Algorithms and their Applications CS2004 (2020-2021)

Dr Mahir Arzoky

12.1 Tabu Search and ILS

Class Tests So Far…

- Class Test CRI: 192 attempts
- Class Test CRII: 90 attempts
- Class Test CRIII: 54 attempts
- Class Test CRIV: (maybe) released next week!
- **All four class tests must be passed to pass Task #1.**
- **Task #1 weighs 30% of the coursework**
- **But, if you do not pass Task #1 you will be capped at D- grade (coursework).**
- **Class tests needs to be completed by 16/02/2021**

Previously On CS2004…

- So far we have looked at:
  - Concepts of Computation and Algorithms
  - Comparing algorithms
  - Some mathematical foundation
  - The Big-Oh notation
  - Computational Complexity
  - Data structures
  - Sorting Algorithms
  - Various graph traversal algorithms
  - Heuristic Search
  - Hill Climbing and Simulated Annealing
  - Parameter Optimisation (Applications)

Introduction

- In this lecture we are going to look into two Heuristic Search methods:
  - Tabu Search (TS)
  - Iterated Local Search (ILS)
- These are **Meta-Heuristic** search methods
- We are then going to look into some implementation details involved in applying ILS to the Scales problem
  - Representation improvements
  - An **Updatable** Fitness Function
  - Performance

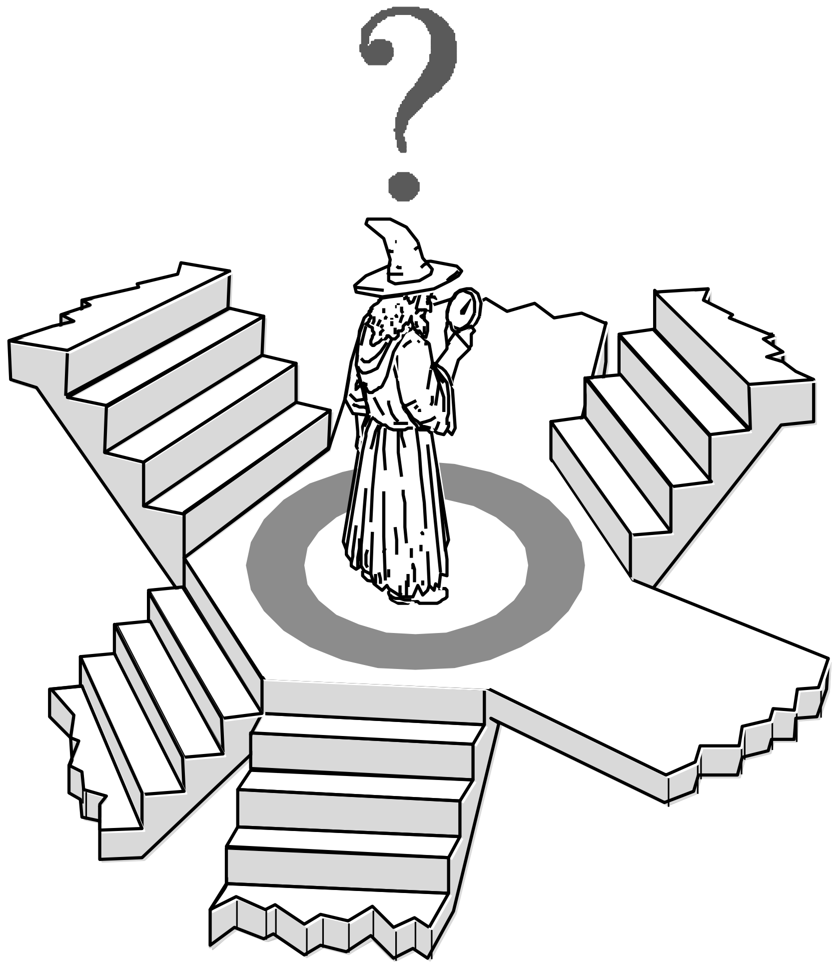
Heuristics – Recap

- A **"rule of thumb"** or **loose set of guidelines**
- No guarantees on quality of solution
- Usually fast and are widely used
- Sometimes we run them multiple times and analyse the results

Popular Heuristics

- Hill Climbing
    - Always go up hill (accept only better quality solutions)
- Simulated Annealing
    - The concepts of annealing and temperature
- Iterated Local Search
    - See later slides
- Tabu Search
    - See later slides
- Genetic Algorithms
    - Simulated evolution
    - See later lecture
- Ant Colony Optimisation
    - Pheromones and cooperation
    - See later lecture
- Particle Swarm Optimisation
    - Swarming
    - See later lecture
- Etc……

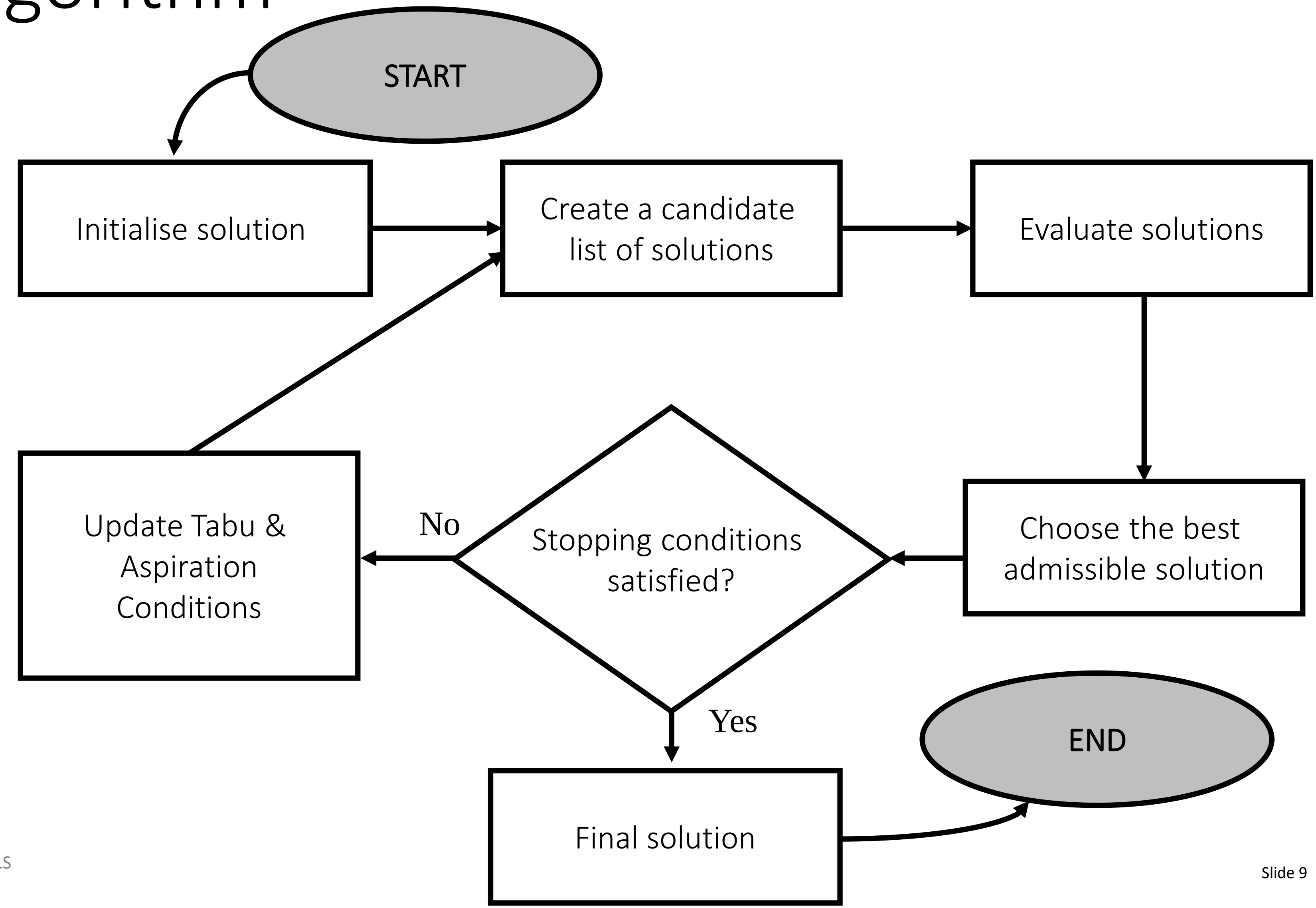
Local View of Search



Tabu (Taboo) Search

- Tabu search tries to model human memory processes
  - Key idea: Use aspects of search history (memory) to escape from local minima
- A **tabu-list** is maintained throughout the search
  - Associate **tabu attributes** with candidate solutions or solution components
  - Moves according to the items on the list are forbidden

Flow Chart of a Standard Tabu Search Algorithm



Tabu Search – Key Concepts

- A search (similar to Hill Climbing) that remembers sets of points in the search space
  - These are called the **Tabu** list and are to be avoided
- The idea is that this helps in avoiding local optima
- However if a point is evaluated to be better than any points discovered so far, then the Tabu list may be updated
  - Aspiration Criteria

Tabu Search Stopping Conditions

- Some immediate stopping conditions:
  - No feasible solution in the neighborhood of solution
  - The maximum amount of iterations or CPU time has been exceeded
  - The number of iterations since the last improvement is larger than a specified number
  - Evidence can be given than an optimum solution has been obtained

Pros and Cons for Tabu Search

- Pros:
  - The use of a Tabu list
  - Can be applied to discrete and continuous solution spaces
  - A meta-heuristic that guides a local search procedure to explore the solution space beyond local optimality
  - For difficult problems (e.g. scheduling and vehicle routing), tabu search obtains solutions that rival / surpass other approaches
- Cons:
  - Too many parameters to be determined
  - Number of iterations could be very large
  - Global optimum may not be found, depends on parameter settings
  - Tabu list can grow out of control

Iterated Local Search (ILS) - Key Concepts

- ILS uses another local search algorithm as part of the algorithm
  - E.g. Hill Climbing
- Built on premise that local search algorithms are easily trapped in local optima!
- It uses information regarding previously discovered local optima (and/or starting points) to locate new (and hopefully better) local optima
  - The current solution is perturbed (changed), and is then used as a new starting point for the search

ILS – Algorithmic Steps

- The key algorithmic steps are as follows:

```
s0 = Generate initial solution
s* = LocalSearch(s0)
history = ϕ
Repeat
  history = history ∪ s* [Remember previous optima and maybe start]
  scurrent = Perturb(s*,history) [Try to avoid similar starting points]
  scurrent* = LocalSearch(scurrent)
  s* = Accept(s*, scurrent*, history) [Often just accept best]
Until termination condition met
```

ILS – Perturbation and Acceptance Criteria

- Perturbation is key
  - Needs to be chosen so that it cannot be undone easily by subsequent local search
  - It may consist of many perturbation steps
  - Strong perturbation: more effective escape from local optima but similar drawbacks as random restart
  - Weak perturbation: short subsequent local search phase but risk of revisiting previous optima
- Acceptance criteria: usually either the more recent or the best

ILS - Pros and Cons

- Pros:
  - Often leads to very good performance
  - Easy to implement a simple ILS
  - State-of-the-art results with further optimisations
- Cons:
  - Deep understanding of the problem and, trial and error may be required for a good perturbation method
  - Using a bad perturbation method: keep returning to the same local optima or our metaheuristic may resemble random restart!

ILS and The Scales – Part 1

- We are going to look at some performance considerations that can be applied to **all** Heuristic Search Methods applied to the **Scales** Problem
- We will briefly look at how we implement ILS and how it performs compared to other heuristic search methods

ILS and The Scales – Part 2

- Representation
  - At the moment we are representing a solution as a **Binary String** or **Array of Integers**
  - Is this a good idea?
  - Each digit or part is either 0 or 1
  - This just needs one bit, but we are using 32 bits!!! (or 64 bits – I assume we are using 32…)
  - This is like writing on a pad of paper by using one sheet per letter…

ILS and The Scales – Part 3

- Representation
  - We can save space and thus efficiency (speed) by just using the number of bits we need
  - For n weights and b bits (32) we would need the following number of integers (m)

$$m = \left\lceil \frac{n}{b} \right\rceil$$

  - to represent our weight/scales allocation
- We would need 32 integers for 1000 weights...

ILS and The Scales – Part 4

- ❑ Luckily the latest version of Java comes with a built in class!

```
import java.util.BitSet;
public class BitSetTest {
   public static void main(String args[]) {
      int n = 25;
      BitSet bs = new BitSet(n);
      bs.set(0,n,true);
      ShowBits(bs,n);
      bs.flip(0,n);
      ShowBits(bs,n);
      bs.set(17,true);
      ShowBits(bs,n);
   }
   private static void ShowBits(BitSet bs,int n) {
      for(int i=0;i<n;++i) System.out.print((bs.get(i))?"1":"0");
      System.out.println();
   }
}
```

Output:

```
1111111111111111111111111
0000000000000000000000000
0000000000000000010000000
```

ILS and The Scales – Part 5

- Fitness
  - Our fitness function for the Scales problem is an $O(n)$ algorithm
  - However note the following:
  - If we record and remember the LHS and RHS totals and the position ($index$) of the last small change
  - If we changed a '1' to a '0' (moved from right to left)
    - $NewLHS = LHS + weight(index)$
    - $NewRHS = RHS - weight(index)$
  - If we changed a '0' to a '1' (moved from left to right)
    - $NewLHS = LHS - weight(index)$
    - $NewRHS = RHS + weight(index)$

ILS and The Scales – Part 6

- Fitness
  - Thus we have created an **updatable fitness function**
    - **New Fitness = Old Fitness + Value based on small change**
  - We have reduced our time complexity from O(*n*) to O(1) [the notation for constant time]
  - I.e. For 1000 weights our program could be up to 1000 times faster!!!
  - Combining this with the more compact representation discussed previously we now have a much more efficient algorithm

ILS and The Scales – Part 7

- Implementation
  - Base our algorithm on RRHC method
  - Reuse any code we might have
  - Need a way of generating starting positions that is not entirely random
  - These starting positions should be different to previous starting points and local optima
    - Generate the first starting point randomly
    - Remember the local optima
    - Bias the way that each starting point is generated based on the recorded starting points and local optima

ILS and The Scales – Part 8

- Results
  - Experiments were conducted on 10,000 weights generated randomly (UR) between 100 and 1000
  - 100 Repeats
  - All methods were allowed 10,000 fitness function calls
    - Worst Fitness = 5524223.536
    - RMHC = 65.418
    - RRHC = 8.248
    - ILSHC = 8.192

ILS and The Scales – Part 9

- Results
  - ILS is less than 1% better than RRHC!
  - But it **is** better…
  - Better ways of combining the starting positions and previously discovered local optima might results in a better performance…

This Weeks Laboratory

- There is no new worksheet this week!!
- Make sure that you use the laboratory to catchup with your worksheets and CodeRunner class tests

Next Lecture

- We will be looking at Genetic Algorithms