Artificial Intelligence

Lecture 4 – Local Search

Outline

- Problems with ‘non-path’ solutions
- Local search algorithms
- State space landscape
- Hill-climbing search
- Stochastic hill-climbing
- Simulated annealing
- Genetic algorithms

Problem Definition

- A *search* problem is defined by:
  - a *state space* (i.e., an initial state or set of initial states and a set of operators)
  - a *set of goal states* (listed explicitly or given implicitly by means of a test that can be applied to a state to determine if it is a goal state)
- A *solution* is a path in the state space from an initial state to a goal state

‘Non-path’ Solutions

- For many problems, the *path* to the goal is irrelevant, e.g., VLSI design, job-shop scheduling, portfolio management
- In such problems the goal state is unknown:
  - goal is specified in the form of a *function* which assigns a (numerical) value to each state
  - aim is to minimise (or maximise) this value
  - the minimum (or maximum) possible value may not be known in advance (optimisation problems)
- The solution *is* the goal state rather than the path to the goal state

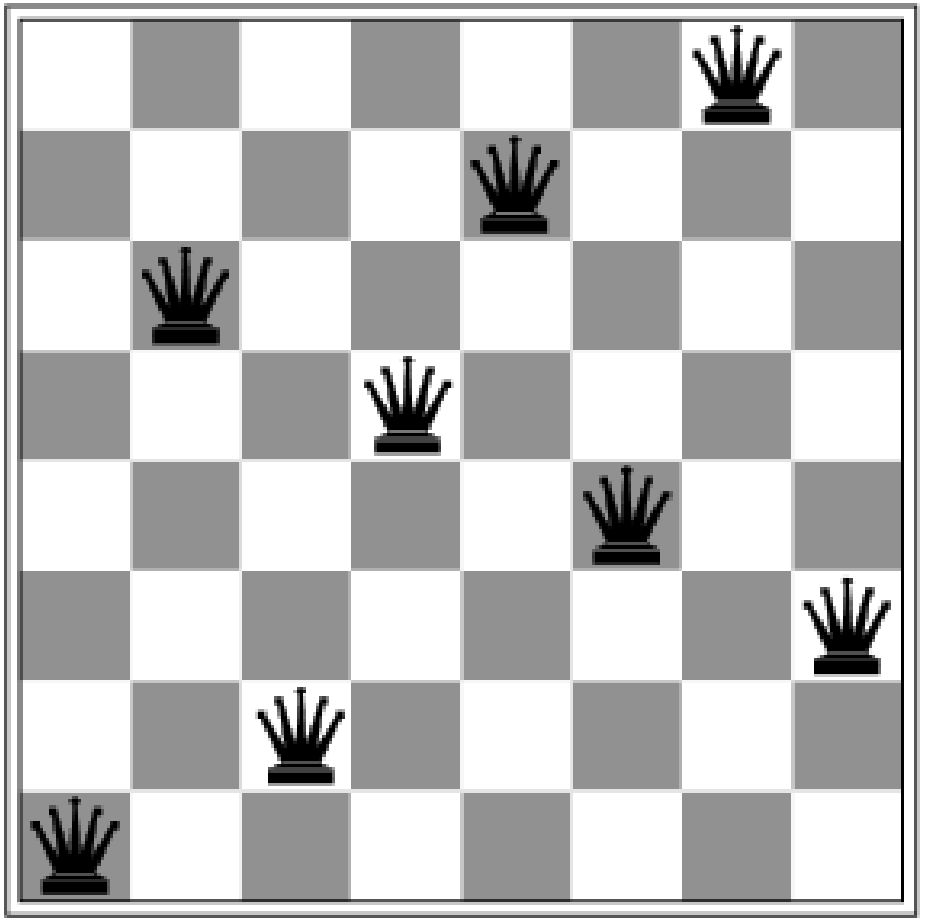
Revised Problem Definition

- A *local search problem* is defined by:
  - a *state space*, i.e., an initial state or set of initial states and a set of operators)
  - a *set of goal states* given implicitly
    - by means of a test that can be applied to a state to determine if it is a goal state; or
    - in the form of an objective function to be minimised or maximised
- A *solution* is a goal state

Example: Eight Queens Problem

- Goal is to place eight queens on an (otherwise empty) chess board so that they can't take each other
  - a queen can take another queen in the same row, column or diagonal
  - the order in which the queens are placed is not of interest, only their final positions
- Since a queen can take another queen in the same column, the problem is often formulated as one of choosing the row for a queen in each column
- Example of a *constraint satisfaction problem*

Example: Eight Queens Problem



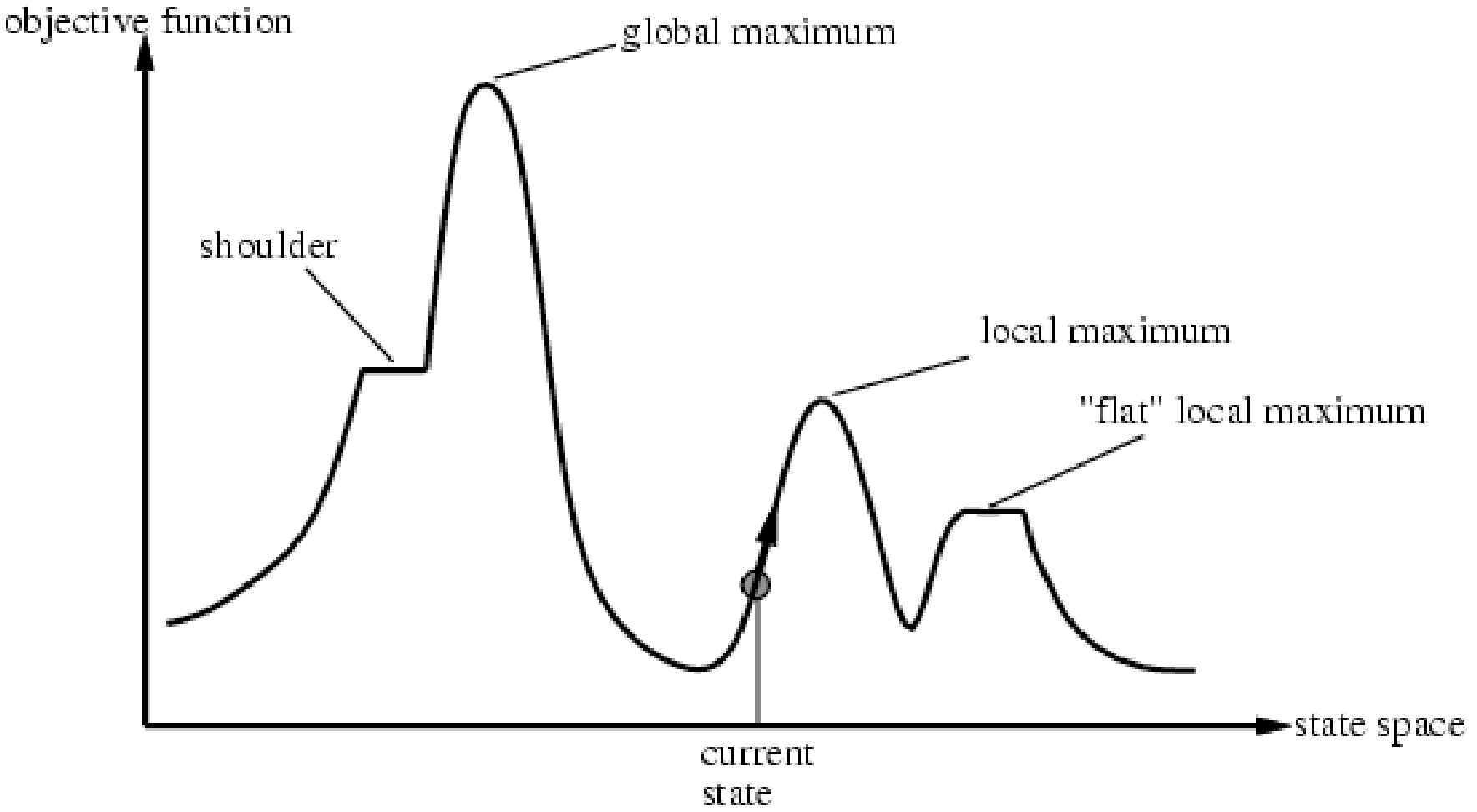
Local Search Algorithms

- Local search algorithms consider only *one state* (or a small number of states) at a time
- After applying operators to the current state, a new current state is chosen from its successors which *replaces* the current state
- Search is not systematic
- Path followed by the search are not retained
- Uses very little memory (usually a constant amount)
- Well suited to problems where the path to the solution doesn’t matter

State-Space Landscape

- Rather than paths in the state space, we have *points* (or sets of points), each with an associated value
- Value is often visualised as the “elevation” of a point in a state space landscape
- If elevation corresponds to *cost*, the aim is to find lowest point in this landscape - a global *minimum*
- If elevation corresponds to solution *quality*, the aim is find the highest point - a global *maximum*

Example: State-Space Landscape



Completeness and Optimality

- Local search algorithms explore this landscape
- A *complete* local search algorithm always finds a solution if one exits
- An *optimal* algorithm always find a global minimum / maximum

Hill-climbing Search

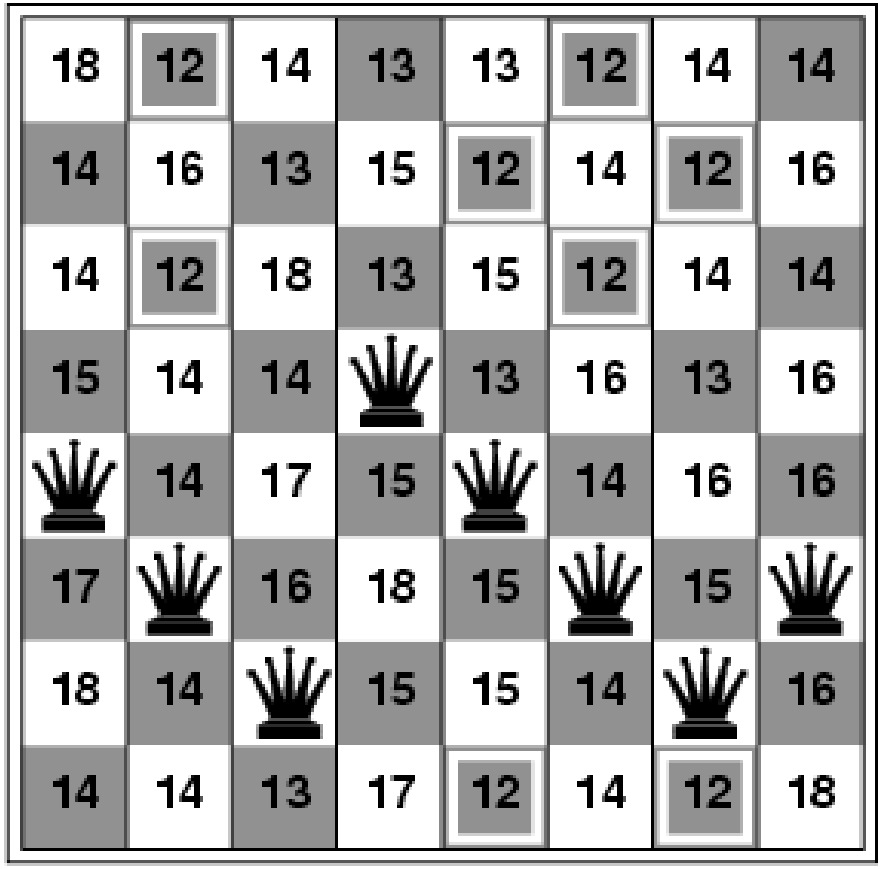
- Expands the initial state *s* and chooses the best of its successors *s′* (if there is more than one best successor, choose one at random)
- *s′* replaces *s* as the current state and is expanded in turn
- If none of the successors of are better (lower cost or higher quality) than the current state, the algorithm halts
- Follows the steepest gradient down (or up) hill from the initial state until no improvement is possible
- Only remembers the current state and its value

Example: Eight Queens

- Local search algorithms typically use a complete state formulation, e.g., in the eight queens problem, each state might have eight queens on the board, one per column
- Operators return all possible states generated by moving a single queen to another square in the same column (so each state has 8 x 7 = 56 successors)
- The *cost* function is the number of pairs of queens that can take each other
- Global *minimum* is zero, when no queen can take any other queen

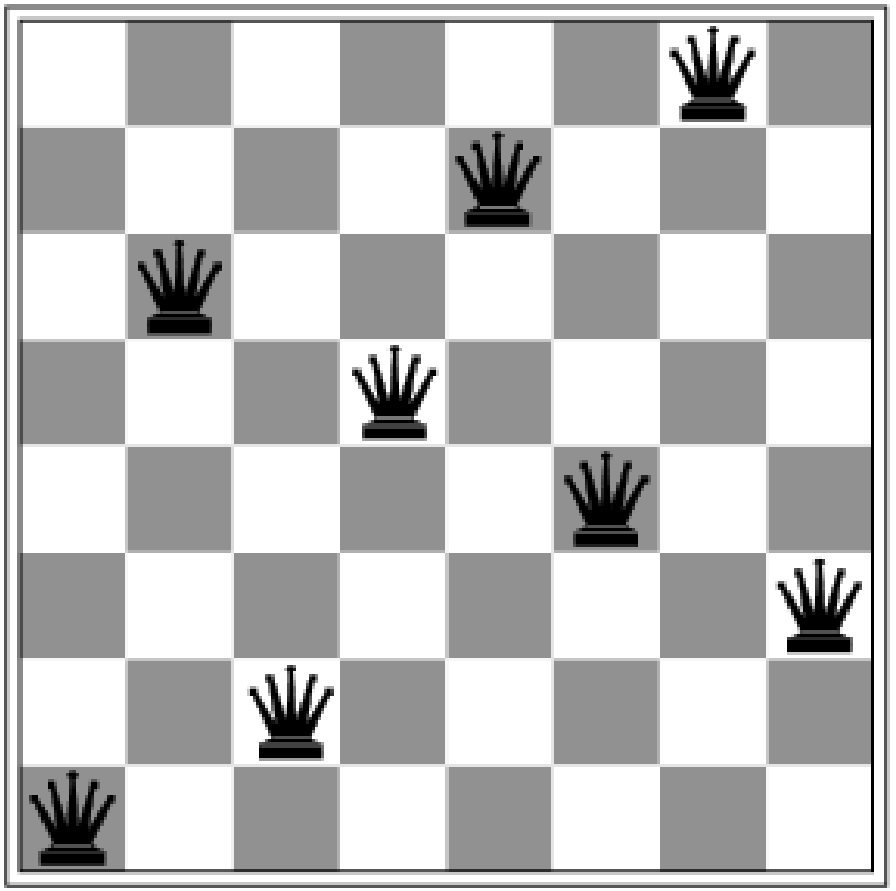
Example: Eight Queens

- Current cost is 17
- Values show the cost of each possible successor obtained by moving a queen to square indicated
- The best moves (value 12) are marked
- One of these is chosen and the process repeats



Example: Eight Queens

- Current cost is 1 (i.e, one pair of queens can take each other)
- Every successor has a higher cost, so the search terminates
- This is called a *local minimum* in the state space landscape



State, SearchProblem & Node

```
// class representing a problem state
class State

// class representing a search problem
class SearchProblem {
    // methods: initialState() and operators()
    // Note: initialState may be chosen randomly
}

// class representing a search node
class Node {
    // methods: state(), cost(), expand()
}
```

Hill-climbing Algorithm

```
// pseudocode implementing hill-climbing search
public Node HillClimbingSearch(SearchProblem problem) {
  Node currentState = new Node(problem.initialState())
  Node bestSuccessorState

  while(true) {
    bestSuccessorState =
        Collections.sort(currentState.expand(problem.operators()),
                                        CostComparator).removeFirst()
    if (bestSuccessorState.cost() < currentState.cost()) then {
      currentState = bestSuccessorState
    } else {
      return currentState
    }
  }
}
```

Limitations of Hill-climbing

- Hill-climbing search often makes rapid progress towards a solution - usually easy to improve a bad state
- However it relies on the state space landscape sloping continuously down (or up) to a goal state
- If it doesn't, hill-climbing often gets stuck:
  - in *local minima* (or maxima) - points that are lower than each of their neighbouring states, but higher than the global minimum
  - on *plateaus* - a set of points with the same elevation
- Hill-climbing is therefore *neither complete* nor *optimal*

Variants of Hill-climbing Search

- To escape local minima and plateaus, a number of variants of hill-climbing have been developed
  - **Sideways moves**: allowing a limited number of moves to states which are no worse than the current state, in the hope that these will take the algorithm to the edge of a shoulder/plateau
  - **First choice hill climbing**: generating successors randomly until one is found that is better than the current state (useful if a state has thousands of successors)
  - **Stochastic hill climbing**: choose at random among the best moves
- While these can increase the number of solutions found, they typically require more operator applications to find a solution and are still incomplete

Stochastic Hill-climbing

- Choose at random from among the best moves (may include states which are no worse than the current state)
- Probability of choosing a given move may be
  - 1/*n* where *n* is the number of good moves
  - proportional to the gradient, i.e., better moves have higher probability of being chosen
- Search terminates when a (local) optimum is found or after a given number of steps
- Typically takes longer to converge on a solution than simple hill-climbing, but finds better solutions in some state space landscapes

Stochastic Hill-climbing Algorithm

```
// pseudocode implementing stochastic hill-climbing search
public Node StochasticHillClimbingSearch(SearchProblem problem) {
   Node currentState = new Node(problem.initialState())
   Node successorState

   while(true) {
      LinkedList<Node> nodes = Collections.sort(currentState.expand(problem.operators()).
                                                add(currentState), CostComparator)
      int c = nodes.indexOf(currentState)
      succssorState = nodes.subList(0, c).shuffle().removeFirst()
      if (successorState.cost() < currentState.cost()) then {
         currentState = successorState
      } else {
         return currentState
      }
   }
}
```

Random Restart Hill-climbing Search

- Performs a *series* of hill-climbing (or stochastic hill-climbing) searches from randomly generated initial states
- Each search terminates when a *local optimum* is found, when no discernable progress has been made for *k* steps (stochastic hill-climbing), or after a fixed number of steps
- Saves the best result found so far from any of the searches
- Can use either a fixed number of restarts or continue until the best result has not been improved for a certain number of restarts

Properties of Random Restart Hill-climbing

- *Complete* with probability approaching 1 (given enough restarts)
- If each hill-climbing search has probability p of success, the expected number of restarts required is $1/p$
  - e.g., starting from a randomly generated eight queens state, steepest descent hill-climbing gets stuck about 86% of the time
  - $p \approx 0.14$ so we need approximately 7 restarts to find a goal state
- However we may not know the probability of success for a hill-climbing search

Properties of Random Restart Hill-climbing

- If there are few local minima/maxima or plateaus, random restart hill climbing will find a good solution very quickly
- NP-hard problems typically have an exponential number of local minima/maxima for the search to get stuck on
- Despite this, a *reasonably good* solution can often be found after a small number of restarts
- Can be more effective to limit the number of steps in each hill-climbing search and perform more searches

Simulated Annealing Search

- Rather than restarting when stuck at a local optimum, *simulated annealing* allows moves to states which are *worse* than the current state
- Picks a move from the current state at random
  - if the resulting state is better than the current state the move is accepted
  - otherwise the move is made with a probability which decreases exponentially with the badness of the move and the "temperature", T
- Probability of making moves to states which are worse than the current state declines as the search progresses - controlled by the *annealing schedule*
- Search terminates when $T = 0$

Simulated Annealing Algorithm

```
// pseudocode implementing simulated annealing search
public Node SimulatedAnnealingSearch(SearchProblem problem) {
   Node currentState = new Node(problem.initialState())
   Node successorState

   while(T > 0) {
      LinkedList<Node> nodes = currentState.expand(problem.operators())
      succssorState = nodes.shuffle().removeFirst()
      if (successorState.cost() < currentState.cost()) then {
         currentState = successorState
      } else{
         // Replace currentState with successorState with probability
         // e**(-(successorState.cost() – currentState.cost())/T)
      }
      // Decrement T according to the annealing schedule
   }
}
```

Properties of Simulated Annealing

- If the annealing schedule lowers T slowly enough, simulated annealing is *complete* and *optimal*
- However finding an optimal (or even a very good) solution may take a *long* time
- Simulated annealing has been used to solve VLSI design problems, factory scheduling and other large-scale optimisation problems

Genetic Algorithms

- Inspired by natural selection
- Considers a set of *k* states ('population') at a time rather than a single current state
- Starts with a set of *k* randomly generated initial states
- Combines (parts of) states ('individuals') in the population to produce new states ('offspring')
- Offspring may either replace the current population, or the best of the offspring may replace the worst of the current population
- Algorithm halts when either the goal is found or after a given number of iterations ('generations')

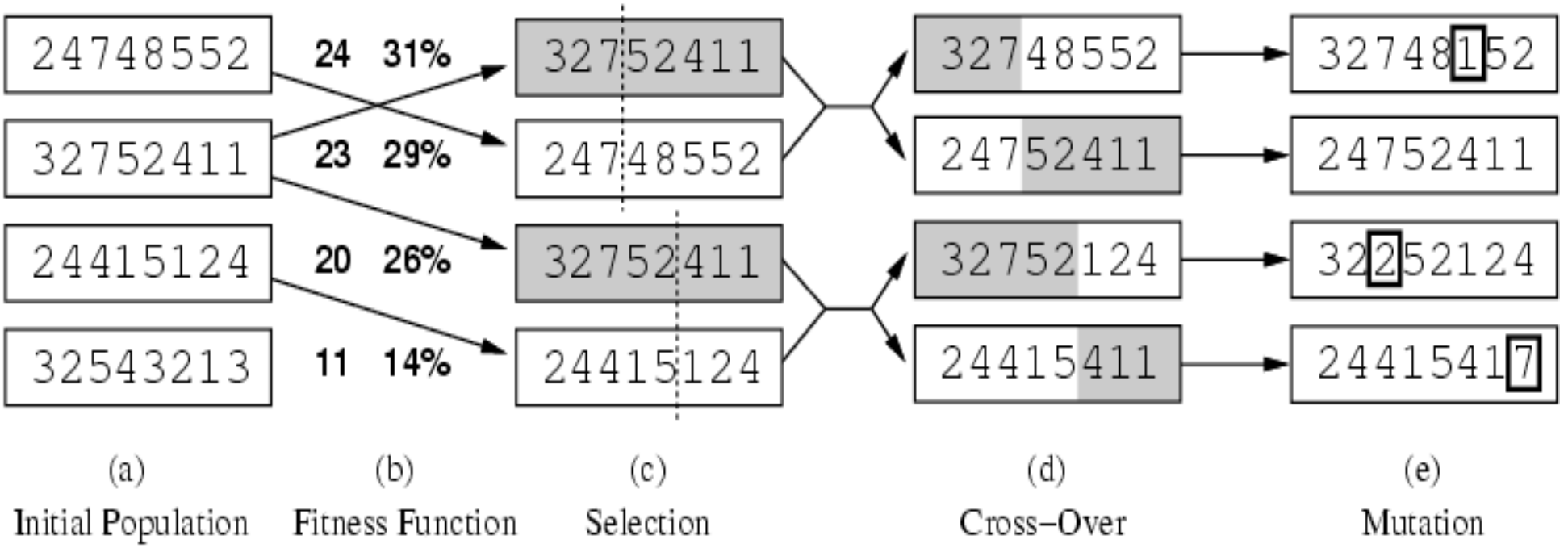
State Representation and Evaluation

- Individuals are represented as strings over a finite alphabet - usually {0,1}, e.g., states in the eight queens problem can be encoded in 24 bits
- The value of each individual is computed using a *fitness function* which returns higher values for better states, e.g, the number of non attacking queens
- *k* pairs of individuals are then selected to 'reproduce' - probability of selection is often taken to be proportional to the fitness value
- Some individuals may be selected multiple times and some not at all

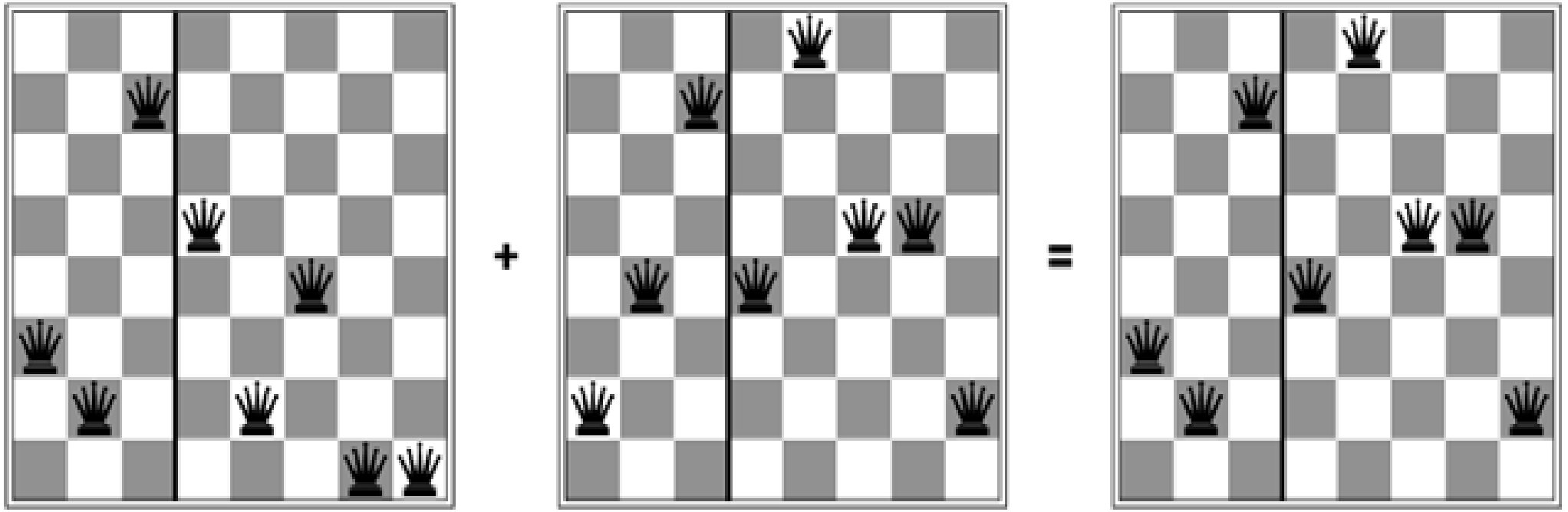
Crossover and Mutation

- For each pair, a *crossover point* is randomly chosen from the positions in the string
- *Offspring* are then created by crossing over the parent strings at the crossover point, e.g. the first child gets the first 3 digits from the first parent and the remaining digits from the second parent
- Each location in the offspring is subject to random *mutation* with a small probability
- The offspring combines part solutions from each parent and possibly some variation
- (Some of) the offspring then replace (some of) the parent population and the next generation begins

Example: Eight Queens GA



Example: Eight Queens Crossover



Properties of Genetic Algorithms

- Key advantage of GAs is the crossover operation
- Useful information can be passed between *k* different solutions
- If the initial population is random, GAs (like simulated annealing) make large moves in the state space initially, and smaller moves later on when the individuals are quite similar
- Intuitively crossover combines substrings which have evolved independently to solve parts of the problem, raising the granularity of the search (*schema theory*)
- GAs work best when schemas correspond to meaningful components of the solution - requires careful engineering of the representation

Summary

- Unlike (some) uninformed search techniques, local search techniques are usually *incomplete* in practice
- However they can successfully be applied to problems which are too large for systematic search techniques (uninformed or informed), e.g, where the number of operators is large
- Often capable of finding good solutions in a reasonable amount of time
- GAs are often out-performed by simpler stochastic hill-climbing techniques