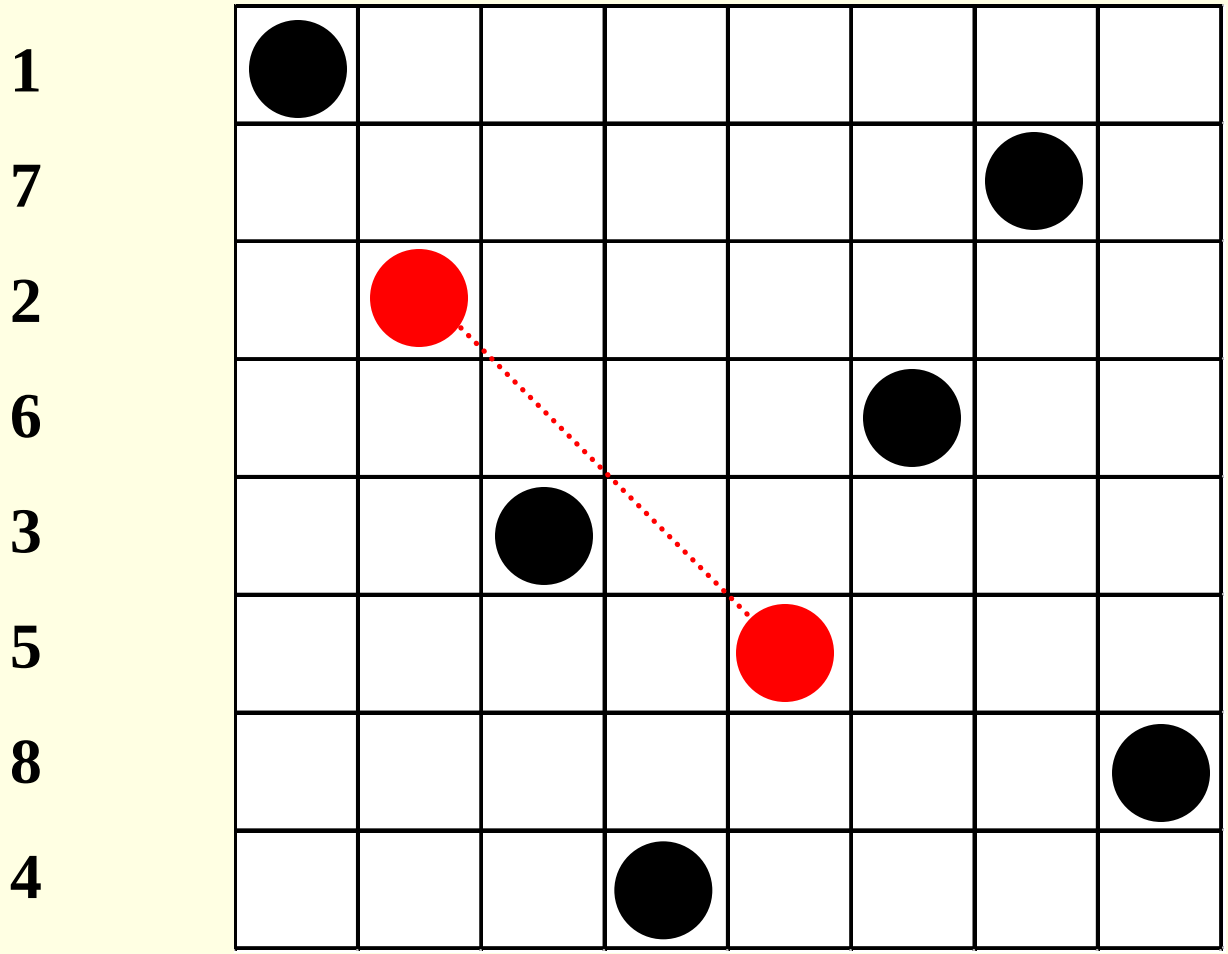
3 Ataques

1 - 3

2 - **8**

3 - 6

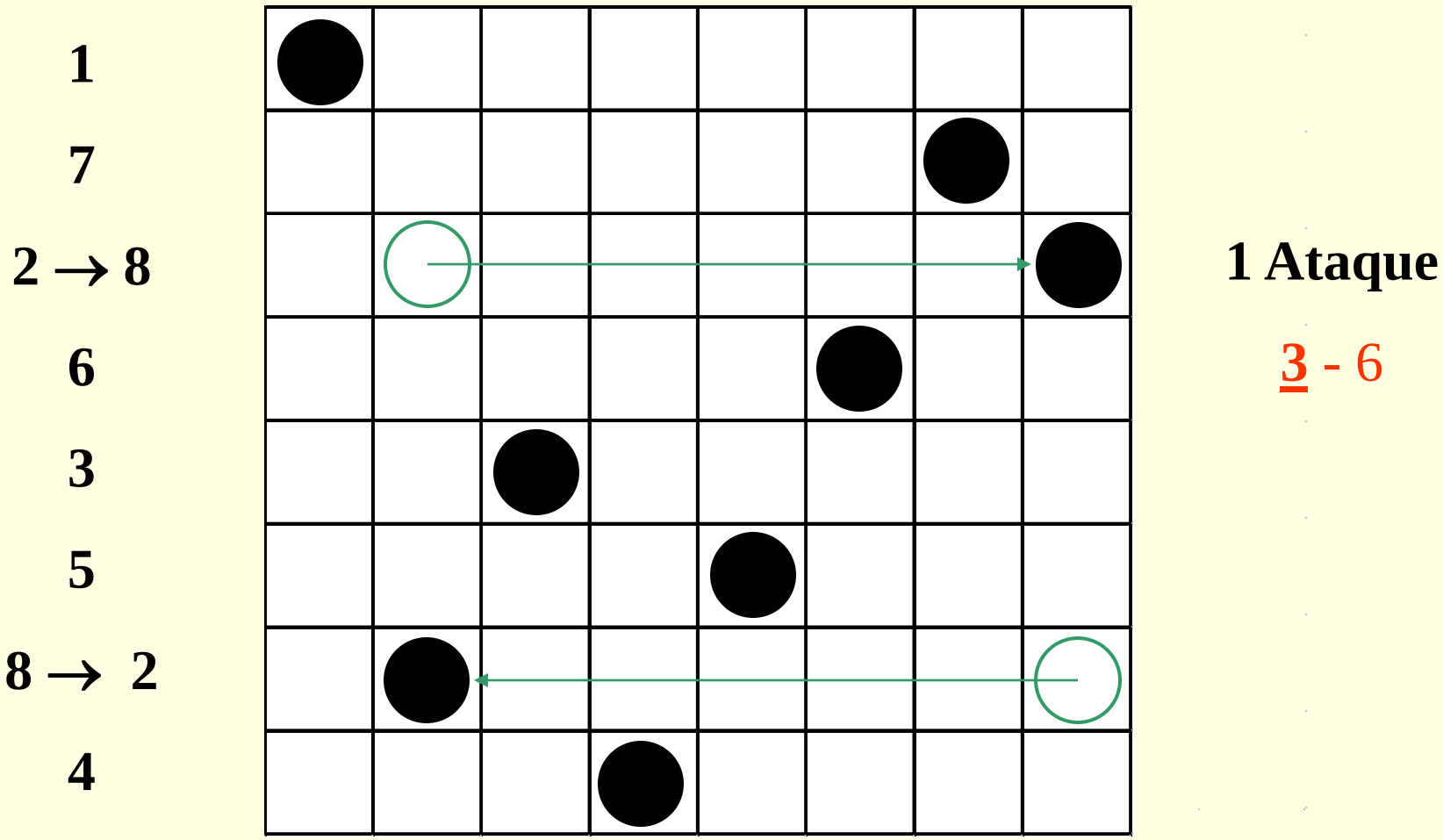
Pesquisa Local - Reparação



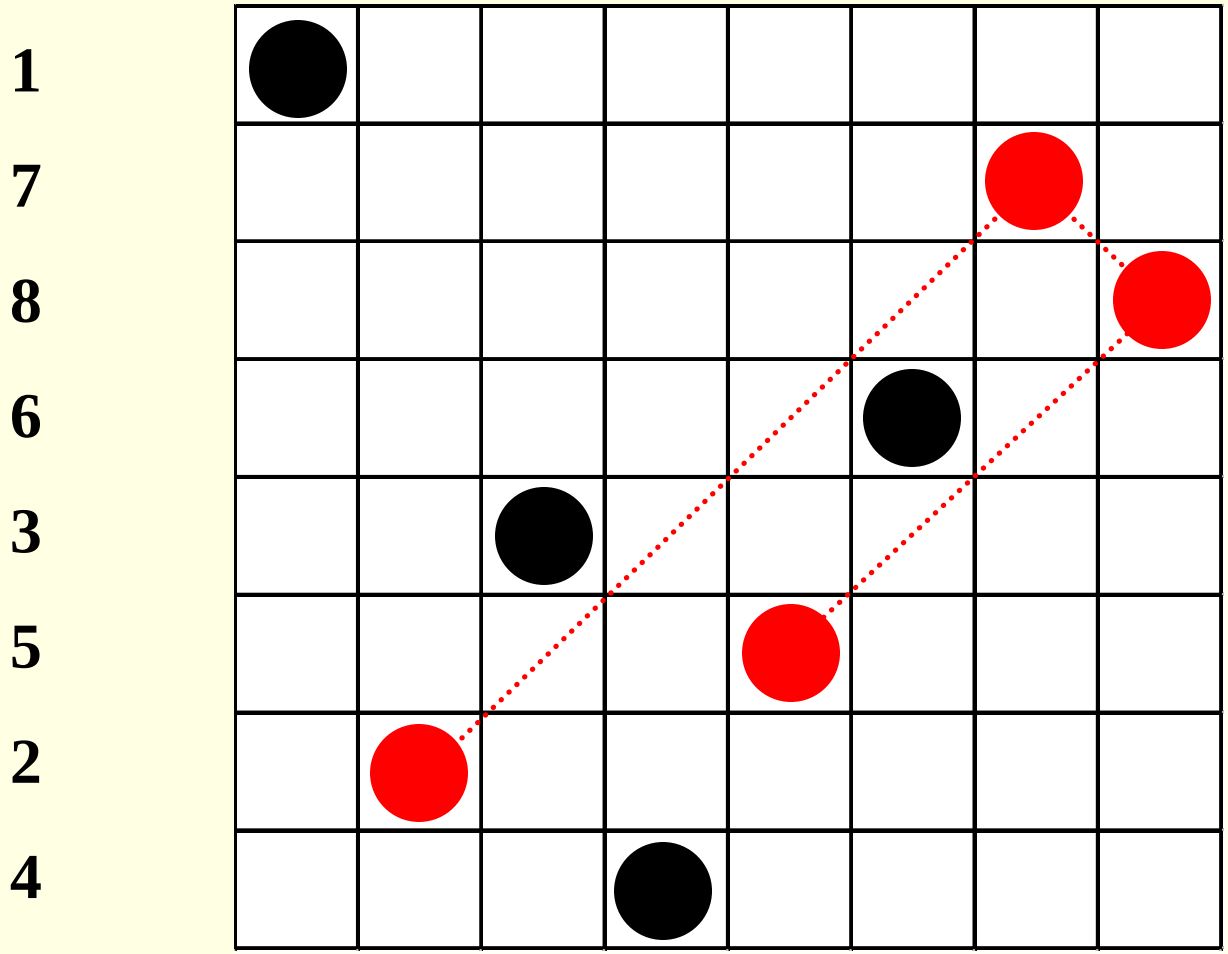
1 Ataque

3 - 6

Pesquisa Local - Reparação



Pesquisa Local - Reparação



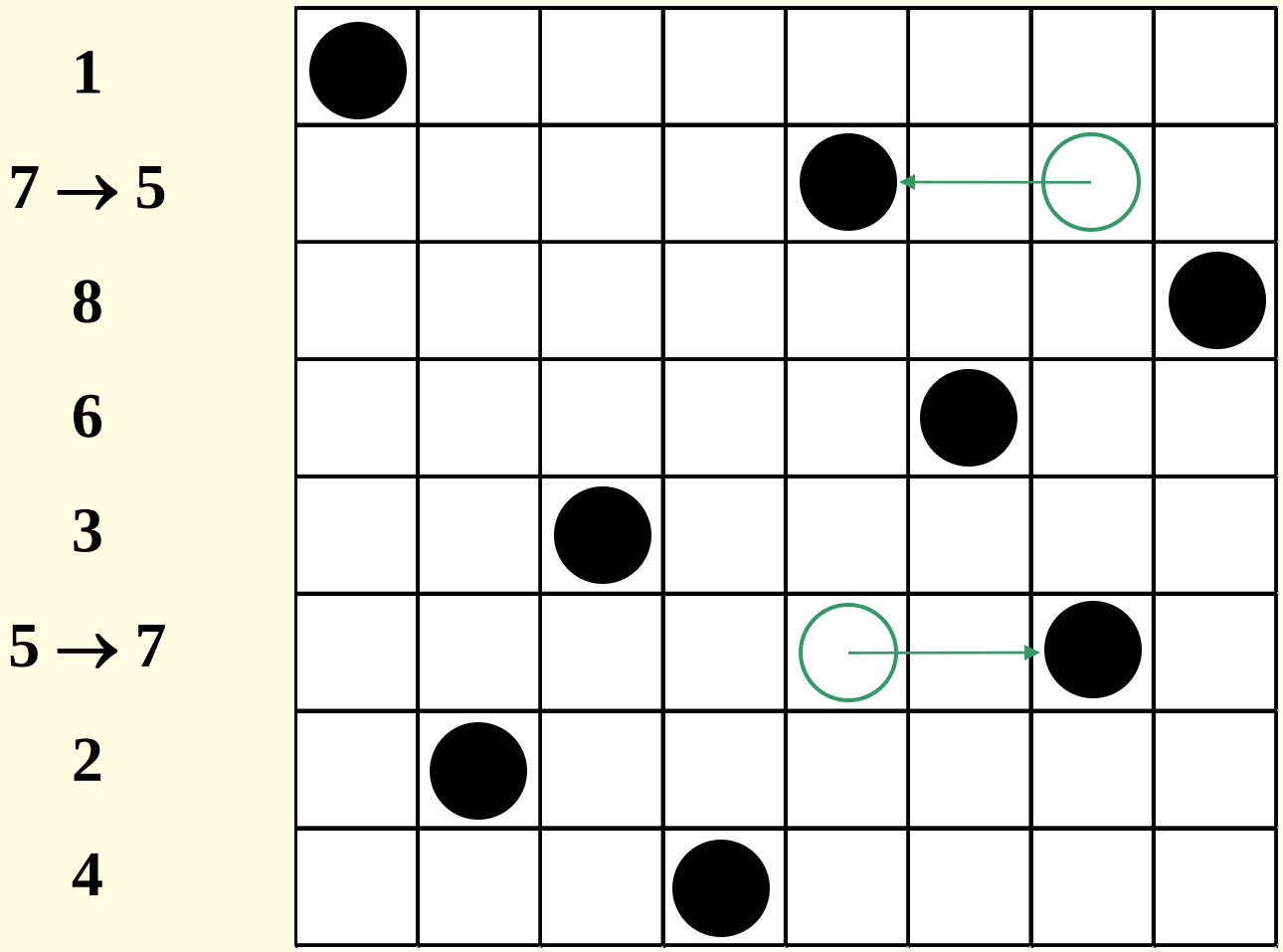
3 Ataques

2 - 3

2 - 7

3 - **6**

Pesquisa Local - Reparação



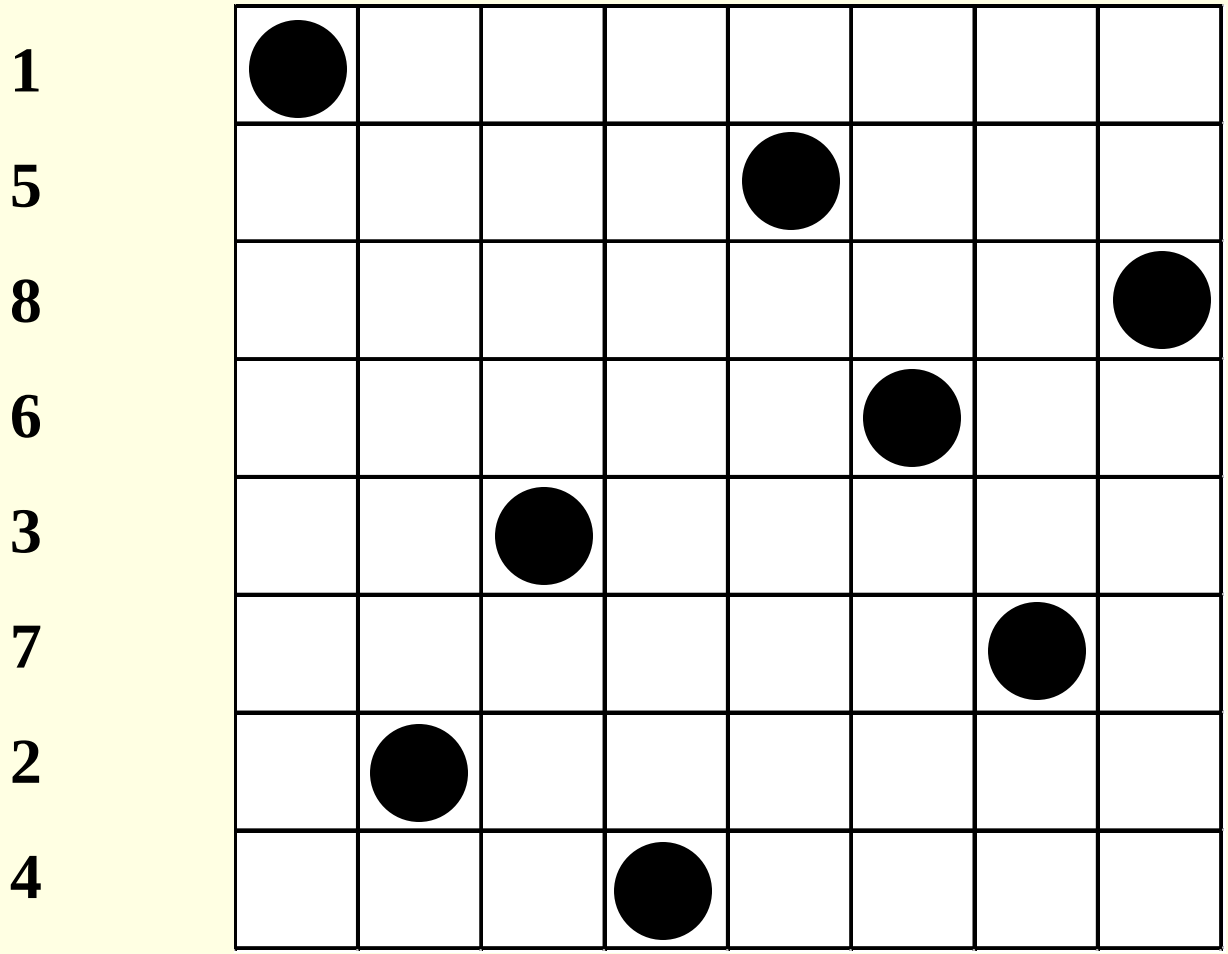
3 Ataques

2 - 3

2 - 7

3 - **6**

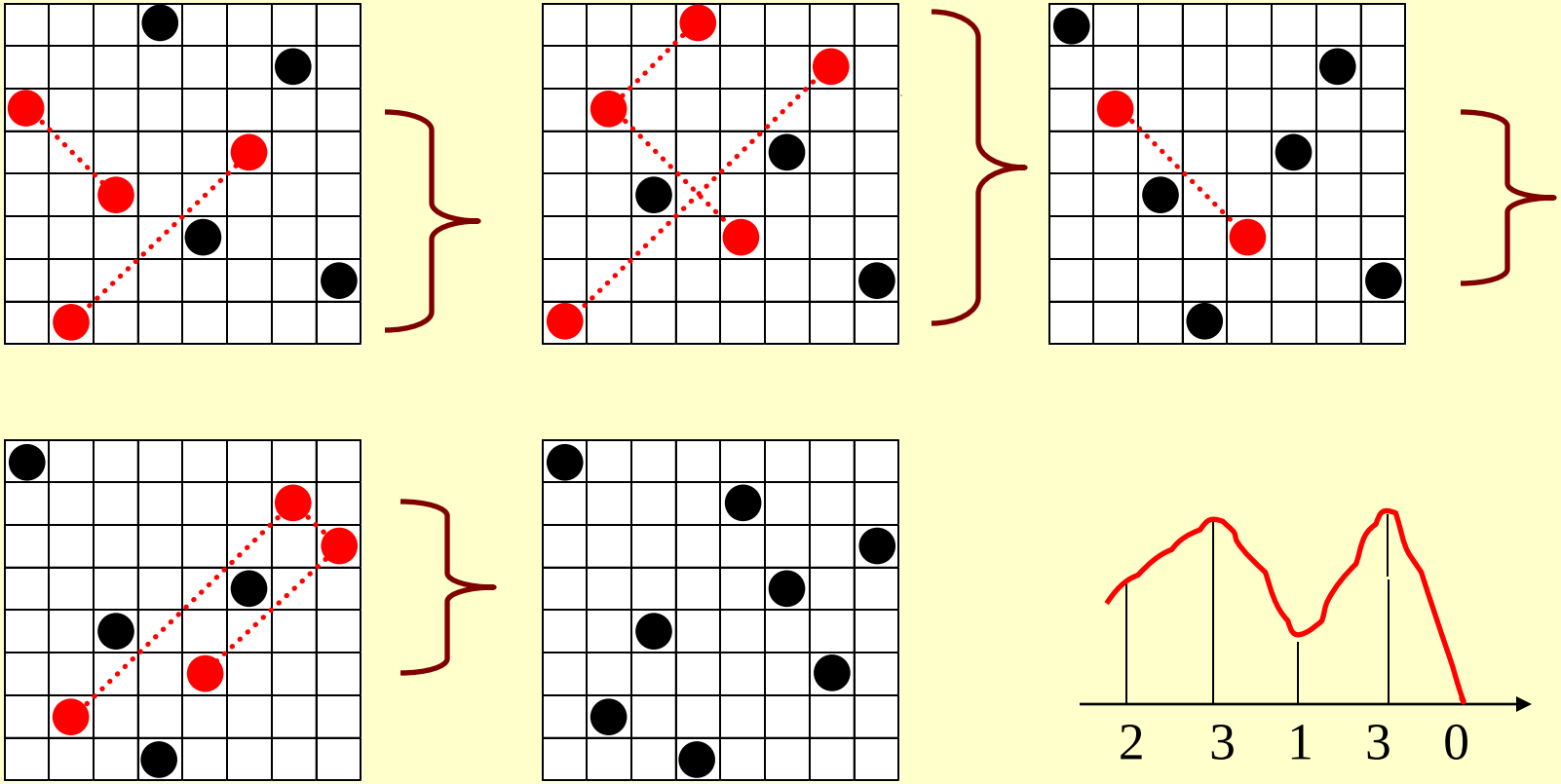
Pesquisa Local – Reparação



0 Ataques

Repairing Methods

- Difficulty - Escaping from Local Optima



- Restarts / Stochastic methods