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

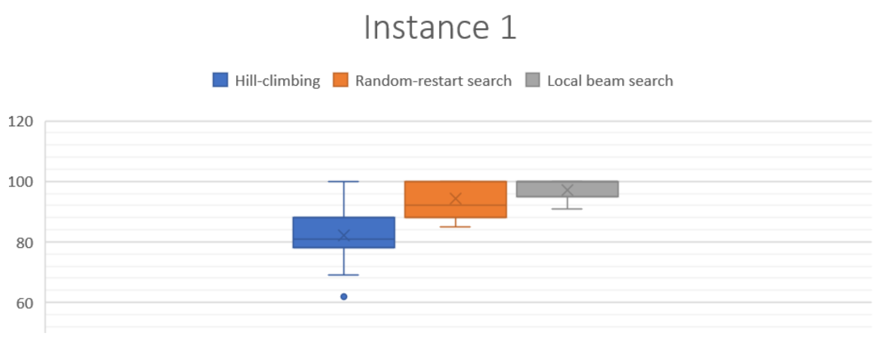

Figure 1 The box plot of three different algorithms on instance 1

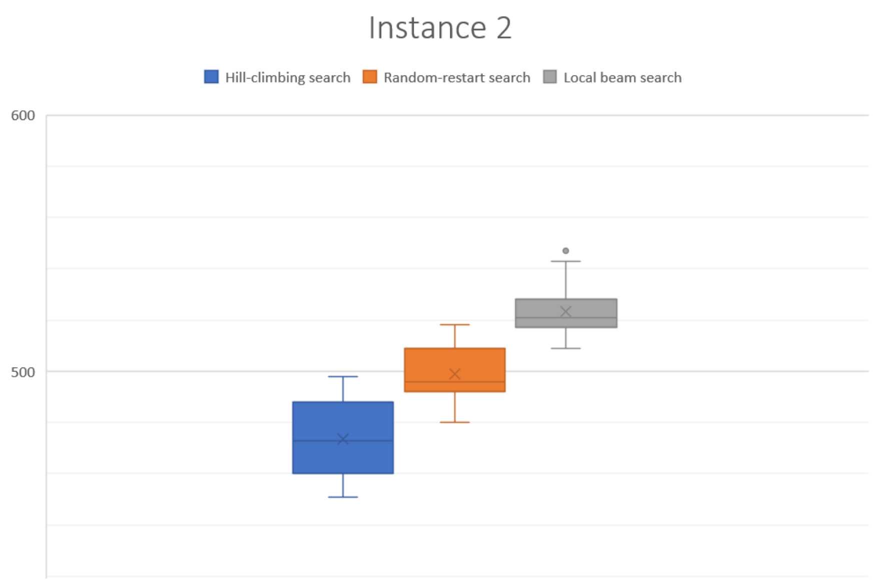

Figure 2 The box plot of three different algorithms on instance 2

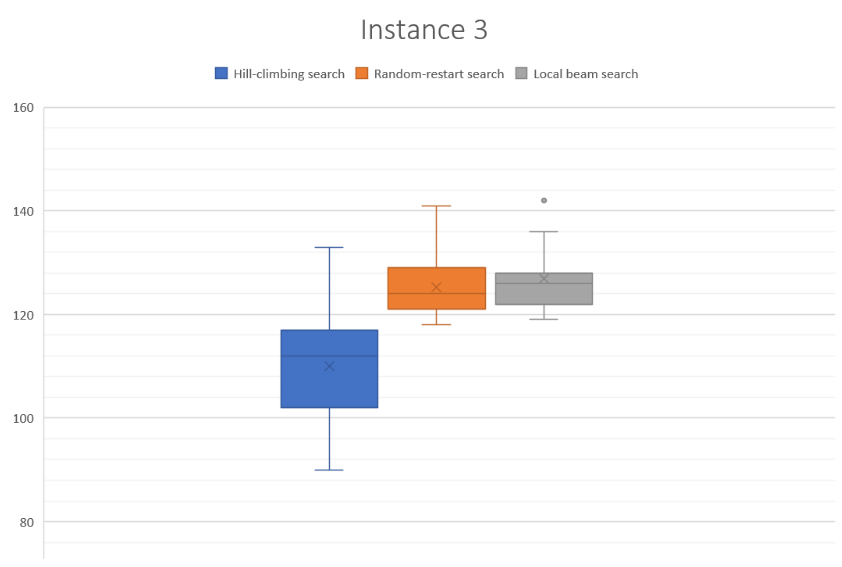

Figure 3 The box plot of three different algorithms on instance 3

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.