CS441 Writeup: Dinner Party Problem

Yiming Lin

yl6@pdx.edu

Local Search

1.1 Agent actions

The agent chooses two people in the table and swap with each other. Specifically, the agent chooses two elements in the 2-d array and swap their position.

1.2 Hill-climbing search

The agent randomly generates a state as its initial state. Based on initial state, the agent does all possible actions, so a list of new states generated (called successors). The agent moves to the state with the best score in the successors, does all possible actions based on the state with the best score in the successors and generate new successors, moves to the state in the new successors with the best score, and repeat. If the agent cannot move to a state that has a greater score than previously moved to, the searching halts.

```
Hill-climbing-search(Agent):
Init-state ← Agent.random-generate()
Current-state ← Init-state
Loop:
        Successors ← Agent.actions(Current-state)
        Successors.sort-descent()
        If successors[0].score < Current-state.score:
                Return Current-state
        Else:
                Current-state ← successors[0]
```

1.3 Random-restart search

The random-restart search is built on top of hill-climbing search. The agent uses hill-climbing search and generate a resulted state when hill-climbing search algorithm halts. Instead of halting the searching directly, the agent randomly generates another initial state and executes hill-climbing search based on new generated initial state. Such a process repeats until a desired score is reached.

```
Random-restart-search(Agent, K)    Input: K – The times of restart execution
Current-state ← None
```

```
Maximum-state ← None
Loop 1..K :
        Current-maximum ← Hill-climbing-search(Agent)
        If (current-maximum.score > maximum-state.score) OR (maximum-state is None):
                Maximum-state ← current-maximum
Return Maximum-state
```

1.4 Local beam search

The agent randomly generates k states, for each state, the agent does all possible actions in parallel and generates k lists of successors. For all k lists of successors, the agent picks up k states with top k scores among all lists of successors; for those k states, the agent does all possible actions in parallel again and picks up k states with top k scores among all lists of successors. The agent repeats such a process until it cannot move to a state with a better score.

```
Thread-action(Agent, state, lock):
Successors ← Agent.actions(state)
Successors.sort()
Acqurie(lock)
Successor-lists.append(successors)
Release(lock)

Local-beam-search(Agent, K)    Input: K – The number of states to act in parallel
Current-states ← []
Init-state-counter ← 0
Maximum-state ← None
Loop until Init-state-counter == K:
        Init-state ← Agent.random-generate()
        If Init-state not in Current-states:
                Current-states.append(init-state)
                Init-state-counter ← Init-state-counter + 1
Loop:
        Successor-lists ← []
        Lock ← Threading.create-lock()
        Threads ← [Threading.create-thread() for 1..K]
        For i in 1..K:
                Args ← [Agent, current-states[i], lock]
                Threads[i].start(Thread-action, args=Args)
        For i in 1..K:
                Threads[i].join()
```

```
Successors ← merge-sort(successor-lists)
If (Successors[0].score > Maximum-state.score) OR (Maximum-state is None):
        Maximum-state ← Successors[0]
Else:
        Return Maximum-state
Current-states ← Successors[0..K]
```

1.4.1 Implementation details

The local beam search algorithm is built with concurrency control in Python 3.7. A concurrent data structure "successor lists" is maintained so that each state generates a list of successors and add that list as an element to the "successor lists". Each list of successors generated by each state is sorted before added into the "successor lists". To merge all lists in "successor lists", the method "merge" in Python 3.7 module "heapq" is used, which merge sorted lists into an iterator.

For each state to act, the program creates k threads, and for each thread the agent does all possible actions and generate a list of successors concurrently. The main thread waits for all other threads to finish generating successors, the main thread picks top k successors for each k threads to act again. The program repeats this process until the score cannot make improvement.

1.5 Experiment results

The following table shows the result of 20 experimentations on hill-climbing search, random-restart search, and local beam search. The result is gotten within the run time 60 seconds.

#	**Hill-climbing search**			**Random-restart search**			**Local beam search**		
	Inst1	Inst2	Inst3	Inst1	Inst2	Inst3	Inst1	Inst2	Inst3
1	92	473	122	92	506	130	100	517	125
2	88	489	118	91	509	124	92	521	125
3	84	482	112	100	480	120	100	523	133
4	79	462	110	92	496	118	100	515	136
5	69	472	102	88	487	127	95	513	121
6	81	479	114	100	518	128	95	522	122
7	78	451	111	100	492	129	95	520	126
8	78	457	90	88	495	124	91	528	119
9	84	488	99	100	500	121	100	509	142
10	100	460	113	100	504	126	100	547	128
11	62	459	133	88	494	118	100	522	126
12	81	477	117	85	511	130	95	517	128

13	91	498	112	100	485	141	95	521	125
14	78	498	94	91	515	121	100	543	121
15	85	460	104	100	493	121	100	531	126

Table 1 Experimentation results for 60 second runtime

The following table summarizes the results from 15 experimentations above.

Alg.	**Hill-climbing search**			**Random-restart search**			**Local beam search**		
Instance	Inst1	Inst2	Inst3	Inst1	Inst2	Inst3	Inst1	Inst2	Inst3
Mean	82	473.67	110.07	94.33	499	125.2	97.2	523.27	126.87
Var	79.07	216.89	113.13	31.02	118.13	33.89	10.16	101.00	34.64
Std	8.89	14.72	10.64	5.57	10.87	5.82	3.19	10.05	5.89
Max	100	498	133	100	518	141	100	547	142
Min	62	451	90	85	480	118	92	509	119

Table 2 Statistical summary of 20 experimentations

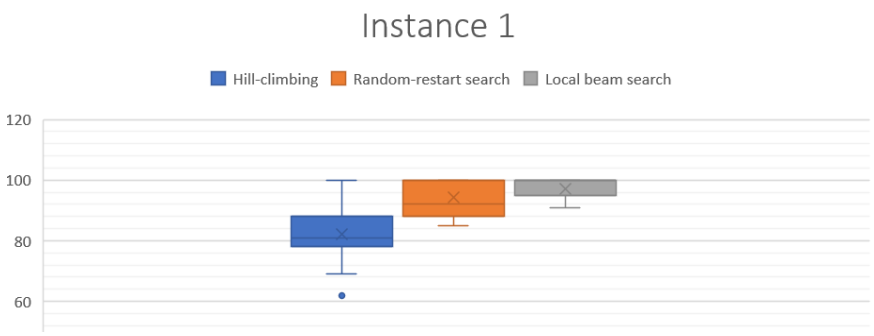

Figure 1 The box plot of three different algorithms on instance 1

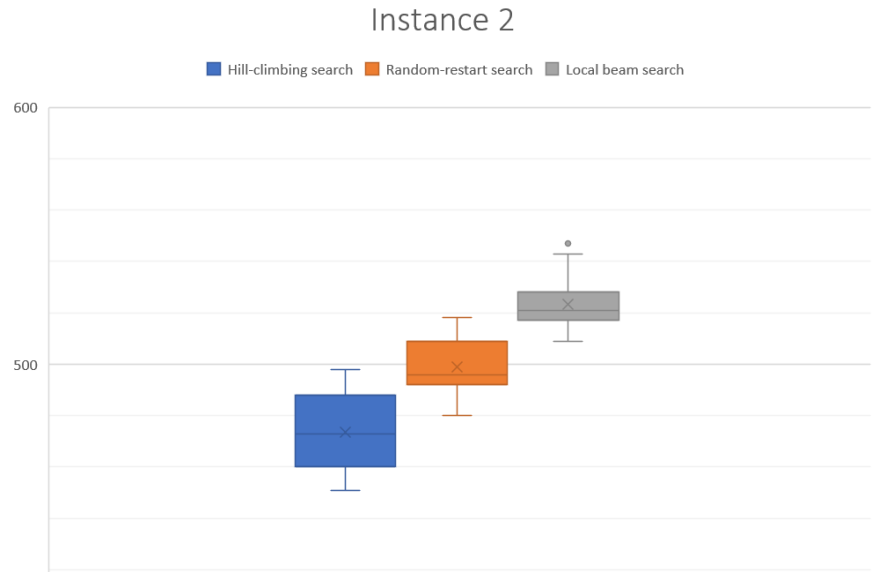

Figure 2 The box plot of three different algorithms on instance 2

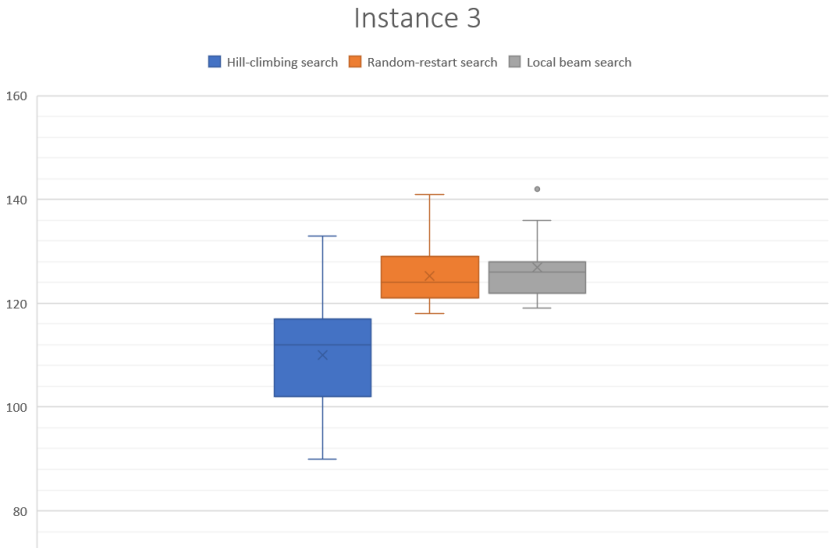

Figure 3 The box plot of three different algorithms on instance 3

1.6 Discussion

From the experimentation results, it is obvious that local beam search works better than random-restart search, and random-restart search works better than hill-climbing search on the dinner party problem. All three algorithms tend to be greedy to get a better score, but the differences are revealed by the statistical summary. The common tendency among all experimentations on three instances is the following: (1) local beam search always results in a largest mean, smallest variance, and smallest standard deviation; (2) hill-climbing search always results in a largest variance, largest standard deviation, and smallest mean; (3) random-restart search always has a medium results.

For hill-climbing search, the agent tries to move to the neighbor node that has a best score improvement. However, the agent doesn't know the landscape of the state space in the local search, it is very easy to trap into a local maximum. Furthermore, since the initial state is randomly generated, the initial state has a huge effect on the path that the agent moves. Sometimes the initial state has a "direct path" to a reasonably good local maximum, so the agent just follows that "direct path"; in other words, the agent keeps greedy to move towards the node that can maximumly improve the score. However, it is not often the case, our agent is easy to trap to a bad local maximum, then get stuck on it. Therefore, the hill-climbing search results in a largest variance and standard deviation among all three algorithms in 15 experimentations.

In section 1.3, it states that random-restart search is built on top of hill-climbing search. Its advantage is that instead of halting the search directly once the agent gets stuck on the local maximum, it re-generates a new initial states and hill-climbing again, until it reaches a reasonably good result or time threshold. An interpretation for this is that the once the agent gets stuck on the local maximum, it goes

to a random place in the state space, and then “greedy moving” again. The agent keeps changing its moving direction once it meets a local maximum. Random-restart search essentially just run hill-climbing search multiple times, but it turns out it improves the stability of searching result a lot based on the experimentation results. Statistical speaking, the result of random-restart search depends on the probability of getting an optimal solution (or reasonably good solution).

Local beam search does the best job among all three algorithms on this problem. Although it just keeps k best states among all successors instead of one state, as Stuart Russell and Peter Norvig wrote in “Artificial Intelligence: A modern approach”, the “useful information” is passed among all threads. In random-restart search, each hill-climbing search is independent to the others; in local beam search, the agent knows all the successors of k states, then it choose best k states among all successors, so it quickly abandons successors that cannot improve the score very much. Hence, it has the most stable results among all three algorithms, and the mean is the largest.

1.7 Getting over local maximum

One way to get over local maximum is to utilize randomization technique. After experimenting with local beam search, its variant algorithm – stochastic beam search is experimented. However, **the experimentation failed**.

1.7.1 Stochastic beam search

The stochastic beam search does the similar thing as local beam search. In stochastic beam search, it keeps k states and does all possible actions based on k states in parallel, and then merge all lists of successors of k states into a “candidate list”. In local beam search, it will pick k states in the candidate list with top k scores; in stochastic beam search, it will pick k states based on the probability density function.

In “Artificial Intelligence: A modern approach”, Stuart Russell and Peter Norvig noted that local beam search was easy to get stuck in the local maximum because the k states in all iteration lacked diversity. In other words, the k states with top k scores in the candidate lists are likely to come from the action of only one state. With the iteration continuing, some of the successors that are generated by some of the previous k states are fully abandoned. Then the agent becomes focusing on a small region in the state space. Hence, the agent gets stuck in the local maximum.

Stochastic beam search aims at adding the diversity of k states. It also requires making improvement. Then an appropriate *probability density function* is critical to this algorithm. The high-level idea is that the larger the score of the state, the larger probability to pick up in the candidate list. The probability is independent of the value of the score, it is dependent of the order of the candidate list.

1.7.2 Probability density function

Fan Wang and Andrew Lim in "A stochastic beam search for the berth allocation problem" [1] proposed a probability density function that is rank-based probability distribution. It ranks the candidate list by three degree. In the first degree, the probability of being picked is highest. In the second degree, the probability is less than the first degree. And in the third degree, the probability is the smallest. If there are M states in the candidate list, and the first degree has S states; then the second degree has 2S states, and the third degree has M-3S states. And the probability density function is the following:

$$p(n) = \begin{cases} \dfrac{5}{6S+M} & if\ n\ \epsilon\ 1st\ degree \\ \dfrac{2}{6S+M} & if\ n\ \epsilon\ 2nd\ degree \\ \dfrac{1}{6S+M} & if\ n\ \epsilon\ 3rd\ degree \end{cases}$$

Then, it has $\sum_{n=1}^{M} p(n) = 1$

1.7.3 Failure of experimentation

The experimentation of 60 seconds runtime is failed. Compared to the results in table 1, the result is extremely small. The observation of the process of stochastic beam search is that the agent is moving to the states with a larger score, but the speed is extremely slow. The following figure shows the result of stochastic beam search on instance 2. The configuration is that the beam width is 6, and the value of S is 3, and the searching halts when the run time reaches 60 seconds (Detailed data is recorded in appendix). The "avg" is the average value of 6 states selected. And "stdev" is the standard deviation of 6 states selected in each iteration. The "maximum" is the maximum score recorded until current iteration.

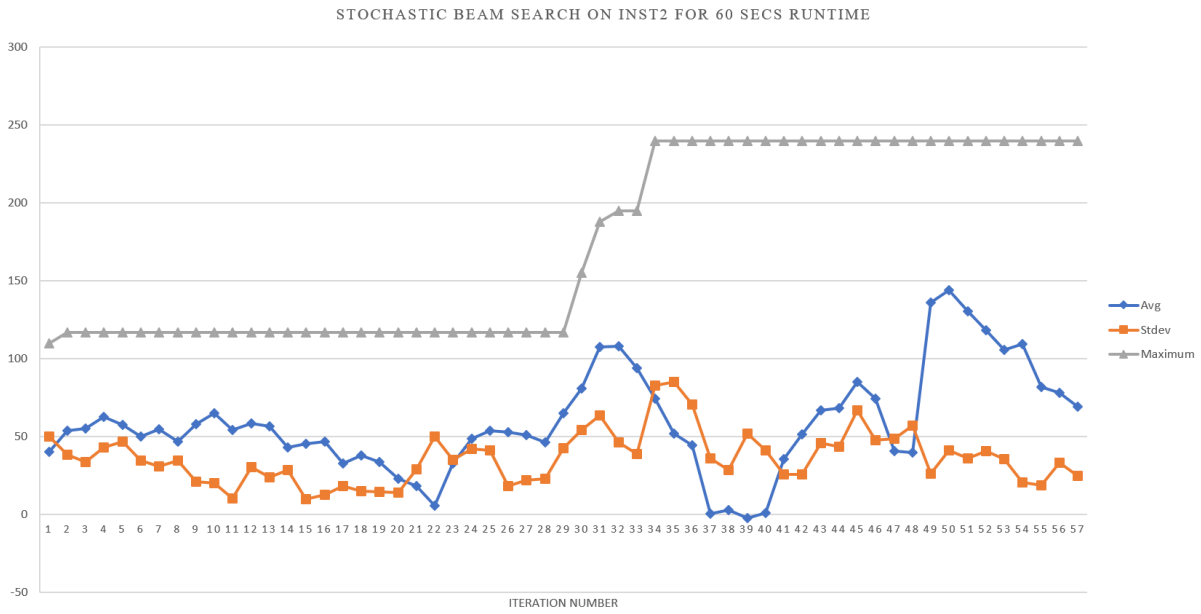

Figure 4 The plot of statistical summary on each iteration of stochastic beam search with 60 seconds runtime

In figure 4, the agent has made some progress of finding a better score as the maximum score is

increasing. However, it takes a lot of iterations in order to find a better score to update maximum score. This project requires to find a solution within 60 seconds runtime, so it is not appropriate for such an algorithm to solve problem.

Informed Search

This section describes the experimentation of A-star algorithm, with a failed heuristic function designing, which results in a **failed experimentation** on informed search.

2.1 Agent actions

The agent adds a person to an empty seat where it is adjacent to an existed person or opposite to an existed person.

2.2 Cost estimation and heuristic function

The goodness of the table is measured by the score: the higher the score, the better. Then, the score is added up by the preference score of each person as well as some bonus points made by adjacent or opposite host-guest role matching. The goodness is that each person becomes closer to the preferred person based on preference score. Hence, the cost is the "un-preferred degree".

In order to satisfy heuristic admissible, which means the cost estimation will never be overestimated. Then, it also means that the un-preferred degree will never be overestimated; thus, it means that *the preferred degree (the score at the goal state) will always be equal to optimal or overestimated*.

In order to estimate the cost to the goal state that all everyone have a seat (the table is full), the following constraint:

> "h(p1, p2) + h(p2, p1) points for every adjacent or opposite pair of people p1, p2"

is relaxed for all people that are not have a seat. Then for all people that are not have a seat, they can seat with three people that has top 3 preference scores in its preference score list. The following figure visualize such an idea:

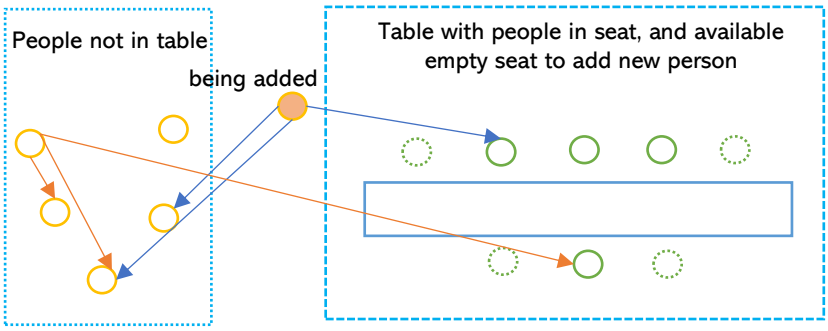

Figure 5 The visual explanation of cost estimation

The green solid border circle denotes the person that has already had a seat. The orange solid border circle denotes the person hasn't been assigned a seat (called *wait list*). The circle filled with orange color is the person that is going to be added to the table.

The estimation is following:

(1) people who are not in table pick top 3 "pair-score" which is calculated by which the preference score plus role-matching bonus;

(2) the person who is being added can pick two people in the *wait list* if its left or right is empty **plus** its opposite seat is empty; otherwise, pick one people in the *wait list* has the maximum "pair-score".

(3) the person who have already been in table cannot be relaxed any constraint.

2.3 Evaluation of heuristic function

Currently, the heuristic function does satisfy heuristic admissible, since everyone in the wait list can seat with their most preferred 3 people, which always overestimates the score of the table, which never overestimates the cost to the goal state. However, such a design causes the cost estimation to be too small to cut off some nodes in the frontier. Although it relaxes the constraint on the point calculation, in some sense it also relaxes the constraint on the shape of the table. In order to let everyone in the wait list to seat with top 3 pair-score, one person in the table can have multiple opposite people instead of just one opposite person. Thus, it is a bad heuristic function, its performance is nearly no difference with Dijkstra's algorithm.

2.4 Informed search vs. Local search

For this problem, personally speaking, using informed search has some internal difficulties. From the initial state in informed search, which picks a random person to add to the table. For a 30-people table, it has 30 kinds of state. However, the goal state has around 30! ÷ 4 kinds of state.

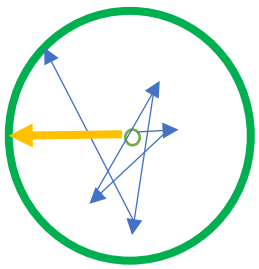

Figure 6 The abstract representation of state space

In figure 6, the inner green circle represents 30 initial states of state space. And the outer circle represents 30! ÷ 4 states which are goal states. The difficulty is that the optimal goal state is unknown, also there are too many goal states, it is difficult to estimate where a goal state is so as to design a heuristic function. If the number of goal state is small, it is easy to find a estimation function, then the orange path in the figure 6 can be found; otherwise, the path will be like blue arrows in figure 6.

Furthermore, the path of finding a goal states doesn't matter in this problem. From the result of local search algorithms, it turns our that local search performs relatively well on this problem. At least it can find some solutions with reasonably good score.

2.5 A* Algorithm

```
A-star(Agent):
Frontier ← priority-queue([]) // ordered by node.score + node.heuristic()
Explored ← Dict()
Node ← Agent.init-state()
Frontier.push(Node)
Loop until Frontier is empty:
        Node = Frontier.pop()
        If Node.is_goal():
                Return Node
        Explored[hash(Node)] = Node
        Successors = Agent.actions(Node)
        For Successor in Successors:
                If hash(Successor) not in explored AND Successor not in Frontier:
                        Frontier.push(Successor)
                Else if Successor in Frontier:
                        Index ← frontier.get-index(Successor)
                        // the greater the Score + Heuristic, the smaller the cost
                        If (frontier[Index].score + frontier[Index].heuristic()
                          < Successor.score + successor.heuristic()):
                                Frontier[Index] ← successor
                                Frontier.heapify()
Return None
```

Appendix

The following table shows the detailed data in the experimentation described in section 1.7.3.

iter #	Beam1	Beam2	Beam3	Beam4	Beam5	Beam6	Maximum
1	110	74	54	44	5	-47	110
2	117	82	56	47	22	0	117
3	108	95	44	35	27	23	117

4	108	108	98	34	21	7	117
5	112	107	91	24	5	5	117
6	94	73	68	56	11	-3	117
7	107	65	65	56	26	10	117
8	87	76	59	49	29	-18	117
9	82	76	62	56	55	17	117
10	93	90	62	57	49	39	117
11	67	60	60	54	49	35	117
12	91	85	72	70	19	14	117
13	93	79	60	48	37	23	117
14	81	69	46	44	25	-7	117
15	62	52	46	42	35	34	117
16	72	48	47	43	42	30	117
17	58	57	30	20	18	13	117
18	59	55	38	32	22	20	117
19	57	44	41	26	20	15	117
20	40	40	27	15	14	1	117
21	46	43	36	25	-5	-35	117
22	81	47	8	-5	-21	-77	117
23	92	54	37	27	6	-19	117
24	100	86	72	48	-2	-11	117
25	114	90	57	57	3	2	117
26	74	70	62	53	35	23	117
27	79	68	60	49	38	11	117
28	78	72	47	38	30	12	117
29	117	117	82	37	21	15	117
30	155	142	99	38	33	17	155
31	188	164	140	100	39	14	188
32	195	125	122	82	65	60	195
33	165	117	84	84	74	40	195
34	240	93	87	14	9	3	240
35	239	37	24	16	14	-18	240
36	190	44	39	33	-11	-29	240
37	61	23	16	-26	-28	-42	240
38	50	24	5	-9	-13	-39	240
39	72	62	-14	-29	-37	-68	240
40	51	42	27	-18	-39	-57	240
41	72	50	43	42	16	-9	240
42	92	76	48	43	36	14	240

43	149	96	63	47	41	5	240
44	139	108	73	36	34	19	240
45	195	129	118	31	30	8	240
46	117	117	116	69	37	-9	240
47	131	47	45	37	18	-33	240
48	136	85	46	2	2	-33	240
49	161	155	148	142	127	83	240
50	217	162	154	136	103	92	240
51	208	126	126	116	104	102	240
52	173	169	115	108	78	66	240
53	170	125	116	78	78	67	240
54	135	128	125	97	93	79	240
55	111	101	80	75	70	55	240
56	129	111	77	70	50	32	240
57	105	94	78	53	44	40	240

Table 3 Detailed data in experimentation described in section 1.7.3

Reference

[1] A stochastic beam search for the berth allocation problem, Fan Wang, Andrew Lim, January 2007 https://www.sciencedirect.com/science/article/pii/S016792360600090X