Local Search

CS 486/686: Introduction to Artificial Intelligence
Fall 2013

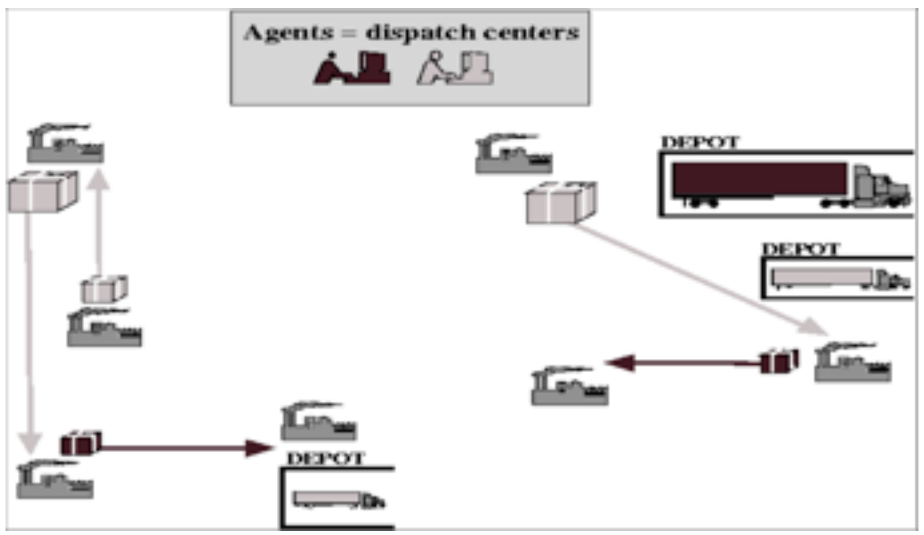
Overview

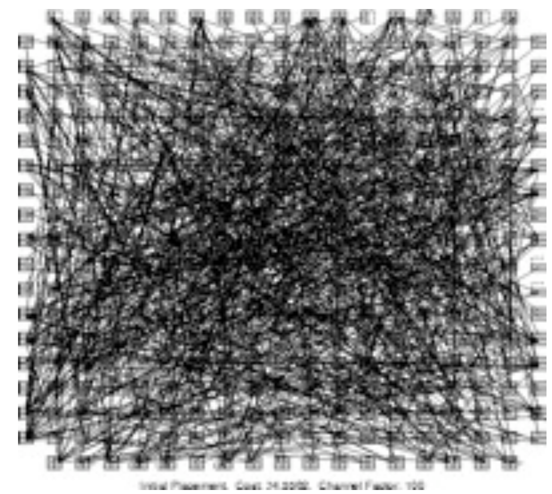
- Uninformed Search
  - Very general: assumes no knowledge about the problem
  - BFS, DFS, IDS
- Informed Search
  - Heuristics
  - A* search and variations
- **Search and Optimization**
  - What are the problem features?
  - Iterative improvement: hill climbing, simulated annealing
  - Genetic algorithms

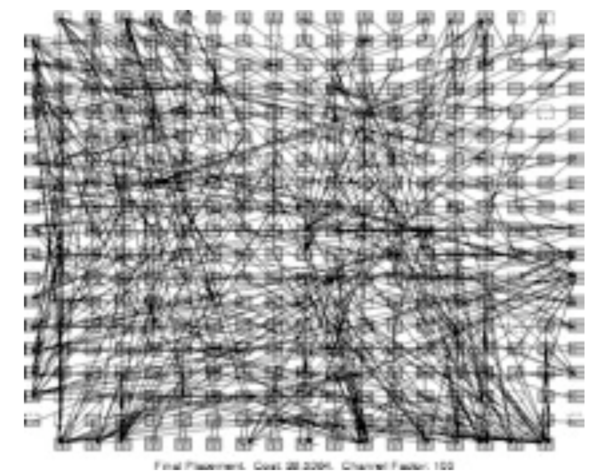
Introduction

- Both uninformed and informed search systematically explore the search space
  - Keep 1 or more paths in memory
  - **Solution is a path to the goal**
- For many problems, the path is unimportant

Examples







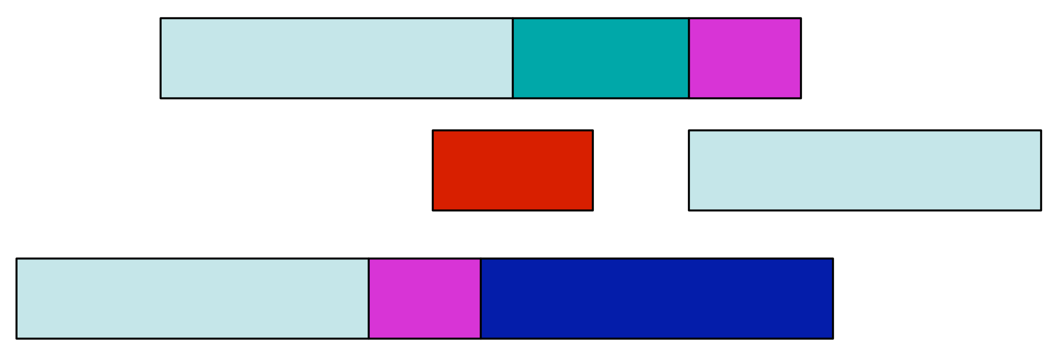
AV ~B V C

~A V C V D

B V D V ~E

~C V ~D V ~E

…

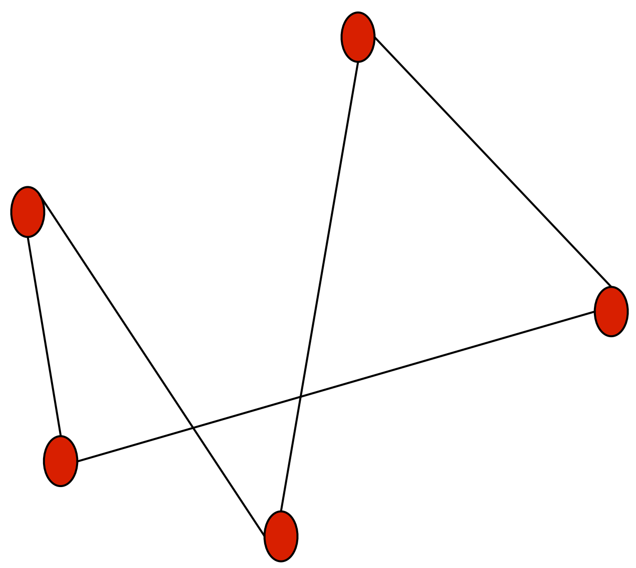


Informal Characterization

- Combinatorial structure being optimized
- Constraints have to be satisfied
- There is a cost function
  - We want to find a **good** solution
- Search all possible states is infeasible
  - Often easy to find **some solution** to the problem
  - Often provably **hard** (NP-complete) to find the **best** solution

Typical Example: TSP

- Goal is to minimize the length of the route
- **Constructive method**: Start from scratch and build up a solution
- **Iterative improvement method**: Start with solution (may be suboptimal or broken) and improve it

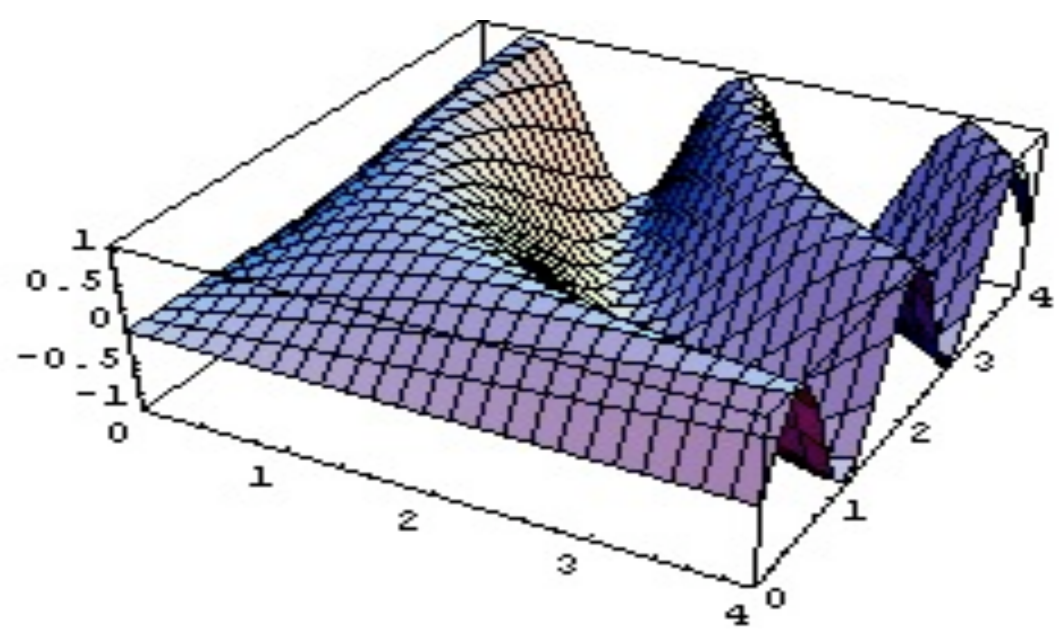


Constructive Methods

- For the optimal solution we can use A*
- But...
- We do not need to know how we got the solution
  - We just want the solution

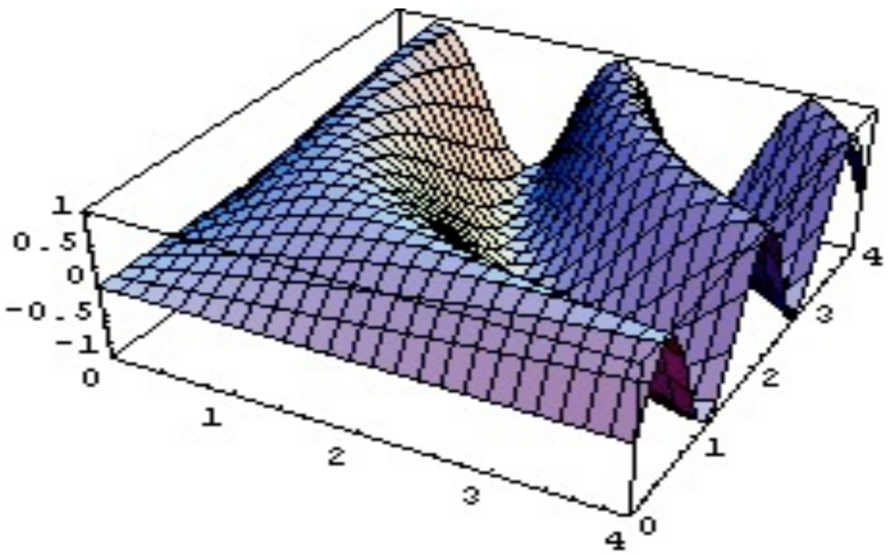
Iterative Improvement Methods

- Idea: Imagine all possible solutions laid out on a landscape
  - Goal: find the highest (or lowest) point



Iterative Improvement Methods

- Start at some random point
- Generate all possible points to move to
- If the set is not empty, choose a point and move to it
- If you are stuck (set is empty), then restart



Iterative Improvement Methods

- What does it mean to “generate points to move to”
  - Generating the **moveset**
- Depends on the application

TSP

2-swap

Hill Climbing (Gradient Descent)

- Main idea
  - Always **take a step** in the direction that **improves** the current solution value the **most**
- Variation of best-first search
- Very popular for learning algorithms

"...like trying to find the top of Mt Everest in a thick fog while suffering from amnesia", Russell and Norvig

Hill Climbing

1. Start with some initial configuration S
2. Let V be the value of S
3. Let S_i, i=1,...,n be neighbouring configs, V_i are corresponding values
4. Let $V_{max}=max_i V_i$ be value of best config and S_{max} is the corresponding config

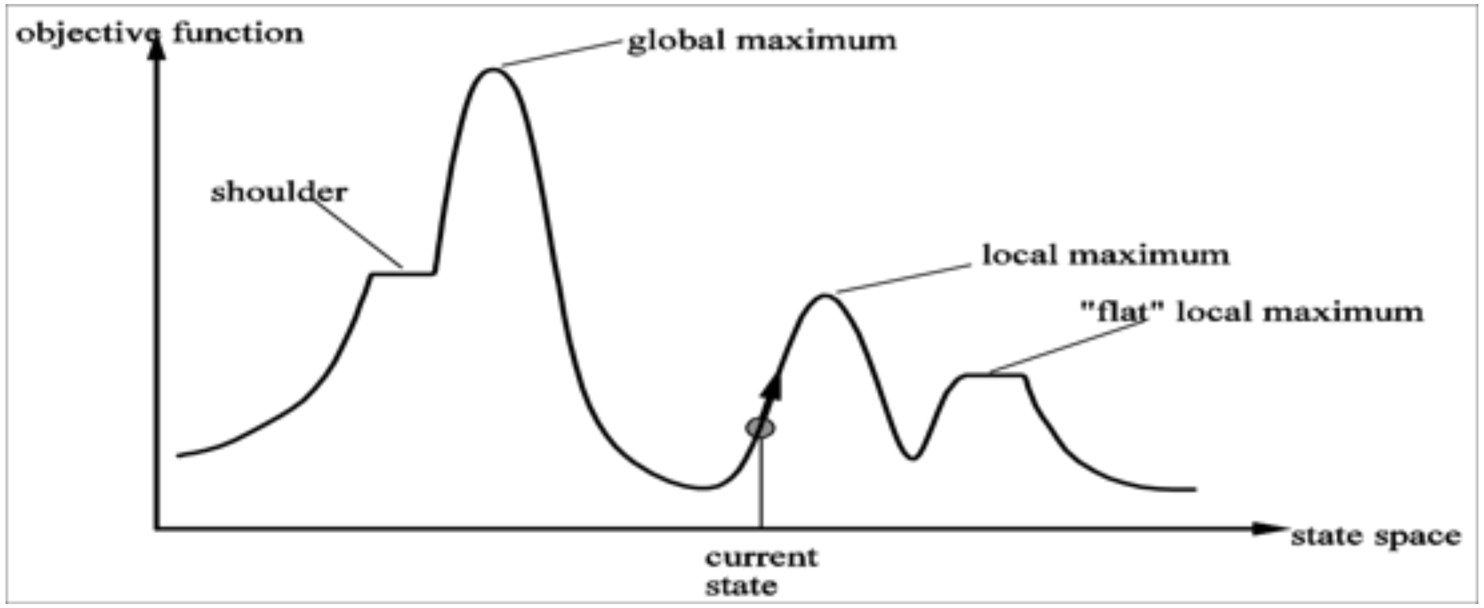
- If $V_{max}<V$ return S (local optimium)
- Let $S \leftarrow S_{max}$ and $V \leftarrow V_{max}$. Go to 3

Judging Hill Climbing

- Good news
  - Easy to program
  - Requires no memory of where we have been
  - Important to have a “good” set of moves
    - Not too many, not too few

Judging Hill Climbing

- Bad news
  - It can get stuck
  - Local maxima/minima
  - Plateaus



Improving Hill Climbing

- Plateaus
  - Allow for sideways moves
    - But be careful since might move sideways forever
- Local Maxima
  - **Random restarts:** *If at first you do not succeed, try, try again!*

Randomized Hill Climbing

- Like hill climbing except
  - You choose a random state from the move set
  - Move to it if it is better than current state
  - Continue until you are bored

More Randomization

- Hill climbing is incomplete
  - can get stuck at local optima
- A random walk is complete
  - but very inefficient
- **New Idea**:
  - Allow the algorithm to make some "**bad**" moves in order to escape local optima

Example: GSAT

AV~BVC	1
~AVCVD	1
BVDV~E	0
~CV~DV~E	1
~AV~CVE	1

Configuration A=1, B=0, C=1, D=0, E=1

Goal is to maximize the number of satisfied clauses: Eval(config)=# satisfied clauses

GSAT Move_Set: Flip any 1 variable

WALKSAT (Randomized GSAT)

Pick a random unsatisfied clause;

Consider flipping each variable in the clause

> If any improve Eval, then accept the best
>
> If none improve Eval, then with prob p pick the move that is least bad; prob (1-p) pick a random one

Simulated Annealing

1. S is initial config and V=Eval(S)
2. Let i be a **random** move from the moveset and let S_i be the next config, V_i=Eval(S_i)
3. If $V < V_i$, then $S=S_i$ and $V=V_i$
4. Else with probability p, $S=S_i$ and $V=V_i$
5. Go to 2 until you are bored

What About p?

- How should we choose the probability of making a “bad” move?
  - p=0.1 (or some fixed value)?
  - Decrease p with time?
  - Decrease p with time and as $V-V_i$ increases?
  - ...

Selecting Moves in Simulated Annealing

- If new value V_i is better than old value V then definitely move to new solution
- If new value V_i is worse than old value V then move to new solution with probability

$$e^{-(V-V_i)/T}$$

Boltzmann Distribution: T>0 is a parameter called temperature. It starts high and decreases over time towards 0. If T is close to 0 then the prob. of making a bad move is almost 0.

Properties to Simulated Annealing

- When T is high:
  - **Exploratory phase**: even bad news have a chance of being picked (random walk)
- When T is low:
  - **Exploitation phase**: “bad” moves have low probability of being chosen (randomized hill climbing)
- If T is decreased slowly enough then simulated annealing is guaranteed to reach optimal solution

Genetic Algorithms

- Populations are encoded into a representation which allows certain operations to occur
  - Usually a bitstring
  - Representation is key - needs to be thought out carefully
- An encoded candidate solution is an **individual**
- Each individual has a **fitness**
  - Numerical value associated with its quality of solution
- A **population** is a set of individuals
- Populations change over **generations** by applying strategies to them
  - Operation: selection, mutation, crossover

Typical Genetic Algorithm

- Initialize: Population P←N random individuals
- Evaluate: For each x in P, compute fitness(x)
- Loop
  - For i=1 to N
    - **Select** 2 parents each with probability proportional to fitness scores
    - **Crossove**r the 2 parents to prodice a new bitstring (child)
    - With some small probability **mutate** child
    - Add child to population
  - Until some child is fit enough or you get bored
- Return best child in the population according to fitness function

Selection

- Fitness proportionate selection: $$P(i) = \frac{\text{fitness}(i)}{\sum_j \text{fitness}(j)}$$
  - Can lead to overcrowding
- Tournament selection
  - Pick i, j at random with uniform probability
  - With probability p select fitter one
- Rank selection
  - Sort all by fitness
  - Probability of selection is proportional to rank
- Softmax (Boltzmann) selection: $$P(i) = \frac{e^{\text{fitness}(i)/T}}{\sum_j e^{\text{fitness}(j)/T}}$$

Crossover

- Combine parts of individuals to create new ones
- For each pair, choose a random crossover point
  - Cut the individuals there and swap the pieces

101|0101 011|1110

Cross over

011|0101 101|1110

Implementation: use a crossover mask m

Given two parents a and b the offspring are

(a^m)V(b^~m) and (a^~m)V (b^m)

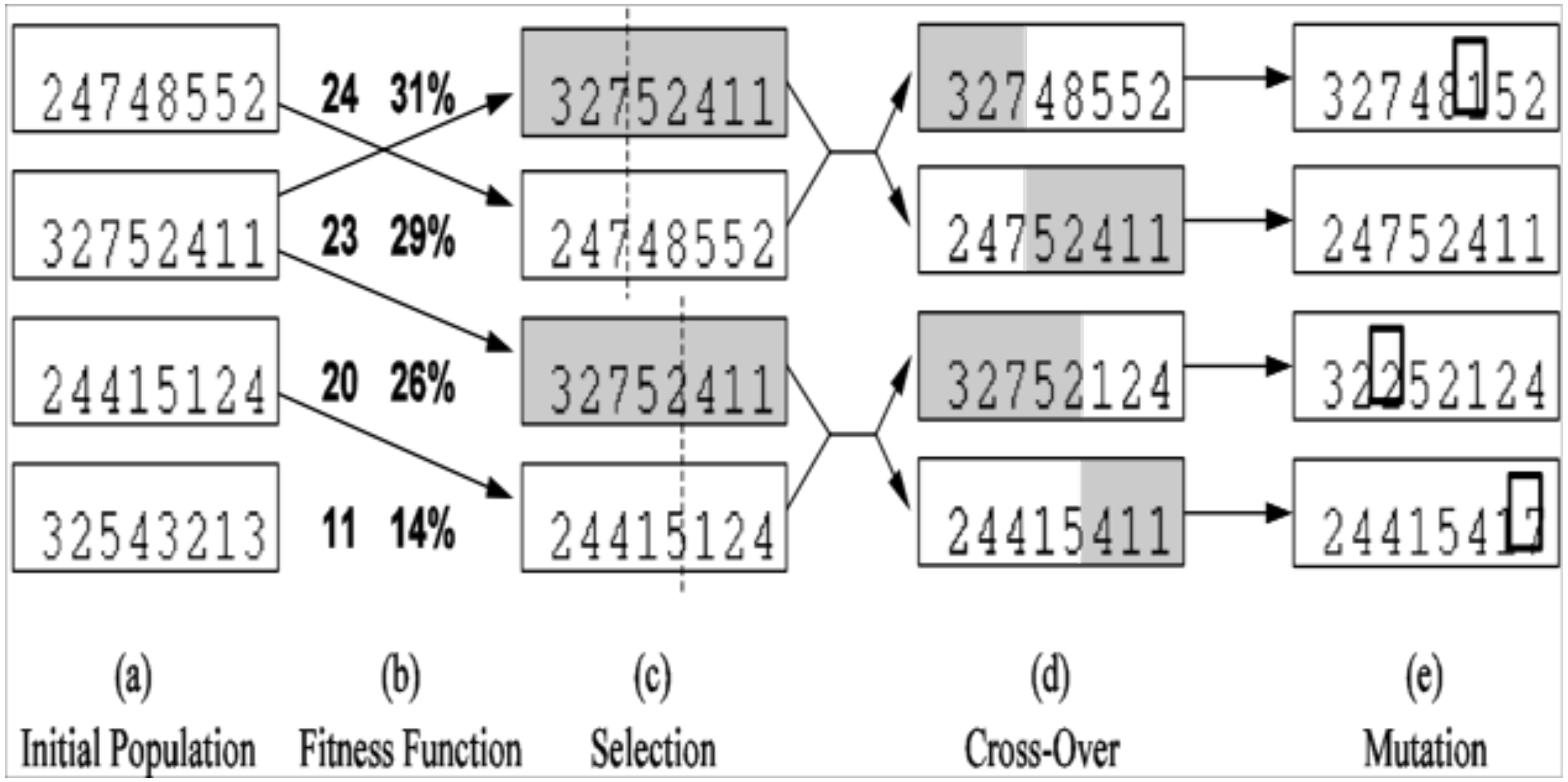
Mutation

- Mutation generates new features that are not present in original population
- Typically means flipping a bit in the string

100111 mutates to 100101

- Can allow mutation in all individuals or just in new offspring

Example


24748552
32752411
24415124
32543213
24 31%
23 29%
20 26%
11 14%
32752411
24748552
32752411
24415124
32748552
24752411
32752124
24415411
24752411
(a)
Initial Population
(b)
Fitness Function
(c)
Selection
(d)
Cross-Over
(e)
Mutation

Summary

- Useful for optimization problems
- Often the second-best way to solve a problem
  - If you can, use A* or linear programming or ...
- Need to think about how to escape from local optima
  - Random restarts
  - Allowing for bad moves
  - ...