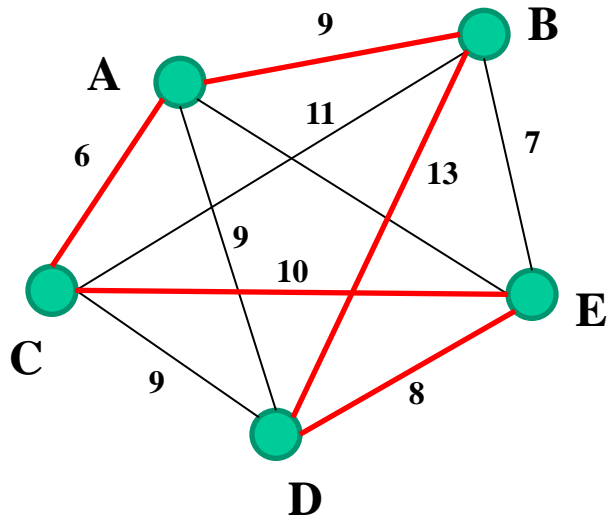


*Not all state space search problems are **routing** and **planning***

- for problems like the missionaries-cannibals, water-jugs, n-puzzle, etc., the problem is to *construct a path from an initial state to a goal state, in a large state space*
- but, there are many other kinds of state space search problem
- sometimes, a path is not the interest, but *rather a state that is good (or best) in some sense*
- I will give a couple of examples, to introduce the idea of “**local search**” in a state space

TSP = Traveling salesman problem

- you probably have seen this problem before, which may be stated simply as follows
- there are N cities where each pair of cities is connected by a road of a certain length [so the cities form *a complete undirected graph*]
- a *tour* is meant to be a route starting with a given city, say A , passing through each of the *other cities exactly once*, and returning to A [tour = *a closed path including each city exactly once*]



- so, a tour is just a **permutation** of the cities, such as

A-C-E-D-B

- from a practical point of view, it may be useful to think of a tour as something like

A-C-E-D-B-A

to make sure *the path is seen as closing at the starting point*

- for the above example, A-C-E-D-B-A is a tour of length 46

- then,

**TSP or Traveling Salesman Problem is the problem of
finding a tour of the smallest length**

- there is an obvious algorithm:
 1. list all tours along with their lengths
 2. see which has the smallest length
- however, since each tour is basically a permutation, if there are N cities, then there are **$N!$ (N factorial) possible tours**,
as per finite/discrete math

- you may remember that $N!$ grows faster than exponentials with a fixed base, such as 2^N , 3^N , etc.
- see next slide, *in case* you have forgotten from “Data Structures”
- clearly, the above simple algorithm is not going to work except for small N [*not* enough space to store the tours, and *not* enough time to enumerate them]
- at this time, nobody knows of a single efficient (e.g. polynomial) algorithm applicable to any TSP, producing an exact answer

```
for i in range(0,31):
    print str(factorial(i)) + " " + str(pow(2,i))
```

i	factorial(i)	2 ⁱ
0	1	1
1	1	2
2	2	4
3	6	8
4	24	16
5	120	32
6	720	64
7	5040	128
8	40320	256
9	362880	512
10	3628800	1024
11	39916800	2048
12	479001600	4096
13	6227020800	8192
14	87178291200	16384
15	1307674368000	32768
16	20922789888000	65536
17	355687428096000	131072
18	6402373705728000	262144
19	121645100408832000	524288
20	2432902008176640000	1048576
21	51090942171709440000	2097152
22	1124000727777607680000	4194304
23	25852016738884976640000	8388608
24	620448401733239439360000	16777216
25	15511210043330985984000000	33554432
26	403291461126605635584000000	67108864
27	10888869450418352160768000000	134217728
28	304888344611713860501504000000	268435456
29	8841761993739701954543616000000	536870912
30	26525285981219105863630848000000	1073741824

N! compared to 2^N

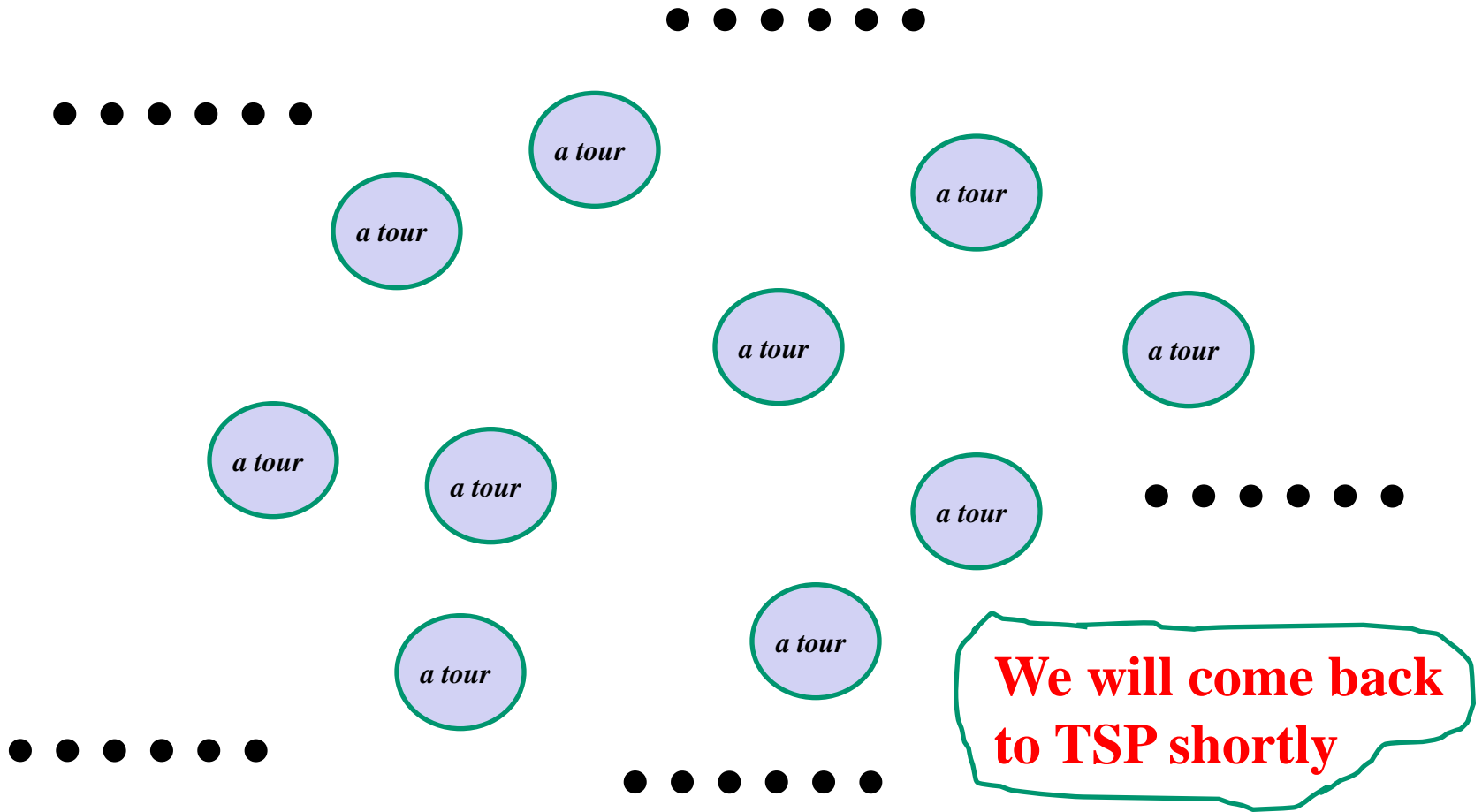
N! grows much faster than 2^N

- by the way, let us recall roughly why an algorithm is “exponential”
- if you recall your finite or discrete math, **the number of subsets of a set of N elements is 2^N** (exponential N base 2)
- hence, *if an algorithm has to consider all or almost all subsets of an set of N things, then it is an “exponential” algorithm*
- similar to where you have to **consider all or almost all permutations of a set of N things, which would be a factorial algorithm, worse than an exponential algorithm**

- there is a whole bunch of *theory* regarding TSP and similar problems, that I will not go into in this module
[TSP is said to be **NP-Hard**, which means that it is at least as hard as so-called NP-complete problems]
- but, ideas related to state space search have been attempted to deal with TSP, *not to find an exact solution*, but **approximate solutions that are good enough in practice**
- the main idea is to do **a local exploration of the state space**, since *considering the entire space is impossible anyway*

- for example,
think of a state as a potential solution to the problem
- starting with such a state, maybe chosen randomly to begin, consider its neighbor(s) to see if it (they) are an improvement
- keep trying this attempt to improve from neighbor to neighbor [until no improvement is possible, or some iteration limit has been reached]
- *see why is it called **local search**?*

- in TSP, a state is then a tour whose neighbor(s) is (are) a slight variation(s) of the tour [e.g. switch 2 or 3 cities]



Knapsack problem

- there are many variations of this problem, so the one below is just one example
- there are N objects to be carried in a knapsack;
each object has a *weight* w_i and a *value* v_i
the knapsack has a *maximum weight it can carry* (not to exceed)
- KP or Knapsack Problem is to select a subset of the objects to carry whose total value is as high as possible

- for example, suppose there are 5 objects A, B, C, D, E with respective weights 2, 4, 5, 8 9 and respective values 25, 12, 16, 26, 15:

objects:	A	B	C	D	E
weights:	2	4	5	8	9
values:	25	12	16	26	15

assume that the maximum weight for the knapsack to carry is **15**

- A-D-E is not a valid choice, although it has a very high value;
A-B-D is a valid choice with value 63;
A-C-D is a valid choice with a better value 67

- a trivial algorithm for KP is:
 1. go through all possible subsets, eliminating those whose total weight exceeds the max, recording also the total value of each kept subset
 2. see which has the highest total value
- as I recalled on a previous slide, **this is exponential**, since **it considers all the subsets of a set**
- just as in TSP, we might try *a local exploration* of the state space [as stated above, KP is also **NP-Hard**]

- **a state in KP, for local exploration, is a potential solution, that is a subset of objects, usually encoded as bit pattern**
- **for this previous example:**

objects:	A	B	C	D	E
weights:	2	4	5	8	9
values:	25	12	16	26	15

a valid state is 01100 (the subset BC) whose value is 28;

another valid state is 11010 (the subset ABD) whose value is 63;

another valid state is 10110 (the subset ACD) whose value is 67

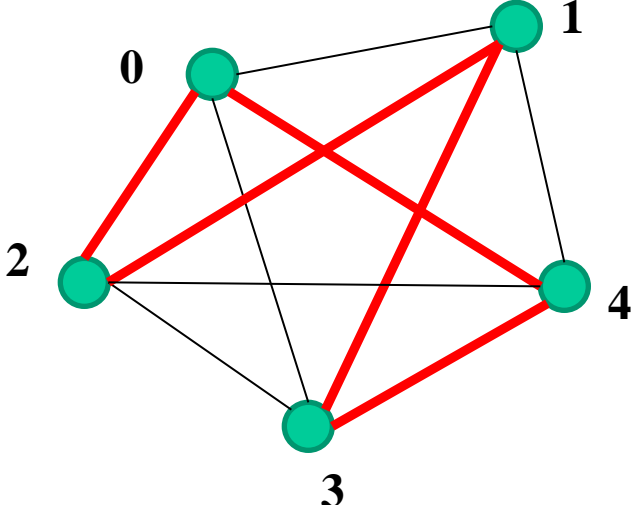
- ***see it?***

Local search algorithms in general

- local search algorithms are not guaranteed to produce exact solutions all the time
- **chapter 4 of the textbook** explains some of the reasons, which I will *not* repeat here
- and there are many of these local algorithms around, and many that are still being researched
- for programming in Python, we first need to say more about **lists in Python**

Manipulating Python lists properly

- as you know already, “lists” of things are *the most fundamental data structure* used in programming
(arrays are just a particular case of lists)
- *much* of what you know about *arrays* from languages like C, C++ and Java applies
- but there are a few things to **pay attention to as you deal with Python**

- recall that **in Python, a literal list consists of values separated by commas, enclosed in square brackets**
[and as you are familiar with, **indexing is 0-based**]
- so, in TSP with 5 cities [5 cities is *too small*, but I use it to explain ideas], I think of a state [a possible tour] as something like
[0, 4, 3, 1, 2, 0]

- I number the cities from 0, and I may assume that the tour begins and ends with city 0 [it must contain 0 and it is closed]

- when setting an initial state then, I *might* say

.... **initial_state=[0, 4, 3, 1, 2, 0]**

- now, in TSP, you must give the *distances between all pairs of cities*; abstractly, it is then a **2D matrix** like

	0	1	2	3	4
0	0	14	15	5	12
1	14	0	9	8	3
2	15	9	0	6	8
3	5	8	6	0	4
4	12	3	8	4	0

In TSP, such a distance matrix must be symmetric around the main diagonal and must have all 0 on the main diagonal

Right?

- now, just as with C and Java, such a 2D matrix is really **a list of lists (a list of rows)** in Python
- so, you might somewhere write something like

```
self.distances = \  
    [ [ 0, 14, 15, 5, 12 ], \  
      [ 14, 0, 9, 8, 3 ], \  
      [ 15, 9, 0, 6, 8 ], \  
      [ 5, 8, 6, 0, 4 ], \  
      [ 12, 3, 8, 4, 0 ] ]
```

- ***get all that?*** Pay attention to all these *brackets, commas and backslashes*

- as you know already, there are zillions of things we do with lists (and arrays)
- for example, we might **slice** them, extracting a single entry, or a bunch of consecutive entries, etc.
- in Python, if s is a one-dimensional list, you use $s[k]$ for the entry at index k , remembering that **indexing is 0-based**
- in Python,

Pay attention to that $m - 1$;
similar to last index being $N - 1$
if the list is of length N

$s[k:m]$ = sublist from index to index $m-1$

$s[:m]$ = sublist from the beginning to index $m-1$

$s[k:]$ = sublist from k to the end

See next slide

```
C:\Windows\system32\cmd.exe - python

C:\>python
Python 2.7.6 <default, Nov 10 2013, 19:24:18> [MSC v.1500 32 bit <Intel>] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
>>>
>>> x = [1, 5, 8, 2, 5, 9, 2, 6, 2, 7, 0]
>>>
>>> x
[1, 5, 8, 2, 5, 9, 2, 6, 2, 7, 0]
>>>
>>> x[2:6]
[8, 2, 5, 9]
>>>
>>> x[7:]
[6, 2, 7, 0]
>>>
>>> x[:5]
[1, 5, 8, 2, 5]
>>>
>>> x[2:47]
[8, 2, 5, 9, 2, 6, 2, 7, 0]
>>>
>>>
```

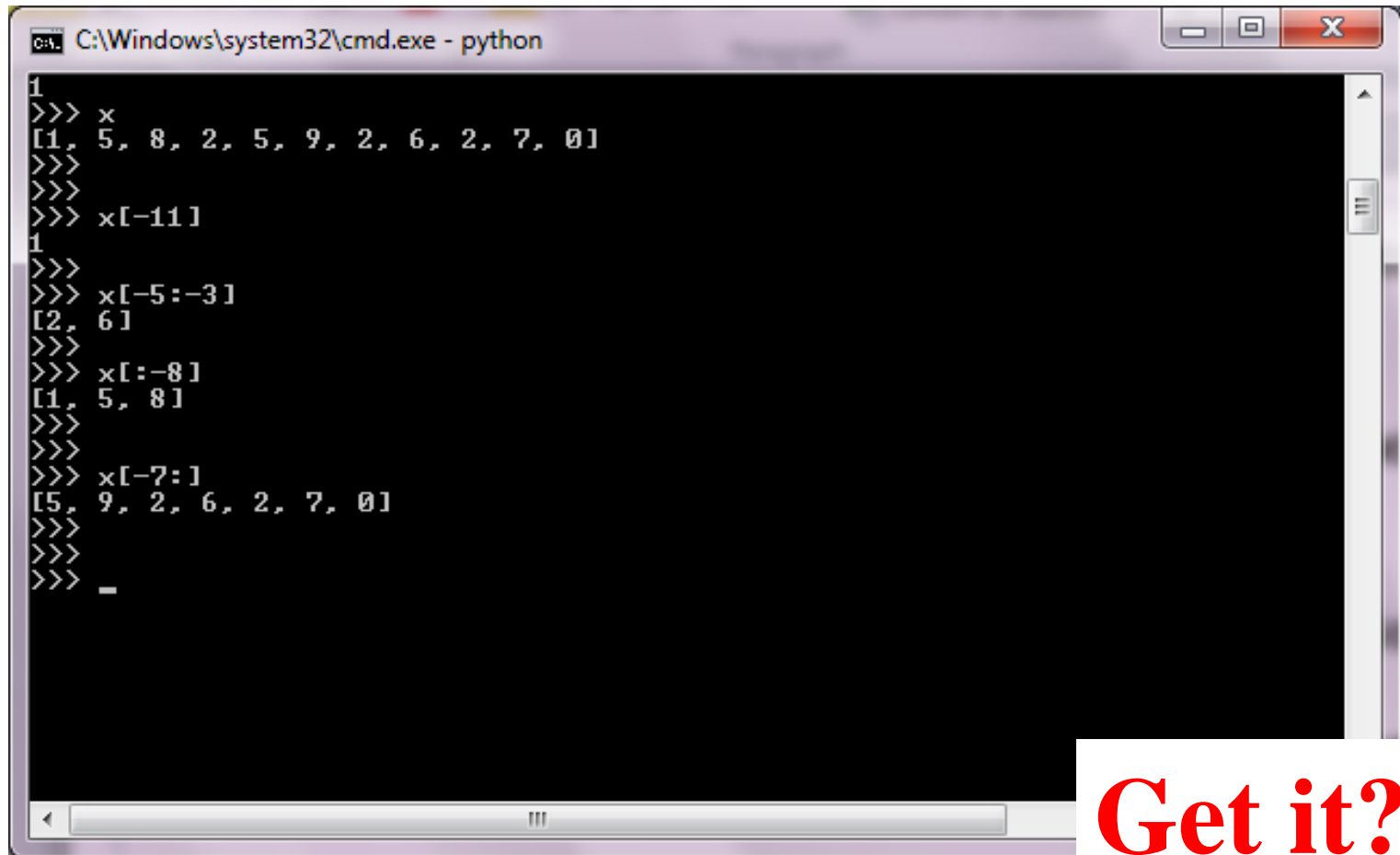
Get it?

- but here is something special about Python
- you **may use negative indices** to refer to the entries in a list
- the first entry of a list x , usually referred to as $x[0]$ may also be referred to as $x[-N]$, where N is the length of the list
- so the indices are as in this example x :

	<i>0</i>	<i>1</i>	<i>2</i>	<i>3</i>	<i>4</i>	<i>5</i>	<i>6</i>	<i>7</i>	← indices
x	781	546	112	690	277	321	972	774	← entries
	-8	-7	-6	-5	-4	-3	-2	-1	← indices

so, $x[3] = 690$, $x[-2] = 972$, etc.

- *Get it?*



```
C:\Windows\system32\cmd.exe - python
1
>>> x
[1, 5, 8, 2, 5, 9, 2, 6, 2, 7, 0]
>>>
>>>
>>> x[-11]
1
>>>
>>> x[-5:-3]
[2, 6]
>>>
>>> x[:-8]
[1, 5, 8]
>>>
>>>
>>> x[-7:]
[5, 9, 2, 6, 2, 7, 0]
>>>
>>>
>>> _
```

Get it?

- often we need to **reverse a list** (possibly along with slicing);
you may do it by writing a loop yourself
- but, you may also use the **reversed()** method, which does not
give you a list, but an iterator
- you may call **list()** on an iterator to produce the actual
reversed list
- see next slide, and remember it

```
cmd C:\Windows\system32\cmd.exe - python
>>>
>>>
>>> x
[1, 5, 8, 2, 5, 9, 2, 6, 2, 7, 0]
>>>
>>> reversed(x)
<listreverseiterator object at 0x01E900F0>
>>>
>>> y = list(reversed(x))
>>> y
[0, 7, 2, 6, 2, 9, 5, 2, 8, 5, 1]
>>>
>>>
>>>
```

Get it?

- by the way, **lists may be concatenated with the + operator**, similar to concatenating strings, as with Java
- in TSP, we might need to make a **random permutation** of a tour (or subset thereof)
- a random permutation of a list (or sub-list) **x** may be gotten through the **random.sample(x, len(x))** in the random module
- see next slide

```
C:\Windows\system32\cmd.exe - python
C:\>python
Python 2.7.6 <default, Nov 10 2013, 19:24:18> [MSC v.1500 32 bit (Intel)] on win
Type "help", "copyright", "credits" or "license" for more information.
>>>
>>> y = [ 2, 7, 1, 8 ]
>>> z = [ 9, 5, 6 ]
>>> x = y + z
>>> x
[2, 7, 1, 8, 9, 5, 6]
>>>
>>> import random
>>> r = random.sample(x, len(x))
>>> r
[2, 8, 1, 5, 9, 6, 7]
>>>
>>> s = random.sample(x, len(x))
>>> s
[6, 1, 9, 7, 5, 8, 2]
>>>
>>> tour = [1,2,3,4]
>>> a = [0] + random.sample(tour, len(tour)) + [0]
>>> a
[0, 1, 3, 4, 2, 0]
>>>
>>> b = [0] + random.sample(tour, len(tour)) + [0]
>>> b
[0, 3, 2, 4, 1, 0]
>>>
>>> _
```

Get it?

- entries from a **2D list** are referred to using a row index and a column index as usual, all 0-based indexing of course
- there are many things you can do with **2D lists**, of course
- I let you experiment yourself, and pay attention to your readings
- see next slide however

```
C:\Windows\system32\cmd.exe - python
>>>
>>> d = \
...  [ [ 0, 14, 15, 5, 12, 1, \
...    [ 14, 0, 9, 8, 3, 1, \
...    [ 15, 9, 0, 6, 8, 1, \
...    [ 5, 8, 6, 0, 4, 1, \
...    [ 12, 3, 8, 4, 0, 1 1
>>>
>>> d
[[0, 14, 15, 5, 12], [14, 0, 9, 8, 3], [15, 9, 0, 6, 8], [5, 8, 6, 0, 4], [12, 3, 8, 4, 0]]
>>>
>>>
>>> for i in range(0,5):
...     print d[i][i]
...
0
0
0
0
0
>>> for i in range(0,5):
...     for j in range(0,5):
...         if d[i][j] > 6: print d[i][j]
...
14
15
12
14
9
8
15
9
8
8
12
8
>>> _
```

Get it?

Local search in *simpleai*

- again, you need to read **chapter 4 of the textbook** for details on *local search algorithms*
- the first kind we deal with is the so-called “**hill-climbing**” (hc) algorithms
- in general,

hill-climbing (hc) associates a value (a score) to each state, where **higher value signifies a better state**

- then, at the current state, a neighboring(s) state is determined, and compared to see if it has a better value
- if so, the neighbor becomes the current state, and the entire scheme repeats; if not, you stop
- you may get stuck at a so-called *local maximum*, not finding the state with the overall maximum, so something like

hill climbing with random restarts

may be a good thing to try, where you randomly select more than one starting state, i.e. **exploring the state space at various random locations**

- now, what is called *neighbor(s)* depends on the problem and your design
- in **simpleai**, the source file **local.py** contains implementations of the algorithms in chapter 4 of the textbook
- you still need what is in **models.py**, because a problem is still *a state space problem* as in chapter 3 and assignment 3
- but there are *new functions* you have to deal with

- similar to what is done in **traditional.py**, most (not all) algorithms in **local.py** use a “private” method called ***_local_search()***
- I want you to read the “hill climbing” algorithms, especially ***hill_climbing()*** and ***hill_climbing_random_restarts()***, and that ***_local_search()*** method
- much of it is really the same as in previous write-ups, and as in *traditional.py*, so I will *not* repeat

Assignment 4

- the assignment is TSP with 12 cities using *hill climbing with random restarts*
- similar to assignment 3 but not exactly the same
- from *simpleai.search* you need to import *SearchProblem* and at least *hill_climbing_random_restarts*
- you also need to import *random* [the random module], as per the above slides

- you need to make a TspProblem class then, sub-classing SearchProblem
- the **constructor** should set *an initial state*;
note that here then, such state is something like
[0, 4, 3, 2, 1, 8, 7, 6, 5, 11, 10, 9, 0]
if you relate to what I said in previous slides
- you need instance variables like *self.numCities* (e.g.12 as value),
and a distance matrix *self.distances*, but make those as arguments of the constructor, as in assignment 3

- to emphasize, **the distances matrix must be a 12-by-12 symmetric matrix, with all zero on the main diagonal**
- **choose your own distance numbers [no need to use big numbers]**
- by the requirements of *hill_climbing_random_restarts()*, you need to write the instance methods

actions()

result()

value()

generate_random_state()

- I also want you to write a “private” method

def _tour_length(self, s)

that returns the length of the state s, which is a tour

- you need to use the *self.distances* matrix, of course

as always, be careful about the indices used in your looping

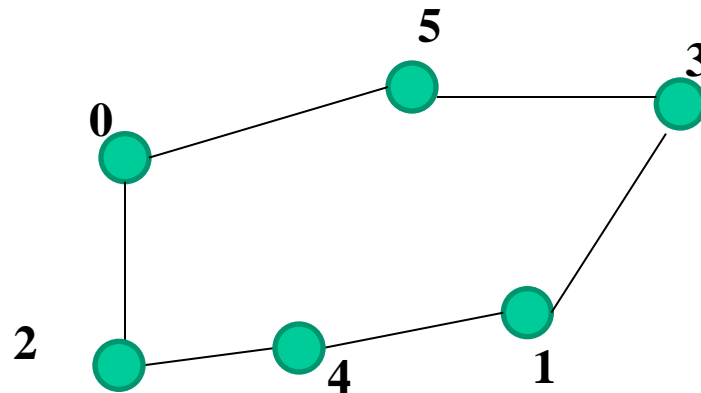
[you may also be able to use other schemes we have seen already, instead of straight looping]

- the **value()** method should be easy, using **_tour_length()**,
since the lower the tour length, the higher the value

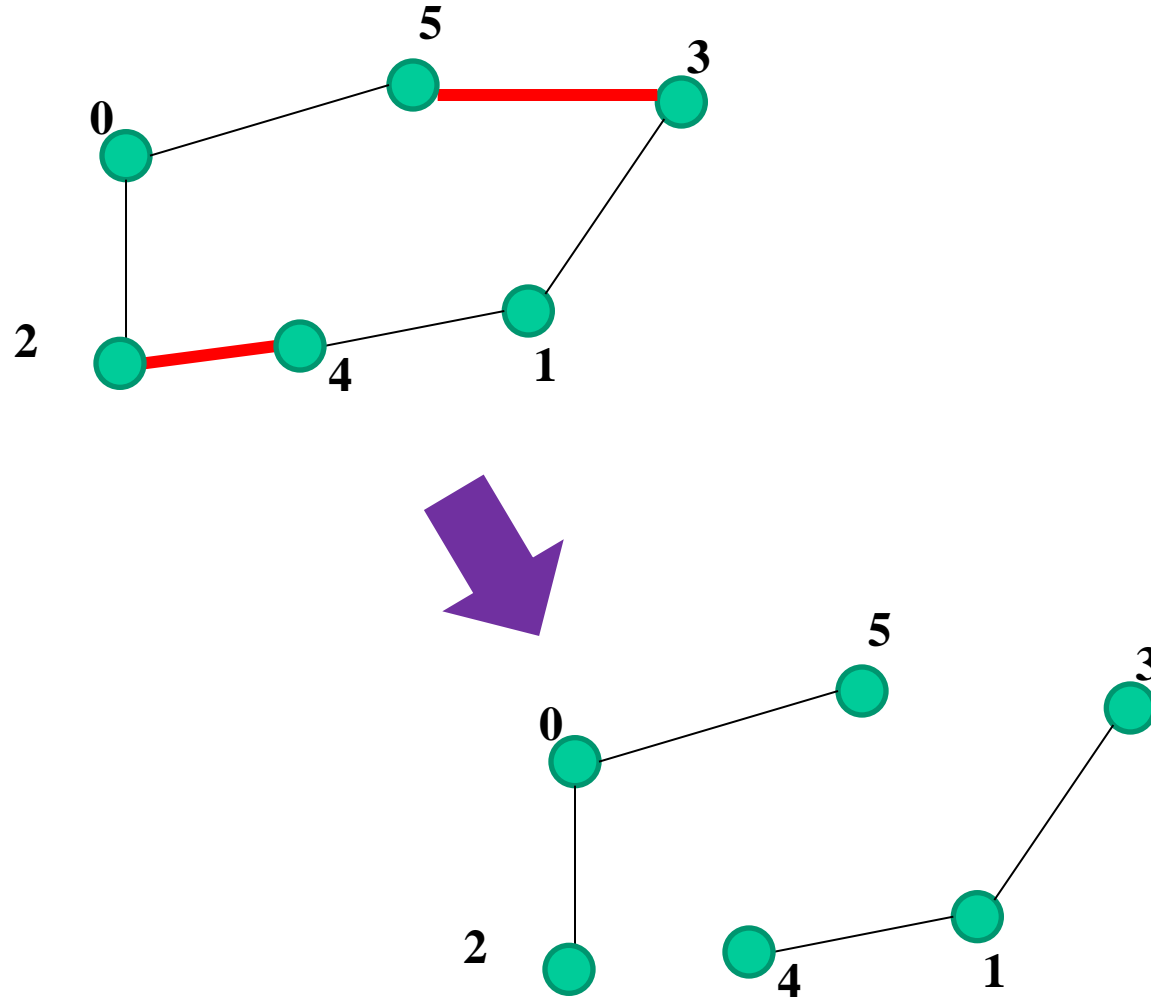
- next, for **def generate_random_state(self)** which must return a random state (a random tour)
- how can you generate a random tour?
- remember that you *keep each tour as starting from 0 and ending at 0*; so, you might randomly sample a list [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11] of the other cities, and concatenate the list [0] at the beginning and at the end
- I already mentioned **random.sample(x, len(x))** before, as well as **list concatenation**

- now, for **actions(self, s)**
- in assignment 3, and *traditional.py*, **actions()** generates all valid actions from state *s*
- but, we cannot do that here, because *we are not attempting to explore the entire state space*
- now, we simply generate one or a few states supposedly neighbor(s) of state *s*, and explore this (those)

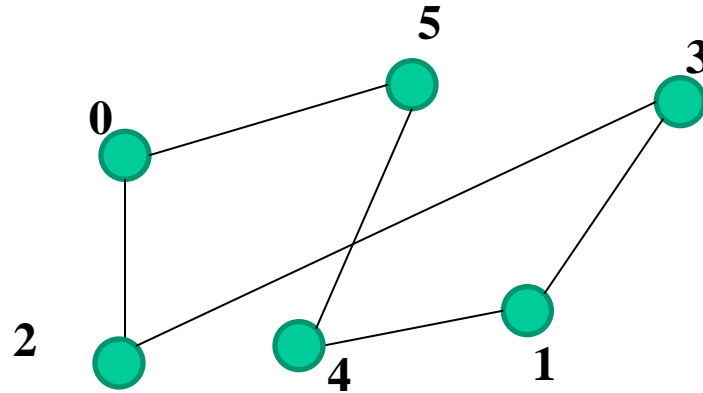
- there are many ways we might do this
- but in assignment 4, I want you to do the following
- I will explain the scheme, called **2-change**, with an example
(I will use 6 cities instead of 12; adapt accordingly)
- say s is $[0, 2, 4, 1, 3, 5, 0]$



- think of randomly selecting 2 distinct edges, and cutting them out



- and then cross connect the end points



- now, from

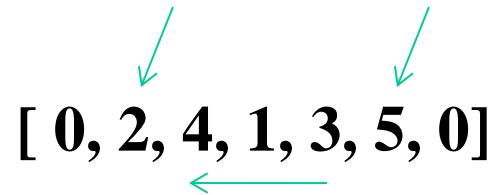
[0, 2, 4, 1, 3, 5, 0]

you got then

[0, 2, 3, 1, 4, 5, 0]

- think about it now as lists
- you randomly selected 2 distinct indices $x = 1$ and $y = 5$

[0, 2, 4, 1, 3, 5, 0]

The diagram shows a list of seven numbers: [0, 2, 4, 1, 3, 5, 0]. Above the number 2, there is a green arrow pointing down to it. Above the number 5, there is a green arrow pointing down to it. Below the number 1, there is a green arrow pointing left towards the number 4.

- then, you just *reversed* the sub-list from $x+1$ to $y-1$
- *right?*
- but, by the previous slides, you know how to do all that

- so, **actions()** in this assignment 4 returns just one neighbor
- but, you must abide by the **sampleai** API; that is, you still must start with `actions = []`
- then, append the one action described above;
the action is, as in assignment 2, a pair (**'description', new state**)
like
('2-change at ' + str(x) + ' and ' + str(y), the new tour described above)
- then return the actions list
- *right?*

- now, how do you select the 2 random numbers x and y where $x < y$ [they should not be the same, *right?*]
- the random module has a **randint()** function;
random.randint(A, B) select a random number between A and B inclusive
- since we assume that a tour always begins and ends at 0, you should **select a random number between 1 and numCities-1**, *right?*


- start off by selecting a first number *a*; then select a second number, looping as long as the second is the same as the first
[repeat: the 2 numbers must not be the same]
- make *x* the *min* of the two, and *y* the *max* of the two;
such functions *min()* and *max()* are in Python already
- now, using **slicing and list(reversed())** as explained above,
you should be able to make that new tour

- finally, there is that **result(self, s, a)** that must be there
- same as in assignment 3:
it returns the state gotten by applying action a to state s
- but **action a** already has that new state, viz. the second part of the pair a
- **all to think about**
- I chose 12 as number of cities, but your program should work with any number, with the appropriate distance matrix
[well, it may not find an exact solution]

- at the end, of course you must make an instance of your TspProblem, passing 12 and some appropriate matrix

- as in

```
d = ..... # distance matrix  
problem = TspProblem( 12, d)  
result = hill_climbing_random_restarts(problem, restarts_limit=200)  
....  
# appropriate print
```

Two red arrows are drawn on the slide. One arrow points from the bottom right towards the 'restarts_limit=200' parameter in the function call. The other arrow points from the bottom left towards the 'problem' parameter in the same function call.

- do read your documentation

```
Python 2.7.6 Shell
File Edit Shell Debug Options Windows Help
Python 2.7.6 (default, Nov 10 2013, 19:24:18) [MSC v.1500 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> ===== RESTART =====
>>>
>>> [(['2-change at 8 and 11', [0, 8, 7, 10, 2, 5, 3, 9, 1, 6, 4, 11, 0]), [0, 8, 7, 1
0, 2, 5, 3, 9, 1, 6, 4, 11, 0])]
14
>>>
>>>
>>> ===== RESTART =====
>>>
>>> [(None, [0, 8, 6, 11, 3, 10, 4, 1, 9, 5, 7, 2, 0])]
13
>>>
>>>
>>> ===== RESTART =====
>>>
>>> [(['2-change at 4 and 5', [0, 3, 6, 9, 1, 11, 4, 8, 10, 7, 5, 2, 0]), [0, 3, 6, 9,
1, 11, 4, 8, 10, 7, 5, 2, 0])]
12
>>>
>>>
>>>
>>> ===== RESTART =====
>>>
>>> [(None, [0, 3, 9, 1, 4, 8, 7, 10, 2, 5, 6, 11, 0])]
12
>>> ===== RESTART =====
>>>
>>> [(None, [0, 11, 10, 4, 7, 8, 6, 3, 1, 9, 5, 2, 0])]
14
>>>
>>>
>>> ===== RESTART =====
>>>
>>> [(['2-change at 5 and 6', [0, 1, 9, 3, 10, 4, 8, 5, 6, 11, 7, 2, 0]), [0, 1, 9, 3,
10, 4, 8, 5, 6, 11, 7, 2, 0])]
15
>>> ===== RESTART =====
>>>
>>> [(['2-change at 10 and 11', [0, 1, 4, 7, 10, 2, 5, 9, 3, 11, 6, 8, 0]), [0, 1, 4,
7, 10, 2, 5, 9, 3, 11, 6, 8, 0])]
14
```

Here are sample runs;
it does not always produce the same tour or an optimal tour
(because of the randomness)

It prints the last action (and last state), not a path