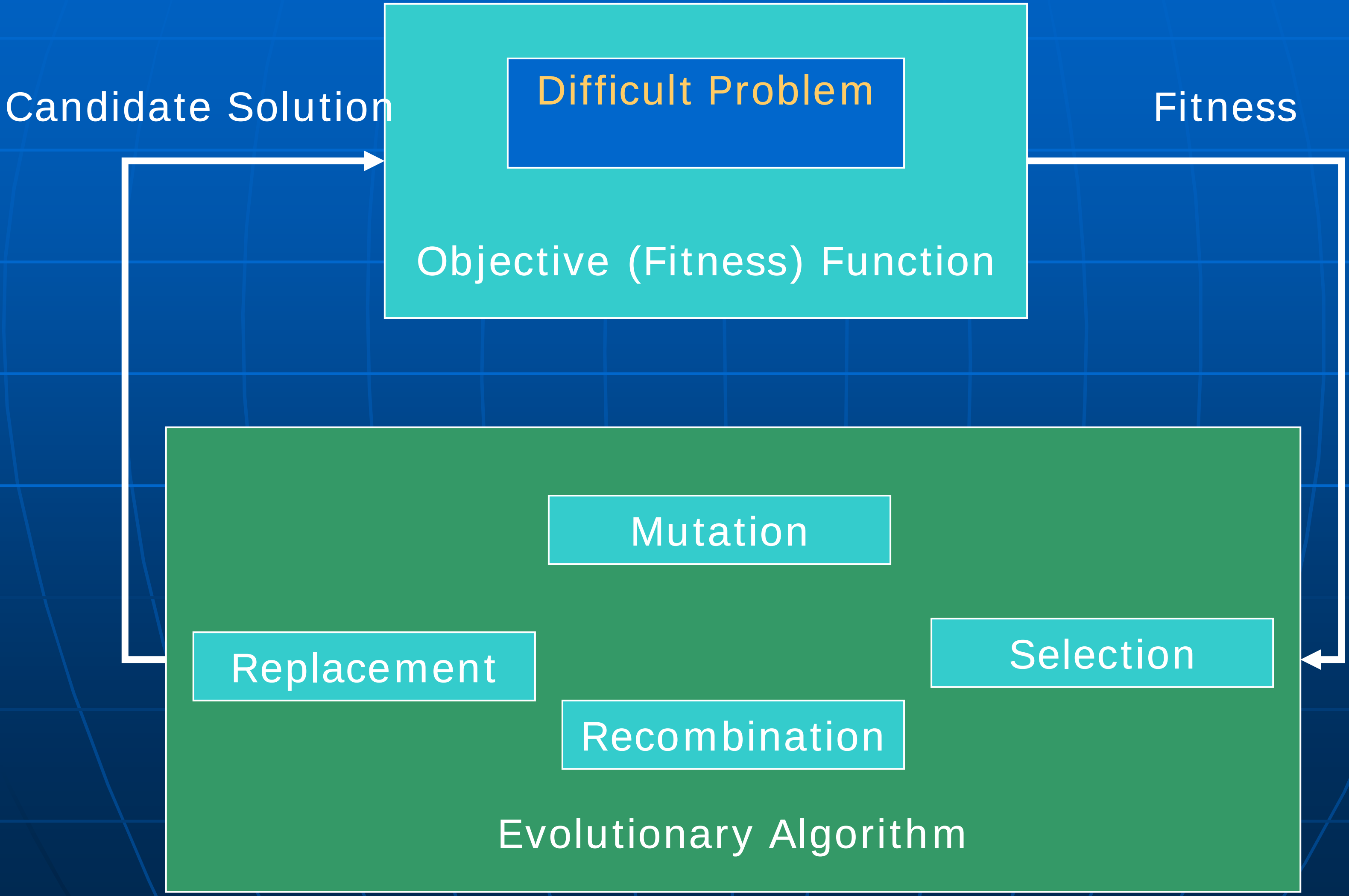
ECM3412/ECMM406
Nature Inspired Computation
Lecture 3

Evolutionary Algorithms in Detail
(The basic variants / local and population-based search / landscapes / examples

Basic Principle of Evolutionary Algorithms



Some Terms Used in Evolutionary Computation

- Algorithm Type
  - Generational
  - Steady State
  - Elitist
- Recombination (Crossover)
  - Single-point
  - Multi-point
  - Uniform
- Selection
  - Rank-Biased
  - Roulette Wheel
  - Tournament
- Replacer
  - Weakest
  - First weaker
- Mutation
- Parameters
  - Population size
  - Generations/iterations
  - Mutation rate
  - Crossover rate
  - Tournament size….
  - Etc

2 Types of Genetic Algorithm

- Generational – genetic operators applied repeatedly to generate new population.
- New solutions in yellow.

Initial Pop

Name	F(S)
S1	0.1
S2	0.5
S3	0.3
S4	0.2
S5	0.9
S6	0.7
S7	0.3
S8	0.4
S9	0.4
S10	0.1

Apply selection & genetic operators 10x to give new population $\longrightarrow$

Generation 1

Name	F(S)
S11	0.5
S12	0.3
S13	0.3
S14	0.7
S15	0.7
S16	0.9
S17	0.4
S18	0.9
S19	0.9
S20	0.3

Apply selection & genetic operators 10x to give new population $\longrightarrow$

Generation 2

Name	F(S)
S21	0.7
S22	0.8
S23	0.9
S24	0.9
S25	0.7
S26	0.8
S27	0.5
S28	0.7
S29	0.6
S30	0.7

2 Types of Genetic Algorithm II

- Steady State – genetic operators applied N times and bad solutions replaced
- New solutions in yellow.

Initial Pop

Name	F(S)
S1	0.1
S2	0.5
S3	0.3
S4	0.2
S5	0.9
S6	0.7
S7	0.3
S8	0.4
S9	0.4
S10	0.1

Apply selection & genetic operators N times to give new individuals

Name	F(S)
S11	0.1
S12	0.5

Replace weak solutions in population

Generation 1

Name	F(S)
S12	0.5
S2	0.5
S3	0.3
S4	0.2
S5	0.9
S6	0.7
S7	0.3
S8	0.4
S9	0.4
S11	0.1

A Standard Evolutionary Algorithm

The algorithm whose pseudocode is on the next slide is a *steady state*, *replace-worst* EA with *tournament selection,* using mutation, but no crossover.

Parameters are *popsize, tournament size, mutation-rate*.

It can be applied to any problem; the details glossed over are all problem-specific.

A *steady state*, *mutation-only*, *replace-worst* EA with *tournament selection*

```
0. Initialise: generate a population of popsize random
      solutions, evaluate their fitnesses.

1.    Run Select to obtain a parent solution X.
2.    With probability mute_rate, mutate a copy of X to
      obtain a mutant M (otherwise M = X)
3.    Evaluate the fitness of M.
4.    Let W be the current worst in the population. If M is
      not less fit than W, then replace W with M. (otherwise
      do nothing)
5.    If a termination condition is met (e.g. we have done
      10,000 evals) then stop. Otherwise go to 2.

Select: randomly choose tsize individuals from the
      population. Let c be the one with best fitness (BTR);
      return X.
```

A *generational*, *elitist*, *crossover+mutation* EA with *Rank-Based selection*

0. Initialise: generate a population G of *popsize* random solutions, evaluate their fitnesses.

1. Run *Select 2*(*popsize – 1) times to obtain a collection I of 2*(popsize-1) parents.
2. Randomly pair up the parents in I (into popsize – 1 pairs) and apply *Vary* to produce a child from each pair. Let the set of children be C.
3. Evaluate the fitness of each child.
4. Keep the best in the population G and delete the rest.
5. Add all the children to G.
6. If a termination condition is met (e.g. we have done 100 or more generations (runs through steps 1—5) then stop. Otherwise go to 1,

A *generational*, *elitist*, *crossover+mutation* EA with *Rank-Based selection*, continued …

Select: sort the contents of G from best to worst, assigning rank *popsize* to the best, *popsize*-1 to the next best, etc …, and rank 1 to the worst.

The ranks sum to F = popsize(popsize+1)/2

Associate a probability Rank_i/F with each individual *i.*

Using these probabilities, choose one individual X, and return X.

Vary:

1. With probability *cross_rate,* do a crossover:
   I.e produce a child by applying a crossover operator to the two parents. Otherwise, let the child be a randomly chosen one of the parents.
2. Apply mutation to the child.
3. Return the mutated child.

Back to Basics

- With your thirst for seeing example EAs temporarily quenched, the story now skips to simpler algorithms.
- This will help to explain what it is about the previous ones which make them work.

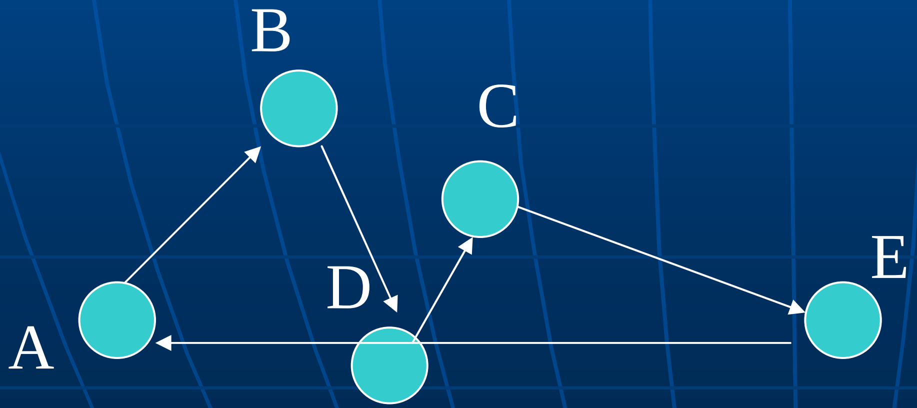
The Travelling Salesperson Problem

An example (hard) problem, for illustration

The Travelling Salesperson Problem
Find the shortest tour through the cities.
You must:

- Visit every city on the 'map'
- Return to where you started

The one below is length: 33



	A	B	C	D	E
A		5	7	4	15
B	5		3	4	10
C	7	3		2	7
D	4	4	2		9
E	15	10	7	9	

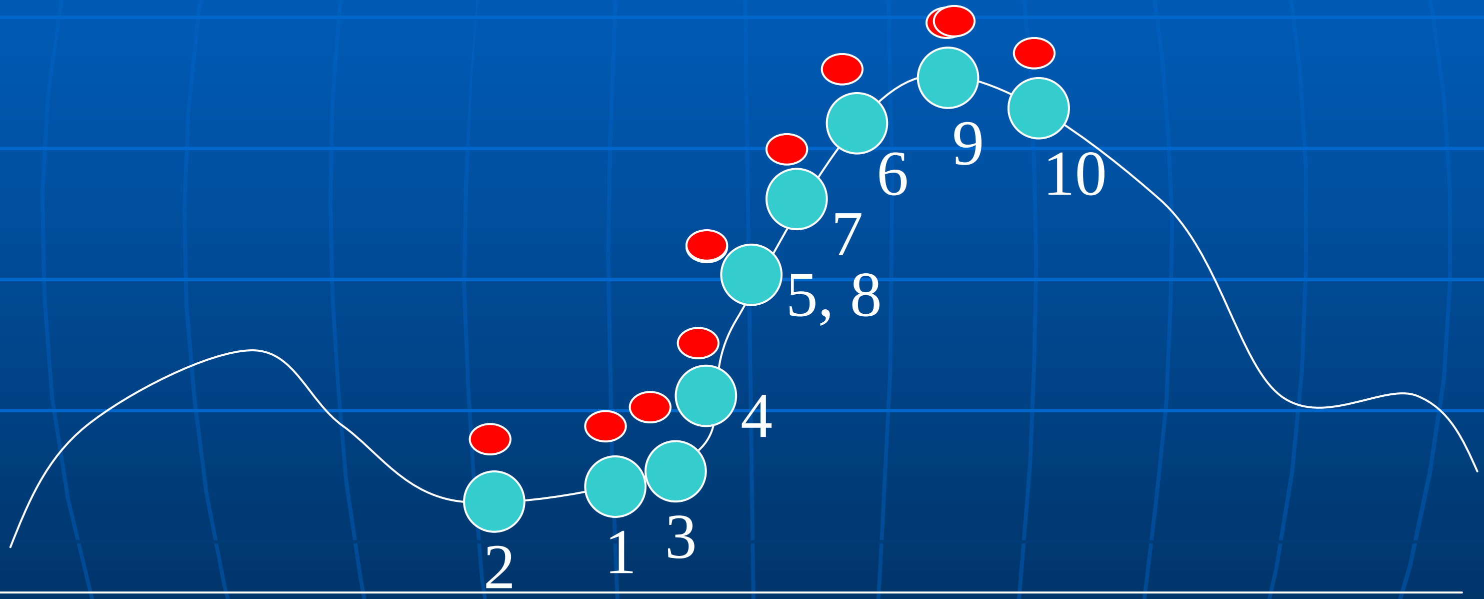
Hillclimbing

```
0.  Initialise:   Generate a random solution *c*;  evaluate its
    fitness, *f*(c).  Call *c* the *current solution.*

1. Mutate a copy of the current solution – call the mutant *m*
   Evaluate fitness of *m*, f(m).

2. If *f(m)* is no worse than *f*(c), then replace *c* with *m*,
   otherwise do nothing (effectively discarding *m*).

3. If a termination condition has been reached, stop.
   Otherwise, go to 1.
```

Note. No population (well, population of 1). This is a very simple version of an EA, although it has been around for much longer.

Why “Hillclimbing”?

Suppose that solutions are lined up along the *x* axis, and that *mutation* always gives you a nearby solutions. Fitness is on the *y* axis; this is a ***landscape***



1. Initial solution; 2. rejected mutant; 3. new current solution,
4. New current solution; 5. new current solution; 6. new current soln
7. Rejected mutant; 8. rejected mutant; 9. new current solution,
10. Rejected mutant, …

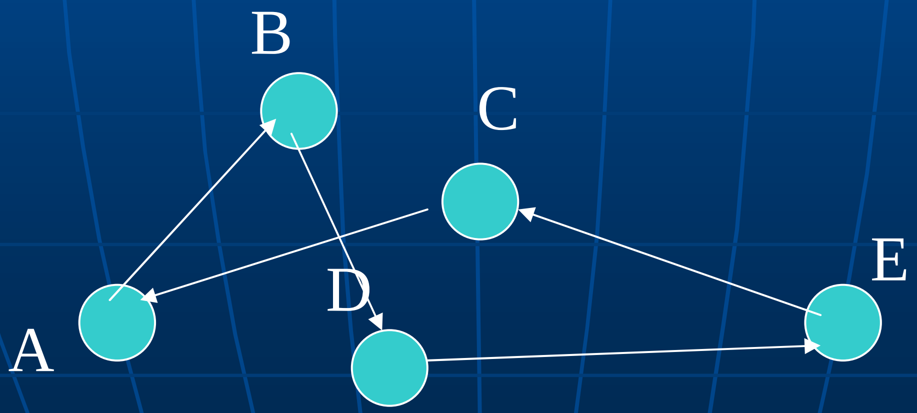
Example: HC on the TSP

We can encode a candidate solution to the TSP as a permutation

Here is our initial random solution ABDEC with fitness 32

	A	B	C	D	E
A		5	7	4	15
B	5		3	4	10
C	7	3		2	7
D	4	4	2		9
E	15	10	7	9	

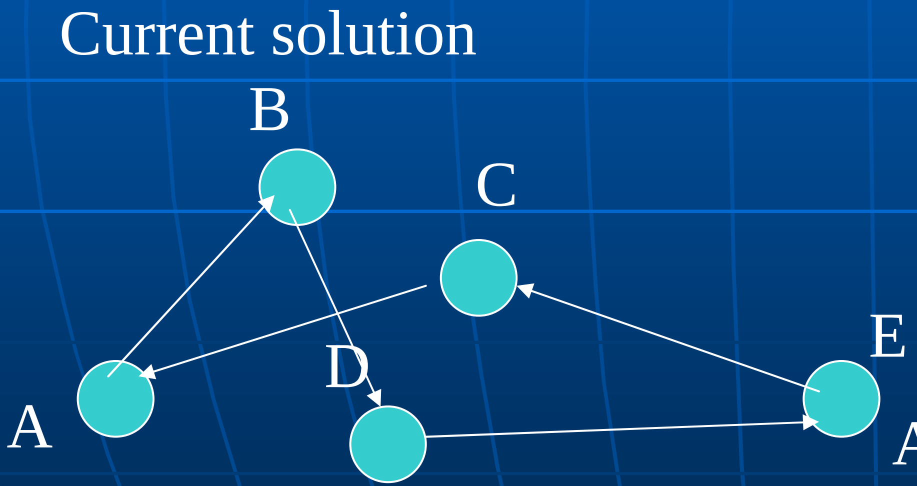
Current solution

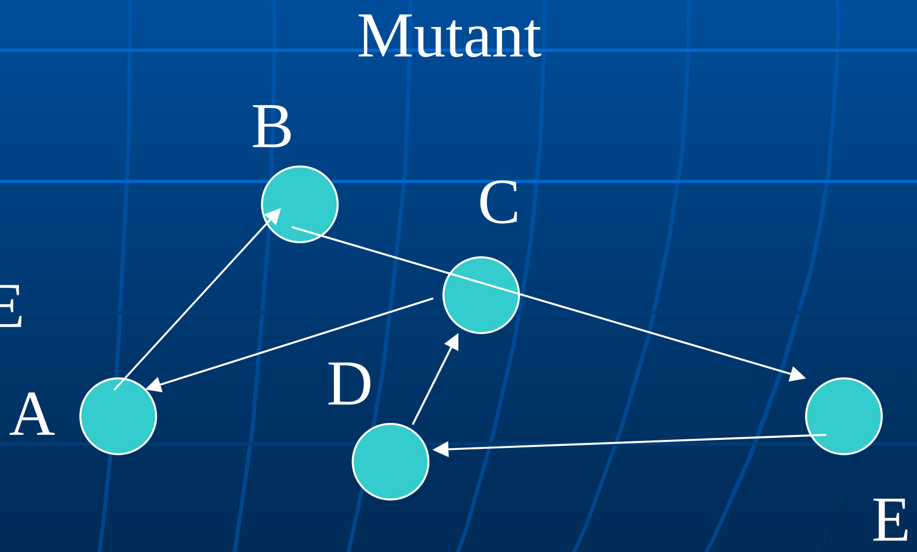


HC on the TSP

We randomly mutate (swap randomly chosen adjacent nodes) current to: ABEDC which has fitness 33 -- so current stays the same

	A	B	C	D	E
A		5	7	4	15
B	5		3	4	10
C	7	3		2	7
D	4	4	2		9
E	15	10	7	9	



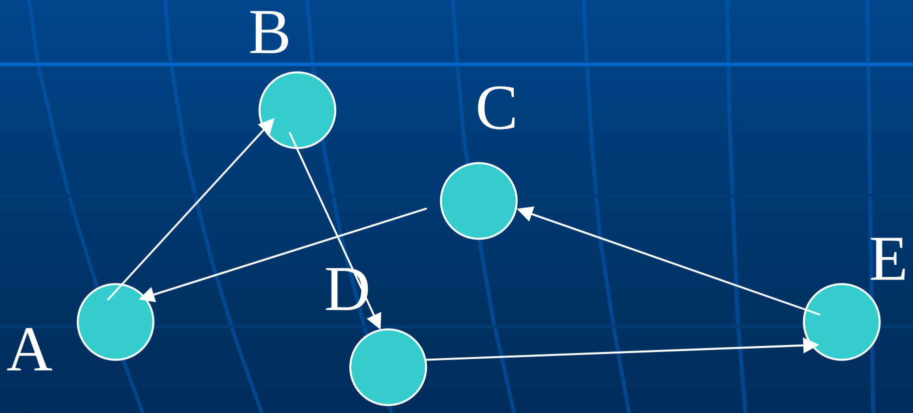


HC on the TSP

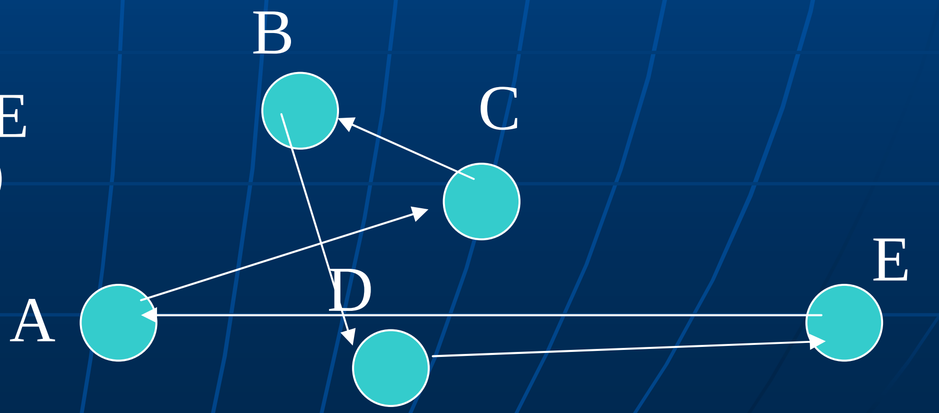
We randomly mutate (swap randomly chosen adjacent nodes) current (ABDEC) to CBDEA which has fitness 38 -- so current stays the same

	A	B	C	D	E
A		5	7	4	15
B	5		3	4	10
C	7	3		2	7
D	4	4	2		9
E	15	10	7	9	

Current solution



Mutant



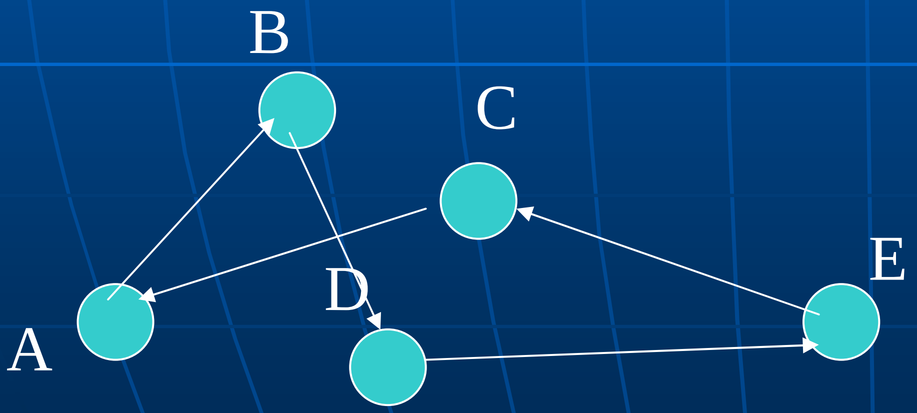
HC on the TSP

We randomly mutate (swap randomly chosen adjacent nodes) current (ABDEC) to BADEC which has fitness 28

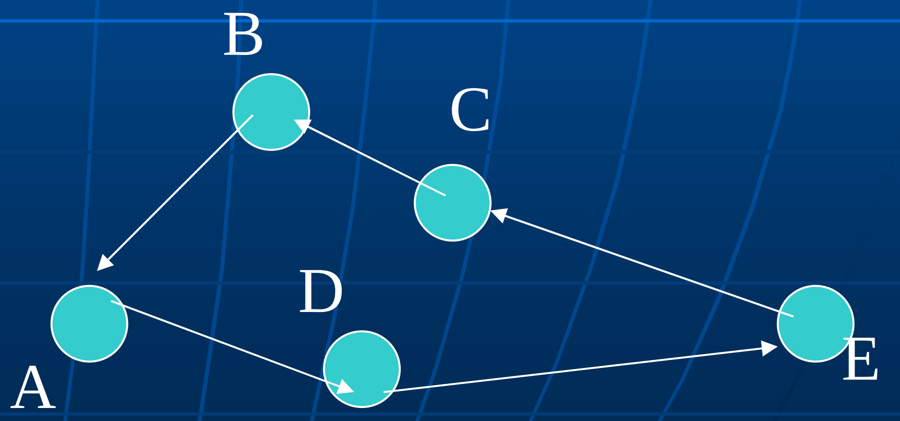
So this becomes the new current solution

	A	B	C	D	E
A		5	7	4	15
B	5		3	4	10
C	7	3		2	7
D	4	4	2		9
E	15	10	7	9	

Current solution



Mutant



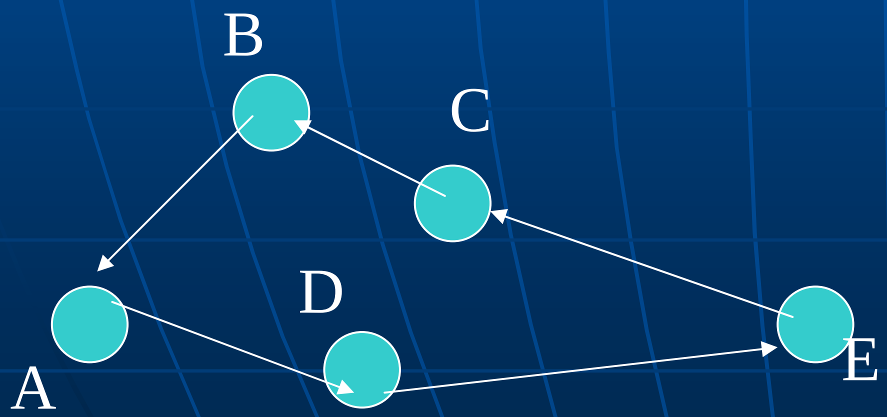
HC on the TSP

We randomly mutate (swap randomly chosen adjacent nodes) current (BADEC) to BADCE which also has fitness 28

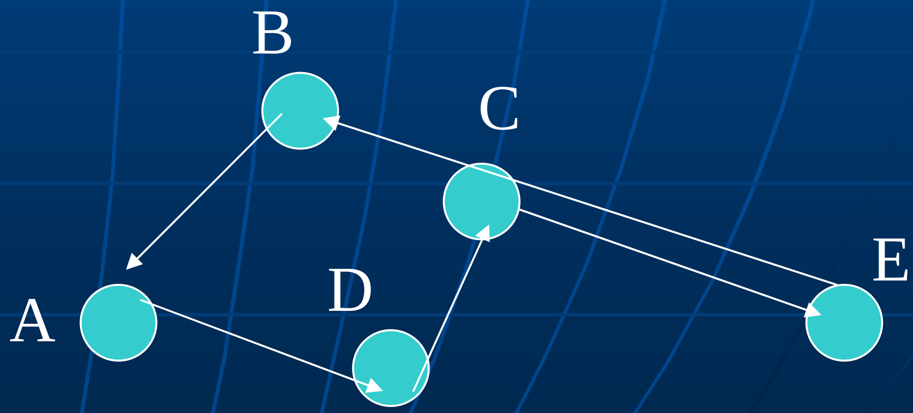
This becomes the new current solution

	A	B	C	D	E
A		5	7	4	15
B	5		3	4	10
C	7	3		2	7
D	4	4	2		9
E	15	10	7	9	

Current Solution

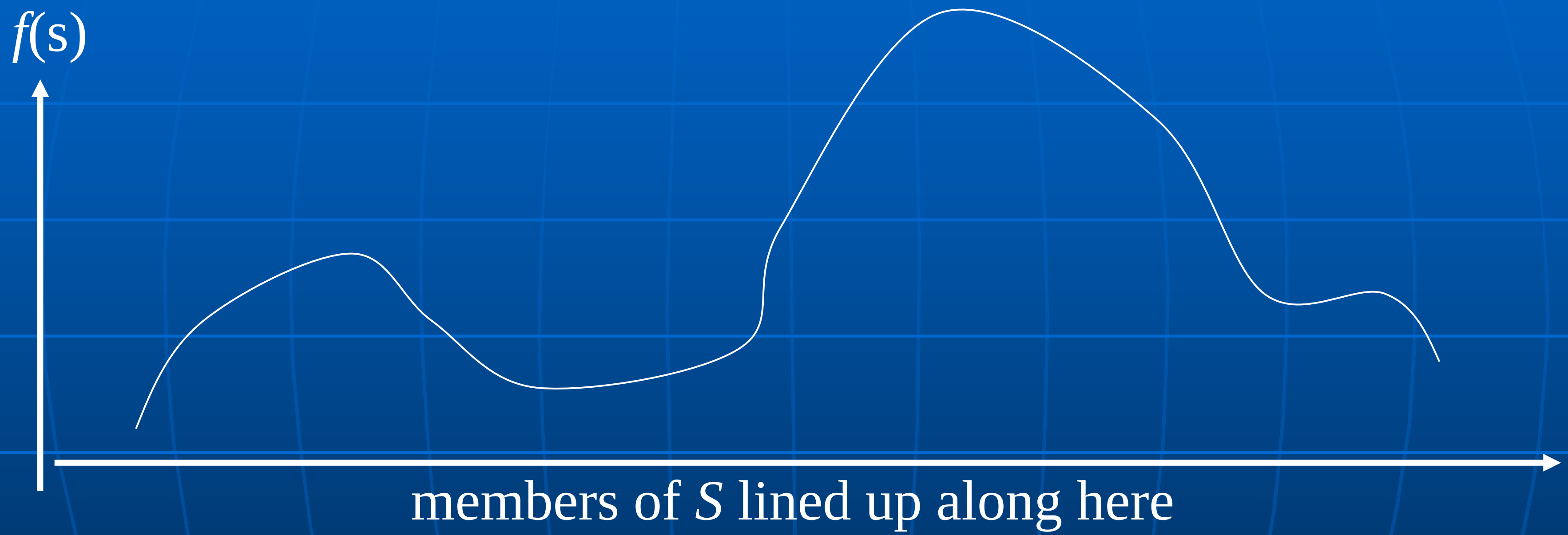


Mutant



Landscapes

Recall *S*, the search space, and *f*(s), the fitness of a candidate in *S*



The structure we get by imposing f(s) on S is called a *landscape*

What does the landscape look like if *f*(s) is a random number generator?
What kinds of problems would have very smooth landscapes?
What is the importance of the mutation operator in all this?

Neighbourhoods

Let *s* be an individual in *S*, *f*(s) is our fitness function, and *M* is our mutation operator, so that *M*(*s*1) → *s*2, where s2 is a mutant of s1.

Given *M*, we can usually work out the *neighbourhood* of an individual point *s* – the neighbourhood of *s* is the set of all possible mutants of *s*

E.g. **Encoding:** permutations of *k* objects (e.g. for *k*-city TSP)
Mutation: swap any adjacent pair of objects.
Neighbourhood: Each individual has *k* neighbours. E.g. neighbours of EABDC are: {**AE**BDC, E**BA**DC, EA**DB**C, EAB**CD**, **C**ABD**E**}
Encoding: binary strings of length *L* (e.g. for *L*-item bin-packing)
Mutation: choose a bit randomly and flip it.
Neighbourhood: Each individual has *L* neighbours. E.g. neighbours of 00110 are: {10110, 01110, 00010, 00100, 00111}

Landscape Topology

- Mutation operators lead to slight changes in the solution, which tend to lead to slight changes in fitness.

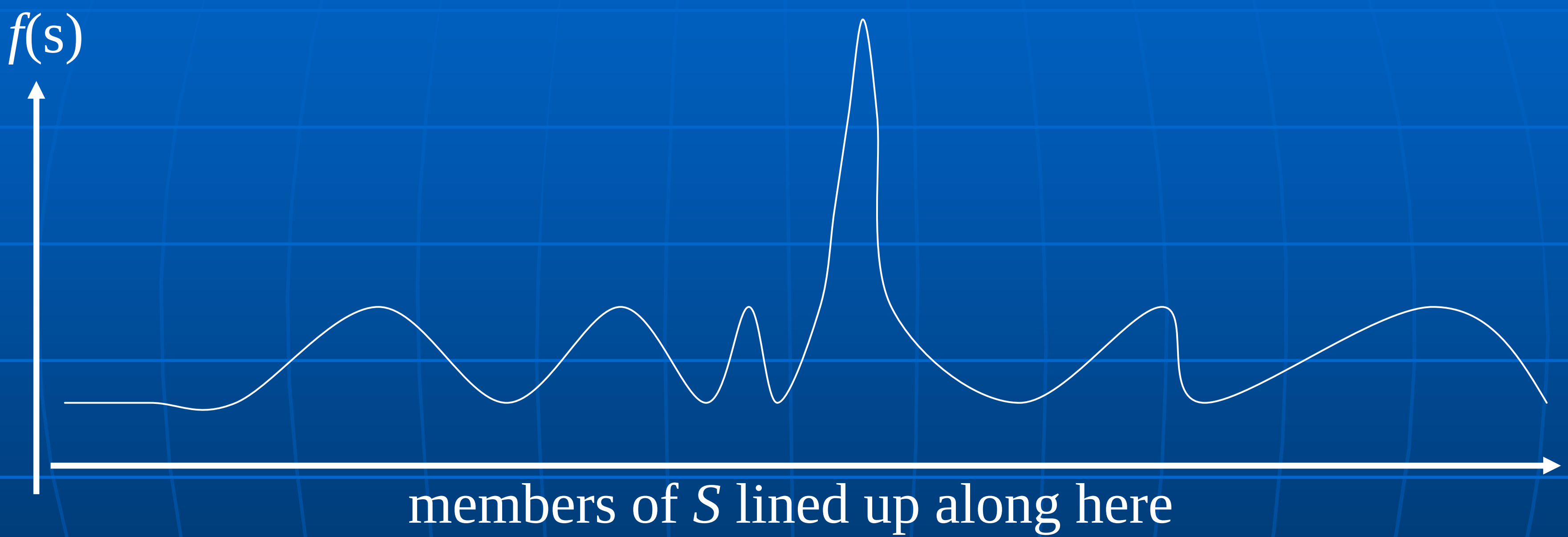
I.e. the fitnesses in the neighbourhood of *s* are often similar to the fitness of *s*.

- Landscapes tend to be *locally smooth*

What about big mutations ??

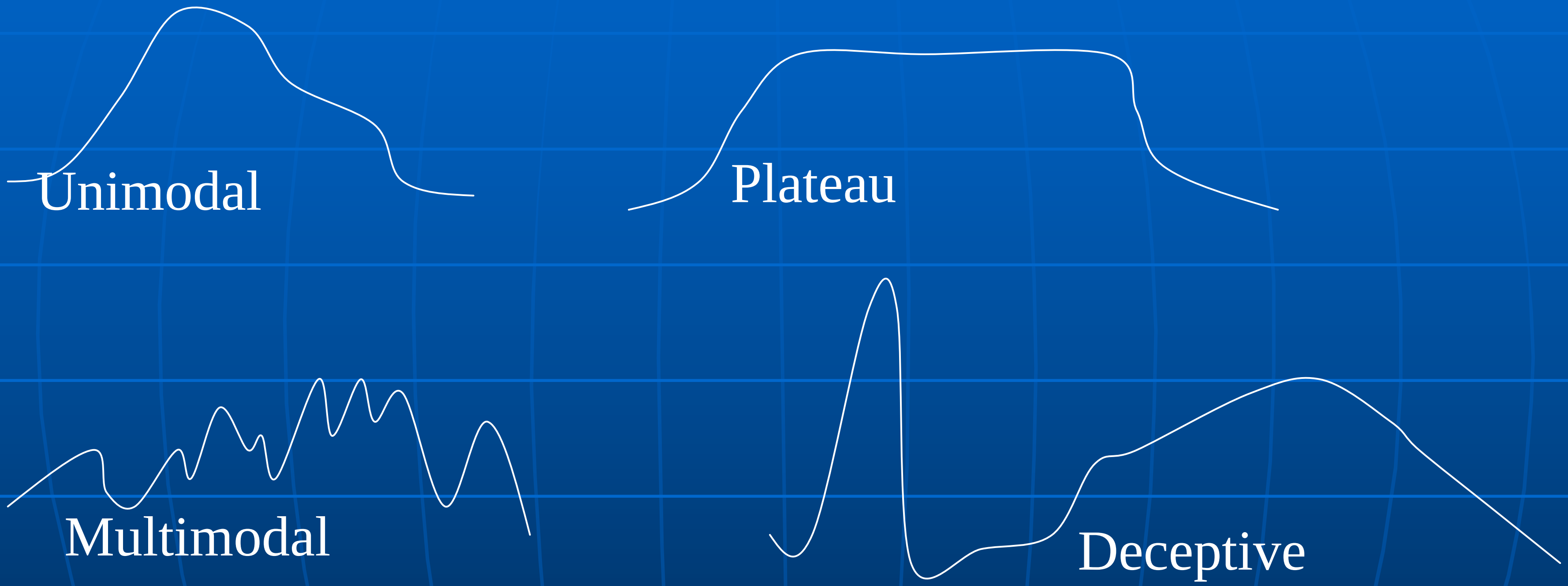
It turns out that ….

Typical Landscapes



Typically, with large (realistic) problems, the huge majority of the landscape has very poor fitness – there are tiny areas where the decent solutions lurk.
So, big random changes are very likely to take us outside the nice areas.

Typical Landscapes II



As we home in on the good areas, we can identify broad types of Landscape feature.

Most landscapes of interest are predominantly multimodal.

Despite being locally smooth, they are globally rugged

Beyond Hillclimbing

HC clearly has problems with typical landscapes:

There are two broad ways to improve HC, from the algorithm viewpoint:

1. Allow downhill moves – a family of methods called Local Search does this in various ways.
2. Have a population – so that different regions can be explored inherently in parallel – I.e. we keep `poor' solutions around and give them a chance to `develop'.

Local Search

```
Initialise:    Generate a random solution c;  evaluate its
               fitness, f(s) = b;  call c the current solution,
               and call b the best so far.

Repeat until termination conditon reached:
      1.  Search the neighbourhood of c, and choose one, m
           Evaluate fitness of m, call that x.
      2. According to some policy, maybe replace c with x, and
         update c and b as appropriate.
```

E.g. Monte Carlo search: 1. same as hillclimbing; 2. If *x* is better, accept it as new current solution;if *x* is worse, accept it with some probability (e.g. 0.1).

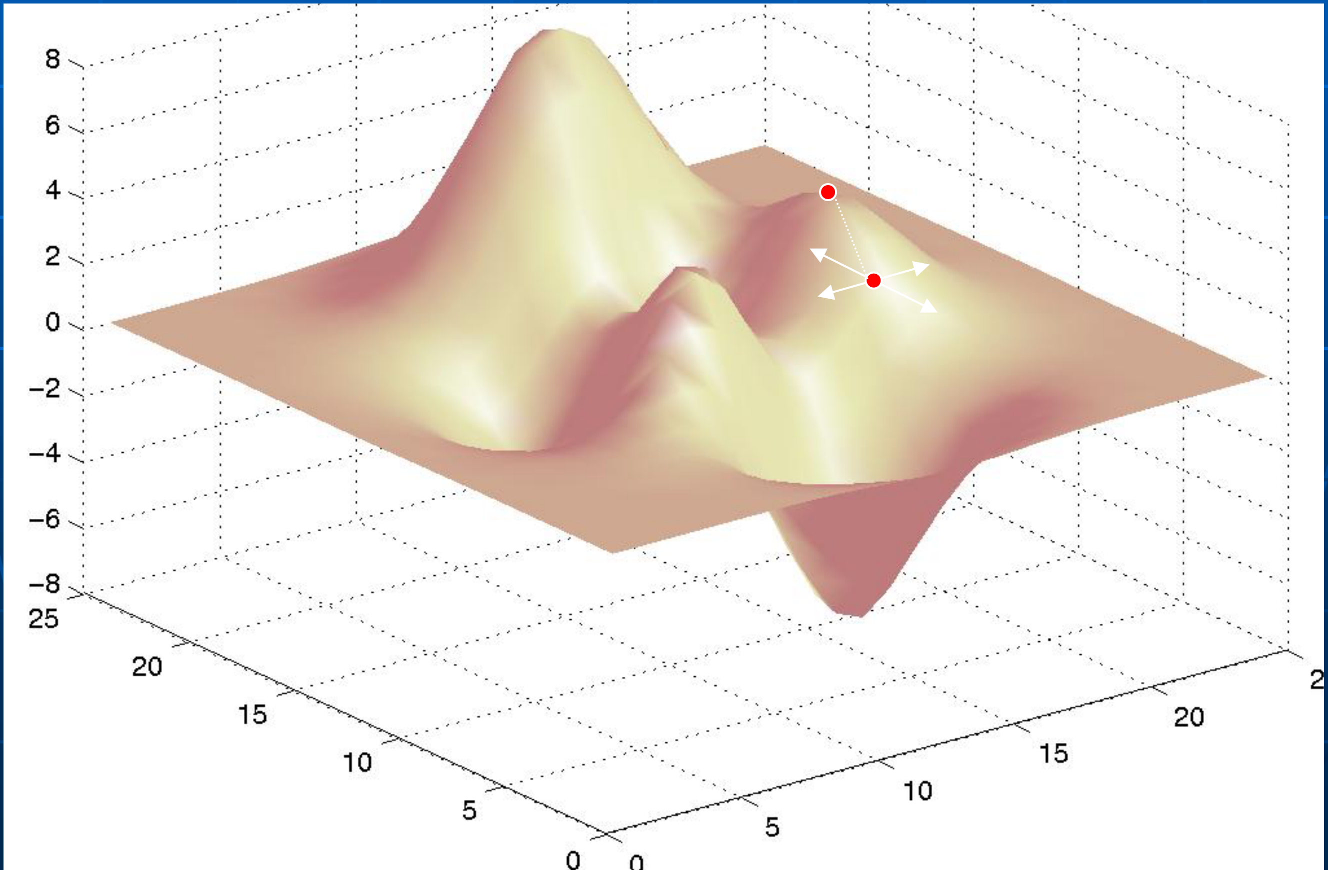
E.g. tabu search: 1. evaluate *all* immediate neighbours of *c*
 2. choose the best from (1) to be the next current solution, unless it is `tabu' (recently visited), in which choose the next best, etc.

Population-Based Search

- Local search is fine, but tends to get stuck in *local optima,* less so than HC, but it still gets stuck.
- In PBS, we no longer have a single `current solution', we now have a *population* of them. This leads directly to the two main algorithmic differences between PBS and LS
  - Which of the set of current solutions do we mutate? We need a *selection* method
  - With more than one solution available, we needn't just mutate, we can [mate, recombine, crossover, etc ...] two or more current solutions.
- So this is an alternative route towards motivating our nature-inspired EAs – and also starts to explain why they turn out to be so good.

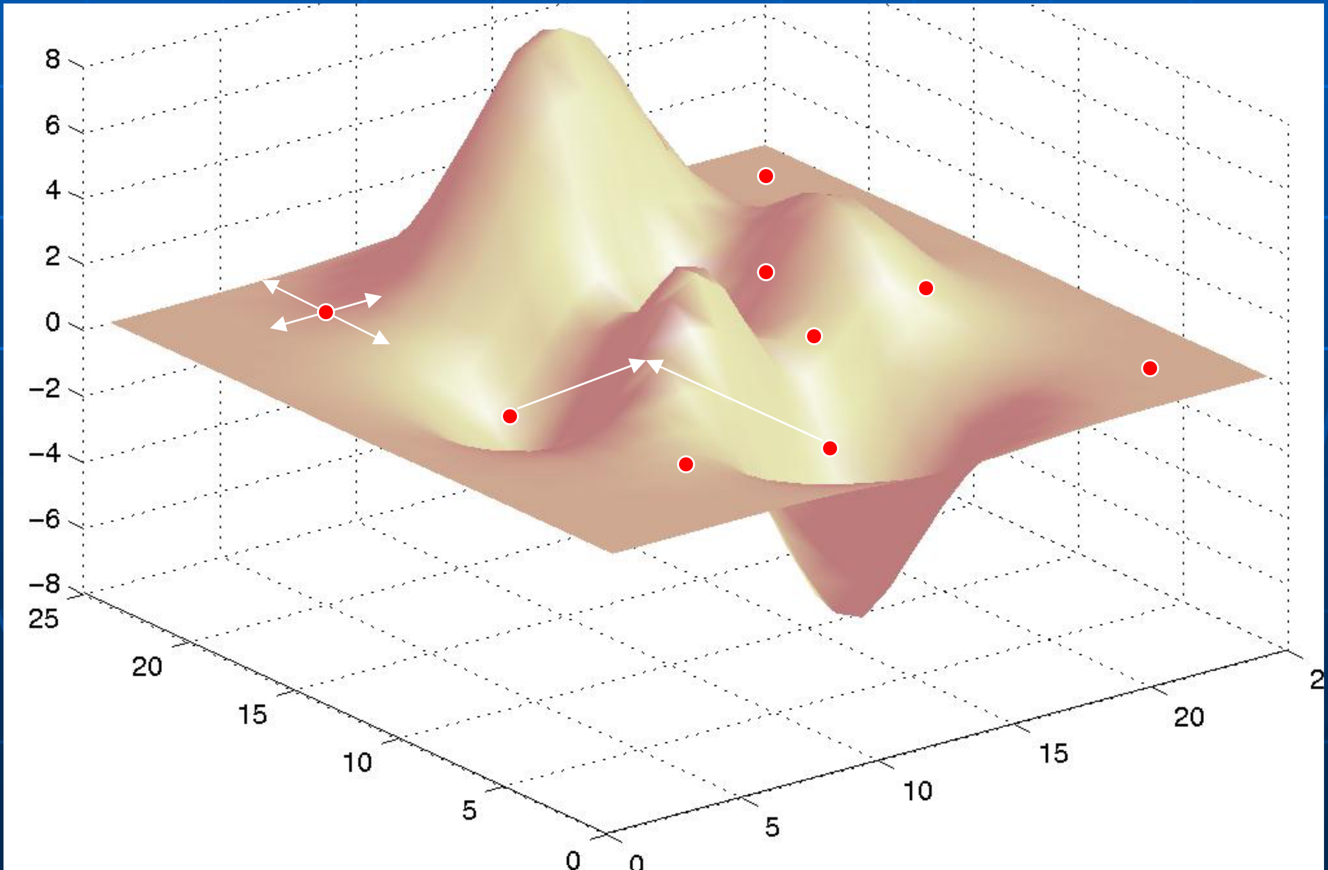
”Traditional” search/optimization algorithms and landscapes

- Gradient search and hill-climbing algorithms work by crawling up a fitness hill to the nearest local optimum – and getting stuck.



Evolutionary algorithms and landscapes

- EAs use populations of candidate solutions
- ”Mutations” correspond to local moves in the fitness landscape
- ”Crossovers” combine two candidate solutions



TSP, this time with an EA

- A steady state EA with mutation-only, running for a few steps on the TSP example, with an unidentified selection method.

Running a Steady State EA -- TSP

	A	B	C	D	E
A		5	7	4	15
B	5		3	4	10
C	7	3		2	7
D	4	4	2		9
E	15	10	7	9	

Let’s encode a solution as a permutation

Initial randomly generated pop of 5:

	ACEBD	DACBE	BACED	CDAEB	ABCED
Evaluation	32	33	32	31	28

Mutant of selected parent CDAEB → ADCEB

Evaluation of mutant: 26

Mutant enters population, replacing worst

Running a Steady State EA -- TSP

	A	B	C	D	E
A		5	7	4	15
B	5		3	4	10
C	7	3		2	7
D	4	4	2		9
E	15	10	7	9	

Generation 2

	ACEBD	ADCEB	BACED	CDAEB	ABCED
Evaluation	32	26	32	31	28

Mutant of selected parent BACED → BDCEA

Evaluation of mutant: 33

Mutant discarded– worse than current worst

Running a Steady State EA -- TSP

	A	B	C	D	E
A		5	7	4	15
B	5		3	4	10
C	7	3		2	7
D	4	4	2		9
E	15	10	7	9	

Generation 3:

	ACEBD	ADCEB	BACED	CDAEB	ABCED
Evaluation	32	26	32	31	28

Mutant of selected parent ABCED → ABECD

Evaluation of mutant: 28

Mutant enters population, replacing worst

Running a Steady State EA -- TSP

	A	B	C	D	E
A		5	7	4	15
B	5		3	4	10
C	7	3		2	7
D	4	4	2		9
E	15	10	7	9	

Generation 4:

	ABECD	ADCEB	BACED	CDAEB	ABCED
Evaluation	28	26	32	31	28

Mutant of selected parent ADCEB → BDCEA

Evaluation of mutant: 33

Mutant is discarded

Running a Steady State EA -- TSP

	A	B	C	D	E
A		5	7	4	15
B	5		3	4	10
C	7	3		2	7
D	4	4	2		9
E	15	10	7	9	

Generation 5:

	ABECD	ADCEB	BACED	CDAEB	ABCED
Evaluation	28	26	32	31	28

Mutant of selected parent ABCED → ABECD

Evaluation of mutant: 28

Mutant enters population, replacing worst

Running a Steady State EA -- TSP

	A	B	C	D	E
A		5	7	4	15
B	5		3	4	10
C	7	3		2	7
D	4	4	2		9
E	15	10	7	9	

Generation 6:

	ABECD	ADCEB	ABECD	CDAEB	ABCED
Evaluation	28	26	28	31	28

And so on.

Note: population starting to converge, genotypically and phenotypically