Parellel Pac-Man Solver

Project Report

Yiran Shen yirans@andrew.cmu.edu
Jiaqi Zuo jiaqizuo@andrew.cmu.edu

December 9, 2019

1 Summary

We implemented a game solver of Pac-Man using the iterative deepening search algorithm. We implemented both the sequential version and the parallel version using OpenMP and MPI. We met some issues during the implementation of the parallel version and applied the parallel window search [2] to resolve them. Both versions of the algorithm are evaluated on a multi-core system, that is, GHC machines. The performance of the parallel version achieves linear speedup corresponding to the sequential while still providing a good solution.

2 Background

2.1 Pac-Man game

Pac-Man is a maze arcade game.[5] Basically, there are three roles in the game: Pac-Man, dots and ghosts. Pac-Man is controlled by the player and needs to eat target dots (usually all dots) without colliding with ghosts. The dots are statically distributed in the maze while ghosts moves randomly in the maze. The solver takes the initial environment containing the maze, the Pac-Man, dots and ghosts as the input and outputs a give a suboptimal solution (path) to eat all target dots.

2.2 Iterative Deepening Search

The core algorithm used in the solver is the iterative deepening depth first search(IDDFS). [4] It is a variant of depth first search with depth limitation. That is, in each iteration, the algorithm expands nodes in the same order as vanilla depth first search but within some limit on the depth. The depth limit increase across iterations until the optimal or target solution is found. One big advantage of IDDFS is that it combines the space efficiency of vanilla depth first search and the

completeness of breadth first search. [4]

2.3 Workload analysis

Basically, the IDDFS is the depth first search with increasing depth limit until the optimal solution is found. Iterations start with the same search tree but expand nodes with different depth limit, so iterations are mutually independent. Besides, the search space increases exponentially as the depth limit increases. When the search space or the depth limit becomes large enough, the workload will become computation intensive. In that case, we can utilize the independence among iterations to apply parallel execution to improve the performance. However, the workload does not have the functional property, so data parallel may not be a good idea. Also, within each iteration, the depth first search cannot exploit vectorization to improve the performance, so SIMD execution may not be a good idea either. Based on the analysis on the workload, we think the shared memory parallelism is the most appropriate model for our project, so we decide to apply OpenMP and MPI to implement the parallel version of IDDFS.

3 Approach

Due to the goal of this project, we decide to focus on the implementation of IDDFS since it is the core algorithm of the Pac-Man solver. Hence, we adapted the infrastructure of the solver from the course CS188 from University of California, Berkeley [1] and implemented the IDDFS within the framework provided by the infrastructure. Since the infrastructure is implemented in Python, we decide to use Python to implement both sequential and parallel version of IDDFS. Note that Python does not have multi-threading due to the GIL, we will use multi-processing in the project. Hence, OpenMP here does not refer to multi-threading strategy but a OpenMP-like multi-processing strategy. Our target machine should be the multi-core system and we do not consider the GPU as our target machine since we cannot find the functional property in the workload. The parallel version is expected to achieve the linear speedup to the sequential version both implemented in the project.

According to the analysis in the previous section, we know that iterations are mutually independent, that means we can apply the parallelization among iterations. However, we do not know in which

depth of the search space we can find the optimal solution. Therefore, we decide to use the task based parallelism, where each task is a depth first search with a fixed depth limit. Then the dynamic scheduling on those tasks is applied to map tasks to working processes.

3.1 Naive parallelization

We first tried the naive parallelization as described above. More detailedly, in the OpenMP implementation, we ran the program in batched iterations, that is, we first run N iterations using the method described above. If we fail to find the optimal solution, we will run next N iterations until we reach the optimal solution. The reason is the lack of communication among working threads in OpenMP. In the MPI implementation, the dynamic schdueling is easy to implement. We first sent the initial search tree to each working process. Then the main process maintains a counter for the next pending task. Once an worker process is ready, the main process sends the task id, which is, the current depth limit to that worker and the worker will do the search and return the result. Once the the target solution is found, the main process will send the stop signal to all worker processes indicating the termination of the whole algorithm.

However, we only get less than 2x speedup no matter how large the maze is. Then we analyzed the workload of our algorithm and found that the search space increases exponentially as the depth limit increases. That is, assume the depth limit is n, the work of the iteration with depth limit n is $\mathsf{O}(b^n)$. That shows the overall performance is always bound by the last iteration since it has the largest amount of work, which corresponds to our results that we cannot get more than 2x speedup.

From the analysis above, we can find that the algorithm is embarrassingly parallelized. Although it is easy to find the independence among iterations, the workload imbalance offsets the benefit of parallelization. Due to the exponentially increased workload, the dynamic scheduling or other smart scheduling methods cannot fix this problem because it is easy to see that

$$b^1 + b^2 + \cdots + b^{n-1} = \frac{b^n - 1}{b - 1} < b^n.$$

That means the total workload of all other iterations is still less than that of the last iteration. That shows the IDDFS is inherently difficult to parallelize.

3.2 Parallel Window Search

From the previous analysis, we know that the naive parallelization cannot get the expected speedup and smart scheduling methods do not work due to the exponentially increased workload. To overcome the workload imbalance issue, we introduce the parallel window search with a special strategy called node ordering. [2] The aim of the node ordering is to reduce the time complexity of the single iteration from exponential to linear to the depth limit of current iteration. [2]

The idea is that there are many overlaps in the search tree among the iterations because iterations with larger depth limit repeat the same search path as smaller depth limit. To utilize this property of the algorithm, results from previous iterations can be used to assign an order to nodes that will be expanded in the current iteration. Then the current iteration can immediately to choose a node to expand to complete the search. This idea is beneficial for iterations with large depth limit because ordering based on results from previous iterations can significantly reduce the search space of the current iteration, which improve the performance.

This idea is based on the perfect ordering, which is an order of all candidate nodes that will be expanded in the current iterations. However, candidate nodes are not available at the same time due to node expanding order of depth first search. [2] One possible solution to that is that we can use some heuristics to assign estimated priorities to those nodes that are not expanded yet. The upside of this approach is that we can get an estimated perfect ordering to reduce the search space. The downside of this approach is that we may not get the optimal solution. That means we need to make a trade-off between the quality of the solution and the performance. After careful consideration, we think it is acceptable to get the sub-optimal solution in order to improve the performance because the solution to the Pac-Man solver does not need to be optimal in this project, while the performance is much more important than the quality of the solution.

The last thing to consider to how to choose a good heuristic. We decide to use the way A^* search [3] to formalize the heuristic function as suggested in [2]. Consider the heuristic function

$$f(k) = g(k) + h(k),$$

where k is the next candidate node ready to expand, $g(k)$ is the cost from the initial node to current node k and $h(k)$ is the estimated cost from k to the target. Note that the newly expanded node

regard to previous iteration in current iteration typically contains nodes with the same f values, we can order the candidate nodes in the increasing order of h values. [2] That means we first expand the node that has the minimum cost to the target.

4 Implementation

In this section, we are going to mention some implementation details to better illustrating the idea of parallel version of the algorithm.

First, the node ordering requires results from the previous iterations. Ideally, for the iteration with depth limit n, it requires results from the iteration with depth limit $n-1$. In the implementation, that requires the communication among working processes. The data to be transferred are all leaf nodes in the result of the iteration with depth limit $n-1$. The data exponentially increases as the depth limit increases, so transferring all leaf nodes is not practical due to the large memory and performance overhead. To reduce the overhead, we can change the perspective to working processes. Instead of storing results from previous iteration, each working process can store the result from previously assigned depth limit by master process under the assumption of dynamic scheduling. In that case, there are no communications among working processes and that will improve the performance a lot. Another issue is the space or memory usage. Rather than store the all leaf nodes, it is better to store part of leaf nodes which have higher priorities, whose order is generated using the heuristics mentioned in the previous section. The number of nodes can be constant or adaptive and we choose to store constant number of nodes because it is easy to implement and the result is good enough. One issue is that the current iteration uses results from the previous assigned depth limit instead of previous depth limit, which may affect the quality of the final solution. Here, we decide to accept that fact and relax the constraint on the quality of the solution since we care more about the performance.

Second, we would like to discuss about the implementation using two parallelization tools, OpenMP and MPI. Most part of the implementation using both tools are the same. The only important difference is how to terminate the algorithm when the target (sub-optimal) solution is found. For MPI, it is easy and obvious because working processes reports the result to the master process and is assigned the next task (depth limit). Reporting results can indicate whether the target solution

is found to the master process. Once the master process knows that the target solution is found, it can send some stop signal to all worker processes. Sending signals can be asynchronous to generate little overhead of the performance. Hence, the MPI can achieve expected performance though it may generate more tasks than necessary.

However, it is not the same case for OpenMP. Since there are no communications between the master process and worker processes. The only interaction is the join of worker processes in the master process. Hence, in our implementation, we use batched tasks to find the target solution: we first create N tasks and apply OpenMP with dynamic scheduling to complete them. If the target solution is found, the algorithm will terminate. Otherwise, we will generate next N tasks to repeat the same process. It is the compromise with the architecture of OpenMP and the batch size N should be carefully tuned for each environment. The performance is usually worse than MPI because it will explore more iterations before terminations than MPI. Results shown in the next sections demonstrate that claim.

5 Result

1. Test metrics and test cases

    i. Because our project is to accelerate a certain algorithm, we chose to measure the result in terms of how many time it spend in seconds. To be specific, we used python time library to record the time spent for the critical section in our parallel implementation. In addition, since we have the sequential version of algorithm as a baseline, we can also measure the speedup versus resources we have used to analyze how efficient our parallel implementation is.

    ii. We mainly used test mazes of four different sizes. The sizes are categorized into tiny, medium and big mazes. Here are some typical example mazes for each size.

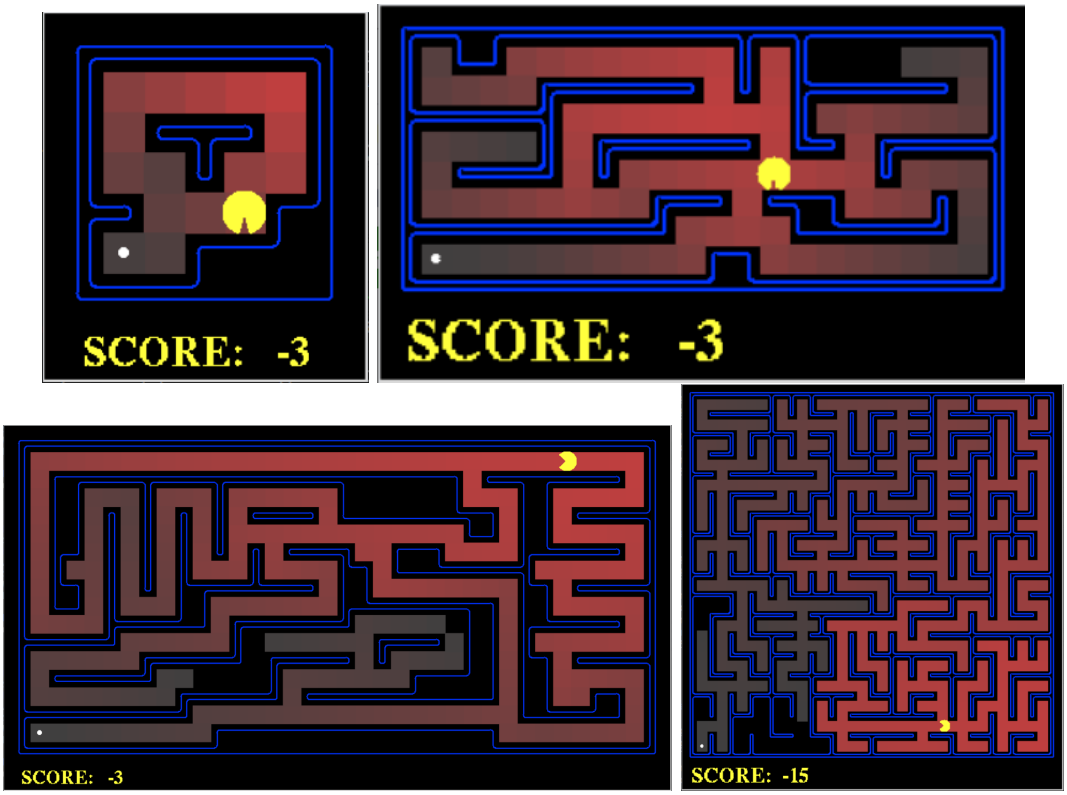

Figure 1: tiny/small/medium/big maze

As you can see above, these mazes differs a lot in size. These test cases are useful to decide how large the multi-process and communication overhead is in our implementation. The baseline run time for a sequential iterative deepening dfs for each of these test mazes are also provided below.

size	baseline(sequential version) run time (s)
tiny	0.0015
small	0.0235
medium	0.3341
big	1.5190

Table 1: Baseline run time for 4 different maze sizes

2. Result for OpenMP (pymp) implementation

   i. Measurements

   We first measured the performance on each type of maze size, in this case, we fixed the batch size to be 300 and number of processes to be 8

size	OpenMP (pymp) implementation run time (s)	speedup
tiny	0.1857	0.008x
small	0.1888	0.124x
medium	0.2323	1.43x
big	0.4170	3.64x

Table 2: OpenMP(pymp) run time for 4 different maze sizes

After getting the above results, we decided that we are more interested on the big maze where our implementation can provide the most significant speedup. We want to know how the amount of resources will influence the result. Therefore, we again measured the runtime for using different number of processes. Notice that the OpenMP is using task-based scheduling, there are actually no upper limit for the number of tasks we want to use. Here we measure the number of task range from 1 to 16.

Here is the plot for runtime versus different amount of tasks used

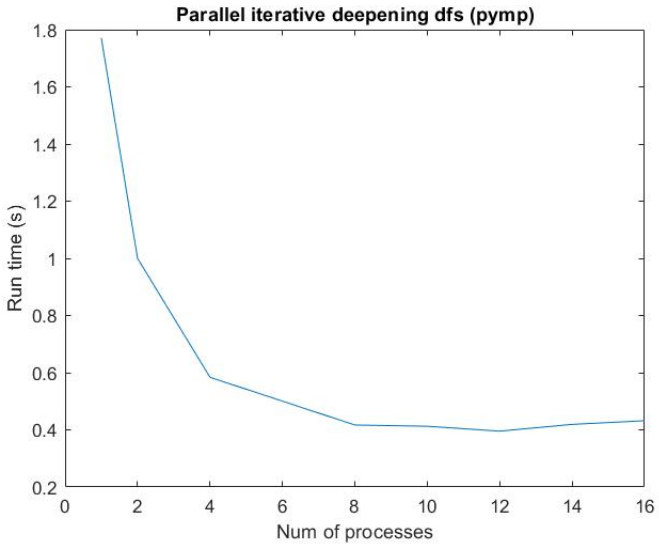

Figure 2: Runtime for pymp with different number of tasks

By comparing with the baseline implementation, we can also calculate the speedup for each number of processes used, the result is shown below:

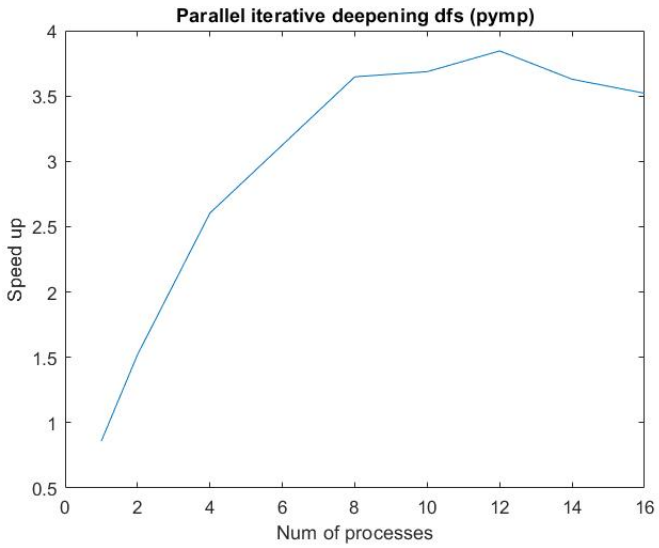

Figure 3: Speedup for pymp with different number of tasks

ii. Analysis and discussion

a. The first thing to discuss is how the input size influences the speedup performance. As you can observe from the above tables, the run time for tiny and small mazes by using our OpenMP implementation is actually worse than the sequential version of the algorithm. After taking this course, we know that this is probably because the overhead for multi-processing is too high. For example, for pymp, the multi-processing is implemented by spawn new processes when we need them. As a common sense, the spawn of processes is expensive, which takes a long time compared with the algorithm itself when the maze is small. This also explains the reason why the run time for tiny and small maze are almost the same. The 0.1857s and 0.1888s should mostly consist of the spawning of the processes. Also, the influence of large overhead can be mitigated when the run time of sequential algorithm gets large. That's mainly the reason why for medium and big, we have a greater-than-one speedup and the speedup for big is even larger maze than that of medium maze.

b. The second thing is how efficient our implementation utilizes the resources. From figure 2, we can observe that the runtime cannot be less than 0.4s as the number of tasks goes up. From the plot, the best performance is when the number of tasks is 8. We believe this makes sense because the machine we are running our tests on has 8 cores and can support 16 threads (each core supports simultaneous multi-threading

with 2 threads). It is clear that the speedup goes up when we fully utilize all the available resources. From figure 3, we can observe that the speedup is not linear with the number of tasks, which means the contention and communication cost is high when number of tasks goes up. In addition, the speedup is not ideal. The ideal speedup should be 8x when we fully utilize all the resources on the device. However, the best speedup we got from our experiments is only about 3.8x. Here are possible reasons: The memory contention is high. Because each task is reading the variable (completion_flag) in the shared address space to know whether there is a solution found globally. That variable is accessed many times during the execution. Although it should be only invalidated once (when a solution is found is some process) in each process's local cache, this still influence the performance a lot. Another important factor is the spawn of processes. As mentioned above, the spawn of 8 processes can take a long and this overhead is not avoidable regardless of the input size, it will probably degrade the potential speedup a lot.

3. Result for MPI (mpi4py) implementation

   i. Measurements

   Again, we first measured the performance on each type of maze size. For MPI implementation, we fixed number of processes to be 8.

size	MPI (mpi4py) implementation run time (s)	speedup
tiny	0.001332	1.126x
small	0.003093	7.59x
medium	0.04518	7.39x
big	0.2290	6.9x

Table 3: MPI implementation run time for 4 different maze sizes

After observing the above results, we decides to do more experiments only on the big maze (the reason will be discussed in analysis and discussion section). We are interested in how the number of MPI processes will influence the result performance. And therefore, can compare it with the OpenMP implementation. Notice that the number of processes we can use in this MPI implementation is restricted by the number of cores a device has. We chose to run the test on GHC machine, which has 8 cores and therefore measured

performance by adjusting different number of processes to start with.

Here is the plot for runtime versus different amount of cores(processes) used

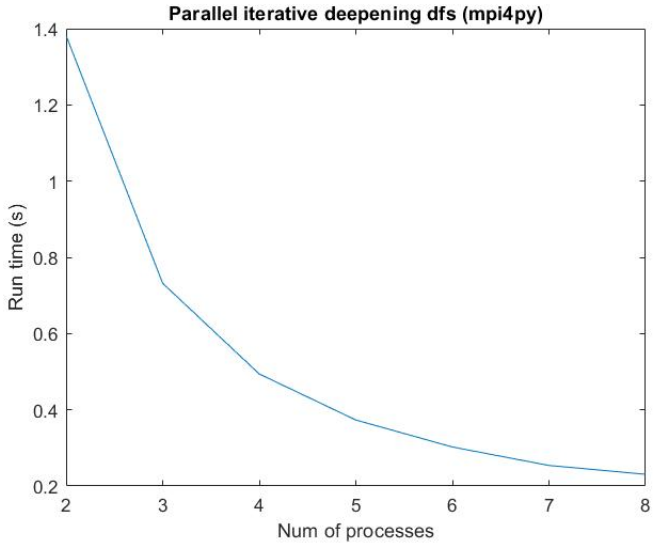

Figure 4: Runtime for MPI with different number of processes

Again, we also calculated speedup for each number of processes

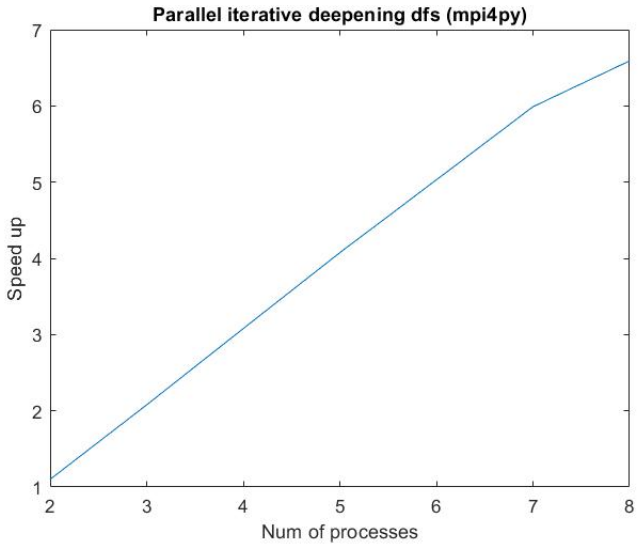

Figure 5: Runtime for MPI with different number of processes

ii. Analysis and discussion

a. Again, let's start from discussing how the input size influences the parallel implementation performance. Form table 3, we can observe that when testing for the tiny mazes, there is only 1.126x speedup for the algorithm to complete. This is much less

than that of test mazes of other sizes. The main reason for this is that the communication overhead made up of most of the run time when the test maze is small. To be specific, it should be about the same level of run time of the algorithm run on such a small maze. As long as the maze is large enough (for MPI implementation, small maze is already large enough), the overhead of communication is insignificant and we can observe almost ideal speedup.

Here are some more comments on the speedup for small and medium test mazes. As we are using 8 process–1 of them acting only as the coordinator, the other 7 acting as actual worker nodes, the ideal speedup for this part should be 7x. However, we measured speedup that is greater than 7x for small and medium test mazes. We believe the main reason for this is that the measured run time fluctuates in a certain range because randomness of execution and dynamic scheduling. When the total run time is small, this influence will be enlarged.

b. For the big maze, we measured the speedup when using different number of processes. The result from Figure 5 is actually exciting. From our measured data, the speedup for each number of processes is just a lot bit below the ideal speedup. For example, when using 7 processes, the ideal speedup is 6x and the measured speedup is 5.9891x, which basically means in this case, the algorithm can be perfectly parallelized. This implies that, when the test input size is large, the communication overhead of MPI can be almost ignored.

   This result is actually a little bit out of our expectation. Because from the original paper and our previous estimation, the speedup should be at least limited by the run time difference between different iteration with different depth limits. Our guess for why the parallel implementation works so well in our situation is that: our branching factor is actually much smaller. When using IDDFS tree search, the upper bound of branching factor is limited by 4 (when current location is surrounded by walls, it will be less than 4, otherwise, it should be 4). However, our IDDFS is using graph search (because in terms of Pac-Man path finding, a path which revisits a same location twice does not make much sense). In this case, the average branching factor can be much smaller than expected. Although it is hard to measure the actual average branching

factor, we can imagine that by observing the large maze example above. In that maze, most locations are surrounded by at least 2 walls, which means the average branching factor can probably be only between 1 and 2! This shrinks the run time difference between different iterations a lot and result in the fact that the parallel implementation can work really well on this specific case. Also, allowing sub-optimal to be returned by the algorithm also somehow contributes to the potential speedup we can observe.

4. Why MPI implementation outperforms OpenMP implementation?

   We think the main reason leads to this difference is the timing of spawning child processes. While all the child processes are spawn at the very beginning as the program starts, OpenMP's spawning of child processes are executed during the running of the algorithm.

6 Division of Work

Equal work was performed by both project members.

References

[1] J. DeNero, D. Klein, B. Miller, N. Hay, and P. Abbeel. Berkeley ai (cs188: Artificial intelligence). `http://ai.berkeley.edu/`, 2019. [Online; accessed 9-December-2019].

[2] C. Powley and R. E. Korf. Single-agent parallel window search. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 13(5):466–477, May 1991. ISSN 1939-3539. doi: 10.1109/34.134045.

[3] Wikipedia contributors. A* search algorithm — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=A*_search_algorithm&oldid=925009518`, 2019. [Online; accessed 9-December-2019].

[4] Wikipedia contributors. Iterative deepening depth-first search — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=Iterative_deepening_depth-first_search&oldid=925129768`, 2019. [Online; accessed 9-December-2019].

[5] Wikipedia contributors. Pac-man — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=Pac-Man&oldid=929876956`, 2019. [Online; accessed 9-December-2019].